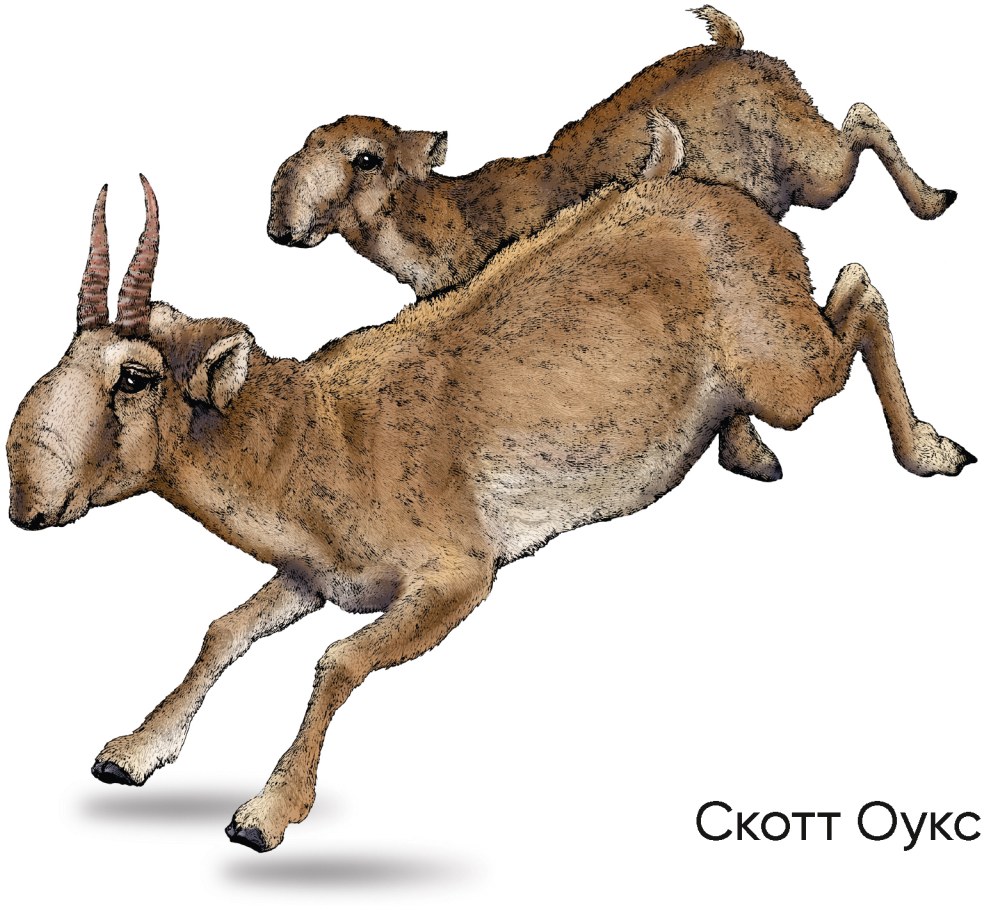
O'REILLY®

Второе издание

Эффективный Java

Тюнинг кода на Java 8, 11 и дальше



Скотт Оукс

SECOND EDITION

Java Performance

In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond

Scott Oaks

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Эффективный Java

Тюнинг кода на Java 8, 11 и дальше

Второе издание

Скотт Оукс

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.973.2-018.1
УДК 004.43
О-90

Оукс Скотт

О-90 Эффективный Java. Тюнинг кода на Java 8, 11 и дальше. — СПб.: Питер, 2021. — 496 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1757-4

Программирование и тестирование обычно принято относить к разным профессиональным сферам. Скотт Оукс — признанный эксперт по языку Java — уверен, что если вы хотите работать с этим языком, то обязаны понимать, как выполняется код в виртуальной машине Java, и знать, какие настройки влияют на производительность.

Вы сможете разобраться в производительности приложений Java в контексте как JVM, так и платформы Java, освоите средства, функции и процессы, которые могут повысить производительность в LTS-версиях Java, и познакомитесь с новыми возможностями (такими как предварительная компиляция и экспериментальные уборщики мусора).

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1492056119 англ.

Authorized Russian translation of the English edition of Java Performance, 2nd Edition ISBN 9781492056119 © 2020 Scott Oaks
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1757-4

Краткое содержание

Оглавление

Предисловие

Когда издательство O'Reilly впервые обратилось ко мне с предложением написать книгу о настройке производительности Java, я засомневался. «Производительность Java, — подумал я. — Разве эта тема еще не исчерпана?» Да, я работаю над улучшением производительности приложений Java (и других приложений) в своей повседневной работе, но я предпочитаю думать, что трачу большую часть своего времени на борьбу с алгоритмическими неэффективностями и узкими местами внешних систем, а не на причины, связанные напрямую с оптимизацией Java.

После некоторых размышлений я понял, что в этом случае (как обычно) обманывал себя. Безусловно, производительность комплексных систем отнимает у меня немало времени, и я иногда сталкиваюсь с кодом, который использует алгоритм $O(n^2)$, хотя мог бы использовать алгоритм со сложностью $O(\log N)$. И мне ежедневно приходится думать об эффективности уборки мусора, или производительности компилятора JVM, или о том, как добиться наилучшей производительности различных Java API.

Я вовсе не собираюсь принижать колоссальный прогресс в области производительности Java и JVM за последние двадцать с лишним лет. Когда я работал в отделе Java-евангелизма в компании Sun в конце 1990-х годов, единственным реальным «эталонным тестом» была программа CaffeineMark 2.0 от Pendragon software. По разным причинам структура этой программы ограничивала ее ценность; и все же в те дни мы обожали говорить всем, что на основании этого теста производительность Java 1.1.8 в 8 раз превышает производительность Java 1.0. И надо сказать, это было чистой правдой — у Java 1.1.8 появился JIT-компилятор, тогда как версия Java 1.0 была практически полностью интерпретируемой.

Затем комитеты по стандартизации начали разрабатывать более строгие тесты, и оценка производительности Java стала строиться вокруг этих тестов. Результатом стало непрерывное улучшение всех областей JVM — уборки мусора, компиляции и API. Конечно, этот процесс продолжается и сегодня, но как ни странно, работа в области производительности становится все сложнее и сложнее. Восьмикратное повышение производительности за счет введения

JIT-компилятора было достаточно прямолинейным технологическим достижением; и хотя компилятор продолжает совершенствоваться, таких улучшений мы уже не увидим. Параллелизация уборщика мусора обеспечила огромный прирост производительности, но последние изменения были не столь значительными.

Этот процесс типичен для приложений (а ведь JVM — всего лишь приложение): в начале проекта достаточно легко обнаруживаются архитектурные недостатки (или ошибки в коде), устранение которых обеспечивает значительный рост производительности. В зрелом приложении отыскать такие возможности для усовершенствований намного труднее.

Эта предпосылка скрывалась за моим исходным убеждением о том, что для технического мира тема производительности Java в значительной степени утратила актуальность. Некоторые обстоятельства убедили меня в том, что я ошибался.

Первое — количество вопросов, которые я получаю ежедневно относительно того, как некоторый аспект JVM работает в тех или иных обстоятельствах. В Java постоянно приходят новые специалисты, и в некоторых областях поведение JVM остается достаточно сложным, чтобы учебное руководство могло принести пользу. Второе — похоже, сами изменения сред современных вычислений изменяют проблемы производительности, с которыми технические специалисты сталкиваются в наши дни.

За несколько последних лет проблемы производительности разделились на два направления. С одной стороны, очень большие машины, на которых могут выполняться JVM с очень большим размером кучи, стали рядовым явлением. Для новых условий в JVM появился новый уборщик мусора (G1), который — как новая технология — требует чуть большей ручной настройки, чем традиционные уборщики мусора. В то же время облачные технологии возродили актуальность небольших однопроцессорных машин: вы можете воспользоваться услугами Oracle, Amazon или другой компании и недорого арендовать однопроцессорную машину для работы небольшого сервера приложения. (На самом деле вы получаете не однопроцессорную машину, а виртуальный образ ОС на очень большой машине, но виртуальная ОС ограничена использованием одного процессора. С точки зрения Java это то же самое, что однопроцессорная машина.) В таких средах правильное управление небольшими объемами памяти начинает играть важную роль.

Платформа Java также продолжает развиваться. Каждое новое издание Java предоставляет новые языковые средства и новые API, повышающие эффективность труда разработчиков (даже если они не всегда повышают эффективность приложений). От эффективного применения этих языковых средств может зависеть судьба приложения — провал или коммерческий успех. А эволюция платформы поднимает ряд интересных вопросов из области производитель-

ности: бесспорно, использование JSON для обмена информацией между двумя программами намного проще оптимизированного закрытого протокола. Экономия времени разработчика важна, но вы должны следить за тем, чтобы эффективность разработки сопровождалась приростом производительности.

Для кого написана эта книга

Эта книга написана для специалистов по производительности и разработчиков, которые желают разобраться в различных аспектах влияния JVM и Java API на производительность. Если сейчас воскресный вечер, в понедельник ваш сайт должен принять первых посетителей, а вы хотите на скорую руку решить проблемы с производительностью — эта книга не для вас.

Если у вас еще нет опыта анализа производительности и вы начинаете заниматься им в Java — эта книга может вам помочь. Конечно, я постараюсь предоставить достаточно информации и контекста, чтобы начинающие специалисты могли понять, как следует применять основные настройки и принципы регулировки производительности в приложениях Java. Тем не менее системный анализ — весьма обширная область. Существует ряд превосходных ресурсов, посвященных системному анализу в целом (и конечно, эти принципы применимы к Java); в этом смысле книга станет полезным приложением к этим текстам.

Однако на фундаментальном уровне, для того чтобы ваши программы Java работали действительно быстро, необходимо глубоко понимать, как работает JVM (и Java API). Существуют сотни флагов настройки Java, и настройка JVM не должна сводиться к тому, чтобы бездумно изменять эти флаги и смотреть, что из этого получится. Вместо этого я постарался подробно рассказать о том, что делают JVM и API, в надежде, что вы поймете суть происходящего, разберетесь в конкретном поведении приложения и осознаете, *почему* оно неэффективно работает. После этого вам останется решить простую (или относительно простую) задачу: избавиться от нежелательного (неэффективного) поведения.

У работы в области производительности Java есть один интересный аспект: по уровню подготовки разработчики часто сильно отличаются от инженеров из группы производительности или контроля качества. Я знаю разработчиков, которые помнят тысячи сигнатур малоизвестных методов, редко используемых в Java API, но понятия не имеют, что делает флаг `-Xmn`. И я знаю инженеров по тестированию, которые могут выжать последнюю каплю производительности установкой различных флагов уборщика мусора, но едва смогут написать программу «Hello World» на Java.

Производительность Java включает обе области: настройку флагов компилятора, уборщика мусора и т. д., а также оптимальное использование API. Я буду счи-

тать, что вы хорошо умеете писать программы на Java. Даже если вас в первую очередь интересуют непрограммные аспекты Java, мы проведем немало времени за обсуждением программ, включая примеры, которые предоставляют тестовые данные для примеров.

Впрочем, если прежде всего вас интересует производительность JVM — то есть изменение поведения JVM без какого-либо программирования, — многие части книги вам все равно пригодятся. Просто пропустите материал, относящийся к программированию, и сосредоточьтесь на областях, которые вас интересуют. Возможно, в процессе чтения вы получите некоторое представление о том, как приложения Java могут влиять на производительность JVM, и начнете предлагать изменения разработчикам, чтобы они могли упростить вашу работу по тестированию производительности.

Благодарности

Хочу поблагодарить всех, кто помогал мне в работе над книгой. Во многих отношениях в ней были собраны знания, накопленные мной за последние 20 лет в Java Performance Group и других технических группах в Sun Microsystems и Oracle, так что список людей, поделившихся своим мнением о книге, весьма обширен. Всем инженерам, с которыми я работал за это время, и особенно тем, которые терпеливо отвечали на мои вопросы в течение последнего года, — спасибо!

Хочу особо поблагодарить Стэнли Гуана (Stanley Guan), Азима Дживу (Azeem Jiva), Ким ЛиЧонга (Kim LiChong), Дипа Сингха (Deep Singh), Мартина Вербурга (Martijn Verburg) и Эдварда Юэ Шун Вона (Edward Yue Shung Wong) за время, потраченное на рецензирование черновиков и предоставление исключительно ценной обратной связи. Уверен, что им не удалось найти все мои ошибки, хотя материал книги сильно улучшился на основании полученной от них информации. Второе издание было серьезно улучшено благодаря всеобъемлющей и вдумчивой помощи, которую предоставили Бен Эванс (Ben Evans), Род Хилтон (Rod Hilton) и Майкл Хангер (Michael Hunger). Мои коллеги Эрик Каспоул (Eric Caspole), Чарли Хант (Charlie Hunt) и Роберт Страут (Robert Strout) из группы производительности Oracle HotSpot также терпеливо помогали мне разобраться с различными проблемами второго издания.

Сотрудники издательства O'Reilly были как всегда отзывчивы и готовы прийти на помощь. Мне посчастливилось работать с редактором Мэг Бланшетт (Meg Blanchette) над первым изданием, а Амелия Блевинс (Amelia Blevins) тщательно и усердно руководила работой над вторым изданием. Спасибо за всю помощь и содействие!

И наконец, хочу поблагодарить своего партнера Джеймса, который мирился с долгими вечерами и обедами на выходных, когда я постоянно пребывал в отвлеченном состоянии.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение! На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Введение

Эта книга посвящена науке и искусству настройки производительности Java.

«Научная» часть этого утверждения никого не удивит; любые обсуждения, относящиеся к производительности, включают множество цифр, метрик и аналитики. Многие специалисты по оптимизации имеют хорошую теоретическую подготовку, а применение научной строгости является важнейшим аспектом обеспечения максимального быстродействия.

Но как насчет «искусства»? Представление о том, что оптимизация сочетает в себе науку и искусство, вряд ли можно назвать новым, но обычно оно не находит явного подтверждения в обсуждениях по поводу производительности. Отчасти это объясняется тем, что концепция «искусства» не является тем, чему нас учили. Однако то, что одним кажется искусством, по большому счету основано на глубоких знаниях и опыте. Говорят, что любая достаточно развитая технология неотличима от магии[1]; и бесспорно, рыцарю Круглого стола мобильный телефон покажется чем-то волшебным. Аналогичным образом результат, которого может добиться хороший специалист по оптимизации, может показаться произведением искусства, однако это искусство на самом деле является результатом применения глубоких знаний, опыта и интуиции.

Что касается опыта и интуиции, книга вам помочь не сможет. Зато она может предоставить глубокие знания — в той перспективе, что систематическое применение знаний со временем способно развить навыки, необходимые для того, чтобы стать хорошим специалистом по производительности Java. Она написана для того, чтобы дать читателю глубокое понимание различных аспектов производительности платформы Java.

Эти знания делятся на две широкие категории. К первой относится производительность самой виртуальной машины Java (JVM, Java Virtual Machine): конфигурация JVM влияет на многие аспекты производительности программы.

[1] Один из так называемых «законов Кларка» — https://ru.wikipedia.org/wiki/Три_закона_Кларка. — *Примеч. пер.*

Разработчиков с опытом работы на других языках необходимость настройки может раздражать, хотя на самом деле процесс настройки JVM полностью аналогичен выбору флагов компилятора в процессе компиляции для программистов C++, настройке соответствующих переменных в файле php.ini для PHP-разработчиков и т. д.

Второй аспект, в котором вы должны хорошо разбираться, — влияние функциональности платформы Java на производительность. Обратите внимание на термин «платформа»: некоторые возможности (например, многопоточность и синхронизация) являются частью языка, тогда как другие (например, операции со строками) относятся к стандартному Java API. Хотя между языком Java и Java API существуют важные различия, в данном контексте они будут рассматриваться на одном уровне. В книге рассматриваются обе грани платформы.

Производительность JVM зависит прежде всего от флагов оптимизации, тогда как производительность платформы в большей степени определяется применением передовых практик в коде приложения. Долгое время они считались разными специализациями: разработчики программируют, а группа производительности проводит тестирование и рекомендует исправления для выявленных проблем с производительностью. Это различие никогда не приносило особой пользы — каждый программист, работающий на Java, должен в равной степени понимать, как код ведет себя в JVM и какие виды оптимизации с большой вероятностью улучшат его производительность. С переходом проектов на модель DevOps эти различия постепенно становятся менее жесткими. Знание сферы в общем придаст вашим работам налет искусства.

Структура книги

Обо всем по порядку: в главе 2 рассматриваются общие методологии тестирования приложений Java, включая некоторые ловушки, встречающиеся при хронометраже Java-кода. Так как анализ производительности требует понимания того, что делает приложение, в главе 3 приводится обзор некоторых средств мониторинга приложений Java.

После этого наступает момент для углубленного анализа быстродействия. Мы начнем с самых распространенных аспектов оптимизации: JIT-компиляции (глава 4) и уборки мусора (главы 5 и 6). В последующих главах внимание будет сосредоточено на оптимальном использовании различных частей платформы Java: использовании памяти в куче Java (глава 7), низкоуровневом использовании памяти (глава 8), быстродействии многопоточных приложений (глава 9), серверных технологиях Java (глава 10), работе с базами данных (глава 11) и общих рекомендациях Java SE API (глава 12).

В приложении приведен список всех флагов оптимизации, рассмотренных в книге, с указанием глав, где можно найти более подробное описание.

Платформы и соглашения

Хотя эта книга посвящена производительности Java, на производительность влияет несколько факторов: прежде всего, это версия Java, а также аппаратная и программная платформа, на которой она работает.

Платформы Java

В этой книге вопросы производительности рассматриваются на примере Oracle HotSpot JVM (Java Virtual Machine) и JDK (Java Development Kit) версий 8 и 11. Также эта комбинация обозначается термином Java SE (Standard Edition). Исполнительная среда JRE (Java Runtime Environment) представляет собой подмножество JDK, содержащее только JVM, но поскольку инструменты из JDK важны для анализа производительности, в этой книге центральное место занимает JDK. В практическом плане это означает, что в ней также рассматриваются платформы, созданные на базе от репозитория OpenJDK этой технологии, включающие JVM, опубликованные в проекте AdoptOpenJDK. Строго говоря, двоичные файлы Oracle требуют лицензии для коммерческого использования, а двоичные файлы AdoptOpenJdK поставляются на условиях лицензии с открытым исходным кодом. Для наших целей эти две версии будут рассматриваться как единое целое, которое мы будем обозначать термином *JDK*, или *платформа Java*[1].

Для этих версий был выпущен целый ряд обновлений с исправлениями ошибок. На момент написания книги текущей версией Java 8 была версия jdk8u222 (версия 222), а текущей версией Java 11 — версия 11.0.5. Важно использовать как минимум эти версии (или более поздние), особенно в случае Java 8. Ранние выпуски Java 8 (приблизительно до jdk8u60) не содержат многие важные улучшения в области производительности и функциональные возможности, которые рассматриваются в книге (особенно в отношении уборки мусора и уборщика мусора G1).

Эти версии JDK были выбраны из-за того, что они пользуются долгосрочной поддержкой (LTS, Long-Term Support) от Oracle. Сообществу Java предоставлено право разрабатывать собственные модели поддержки, но до настоящего времени

[1] В отдельных редких случаях эти два понятия различаются; например, в версиях Java из AdoptOpenJDK присутствуют новые уборщики мусора из JDK 11. Я буду указывать на эти различия там, где они будут актуальны.

следовали модели Oracle. Таким образом, эти версии будут поддерживаться и оставаться доступными в течение некоторого времени: по крайней мере до 2023 года для Java 8 (через AdoptOpenJDK; позднее для продленных контрактов поддержки Oracle) и по крайней мере до 2022 года для Java 11. Ожидается, что следующая долгосрочная версия будет выпущена в конце 2021 года.

Что касается промежуточных версий, в обсуждение Java 11 были включены возможности, которые были изначально доступны в Java 9 или Java 10, несмотря на то, что эти версии не поддерживаются как компанией Oracle, так и сообществом в целом. При обсуждении этих возможностей я выражаюсь несколько неточно; может показаться, что я говорю, будто функции X и Y появились в Java 11, тогда как они были доступны в Java 9 или 10. Java 11 стала первой LTS-версией, поддерживающей эти возможности, и важно именно это: так как Java 9 и 10 не используются, на самом деле не так важно, где впервые появилась та или иная возможность. Аналогичным образом, хотя к моменту выхода книги уже появится Java 13, в книге Java 12 и Java 13 почти не рассматриваются. Вы можете использовать эти версии в своих приложениях, но только в течение полугода, после чего необходимо будет обновиться до новой версии (так что к тому моменту, когда вы будете читать эти слова, Java 12 уже не поддерживается, а версия Java 13, если и поддерживается, скоро будет заменена Java 14). Я расскажу о некоторых возможностях этих промежуточных версий, но поскольку в большинстве сред они вряд ли будут задействованы в рабочем коде, основное внимание будет уделяться Java 8 и 11.

Существуют и другие реализации спецификации языка Java, включая ответвления реализации с открытым кодом. Одну из них предоставляет AdoptOpenJDK (Eclipse OpenJ9), другие доступны от других разработчиков. Чтобы иметь возможность использовать имя Java, все эти платформы должны пройти тест на совместимость, однако эта совместимость не всегда распространяется до аспектов, рассматриваемых в книге. Это относится в первую очередь к флагам оптимизации. В любую реализацию JVM входит один или несколько уборщиков мусора (GC), но флаги для настройки реализаций GC от каждого разработчика зависят от конкретного продукта. Таким образом, хотя основные концепции книги актуальны для любой реализации Java, конкретные флаги и рекомендации относятся только к HotSpot JVM.

Данное предупреждение относится и к ранним версиям HotSpot JVM — флаги и их значения по умолчанию изменяются от версии к версии. Флаги, рассмотренные в тексте, действительны для Java 8 (конкретно версии 222) и 11 (конкретно 11.0.5). В более поздних версиях часть информации может измениться. Всегда проверяйте сопроводительную документацию новых версий — в ней могут быть описаны важные изменения.

На уровне API разные реализации JVM обладают намного большей совместимостью, хотя даже в этом случае могут существовать нетривиальные различия

между реализациями конкретного класса на платформе Oracle HotSpot и на альтернативных платформах. Классы должны быть функционально эквивалентными, но фактические реализации могут изменяться. К счастью, такие различия встречаются редко, а вероятность того, что они радикально повлияют на быстродействие, невелика.

В оставшейся части книги термины *Java* и *JVM* следует понимать как относящиеся к конкретной реализации Oracle HotSpot. Строго говоря, фраза «JVM не компилирует код при первом выполнении» неточна — некоторые реализации Java *компилируют* код при первом выполнении. Использовать эту сокращенную запись намного проще, чем постоянно писать (и читать) «Oracle HotSpot JVM...».

Флаги оптимизации JVM

JVM (с немногочисленными исключениями) получает флаги двух типов: логические флаги и флаги с параметром.

Логические флаги используют следующий синтаксис: `-XX:+`*ИмяФлага* устанавливает флаг, а `-XX:-`*ИмяФлага* сбрасывает его.

Флаги с параметром используют синтаксис `-XX:`*ИмяФлага*`=`*значение*; в этом случае *ИмяФлага* присваивается указанное *значение*. В тексте книги конкретное значение флага обычно заменяется чем-то, указывающим на произвольное значение. Например, `-XX:NewRatio=`*N* означает, что флагу `NewRatio` будет присвоено произвольное значение *N* (а последствия выбора *N* станут темой обсуждения).

Значения по умолчанию для всех флагов рассматриваются при первом упоминании флага. Значение по умолчанию часто определяется несколькими факторами: платформой, на которой работает JVM, а также аргументами командной строки JVM. Если у вас возникнут сомнения, в разделе «Основная информация VM» на с. 87 показано, как использовать флаг `-XX:+PrintFlagsFinal` (по умолчанию `false`) для определения значения по умолчанию конкретного флага в конкретной среде для заданной командной строки. Процесс автоматической оптимизации флагов в зависимости от параметров среды называется *эргономикой*.

Реализация JVM, загруженная с сайтов Oracle и AdoptOpenJDK, называется *рабочей сборкой* JVM. При сборке JVM из исходного кода можно построить много разных вариантов сборки: отладочные сборки, сборки для разработчиков и т. д. Такие сборки часто содержат дополнительную функциональность. В частности, сборки для разработчиков включают расширенный набор флагов оптимизации, чтобы разработчики могли поэкспериментировать с самыми незначительными аспектами различных алгоритмов, используемых JVM. Такие флаги обычно в книге не рассматриваются.

Аппаратные платформы

При выходе первого издания книги ситуация с оборудованием была совсем не такой, как сегодня. Многоядерные машины были популярными, но 32-разрядные платформы и однопроцессорные платформы все еще были широко распространены. Другие платформы, используемые в наше время, — виртуальные машины и программные контейнеры — еще только занимали свое место. Ниже приведен краткий обзор того, как эти платформы влияют на материал книги.

Многоядерное оборудование

Процессоры практически всех современных машин содержат несколько ядер, которые JVM (или любая другая программа) воспринимает как разные процессоры. Обычно все ядра включаются в систему гиперпоточного использования. Компания Intel предпочитает термин «*гиперпоточность*» (hyper-threading), хотя AMD (и другие) используют термин «*одновременная многопоточность*» (simultaneous multithreading), а некоторые производители микросхем говорят об «аппаратных цепях выполнения» в отдельных ядрах. Все это одно и то же, и я буду называть эту технологию «гиперпоточностью».

С точки зрения производительности важнейшей характеристикой машины является количество ядер ее процессора. Возьмем базовую 4-ядерную машину: каждое ядро может работать (в целом) независимо от остальных, так что машина с четырьмя ядрами может выдавать результаты до 4 раз быстрее, чем машина с одним ядром. (Конечно, это зависит от особенностей программного обеспечения.)

В большинстве случаев каждое ядро содержит два гиперпотока. Эти потоки не могут считаться независимыми друг от друга: в любой момент времени ядро может выполнять только один из них. Часто выполнение потока приостанавливается: например, потоку может потребоваться загрузка значения из основной памяти, а этот процесс может занять несколько тактов процессора. На ядре с одним потоком этот единственный поток будет приостановлен, а такты процессора будут потеряны. Если же ядро поддерживает два потока, оно может переключиться на другой поток и выполнять команды из этого потока.

Таким образом, все выглядит так, словно 8-ядерная машина с гиперпоточностью может выполнять команды из 8 потоков одновременно (хотя технически она может выполнять только четыре команды на такт процессора). С точки зрения операционной системы, а следовательно, Java и других приложений, машина оснащена восемью процессорами. Тем не менее не все эти процессоры равны с точки зрения производительности. Если запустить одну счетную задачу, создающую интенсивную нагрузку на процессор, она будет использовать

одно ядро; вторая счетная задача будет использовать второе ядро, и так далее до четырех. Вы можете запустить четыре независимые счетные задачи и добились четырехкратного повышения быстродействия.

Если добавить пятую задачу, она сможет выполняться только во время приостановки одной из остальных задач, что в среднем происходит где-то от 20% до 40% времени. Каждая дополнительная задача сталкивается с той же проблемой. Таким образом, добавление пятой задачи приведет к повышению производительности всего около 30%; в конечном итоге восемь процессоров обеспечат повышение производительности только в 5–6 раз по сравнению с одним ядром (без гиперпоточности).

Этот пример еще встретится вам в книге. Уборка мусора в значительной мере является счетной задачей, поэтому в главе 5 будет показано, как гиперпоточность влияет на параллелизацию алгоритмов уборки мусора. В главе 9 обсуждаются общие аспекты эффективного использования средств многопоточности Java, поэтому в ней также будет представлен пример масштабирования гиперпоточных ядер.

Программные контейнеры

Самое значительное изменение в механизмах развертывания Java за последние годы заключается в том, что теперь они часто развертываются в программных контейнерах. Конечно, это изменение не ограничивается Java; это тенденция в отрасли, которая только ускоряется с переходом на облачные вычисления.

Особенно важны два контейнера. Первый — виртуальная машина, которая создает полностью изолированную копию операционной системы на подмножестве оборудования, на котором запускается виртуальная машина. Эта концепция лежит в основе облачных вычислений: у вашего провайдера облачных вычислений имеется вычислительный центр с очень мощными машинами. Такие машины могут иметь до 128 ядер, хотя обычно их меньше по соображениям эффективности затрат. С точки зрения виртуальной машины это не столь важно: виртуальная машина получает доступ к подмножеству этого оборудования. А следовательно, отдельная виртуальная машина может содержать два ядра (и четыре логических процессора, поскольку ядра обычно являются гиперпоточными) и 16 Гбайт памяти.

С точки зрения Java (и других приложений) эта виртуальная машина неотличима от обычной машины с двумя ядрами и 16 Гбайт памяти. Для целей оптимизации и анализа производительности достаточно рассматривать ее именно с этих позиций.

Второй контейнер, заслуживающий внимания, — контейнер Docker. Процесс Java, работающий в контейнере Docker, может не знать, что он находится в та-

ком контейнере (хотя и может определить этот факт средствами инспекции), но контейнер Docker представляет собой всего лишь процесс (возможно, с ограниченными ресурсами) внутри работающей ОС. Как следствие, уровень его изоляции от использования процессоров и памяти другими процессами несколько отличается. Как вы вскоре увидите, подход к решению этой проблемы отличается в ранних версиях Java 8 (до обновления 192) и более поздней версии Java 8 (и всех версиях Java 11).

ИЗБЫТОЧНОЕ РЕЗЕРВИРОВАНИЕ ВИРТУАЛЬНЫХ МАШИН

Провайдеры облачных вычислений имеют возможность осуществить избыточное резервирование виртуальных машин на физическом оборудовании. Допустим, физическая машина оснащена 32 ядрами; провайдер обычно решает развернуть на ней восемь 4-ядерных виртуальных машин, чтобы каждая виртуальная машина имела четыре выделенных ядра.

Для экономии провайдер также может развернуть шестнадцать 4-ядерных виртуальных машин. Теоретически маловероятно, чтобы все 16 виртуальных машин были заняты одновременно; если занята только половина из них, то физических ядер будет достаточно для удовлетворения их потребностей. Но если занятых машин окажется слишком много, они начнут конкурировать за ресурсы процессоров и их быстродействие пострадает.

Также провайдеры облачных вычислений могут регулировать использование процессора виртуальной машиной; это позволит виртуальной машине работать в активных периодах, в которых она потребляет мощности выделенного процессора, но такой режим не будет сохраняться с течением времени. Такой режим часто встречается в бесплатных или пробных предложениях, для которых типичны другие ожидания по быстродействию.

Конечно, такие факторы сильно влияют на производительность. Однако их эффект не ограничивается Java и влияет на Java в точно такой же степени, как и на все остальное, что работает на виртуальной машине.

По умолчанию контейнер Docker может свободно использовать все ресурсы машины: все доступные процессоры и всю доступную память на машине. И это нормально, если вы хотите использовать Docker исключительно для упрощения развертывания единственного приложения на машине (то есть на машине работает только этот контейнер Docker). Но довольно часто требуется развернуть несколько контейнеров Docker и ограничить ресурсы для каждого контейнера. Допустим, на 4-ядерной машине с 16 Гбайт памяти мы

хотим запустить два контейнера Docker, каждому из которых доступны два ядра и 8 Гбайт памяти.

Настроить Docker для этой цели несложно, но на уровне Java могут возникнуть затруднения. Многочисленные ресурсы Java настраиваются автоматически (или эргономически) в зависимости от размера машины, на которой работает JVM. К их числу относится размер кучи по умолчанию и количество потоков, используемых уборщиком мусора (подробнее см. в главе 5), а также некоторые параметры пула потоков, упоминаемые в главе 9.

Если вы используете новую версию Java 8 (версия 192 и выше) или Java 11, то JVM поступит так, как вы ожидаете: если ограничить контейнер Docker для использования только двух ядер, значения, заданные эргономически на основании количества процессоров на машине, определяются с учетом ограничений контейнера Docker[1]. Аналогичным образом размер кучи и другие параметры, которые по умолчанию определяются на основании объема памяти на машине, основаны на ограничениях памяти, определенных для контейнера Docker.

В ранних версиях Java 8 JVM не располагает информацией об ограничениях, устанавливаемых контейнером: в процессе анализа окружения для определения объема доступной памяти и вычисления размера кучи по умолчанию JVM видит всю память на машине (вместо объема памяти, разрешенной для использования контейнеру Docker, как бы вам хотелось). При проверке количества доступных процессоров для оптимизации уборки мусора JVM видит все процессоры на машине вместо количества, выделенного контейнеру Docker. В результате JVM будет работать субоптимально: она запускает слишком много потоков и выделяет слишком большую кучу. Превышение количества потоков ведет к некоторому снижению производительности, но настоящая проблема кроется в использовании памяти: максимальный размер кучи может превысить объем памяти, выделенной контейнеру Docker. Когда куча увеличится до этого размера, контейнер Docker (а следовательно, и JVM) будет уничтожен.

В ранних версиях Java 8 параметры использования памяти и процессоров можно задать вручную. Когда в книге будут встречаться такие настройки, я буду особо отмечать те, которые необходимо отрегулировать в такой ситуации, но лучше просто обновиться до более новой версии Java 8 (или Java 11).

Контейнеры Docker создают одну дополнительную проблему для Java: в поставку Java включен обширный набор инструментов для диагностики проблем с производительностью. Часто они недоступны в контейнерах Docker. Эта проблема будет более подробно рассмотрена в главе 3.

---

[1] Ограничениям процессоров в Docker можно присвоить дробные значения. Java округляет все дробные значения до ближайшего целого.

Производительность: общая картина

В книге рассказано прежде всего о том, как лучше использовать API платформы Java и JVM, чтобы программы работали быстрее. На производительность также влияют многие внешние факторы. Эти факторы время от времени упоминаются в книге, но поскольку они не относятся к специфике Java, они не всегда обсуждаются подробно. Производительность JVM и платформы Java — лишь небольшой шаг на пути к высокой производительности выполнения.

В этом разделе представлены внешние факторы, которые по крайней мере не уступают по важности аспектам оптимизации Java, рассмотренным в книге. Подход, основанный на знании Java, дополняет эти факторы, но многие из них выходят за рамки материала книги.

Пишите более качественные алгоритмы

Многие подробности Java влияют на производительность приложения, и многие флаги оптимизации описаны в книге. Но никакого волшебного флага `-XX:+RunReallyFast` нет.

В конечном счете быстродействие приложения зависит от того, насколько качественно оно написано. Если программа перебирает все элементы массива, то JVM оптимизирует проверку границ массива, чтобы цикл выполнялся быстрее, а JVM может выполнить раскрутку цикла для дополнительного ускорения. Но если цикл предназначен для поиска конкретного элемента, никакая оптимизация в мире не заставит код массива работать так же быстро, как другая версия с хешированием.

Хороший алгоритм — самый важный фактор высокой производительности.

Пишите меньше кода

Одни из нас пишут программы ради денег, другие — для удовольствия, третьи — для поддержки сообщества, но так или иначе все мы пишем программы (или работаем в командах, которые их пишут). Трудно почувствовать, что вы вносите свой вклад в проект, сокращая код, а некоторые руководители даже оценивают разработчиков по объему написанного ими кода.

Все это понятно, но проблема в том, что маленькая хорошо написанная программа выполняется быстрее большой хорошо написанной программы. Это относится ко всем компьютерным программам вообще и к Java-программам в частности. Чем больше кода приходится компилировать, тем больше времени пройдет, прежде чем этот код начнет работать быстро. Чем больше объектов приходится создавать

и уничтожать, тем больше работы придется выполнять уборщику мусора. Чем больше объектов выделяется и хранится в памяти, тем больше времени занимает цикл уборки мусора. Чем больше классов загружается с диска в JVM, тем больше времени занимает запуск программы. Чем больше кода выполняется, тем меньше вероятность того, что он поместится в аппаратных кэшах на машине. И чем больше кода выполняется, тем больше времени займет выполнение.

НА ПУТИ К ПОРАЖЕНИЮ

У производительности есть один аспект, который может показаться парадоксальным (и угнетающим): можно ожидать, что производительность каждого приложения будет снижаться со временем (то есть с новыми циклами выпуска приложения). Часто различия остаются незамеченными, потому что усовершенствования аппаратной части позволяют новым программам выполняться с приемлемой скоростью.

Представьте, что вам приходится работать с интерфейсом Windows 10 на компьютере, который использовался для работы с Windows 95. Моим любимым компьютером всех времен был Mac Quadra 950, но на нем не работала macOS Sierra (причем даже если бы работала — она была бы очень, очень медленной по сравнению с Mac OS 7.5). На более низком уровне может показаться, что Firefox работает быстрее более ранних версий, но все эти версии по сути были второстепенными выпусками одного продукта. Firefox со всеми своими возможностями — поддержкой вкладок при просмотре, синхронизацией прокрутки и средствами безопасности — намного мощнее любой версии Mosaic, но Mosaic загружает HTML-файлы с жесткого диска приблизительно на 50% быстрее Firefox 69.0.

Конечно, Mosaic не сможет загружать реальные URL-адреса практически ни с одного популярного сайта; использовать Mosaic как основной браузер уже не получится. И этот факт тоже подтверждает более общую точку зрения: код (особенно между второстепенными выпусками) может оптимизироваться и выполняться быстрее. Мы, специалисты по производительности, должны сосредоточиться именно на этой задаче, и если мы достаточно хорошо справляемся со своим делом — сможем добиться успеха.

И этот момент очень важен; я вовсе не хочу сказать, что мы не должны работать над повышением производительности существующих приложений. Но как ни парадоксально, с добавлением новых возможностей и поддержкой новых стандартов (а это необходимо для того, чтобы не отстать от конкурирующих продуктов) программы обычно увеличиваются в размерах и начинают медленнее работать.

Я называю это принципом «смерти от тысячи порезов». Разработчик скажет, что всего лишь добавляет очень незначительную возможность, и на это времени вообще не потребуется (особенно если эта возможность не используется). А потом другие разработчики в том же проекте говорят то же самое, и внезапно производительность ухудшается на несколько процентов. Цикл повторяется в следующей версии, отчего общая производительность снижается на 10%. Пару раз в процессе разработки тестирование производительности может обнаружить превышение некоторого порога использования ресурсов — критическую точку в использовании памяти, переполнение кэша команд или что-нибудь в этом роде. В таких случаях обычные тесты производительности выявляют это конкретное условие, а группа производительности исправляет то, что выглядит как серьезная регрессия. Но со временем в проект проникает все больше мелких регрессий, а исправлять их становится все труднее и труднее.

Это вовсе не означает, что вы не должны добавлять в свой продукт новые возможности или новый код; разумеется, совершенствование программ приносит пользу. Но вы должны знать о тех компромиссах, на которые вам придется пойти, и упрощать свой код там, где это возможно.

Применяйте преждевременную оптимизацию

Термин «преждевременная оптимизация» часто приписывают Дональду Кнуту. Обычно разработчики используют этот термин, чтобы заявить, что от эффективности их кода ничего не зависит, а если и зависит — они все равно не узнают об этом, пока код не заработает. Полная цитата, если она вдруг вам еще никогда не попадалась, звучит так: «Преждевременная оптимизация — корень всех зол»[1].

Суть данного высказывания заключается в том, что в конечном итоге вы должны писать чистый, прямолинейный код, легкочитаемый и понятный. В этом контексте под оптимизацией понимается применение алгоритмических и структурных изменений, которые усложняют структуру программы, но улучшают производительность. Такие оптимизации действительно лучше отложить на будущее — например, до того момента, когда профилирование программы покажет, что усовершенствование принесет ощутимую пользу.

Однако в этом контексте оптимизация *не* означает отказа от программных конструкций, которые заведомо плохо сказываются на производительности.

[1] Нет полной ясности относительно того, кто сказал это впервые: Дональд Кнут или Топи Хоар (Тору Hoare). Но данное высказывание встречается в статье Кнута «Structured Programming with goto Statements». И в контексте оно может рассматриваться как аргумент в пользу оптимизации кода, даже если для этого требуются такие неэлегантные решения, как команды goto.

Каждая строка кода подразумевает выбор, и если у вас есть выбор между двумя простыми, прямолинейными решениями, выбирайте более эффективный вариант.

На определенном уровне этот принцип хорошо понятен опытным Java-разработчикам (пример того самого искусства, которое постигается со временем). Возьмем следующий фрагмент:

```
log.log(Level.FINE, "I am here, and the value of X is "
        + calcX() + " and Y is " + calcY());
```

Код выполняет конкатенацию строк, которая, скорее всего, окажется излишней, потому что сообщение будет выводиться в журнал только при достаточно высоком уровне протоколирования. Если сообщение не выводится, вызовы методов `calcX()` и `calcY()` также будут избыточными. Опытные Java-разработчики инстинктивно отказываются от таких решений; некоторые IDE даже помечают код и предлагают изменить его. (Впрочем, инструментарии не идеальны: скажем, NetBeans IDE помечает конкатенацию, но в предлагаемом улучшении сохраняются лишние вызовы методов.)

Код вывода сообщений в журнал лучше записать в следующем виде:

```
if (log.isLoggable(Level.FINE)) {
    log.log(Level.FINE,
            "I am here, and the value of X is {} and Y is {}",
            new Object[]{calcX(), calcY()});
}
```

В этом случае конкатенация вообще не задействована (формат сообщения не обязательно будет более производительным, но он проще), а вызовы методов или выделение памяти под массив объектов не будут выполняться при отключенном протоколировании.

Написанный таким образом код остается чистым и удобочитаемым; чтобы написать его, потребуется не больше усилий, чем с исходной версией. Ладно, соглашусь: он потребует больше нажатий клавиш и дополнительной строки с логическим условием. Но это не та преждевременная оптимизация, которой следует избегать; это решение из тех, которое учится принимать любой хороший программист.

Вырванные из контекста догмы классиков не должны мешать вам размышлять о коде, который вы пишете. Другие примеры такого рода неоднократно встречаются в книге — в том числе в главе 9, в которой обсуждается производительность безобидной на первый взгляд конструкции цикла для обработки вектора объектов.

Ищите в других местах: база данных всегда является узким местом

Если вы разрабатываете автономные приложения Java, которые не используют внешние ресурсы, для вас важна (в целом) только производительность самого приложения. При добавлении внешнего ресурса (например, базы данных) приходится учитывать производительность обеих программ. А в распределенной среде — допустим, с REST-сервером Java, распределителем нагрузки, базой данных и информационной системой предприятия — производительность сервера Java может оказаться наименьшей из проблем с производительностью.

> **ОШИБКИ И ПРОБЛЕМЫ С ПРОИЗВОДИТЕЛЬНОСТЬЮ НЕ ОГРАНИЧИВАЮТСЯ JVM**
>
> В этом разделе проблемы производительности рассматриваются на примере базы данных, но источником этих проблем может стать любая часть среды.
>
> Один из моих клиентов установил новую версию сервера приложения, после чего тестирование показало, что обработка запросов к серверу занимала все больше и больше времени. Бритва Оккама заставила меня рассмотреть все аспекты сервера приложения, которые могли быть источником проблемы.
>
> После того как многие потенциальные причины были отклонены, проблема оставалась, причем у нас не было базы данных, которую можно было обвинить в происходящем. Следующей наиболее вероятной причиной была тестовая конфигурация, а профилирование определило, что источником регрессии был генератор нагрузки Apache JMeter: каждый ответ сохранялся в списке, а при поступлении нового ответа он обрабатывал весь список для вычисления 90% времени отклика (если этот термин вам не знаком, обращайтесь к главе 2).
>
> Проблемы с производительностью могут быть обусловлены любой частью системы, в которой развернуто приложение. Анализ типичного случая рекомендует начать с рассмотрения самой новой части системы (которой часто оказывается приложение в JVM), но вы должны быть готовы рассматривать все возможные компоненты среды.

Глобальная производительность системы не является основной темой книги. В такой среде необходимо применять структурированный подход ко всем аспектам системы. Следует измерить и проанализировать нагрузку на процессор, задержку ввода/вывода и пропускную способность всех частей системы; только

после этого можно определить, какой компонент создает узкое место в эффективности системы. Существует немало отличных ресурсов по этой теме, причем эти методы и инструменты не являются специфическими для Java. Будем считать, что вы уже провели этот анализ и определили, что в вашей среде в улучшении нуждается именно компонент Java.

С другой стороны, вы не должны забывать об этом исходном анализе. Если база данных является узким местом (подсказка: это частое явление), оптимизация приложения Java, работающего с базой данных, не улучшит общего быстродействия. Более того, она может привести к противоположному результату. Как правило, при повышении нагрузки в перегруженной системе производительность такой системы ухудшается. Если изменения в приложении Java сделают его более производительным — что только повысит нагрузку на уже перегруженную базу данных, — общая производительность может ухудшиться. И тогда вы можете прийти к ошибочному выводу, что от конкретного улучшения JVM лучше отказаться.

Этот принцип — повышение нагрузки на компонент в плохо работающей системе приводит к замедлению всей системы — не ограничивается базой данных. В частности, он действует при добавлении на сервер нагрузки, требующей интенсивных вычислений, или если большее количество потоков начнет запрашивать блокировку, которую уже ожидают другие потоки, а также в множестве других сценариев. Яркий пример такого рода, в котором задействована только JVM, приведен в главе 9.

Оптимизация для типичного случая

Появляется искушение — особенно с учетом синдрома смерти от тысячи порезов — рассматривать все аспекты производительности как одинаково важные. Однако вы должны сосредоточиться на типичных сценариях использования. Этот принцип имеет несколько проявлений:

Оптимизируйте код, проводя его профилирование и концентрируясь на операциях, занимающих наибольшее время в результатах профилирования. Однако это вовсе не означает, что вы должны ограничиться рассмотрением только листовых методов в профиле (см. главу 3).

Применяйте бритву Оккама для выявления проблем с производительностью. Простейшее объяснение проблем с производительностью обычно оказывается наиболее вероятным: дефект производительности в новом коде более вероятен, чем проблема конфигурации на машине, которая в свою очередь более вероятна, чем ошибка в JVM или операционной системе. Скрытые ошибки в ОС и JVM существуют, и с исключением более вероятных причин проблем с производительностью появляется вероятность того, что рассматриваемый тестовый сце-

нарий каким-то образом активизировал эту скрытую ошибку. Однако начинать с этой маловероятной ситуации не стоит.

Пишите простые алгоритмы для самых распространенных операций в приложении. Допустим, программа вычисляет результат по математической формуле, а пользователь может выбрать, получить ли ответ с погрешностью 10% или 1%. Если большинству пользователей будет достаточно 10% погрешности, оптимизируйте эту ветвь — даже если это приведет к замедлению кода, обеспечивающего 1%-ную погрешность.

Итоги

В Java есть множество возможностей и инструментов, которые позволяют добиться наилучшего быстродействия от приложений Java. Эта книга покажет вам, как лучше использовать все возможности JVM, для того чтобы добиться от программы высокой производительности работы.

Впрочем, во многих случаях необходимо помнить, что JVM — всего лишь малая часть общей картины производительности. Среды Java, в которых производительность баз данных и других служебных подсистем не менее важна, чем производительность JVM, требуют систематического подхода к оптимизации. Этот уровень анализа производительности не является основной темой книги — предполагается, что вы провели тщательный анализ, который показал, что критичным узким местом в системе является именно компонент Java.

Не менее важную роль играет режим взаимодействия между JVM и другими областями системы — является ли это взаимодействие прямым (например, оптимальный режим работы с базой данных) или косвенным (например, оптимизация низкоуровневого использования памяти в приложении, которое использует машину совместно с другими компонентами большей системы). Информация, приведенная в книге, поможет вам избавиться от проблем с производительностью и в этих направлениях.

ГЛАВА 2

Тестирование производительности

В этой главе рассматриваются четыре главных принципа получения результатов при тестировании производительности: тестирование реальных приложений; понимание пропускной способности, пакетирования и времени отклика; понимание дисперсии; раннее и частое тестирование. Эти принципы закладывают основу для рекомендаций, приведенных в последующих главах. Они предоставляют теоретическую часть науки управления производительностью. Выполнение тестов производительности с приложениями — полезное дело, но без научного анализа эти тесты часто приводят к ошибочному или неполному анализу. В этой главе рассказано, как убедиться в правильности аналитических результатов, полученных при тестировании.

Многие примеры из последующих глав используют общее приложение, эмулирующее систему биржевых котировок; это приложение также описано в этой главе.

Тестирование реального приложения

Первый принцип гласит, что тестирование производительности должно выполняться с реальным продуктом по той схеме, по которой этот продукт будет использоваться. Упрощенно говоря, для тестирования производительности могут использоваться три категории тестового кода: *микробенчмарки*, *макробенчмарки* и *мезобенчмарки*. Каждая категория обладает своими достоинствами и недостатками. Категория, включающая реальное приложение, предоставит наилучшие результаты.

Микробенчмарки

Микробенчмарк (microbenchmark) — тест, предназначенный для измерения небольшой единицы производительности для принятия решения о том, какая из

нескольких альтернативных реализаций является предпочтительной: затраты на создание потока в сравнении с использованием пула потоков, время выполнения одного арифметического алгоритма в сравнении с альтернативной реализацией и т. д.

На первый взгляд микробенчмарки выглядят заманчиво, но те самые особенности Java, которые делают эту платформу привлекательной для программистов, — а именно JIT-компиляция и уборка мусора — затрудняют правильное написание микробенчмарков.

Микробенчмарки должны использовать свои результаты

Микробенчмарки отличаются от обычных программ во многих отношениях. Во-первых, поскольку код Java интерпретируется при нескольких начальных выполнениях, чем дольше он выполняется, тем быстрее он начинает работать. По этой причине все хронометражные тесты (не только микробенчмарки) обычно включают период *разогрева* (warm-up), во время которого JVM может откомпилировать свой код в оптимальное состояние.

Это оптимальное состояние может включать многочисленные оптимизации. Например, ниже приведен простой на первый взгляд цикл для хронометражного тестирования реализации метода, вычисляющего 50-е число Фибоначчи:

```
public void doTest() {
    // Основной цикл
    double l;
    for (int i = 0; i < nWarmups; i++) {
        l = fibImpl1(50);
    }
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(50);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

Этот код собирается измерить время выполнения метода `fibImpl1()`, поэтому он сначала разогревает компилятор, а затем проводит измерения с откомпилированным к этому моменту методом. Но скорее всего, это время будет равно 0 (или более вероятно — времени выполнения цикла без тела). Так как значение `l` нигде не читается, компилятор решает полностью пропустить его вычисление. Это зависит от того, что еще происходит в методе `fibImpl1()`, но если это простая арифметическая операция, она может быть полностью пропущена. Также не исключено, что пропущены будут только отдельные части метода — возможно,

даже с получением неправильного значения `l`; так как это значение нигде не читается, об этом никто не узнает. (Более подробная информация о том, как исключается этот цикл, приводится в главе 4.)

У этой конкретной проблемы есть обходное решение: нужно убедиться в том, что каждый результат не только записывается, но и читается. На практике изменение определения `l` с локальной переменной на переменную экземпляра (объявленную с ключевым словом `volatile`) позволит оценить производительность метода. (Причина, по которой переменная экземпляра `l` должна быть объявлена как `volatile`, также приводится в главе 9.)

МНОГОПОТОЧНЫЕ МИКРОБЕНЧМАРКИ

Необходимость в использовании `volatile`-переменной в данном примере существует даже в том случае, если микробенчмарк является многопоточным.

Будьте особенно осторожны при написании многопоточных микробенчмарков. Когда несколько потоков выполняют небольшие фрагменты кода, риск появления синхронизационных узких мест (и других артефактов многопоточности) весьма велик. Результаты многопоточных микробенчмарков часто приводят к тому, что разработчик тратит много времени на устранение синхронизационных узких мест, редко встречающихся в реальном коде — за счет других, более насущных проблем производительности.

Допустим, два потока вызывают синхронизированный метод в микробенчмарке. Большая часть времени будет проходить за выполнением этого синхронизированного метода, потому что код теста достаточно мал. Даже если всего 50% микробенчмарка проходит внутри синхронизированного метода, есть высокая вероятность того, что не менее двух потоков попытаются выполнить синхронизированный метод одновременно. В результате тест будет работать медленно, так как с добавлением новых потоков проблемы с производительностью, обусловленные повышением уровня конкуренции, только усугубляются. Как следствие, тест будет выполнять измерение эффективности того, несколько хорошо JVM справляется с конкуренцией, а не цель микробенчмарка.

Микробенчмарки должны тестировать набор входных значений

Даже после этого остаются потенциальные ловушки. Код выполняет только одну операцию: вычисление 50-го числа Фибоначчи. Умный компилятор может

понять это и выполнить цикл только один раз — или по крайней мере отбросить часть итераций цикла, поскольку эти операции избыточны.

Кроме того, производительность `fibImpl1(1000)`, скорее всего, будет сильно отличаться от производительности `fibImpl1(1)`; если цель заключается в сравнении производительности разных реализаций, необходимо учесть диапазон входных значений.

Набор входных значений может быть случайным:

```
for (int i = 0; i < nLoops; i++) {
    l = fibImpl1(random.nextInteger());
}
```

Скорее всего, это не то, что вам нужно. Время вычисления случайных чисел включается во время выполнения цикла; получается, что тест теперь измеряет время вычисления последовательности Фибоначчи `nLoops` раз, плюс время генерирования `nLoops` случайных целых чисел.

Лучше вычислить входные значения заранее:

```
int[] input = new int[nLoops];
for (int i = 0; i < nLoops; i++) {
    input[i] = random.nextInt();
}
long then = System.currentTimeMillis();
for (int i = 0; i < nLoops; i++) {
    try {
        l = fibImpl1(input[i]);
    } catch (IllegalArgumentException iae) {
    }
}
long now = System.currentTimeMillis();
```

Микробенчмарки должны проводить измерения для правильных входных данных

Вероятно, вы заметили, что тесту теперь приходится проверять исключение при вызове метода `fibImpl1()`: во входной набор включаются отрицательные числа (для которых числа Фибоначчи не определены) и числа, превышающие 1476 (они дают результат, который не может быть представлен в формате `double`).

Если этот код используется в реальном приложении, появятся ли эти значения во входном наборе? В данном примере — скорее всего, нет; в ваших собственных тестах ситуация может быть иной. Но представьте, что вы тестируете две реализации этой операции. Первая может довольно быстро вычислить число Фибоначчи, но не проверяет свое входное значение. Вторая выдает исключе-

ние, если входной параметр не входит в допустимый набор, но затем выполняет медленную рекурсивную операцию для вычисления числа Фибоначчи:

```
public double fibImplSlow(int n) {
    if (n < 0) throw new IllegalArgumentException("Must be > 0");
    if (n > 1476) throw new ArithmeticException("Must be < 1476");
    return recursiveFib(n);
}
```

Сравнение этой реализации с исходной реализацией по широкому набору входных значений показывает, что эта новая реализация работает намного быстрее исходной — просто из-за проверок диапазона в начале метода.

Если в реальной ситуации пользователи всегда будут передавать методу значения, меньшие 100, это сравнение даст неправильный ответ. В типичном случае метод `fibImpl1()` работает быстрее, и, как объяснялось в главе 1, оптимизацию следует проводить для типичного случая. (Конечно, это искусственный пример, и простое включение проверки границ в исходную реализацию все равно улучшит ее. В общем случае это может быть невозможно.)

Поведение кода микробенчмарка может измениться в реальном приложении

Проблемы, которые рассматривались до настоящего момента, могут быть решены тщательным написанием микробенчмарков. На конечный результат кода после его включения в большую программу также будут влиять другие факторы. Компиляторы используют данные профилирования кода для определения оптимизаций, которые должны быть применены при компиляции метода. Данные профилирования включают информацию о том, какие методы вызываются чаще остальных, на какой глубине стека они вызываются, фактические типы (включая подклассы) их аргументов и т. д., — это зависит от среды, в которой фактически выполняется код.

Следовательно, в микробенчмарках компилятор часто оптимизирует код совсем не так, как он оптимизирует тот же код в большем приложении.

Микробенчмарки также могут проявлять совершенно иное поведение в контексте уборки мусора. Возьмем две реализации микробенчмарка: первая быстро выдает результаты, но также создает много объектов с коротким сроком жизни. Вторая работает медленнее, но создает меньше краткосрочных объектов.

Если запустить маленькую программу для тестирования этих реализаций, скорее всего, первая будет работать быстрее. Даже при том, что при ее выполнении уборка мусора будет активизирована чаще, она сможет быстро уничтожать кратковременные объекты в молодом поколении, поэтому по общему времени выполнения

предпочтительной окажется именно эта реализация. Если запустить этот код на сервере с несколькими потоками, работающими одновременно, профиль GC будет выглядеть иначе: несколько потоков будут быстрее заполнять молодое поколение. Следовательно, многие краткосрочные объекты, которые быстро уничтожались в случае микробенчмарка, могут быть повышены до старого поколения в среде многопоточного сервера. В свою очередь, это приведет к более частой (и более затратной) уборке мусора. И в этом случае из-за большего времени, затрачиваемого на полную уборку мусора, первая реализация станет менее производительной, чем вторая, «медленная» реализация, порождающая меньше мусора.

Наконец, есть проблема того, что же следует понимать под термином «микробенчмарк». Общая разность по времени в таком тесте, как описанный выше, может составить секунды для множества циклов, но для одной итерации разность часто измеряется в наносекундах. Да, наносекунды накапливаются, и «мерть от тысячи порезов часто создает проблемы с производительностью. Однако следует задуматься над тем, насколько оправданно отслеживание результатов на наносекундном уровне — особенно в области регрессионного тестирования. Экономия нескольких наносекунд при каждом обращении к коллекции (которых может быть несколько миллионов — см. главу 12) может играть важную роль. Для операций, которые выполняются реже, — например, один раз на запрос для REST-вызова — исправление наносекундной регрессии, обнаруженной микробенчмарком, отнимет время, которое можно было бы с большей эффективностью потратить на оптимизацию других операций.

Тем не менее при всех своих скрытых рисках микробенчмарки настолько популярны, что в OpenJDK появился фреймворк для разработки микробенчмарков: Java Microbenchmark Harness (jmh). jmh используется разработчиками JDK для построения регрессионных тестов для самого JDK, а также предоставляет основу для разработки общих хронометражных тестов. jmh будет более подробно рассматриваться в следующем разделе.

КАК НАСЧЕТ ПЕРИОДА РАЗОГРЕВА?

Одна из характеристик производительности Java проявляется в том, что при большем количестве выполнений код начинает работать более эффективно — эта тема рассматривается в главе 4. По этой причине микробенчмарки должны включать период разогрева, который дает компилятору возможность сгенерировать оптимальный код.

Преимущества периода разогрева подробно обсуждаются позднее в этой главе. Для микробенчмарков необходим период разогрева; в остальных случаях микробенчмарк измеряет производительность компиляции вместо производительности кода, которую он пытается измерить.

Макробенчмарки

Производительность приложения лучше всего измерять по самому приложению вместе со всеми внешними ресурсами, которые оно использует. Такой вариант тестирования называется *макробенчмарком*. Если приложение обычно проверяет идентификационные данные пользователя, обращаясь с вызовом к службе каталогов (например, через протокол LDAP), оно должно быть протестировано в этом режиме. Подстановка заглушек для вызовов LDAP может быть оправданна для тестирования модульного уровня, но приложение должно тестироваться в своей полной конфигурации.

С ростом приложений соблюдать этот принцип становится все важнее — и все сложнее. Сложная система представляет собой нечто большее, чем простую сумму своих частей; после сборки эти части начинают вести себя иначе. Например, макетирование обращений к базе данных может означать, что вам уже не придется беспокоиться о производительности баз данных — в конце концов, вы специалист по Java; с чего бы вам беспокоиться о проблемах производительности, относящихся к сфере квалификации администратора баз данных? Однако подключения к базам данных потребляют большой объем пространства кучи для хранения своих буферов; передача больших объемов данных повышает нагрузку на сеть; код, вызывающий более простой набор методов (вместо сложного кода в драйвере JDBC), оптимизируется иначе; короткие ветви кода кэшируются и подвергаются конвейерной обработке с большей эффективностью, чем длинные; и т. д.

Другая причина для тестирования полного приложения связана с выделением ресурсов. В идеальном мире разработчику хватало бы времени для оптимизации каждой строки кода в приложении. В реальном мире существуют предельные сроки, а оптимизация только одной части сложной среды может не принести немедленной пользы.

Возьмем поток данных, показанный на рис. 2.1. Данные поступают от пользователя, используются для проведения специальных бизнес-вычислений, соответствующая информация загружается из базы данных, с ними проводятся новые специальные вычисления, измененные данные сохраняются в базе данных, и ответ возвращается пользователю. Число в каждом прямоугольнике — количество запросов в секунду (RPS, Requests Per Second), которые могут обрабатываться модулем при тестировании в изоляции.

С точки зрения бизнеса специальные вычисления — самая важная составляющая; именно ради них существует программа, и из-за них нам платят деньги. Тем не менее ускорение их на 100% не принесет абсолютно никакой пользы в этом примере. Любое приложение (включая отдельную автономную JVM) может быть смоделировано в виде серии таких шагов, в которых данные выходят из блока (модуля, подсистемы и т. д.) со скоростью, определяемой эффективностью этого

блока. Данные поступают в подсистему со скоростью, определяемой выходной скоростью предыдущего блока.

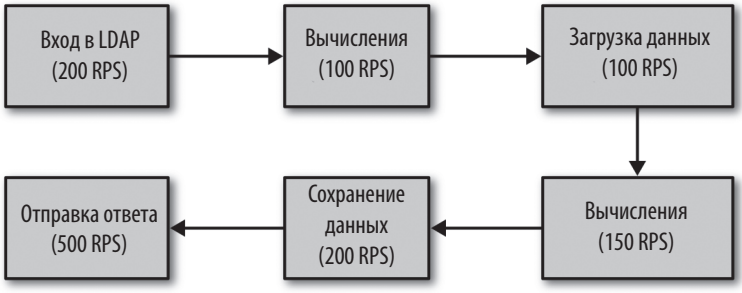


Рис. 2.1. Типичный поток выполнения программы

ПОЛНОСИСТЕМНОЕ ТЕСТИРОВАНИЕ С НЕСКОЛЬКИМИ JVM

Один особенно важный случай тестирования всего приложения встречается при одновременном запуске нескольких приложений на одном оборудовании. Многие аспекты JVM по умолчанию настраиваются в предположении, что JVM доступны все ресурсы машины и при тестировании в изоляции такие JVM будут работать нормально. Но если они будут тестироваться при наличии других приложений (включая, среди прочего, другие JVM), их производительность будет совершенно иной.

Примеры такого рода приводятся в последующих главах, но я кратко упомяну один из них: при выполнении цикла GC одна JVM (в своей конфигурации по умолчанию) доводит загрузку всех процессоров на машине до 100%. Если измерять нагрузку как среднее значение за время выполнения программы, средняя нагрузка может составить 40% — но в действительности это означает, что какое-то время процессор занят на 30%, а в другое время на 100%. Когда JVM работает в изоляции, это может быть нормально, но при выполнении JVM параллельно с другими приложениями она не сможет получить 100% ресурсов процессора в ходе уборки мусора. Ее производительность будет ощутимо отличаться от изолированного выполнения.

Это еще одна причина, по которой микробенчмарки и хронометражные тесты уровня модулей не всегда предоставляют полную картину производительности приложения.

Допустим, в бизнес-вычисления вносится алгоритмическое улучшение, вследствие которого они смогут обрабатывать 200 RPS; соответственно повышается нагрузка, поступающая в систему. Система LDAP может справиться с увеличившейся нагрузкой; пока все хорошо, 200 RPS входят в модуль вычислений, который выдает также 200 RPS.

Однако загрузка данных по-прежнему может обработать только 100 RPS. И хотя на вход базы данных поступают 200 RPS, из нее в другие модули выходят только 100 RPS. Общая пропускная способность системы остается равной 100 RPS, несмотря на удвоение эффективности бизнес-логики. Дальнейшие попытки улучшения бизнес-логики будут неэффективными, пока разработчики не потратят время на улучшение других аспектов среды.

Время, потраченное на оптимизацию вычислений в этом примере, не будет полностью потеряно; после того как разработчики потрудятся над узкими местами в других компонентах системы, выигрыш в производительности наконец-то станет очевидным. Скорее, речь идет о приоритетах; без тестирования всего приложения невозможно сказать, в какой момент работа по оптимизации производительности наконец-то окупится.

Мезобенчмарки

Мезобенчмарки (mesobenchmark) — тесты, занимающие промежуточное положение между микробенчмарками и полным приложением. Я работаю с разработчиками над производительностью как Java SE, так и больших приложений Java, и в каждой группе есть набор тестов, который группа рассматривает как микробенчмарки. У специалистов Java SE этот термин ассоциируется с примером еще меньшим, чем в первом разделе: измерением некоторой очень малой величины. Разработчики приложений обычно используют этот термин для обозначения чего-то другого: тестов, которые измеряют один аспект производительности, но при этом выполняют большой объем кода.

Примером микробенчмарка приложения может послужить тест, измеряющий скорость получения ответа от сервера для простого REST-вызова. Код для такого запроса получается достаточно большим по сравнению с традиционным микробенчмарком: достаточно значительный код управления сокетами, код чтения запроса, код записи ответа и т. д. С традиционной точки зрения это не является микробенчмарком. Такие тесты также не являются макробенчмарками: здесь отсутствуют средства безопасности (например, пользователь не вводит свои регистрационные данные в приложении), отсутствует управление сеансом и не используются многие другие функции приложения. Так как используется всего лишь подмножество реального приложения, этот случай находится где-то в середине — то есть является *мезобенчмарком*. Я буду использовать этот термин

для обозначения тестов, которые выполняют некоторую реальную работу, но не являются полноценными приложениями.

Мезобенчмарки скрывают меньше «подводных камней», чем микробенчмарки, и с ними проще работать, чем с макробенчмарками. Скорее всего, мезобенчмарки не будут содержать большого объема мертвого кода, который может быть исключен компилятором в ходе оптимизации (если только мертвый код не существует в приложении — в этом случае его исключение можно только приветствовать). Мезобенчмарки создают меньше проблем с многопоточным выполнением: с другой стороны, они скорее столкнутся с узкими местами синхронизации, чем при выполнении в полном приложении. Впрочем, эти узкие места все равно рано или поздно встретятся в реальном приложении в большой аппаратной системе при высокой нагрузке.

И все же мезобенчмарки не идеальны. Разработчик, использующий такие тесты для сравнения производительности двух серверов приложений, легко может быть введен в заблуждение. Возьмем гипотетические времена отклика двух REST-серверов, приведенные в табл. 2.1.

Таблица 2.1. Гипотетические времена отклика для двух REST-серверов

Тест	Сервер 1	Сервер 2
Простой вызов REST	19 ± 2,1 мс	50 ± 2,3 мс
Вызов REST с авторизацией	75 ± 3,4 мс	50 ± 3,1 мс

Разработчик, использующий только простой вызов REST для сравнения производительности двух серверов, может не сознавать, что сервер 2 автоматически выполняет авторизацию для каждого запроса. Он может заключить, что сервер 1 обеспечит наибольшую производительность. Но если приложению необходима авторизация (что типично), разработчик примет неправильное решение, потому что серверу 1 требуется намного больше времени для проведения авторизации.

Даже при этом мезобенчмарки предоставляют разумную альтернативу для тестирования полномасштабного приложения; их характеристики производительности намного ближе к реальному приложению, чем характеристики производительности микробенчмарков. И конечно, здесь есть континуум. В одном из последующих разделов этой главы описано типичное приложение, которое будет использоваться во многих примерах последующих глав. Это приложение работает в серверном режиме (как для REST, так и для серверов Jakarta Enterprise Edition), но эти режимы не используют такие серверные функции, как аутентификация, и хотя оно может обращаться к ресурсам корпоративного

уровня (например, к базам данных), в большинстве примеров обращения к базам данных просто заменяются произвольными данными. В пакетном режиме имитируются реальные (хотя и быстрые) вычисления: например, не поддерживается взаимодействие с пользователем или GUI.

Мезобенчмарки также хорошо подходят для автоматизированного тестирования, особенно на уровне модулей.

РЕЗЮМЕ

- Хорошие микробенчмарки трудно написать без соответствующего фреймворка.
- Только тестирование всего приложения позволит узнать, как будет работать код в реальных условиях.
- Изоляция производительности на уровне модулей или операций с применением мезобенчмарков является разумным методом, но не заменяет тестирования всего приложения.

Пропускная способность, пакетирование и время отклика

Второй принцип требует понимания и выбора подходящей тестовой метрики для приложения. Производительность может измеряться в форме пропускной способности (RPS), затраченного времени (пакетные измерения) или времени отклика, причем эти три метрики взаимосвязаны. Понимание этих отношений позволит вам сосредоточиться на правильной метрике в зависимости от цели приложения.

Измерения затраченного времени

Простейший способ измерения производительности заключается в определении продолжительности выполнения некоторой задачи. Допустим, вы хотите прочитать историю изменения 10 000 видов акций за 25-летний период и вычислить стандартное отклонение цен, построить отчет по заработной плате для 50 000 работников корпорации или выполнить цикл 1 000 000 раз.

В языках со статической компиляцией такое тестирование происходит достаточно прямолинейно: разработчик пишет приложение и измеряет время его выполнения. В мире Java к картине добавляется один нюанс: JIT-компиляция. Этот процесс описан в главе 4; по сути это означает, что для полной оптимизации

кода и его выполнения с максимальной производительностью может потребоваться от нескольких секунд до нескольких минут (и более). По этой (и другим) причинам в анализе производительности Java учитываются так называемые *периоды разогрева*: производительность чаще всего измеряется только после того, как тестируемый код будет выполнен в течение времени, достаточного для его компиляции и оптимизации.

ДРУГИЕ ФАКТОРЫ РАЗОГРЕВА ПРИЛОЖЕНИЙ

Разогрев приложения чаще всего обсуждается в контексте ожидания оптимизации кода компилятором. Тем не менее другие факторы также могут влиять на производительность кода в зависимости от длительности его выполнения.

Например, Java (или Jakarta) Persistence API (JPA) обычно кэширует данные, прочитанные из базы данных (см. главу 11); при втором использовании этих данных операция будет выполнена быстрее, потому что данные можно прочитать из кэша без затратного обращения к базе данных. Аналогичным образом, когда приложение читает файл, операционная система обычно загружает этот файл в память. Тест, который после этого читает тот же файл (например, в цикле измерения производительности), во второй раз будет выполнен быстрее, потому что данные уже находятся в оперативной памяти компьютера и читать их с диска не придется.

В общем случае есть много мест (причем далеко не всегда очевидных), в которых данные могут кэшироваться, и период разогрева может сыграть важную роль.

С другой стороны, во многих случаях важна производительность приложения от начала до конца. Генератор отчетов, который обрабатывает десятки тысяч элементов данных, завершается за определенный промежуток времени; конечного пользователя не интересует, что первые пять тысяч элементов были обработаны на 50% медленнее последних пяти тысяч элементов. И даже с такими системами, как REST-сервер, — у которых производительность сервера наверняка улучшится со временем, — исходная производительность важна. Серверу потребуется некоторое время для достижения пиковой производительности; для пользователей, обращающихся к приложению за это время, производительность во время периода разогрева несущественна.

По этим причинам многие примеры в книге являются пакетно-ориентированными (даже если это несколько нетипично).

Измерения пропускной способности

Измерения пропускной способности основаны на объеме работы, которая может быть выполнена за определенный период времени. Хотя в самых распространенных примерах измерений пропускной способности задействован сервер, обрабатывающий данные, поставляемые клиентом, это не является строго необходимым: одно автономное приложение может измерять пропускную способность так же легко, как оно измеряет затраченное время.

В клиентско-серверном тесте измерения пропускной способности означают, что у клиента отсутствует время обработки. Если клиент только один, он отправляет запрос серверу. Получив ответ, клиент немедленно отправляет новый запрос. Этот процесс продолжается; в конце теста клиент сообщает общее количество выполненных операций. Как правило, на стороне клиента несколько потоков делают одно и то же, а пропускная способность является сводной метрикой количества операций, выполненных всеми клиентами. Обычно измеряется количество операций в секунду, а не как общее количество операций за период измерений. Данная метрика часто обозначается сокращениями TPS (Transactions Per Second), RPS (Requests Per Second) и OPS (Operations Per Second).

Конфигурация клиента играет важную роль в тестах «клиент/сервер»; необходимо позаботиться о том, чтобы клиент достаточно быстро поставлял данные серверу. Скорость поставки может быть нарушена из-за того, что на машине клиента не хватает процессорного времени для выполнения нужного количества клиентских потоков, или клиенту приходится тратить много времени на обработку запроса, прежде чем он сможет отправить новый запрос. В таких случаях тест фактически измеряет производительность клиента, а не производительность сервера — обычно это не то, что вам нужно.

Риск возникновения подобных ситуаций зависит от объема работы, выполняемой каждым клиентом (а также размера и конфигурации клиентской машины). Тесты с нулевым временем обработки (ориентированные на определение пропускной способности) скорее столкнутся с такой ситуацией, поскольку каждый клиентский поток выполняет больше запросов. Следовательно, тесты на пропускную способность обычно выполняются с меньшим количеством клиентских потоков (меньшей нагрузкой), чем соответствующие тесты для измерения времени отклика.

Тесты, измеряющие пропускную способность, также часто вычисляют среднее время отклика запросов. Эти данные представляют интерес, но изменения числа не указывают на проблему с производительностью, если только пропускная способность не остается неизменной. Сервер, способный обеспечить 500 OPS при времени отклика 0,5 секунды, работает более эффективно, чем сервер, выдающий время отклика 0,3 секунды при 400 OPS.

Измерения пропускной способности почти всегда выполняются после должного периода разогрева именно потому, что измеряемый объем работы не является фиксированным.

Тесты на определение времени отклика

Последняя распространенная разновидность тестов — тесты на измерение *времени отклика*: времени, проходящего между отправкой запроса клиентом и получением ответа. Различие между тестом на определение времени отклика и тестом на пропускную способность (предполагается, что последний основан на модели «клиент/сервер») состоит в том, что потоки клиента в тесте на определение времени отклика приостанавливаются между операциями. Это время называется *временем обработки* (think time). Тест на определение времени отклика проектируется для более точной имитации действий пользователя: пользователь вводит URL-адрес в браузере, проводит некоторое время за чтением полученной страницы, щелкает на ссылке на этой странице, проводит время за чтением новой страницы и т. д.

Когда в тест вводится время обработки, пропускная способность фиксируется: заданное количество клиентов, выполняющих запросы с заданным временем обработки, всегда будет давать одно значение TPS (с небольшими отклонениями; см. врезку). В этот момент важной метрикой становится время отклика для запроса: эффективность сервера определяется тем, с какой скоростью он отреагирует на эту фиксированную нагрузку.

Время отклика можно измерять двумя способами. Оно может вычисляться как средняя величина: отдельные значения времени суммируются, а сумма делится на число запросов. Время отклика также может вычисляться в форме процентилей — например, 90%-ное время отклика. Если 90% ответов приходят менее чем за 1,5 секунды, а 10% ответов — более чем за 1,5 секунды, то 1,5 секунды является 90%-ным временем отклика.

Одно отличие между средним временем отклика и временем отклика в процентилях проявляется во влиянии статистических выбросов на вычисление среднего: так как они включаются как часть среднего, большие выбросы окажут значительное влияние на среднее время отклика.

На рис. 2.2 изображен граф из 20 запросов с относительно типичным диапазоном времен отклика. Время отклика лежит в диапазоне от 1 до 5 секунд. Среднее время отклика (представленное нижней толстой линией, параллельной оси x) составляет 2,35 секунды, а 90% ответов приходят за 4 секунды и менее (верхняя толстая линия, параллельная оси x).

Этот набор данных включает серьезный выброс: один запрос занял 100 секунд. В результате позиции 90% и среднего времени отклика поменялись. Среднее

ВРЕМЯ ОБРАБОТКИ И ПРОПУСКНАЯ СПОСОБНОСТЬ

Пропускная способность теста, в котором клиенты используют время обработки, может измеряться двумя способами. Самый простой способ — приостановка клиентов на некоторый период времени между запросами:

```
while (!done) {
    executeOperation();
    Thread.currentThread().sleep(30*1000);
}
```

В этом случае пропускная способность в некоторой степени зависит от времени отклика. Если время отклика составляет 1 секунду, это означает, что клиент будет отправлять запрос каждую 31 секунду, что дает пропускную способность 0,032 OPS. Если время отклика составляет 2 секунды, каждый клиент отправляет запрос каждые 32 секунды, а пропускная способность составит 0,031 OPS.

Другая альтернатива называется временем цикла (в отличие от времени обработки). Время цикла устанавливает общее время между запросами равным 30 секундам, так что время приостановки клиента зависит от времени отклика:

```
while (!done) {
    time = executeOperation();
    Thread.currentThread().sleep(30*1000 - time);
}
```

Эта альтернатива обеспечивает фиксированную пропускную способность 0,033 OPS на клиента независимо от времени отклика (предполагается, что время отклика в данном примере всегда меньше 30 секунд).

Средства тестирования нередко сознательно изменяют время обработки; они вычисляют конкретное среднее значение, но используют случайную поправку, чтобы лучше моделировать действия пользователя. Кроме того, планирование потоков никогда не осуществляется полностью в реальном времени, так что фактическое время между запросами, отправляемыми клиентом, будет слегка изменяться. В результате даже при использовании инструмента, обеспечивающего заданное время цикла, вместо времени обработки, вычисленная пропускная способность между запусками будет слегка различаться. Но если пропускная способность далека от ожидаемой, значит, что-то пошло не так в ходе выполнения теста.

время отклика составляет немыслимые 5,95 секунды, тогда как 90%-ное время отклика составляет 1,0 секунды. В этом случае следует сосредоточиться на сокращении влияния выброса (которое ухудшает среднее время отклика).

Рис. 2.2. Типичный набор значений времени отклика

Изображенная ситуация типична для теста с нормальным поведением. Выбросы могут исказить этот анализ, как показывают данные на рис. 2.3.

Рис. 2.3. Набор значений времени отклика с выбросом

ГЕНЕРАТОРЫ НАГРУЗКИ

Существует много средств генерирования нагрузки — как коммерческих, так и с открытым кодом. В примерах этой главы используется Faban — генератор нагрузки с открытым кодом на базе Java. Этот код лежит в основе средств управления нагрузкой таких коммерческих хронометражных тестов, как SPECjEnterprise от SPEC.

В поставку Faban входит простая программа (`fhb`), которая может использоваться для измерения производительности обращений к простому URL-адресу:

```
% fhb -W 1000 -r 300/300/60 -c 25  http://host:port/StockServlet?stock=SDO
ops/sec: 8.247
% errors: 0.0
avg. time: 0.022
max time: 0.045
90th %: 0.030
95th %: 0.035
99th %: 0.035
```

В этом примере измерения проводятся с 25 клиентами (`-c 25`), выдающими запросы к сервлету (для биржевого сокращения SDO); каждый запрос имеет 1-секундное время цикла (`-W 1000`). Тест имеет 5-минутный (300-секундный) период разогрева, за которым следует 5-минутный период измерений и 1-минутный период снижения (`-r 300/300/60`). После теста fhb сообщает значение OPS и различные времена отклика для теста (а поскольку этот пример включает время обработки, время отклика является важной метрикой, тогда как значения OPS будут более или менее постоянными).

fhb хорошо подходит для подобных простых тестов (укладывающихся в одну командную строку). Также существуют более сложные средства генерирования нагрузки: Apache JMeter, Gatling, MicroFocus LoadRunner и многие другие.

Подобные выбросы могут происходить по разным причинам, и они с большей вероятностью встречаются в приложениях Java из-за задержек, обусловленных GC[1]. В тестировании производительности основное внимание обычно уделяется 90%-ному времени отклика (и даже 95%-ному или 99%-ному времени отклика;

[1] Вряд ли уборка мусора может создать 100-секундную задержку, но паузы из-за уборки мусора, особенно для тестов с малым средним временем отклика, могут привести к появлению значительных выбросов.

в пороге 90% нет ничего особенного). Если вы можете сосредоточиться только на одном числе, процентиль лучше подойдет, так как ускорение отклика принесет пользу большей группе пользователей. И все же лучше рассматривать как среднее время отклика, так и по крайней мере одно время отклика с процентилем, чтобы не упустить случаи с большими выбросами.

РЕЗЮМЕ

- Пакетно-ориентированные тесты (или любые тесты без периода разогрева) относительно редко используются при тестировании производительности Java, но могут дать ценные результаты.
- Другие тесты могут измерять пропускную способность или время отклика в зависимости от того, поступает ли нагрузка в фиксированном темпе (то есть на основании эмулированного времени обработки в клиенте).

Дисперсия

Третий принцип — понимание того, как результаты тестов изменяются со временем. Программы, которые обрабатывают в точности одинаковые наборы данных, могут выдавать разные результаты при каждом их запуске. Фоновые процессы на машине влияют на приложение, сеть может быть в большей или меньшей степени загружена при запуске программы, и т. д. Хорошие тесты также никогда не обрабатывают в точности одинаковые наборы данных при каждом запуске; в тест будет встроено случайное поведение, моделирующее внешний мир. И этот факт создает проблему: чем объясняются различия при сравнении результатов двух запусков — регрессией или случайными изменениями в тесте?

Проблема может быть решена многократным запуском теста с усреднением полученных результатов. Затем при внесении изменений в тестируемый код тест снова запускается несколько раз, результаты усредняются и два средних значения сравниваются. На первый взгляд все просто.

К сожалению, в реальности все сложнее. Понять, когда различие обусловлено реальной регрессией, а когда — случайным изменением, достаточно сложно. В этой ключевой области наука прокладывает путь, но искусство тоже играет свою роль.

При сравнении средних результатов тестов невозможно с абсолютной уверенностью определить, реальны ли различия средних значений или они обусловлены случайными флюктуациями. Лучшее, что можно сделать — выдвинуть гипотезу «средние значения равны», а затем определить вероятность того, что

это утверждение истинно. Если утверждение ложно с высокой вероятностью, можно с высокой степенью достоверности утверждать, что средние значения различаются (хотя 100%-ной уверенности не будет никогда).

Тестирование кода при подобных изменениях называется *регрессионным тестированием*. В регрессионных тестах исходный код называется *эталоном*, а новый код — *образцом*. Возьмем пакетную программу, в которой эталон и образец запускаются по три раза. Полученные результаты объединены в табл. 2.2.

Таблица 2.2. Гипотетическое время выполнения двух тестов

	Эталон	**Образец**
Первая итерация	1,0 секунды	0,5 секунды
Вторая итерация	0,8 секунды	1,25 секунды
Третья итерация	1,2 секунды	0,5 секунды
Среднее значение	1 секунда	0,75 секунды

Средние значения для образца указывают на 25%-ное улучшение кода. Насколько можно быть уверенным в том, что тест действительно отражает 25%-ное улучшение? Все выглядит хорошо: два из трех значений образца меньше среднего значения эталона, а степень улучшения велика. Тем не менее после проведения с этими результатами анализа, описанного в этом разделе, выясняется, что вероятность одинаковой производительности у образца и базовой линии составляет 43%. Иначе говоря, при таких показателях в 43% случаев производительность двух тестов совпадает, а различается только в 57% случаев. Кстати, это утверждение не совсем эквивалентно тому, что в 57% случаев производительность будет на 25% лучше, но об этом будет рассказано позднее в этом разделе.

Причина, по которой эти вероятности могут отличаться от наших ожиданий, заключается в большой дисперсии результатов. В общем случае чем больше дисперсия в наборе результатов, тем труднее определить вероятность того, реальны ли различия в средних значениях или они обусловлены случайными отклонениями[1].

Это число (43%) основано на результатах *t-критерия Стьюдента* — метода статистического анализа, основанного на выборках и их дисперсии[2]. В результате применения t-критерия определяется число, называемое *p-значением*; оно описывает вероятность истинности нулевой гипотезы для этого теста.

[1] С тремя точками данных этот пример проще понять, однако трех точек слишком мало, чтобы обеспечить достоверные результаты в любой реальной системе.

[2] Стьюдент — псевдоним ученого, впервые опубликовавшего описание метода.

В регрессионном тестировании *нулевой гипотезой* считается гипотеза о том, что два теста обладают одинаковой производительностью. p-значение для данного примера составляет приблизительно 43%; это означает, что наша уверенность в том, что обе серии результатов сходятся к одному среднему, равна 43%. И наоборот, наша уверенность в том, что серии не сходятся к общему среднему, составляет 57%.

Как же понимать утверждение о том, что в 57% случаев серии не сходятся к одному среднему? Строго говоря, это не означает, что мы можем на 57% быть уверенными в 25%-ном улучшении результата, — а лишь то, что мы можем на 57% быть уверенными в том, что результаты различаются. Улучшение может составлять 25, но может достигнуть и 125%; нельзя даже исключать, что образец обладает худшей производительностью, чем эталон. Наиболее вероятный вариант — что различия в тесте близки к полученным данным (особенно с понижением p-значения), но стопроцентной уверенности достичь никогда не удастся.

t-критерий обычно используется в сочетании с α-значением, которое представляет собой (относительно произвольно выбранную) точку, в которой результат считается статистически значимым. α-значение обычно задается равным 0,1 — это означает, что результат считается статистически значимым, если образец и эталон совпадают только в 10% (0,1) случаев (или наоборот — что в 90% случаев образец и эталон различаются). Другие часто встречающиеся α-значения — 0,05 (95%) и 0,01 (99%). Критерий считается статистически значимым, если p-значение больше, чем 1 – α-значение.

СТАТИСТИКА И СЕМАНТИКА

Правильное представление результатов применения t-критерия выглядит следующим образом: есть 57%-ная вероятность того, что образец отличается от эталона, а лучшая оценка разности между ними составляет 25%.

На практике часто говорят, что с 57%-ным уровнем достоверности есть 25%-ное улучшение результата. Хотя это не совсем то же самое, а специалиста по статистике такие утверждения выведут из себя, это удобная короткая формулировка, которая не так уж далека от истины. Вероятностный анализ всегда подразумевает некоторую неопределенность, и эта неопределенность становится более понятной при точном изложении семантики. Но в тех областях, в которых суть возникающих проблем хорошо понятна, неизбежно появляются семантические сокращения.

Итак, правильный способ поиска регрессий в коде заключается в определении уровня статистической значимости (допустим, 90%) и последующем применении t-критерия для определения того, выходят ли различия между образцом и эталоном за пределы статической значимости. При этом вы должны хорошо

понимать, что означает нарушение критерия статистической значимости. В приведенном примере p-значение равно 0,43; нельзя утверждать, что со статической значимостью в пределах 90% уровня достоверности можно утверждать, что результат указывает на то, что средние значения различны. Тот факт, что проверка не является статистически значимой, не означает, что этот результат не обладает никакой значимостью; он означает лишь то, что проверка не позволяет сделать окончательных выводов.

СТАТИСТИЧЕСКАЯ ВАЖНОСТЬ

Статистическая значимость не означает *статической важности*. Эталон с низкой дисперсией и образец с низкой дисперсией, для которых среднее значение составляет 1,0 и 1,01 секунды соответственно, могут иметь p-значение 0,01: это 99%-ная вероятность присутствия различий в результате.

Само различие составляет всего 1%. Теперь допустим, что другая проверка указывает на 10%-ную регрессию между образцом и эталоном, но с p-значением 0,2, которое не является статистически значимым. Какая проверка заслуживает выделения самого драгоценного ресурса — дополнительного времени для проведения анализа? Хотя в случае с 10%-ным различием уверенность ниже, время лучше потратить на анализ именно этой проверки (и начать по возможности с получения дополнительных данных, по которым можно будет определить, действительно ли результат является статистически значимым). Из того факта, что 1%-ная разность более вероятна, еще не следует, что она более важна.

Типичная причина статистической незавершенности выборки — недостаточность данных в образцах. До настоящего момента в наших примерах рассматривались выборки с тремя результатами в эталоне и образце. Что произойдет при добавлении трех дополнительных результатов? Полученные данные представлены в табл. 2.3.

С дополнительными данными p-значение падает с 0,43 до 0,11: вероятность того, что результаты различны, повысилась с 57% до 89%. Средние значения не изменились; мы просто можем с большей уверенностью утверждать, что различия не обусловлены случайными отклонениями.

Выполнение дополнительных тестов до достижения уровня статистической значимости не всегда оправданно с практической точки зрения. Строго говоря, оно и не является необходимым. α-значение, которое определяет статистическую значимость, выбирается произвольно, хотя некоторые значения часто встречаются на практике. p-значение 0,11 не является статистически значимым с 90%-ным уровнем достоверности, но оно становится статистически значимым с 89%-ным уровнем достоверности.

Таблица 2.3. Расширенная выборка данных гипотетического времени

	Эталон	Образец
Первая итерация	1,0 секунды	0,5 секунды
Вторая итерация	0,8 секунды	1,25 секунды
Третья итерация	1,2 секунды	0,5 секунды
Четвертая итерация	1,1 секунды	0,8 секунды
Пятая итерация	0,7 секунды	0,7 секунды
Шестая итерация	1,2 секунды	0,75 секунды
Среднее значение	1 секунда	0,75 секунды

Регрессионное тестирование играет важную роль, но это не является формальным научным методом. Нельзя взглянуть на серию чисел (или их средних значений) и вынести решение, сравнивающее эти числа, без проведения статистического анализа для понимания того, что же означают эти числа. Даже этот анализ не сможет дать полностью определенный ответ из-за теории вероятности. Специалист по производительности должен взглянуть на эти числа, понять вероятности и определить, на что лучше потратить время, на основании всех доступных данных.

РЕЗЮМЕ

- Чтобы правильно определить, различаются ли результаты двух тестов, следует провести статистический анализ, который подтвердит, что различия не обусловлены случайными отклонениями.
- Формальный метод решения этой задачи основан на применении t-критерия Стьюдента для сравнения результатов.
- Данные t-критерия сообщают нам вероятность существования регрессии, но не говорят о том, на какие регрессии можно не обращать внимания, а какими следует заняться. Поиск этого баланса — часть искусства управления производительностью.

Тестируйте рано, тестируйте часто

Четвертый принцип, который так любят рекомендовать специалисты по производительности (включая меня), гласит, что тестирование производительности должно стать неотъемлемой частью цикла разработки. В идеальном мире тесты производительности должны выполняться как часть процесса при сохранении

кода в центральном репозитории; если код создает регрессии по производительности, попытка его сохранения будет заблокирована.

Между этой рекомендацией и другими рекомендациями, приведенными в этой главе, существуют некоторые внутренние трения — как и между этой рекомендацией и реальным миром. Хороший тест производительности охватывает большой объем кода — по крайней мере мезобенчмарк среднего размера. Его необходимо повторить несколько раз, чтобы укрепить уверенность в том, что любые различия между старым и новым кодом реальны, а не просто являются следствием случайных отклонений. В большом проекте это может занять несколько дней или неделю, из-за чего проведение тестов производительности кода перед его сохранением в репозитории становится нереалистичным. Типичный цикл разработки только усложняет ситуацию. В графике проекта часто устанавливается дата прекращения расширения функциональности: все функциональные изменения в коде должны быть сохранены в репозитории в более ранней точке цикла разработки, а оставшаяся часть цикла будет посвящена устранению ошибок (включая проблемы производительности) в новом выпуске. Этот факт создает две проблемы при раннем тестировании:

- Время, в течение которого разработчик должен сохранить код в репозитории для соблюдения графика, ограничено. Разработчику не захочется тратить время на устранение проблемы с производительностью, когда в графике предусмотрено время для этой задачи после сохранения всего исходного кода в репозитории. Разработчик, который в ранней точке цикла занес в репозиторий код, вызывающий 1%-ную регрессию, столкнется с нехваткой времени для решения этой проблемы. Разработчик, дождавшийся вечера, в который дальнейшее расширение функциональности будет заблокировано, может сохранить код, вызывающий 20%-ную регрессию, и разобраться с ним позднее.
- Производительные характеристики кода изменяются с течением времени. Здесь проявляется тот же принцип, который приводился в качестве аргумента для тестирования всего приложения (в дополнение к любым тестам уровня модулей): изменяется использование кучи, изменяется компиляция кода и т. д.

Несмотря на все сказанное, частое тестирование производительности в процессе разработки играет важную роль — даже в том случае, если эти проблемы не могут быть решены немедленно. У разработчика, создавшего код с 5%-ной регрессией, может существовать план устранения этой регрессии в процессе разработки; возможно, код зависит от еще не интегрированной функциональности, а когда эта функциональность появится — небольшое исправление заставит регрессию исчезнуть. Это разумная позиция, хотя она и означает, что тестам производительности придется мириться с 5%-ной регрессией в течение нескольких недель (и неприятным, но неизбежным фактом того, что указанная регрессия скрывает другие регрессии).

С другой стороны, если новый код порождает регрессию, которая может быть устранена только архитектурными изменениями, лучше выявить эту регрессию и разобраться с ней на ранней стадии — до того, как остальной код начнет зависеть от новой реализации. По сути это поиск баланса, требующий аналитических, а часто и политических навыков.

Как правило, раннее тестирование приносит наибольшую пользу при соблюдении следующих рекомендаций:

Автоматизируйте все, что можно

Все тестирование производительности должно быть реализовано в виде сценария (или запрограммировано, хотя сценарный вариант обычно проще). Сценарии должны уметь устанавливать новый код, настраивать его для формирования полноценной рабочей среды (создание подключений к базам данных, настройка учетных записей пользователей и т. д.) и выполнять набор тестов. Но на этом их работа не останавливается: сценарии должны уметь выполнять тесты многократно, анализировать результаты по t-критерию, выдавать отчет с полученным уровнем достоверности равенства результатов и измеренными различиями, если результаты не совпадают.

Автоматизация должна убедиться в том, что машина находится в известном состоянии перед выполнением тестов: она должна проверить, что в системе не выполняется никаких посторонних процессов, что конфигурация ОС правильна и т. д. Тест производительности воспроизводим только в том случае, если среда остается неизменной между запусками; автоматизация должна позаботиться об этом.

Измеряйте все, что можно

Автоматизация должна собрать все данные, которые могут пригодиться для последующего анализа. В частности, сюда входит системная информация, собранная при запуске: уровень использования процессора, использование диска, использование памяти и т. д. Сюда входят данные журналов приложения — как сгенерированные приложением, так и сгенерированные уборщиком мусора. В идеале информация должна включать записи JFR (Java Flight Recorder) (см. главу 3) или другие профильные данные с малым воздействием на среду выполнения, периодические стеки потоков или другие данные анализа кучи — например, гистограммы или полные дампы кучи (хотя полные дампы кучи занимают много места и не всегда подходят для долгосрочного хранения).

Информация мониторинга также должна по возможности включать данные от других частей системы: например, если программа использует базу данных, она должна включать системную статистику с сервера базы данных, а также весь диагностический вывод базы данных — включая такие

отчеты производительности, как Oracle-отчеты AWR (Automatic Workload Repository).

Эти данные будут управлять анализом любых выявленных регрессий. Если уровень использования процессора возрос, следует проверить профильные данные и определить, что занимает больше времени. Если возросло время, проведенное за уборкой мусора, следует обратиться к профилям кучи и определить, что потребляет больше памяти. Если процессорное время и время уборки мусора сократилось, возможно, производительность сократилась из-за конкуренции за ресурсы: данные стека могут указывать на узкие места синхронизации (см. главу 9), записи JFR помогут выявить задержки в приложении, а журналы базы данных могут указать на фактор, повысивший конкуренцию за ресурсы базы данных.

При определении источника регрессии вы фактически выступаете в роли сыщика. Чем больше данных, тем больше улик для дальнейшего расследования. Как упоминалось в главе 1, источником регрессии не всегда является JVM. Измеряйте все и везде, чтобы иметь возможность провести исчерпывающий анализ.

Выполняйте в целевой системе

Тест, выполняемый на одноядерном ноутбуке, и тест, выполняемый на машине с 72 ядрами, будут вести себя совершенно по-разному. Это должно быть очевидно хотя бы с учетом последствий многопоточности: на более мощной машине будет одновременно выполняться большее количество потоков, что сократит конкуренцию потоков приложений за доступ к процессору. В то же время в более мощной системе проявятся узкие места синхронизации, которые останутся незамеченными на слабом ноутбуке.

Другие различия в производительности не менее важны, хотя они не столь очевидны. Многие важные флаги оптимизации вычисляют свои значения по умолчанию на основании состава оборудования, на котором работает JVM. Код компилируется по-разному в зависимости от платформы. Кэши — программные и, что еще важнее, аппаратные — ведут себя по-разному в разных системах и при разных нагрузках. И так далее...

Следовательно, производительность конкретной рабочей среды может быть полностью определена только после тестирования предполагаемой нагрузки на предполагаемом оборудовании. Приближения и экстраполяции могут делаться на основе выполнения меньших тестов на другом тестовом оборудовании, а в реальном мире дублирование рабочей среды для тестирования может оказаться слишком сложным или затратным. Но экстраполяции — всего лишь прогнозы, а прогноз даже в лучшем случае может оказаться ошибочным. Крупномасштабная система представляет собой нечто большее, чем простую сумму частей, и у адекватного нагрузочного тестирования на целевой платформе попросту нет альтернативы.

РЕЗЮМЕ

- Частое тестирование производительности играет важную роль, но оно не происходит в вакууме; нормальный цикл разработки сопряжен с различными компромиссами, которые вам придется учитывать.
- Автоматизированная система тестирования, собирающая всю возможную статистику от всех машин и программ, предоставит необходимую информацию о любых регрессиях производительности.

Примеры хронометражных тестов

Некоторые примеры в книге используют jmh для создания микробенчмарков. В этом разделе разработка одного такого теста подробно рассматривается как пример написания ваших собственных тестов jmh. Однако многие примеры в книге основаны на разновидностях мезобенчмарка — теста, достаточно сложного для тестирования различных возможностей JVM, но уступающего по сложности реальному приложению. По этой причине после знакомства с jmh будут рассмотрены некоторые примеры кода мезобенчмарка, встречающегося в дальнейших главах, чтобы эти примеры существовали в контексте.

JMH

jmh — набор классов, образующих основу для написания хронометражных тестов. Буква «m» в jmh когда-то обозначала микробенчмарк (Microbenchmark), хотя теперь jmh рекламируется как фреймворк, подходящий для создания нано-/микро-/милли-/макробенчмарков. Впрочем, его типичной областью применения остается создание малых (микро-) тестов. И хотя о выходе jmh было объявлено в сочетании с Java 9, фреймворк в действительности не привязан ни к какой конкретной версии Java, и никакие инструменты в JDK не поддерживают jmh. Библиотеки классов, входящие в jmh, совместимы с JDK 8 и последующими версиями.

jmh устраняет долю неопределенности из написания хорошего хронометражного теста, но это не панацея; вы все еще должны понимать, что именно измеряется и как написать хороший код теста. Но функциональность jmh разрабатывалась именно для того, чтобы упростить эту задачу.

jmh используется в нескольких примерах этой книги, включая тест для параметров JVM, которые влияют на интернирование строк, описанное в главе 12. Мы воспользуемся этим примером для того, чтобы понять, как научиться писать тесты с jmh.

Тесты jmh можно писать с нуля, но проще начать с класса, предоставленного jmh, и написать только код, принадлежащий конкретному тесту. И хотя необходимые классы jmh можно получить при помощи различных средств (и даже некоторых интегрированных сред), обычно для этого используется Maven. Следующая команда создает проект Maven, в который можно добавить код теста:

```
$ mvn archetype:generate \
      -DinteractiveMode=false \
      -DarchetypeGroupId=org.openjdk.jmh \
      -DarchetypeArtifactId=jmh-java-benchmark-archetype \
      -DgroupId=net.sdo \
      -DartifactId=string-intern-benchmark \
      -Dversion=1.0
```

Команда создает проект Maven в каталоге string-intern-benchmark; в нем создается каталог с именем `groupId` и заготовка класса теста с именем `MyBenchmark`. В этом имени нет ничего особенного; вы можете создавать другие классы, поскольку jmh определяет, какие классы должны тестироваться, при помощи аннотации с именем `Benchmark`.

Нас интересует тестирование производительности метода `String.intern()`, поэтому первый метод теста, который мы напишем, выглядит так:

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.infra.Blackhole;

public class MyBenchmark {
    @Benchmark
    public void testIntern(Blackhole bh) {
        for (int i = 0; i < 10000; i++) {
            String s = new String("String to intern " + i);
            String t = s.intern();
            bh.consume(t);
        }
    }
}
```

Базовая структура метода `testIntern()` выглядит разумно: тестируется время создания 10 000 интернированных строк. Класс `Blackhole`, используемый здесь, используется jmh для решения одной из потенциальных проблем микробенчмарков: если значение операции не используется, компилятор может исключить операцию при оптимизации. Поэтому мы обеспечиваем использование значений, передавая их методу `consume()` класса `Blackhole`.

В этом примере класс `Blackhole` не является строго необходимым: нас в действительности интересуют только побочные эффекты вызова метода `intern()`, который вставляет строку в глобальную хеш-таблицу. Это изменение состояния

не будет исключено компилятором, даже если возвращаемое значение самого метода `intern()` не используется. Впрочем, вместо того чтобы подолгу разбираться в том, обязательно или нет потреблять конкретное значение, лучше сразу принимать меры к тому, чтобы операция выполнялась так, как ожидалось, и потребляла вычисленные значения. Чтобы откомпилировать и запустить тест, выполните следующие команды:

```
$ mvn package
... вывод mvn...
$ java -jar target/benchmarks.jar
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: net.sdo.MyBenchmark.testIntern

# Run progress: 0.00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration   1: 189.999 ops/s
# Warmup Iteration   2: 170.331 ops/s
# Warmup Iteration   3: 195.393 ops/s
# Warmup Iteration   4: 184.782 ops/s
# Warmup Iteration   5: 158.762 ops/s
Iteration   1: 182.426 ops/s
Iteration   2: 172.446 ops/s
Iteration   3: 166.488 ops/s
Iteration   4: 182.737 ops/s
Iteration   5: 168.755 ops/s

# Run progress: 20.00% complete, ETA 00:06:44
# Fork: 2 of 5

.... аналогичный вывод ...
Result "net.sdo.MyBenchmark.testIntern":
  177.219 ±(99.9%) 10.140 ops/s [Average]
  (min, avg, max) = (155.286, 177.219, 207.970), stdev = 13.537
  CI (99.9%): [167.078, 187.359] (assumes normal distribution)


Benchmark                Mode  Cnt    Score    Error  Units
MyBenchmark.testIntern  thrpt   25  177.219 ± 10.140  ops/s
```

Из вывода видно, что jmh помогает избежать потенциальных проблем, о которых мы говорили ранее в этой главе. Начнем с выполнения пяти итераций разогрева по 10 секунд каждая, за которыми следуют пять итераций измерений (также по 10 секунд). Итерации разогрева позволяют компилятору полностью оптимизировать код, после чего тестовая оснастка будет выдавать информацию только по итерациям откомпилированного кода.

Затем идут разные ветви (всего пять). Оснастка повторяет тест пять раз, каждый раз в отдельной (ответвленной) JVM для определения воспроизводимости результатов. Каждая JVM должна сначала разогреться, а затем провести измерения с кодом. Ответвленный тест (с интервалами разогрева и измерений) называется *испытанием* (trial). В итоге каждый тест занимает 100 секунд для 5 циклов разогрева и 5 циклов измерений; все это повторяется 5 раз, и общее время выполнения составляет 8 минут 20 секунд.

Наконец, имеется сводный вывод: в среднем метод `testIntern()` выполнялся 177 раз в секунду. С доверительным интервалом 99,9% можно утверждать, что статистическое среднее лежит в пределах от 167 до 187 операций в секунду. Таким образом, jmh также помогает выполнить статистический анализ, необходимый для понимания того, имеет ли полученный результат приемлемую дисперсию.

JMH и параметры

Часто для теста требуется диапазон входных значений; в данном примере нам хотелось бы увидеть эффект от интернирования 1 или 10 000 (и даже 1 миллиона) строк. Вместо того чтобы фиксировать это значение в методе `testIntern()`, мы введем параметр:

```
@Param({"1","10000"})
private int nStrings;

@Benchmark
public void testIntern(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String s = new String("String to intern " + i);
        String t = s.intern();
        bh.consume(t);
    }
}
```

Теперь jmh выводит результаты для обоих значений параметра:

```
$ java -jar target/benchmarks.jar
...длинный вывод...
Benchmark                (nStrings)   Mode  Cnt        Score        Error  Units
MyBenchmark.testMethod            1  thrpt   25  2838957.158 ± 284042.905  ops/s
MyBenchmark.testMethod        10000  thrpt   25      202.829 ±     15.396  ops/s
```

Как и следовало ожидать, с размером цикла 10 000 количество выполнений цикла в секунду сокращается в 10 000 раз. Результат для 10 000 строк меньше 283, как можно было ожидать, что объясняется способом масштабирования таблицы интернирования строк (эта тема рассматривается при использовании этого теста в главе 12).

Обычно бывает проще определить одно простое значение для параметра в исходном коде и использовать его при тестировании. При выполнении теста можно передать ему список значений каждого параметра, переопределяя значение, фиксированное в коде Java:

```
$ java -jar target/benchmarks.jar -p nStrings=1,1000000
```

Сравнение тестов

Этот тест создавался для того, чтобы выяснить, можно ли ускорить интернирование строк при использовании разных оптимизаций JVM. Для этого мы будем запускать тест с разными аргументами JVM, передавая эти аргументы в командной строке:

```
$ java -jar target/benchmarks.jar
... вывод теста 1 ...
$ java -jar target/benchmarks.jar -jvmArg -XX:StringTableSize=10000000
... вывод теста 2 ...
```

После этого можно вручную проанализировать и сравнить последствия от применения оптимизации на результаты.

На практике чаще требуется сравнить две реализации кода. Интернирование строк работает хорошо, но нельзя ли добиться еще лучших результатов, воспользовавшись простой хеш-картой и управляя ею вручную? Чтобы протестировать такое решение, мы определим еще один метод в классе и пометим его аннотацией `Benchmark`. Первая (субоптимальная) попытка могла бы выглядеть примерно так:

```
private static ConcurrentHashMap<String,String> map = new ConcurrentHashMap<>();
@Benchmark
public void testMap(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String s = new String("String to intern " + i);
        String t = map.putIfAbsent(s, s);
        bh.consume(t);
    }
}
```

jmh прогоняет все методы, снабженные аннотацией, через ту же серию итераций разогрева и измерения (всегда во вновь ответвленных JVM) и выдает удобную сравнительную сводку:

```
Benchmark               (nStrings)   Mode  Cnt    Score     Error  Units
MyBenchmark.testIntern       10000  thrpt   25  212.301 ± 207.550  ops/s
MyBenchmark.testMap          10000  thrpt   25  352.140 ±  84.397  ops/s
```

Ручное управление интернированными объектами обеспечило неплохое улучшение (хотя учтите, что в приведенном тесте остаются проблемы; это еще не последняя версия).

Подготовительный код

Приведенный ранее вывод был сокращен для формата книги, но при выполнении тестов jmh перед выводом результатов будет выведено длинное предупреждение. Суть этого предупреждения примерно такова: «Из того, что вы включили свой код в jmh, не следует, что вы написали хороший тест: убедитесь в том, что он тестирует именно то, что вы ожидали».

А теперь вернемся к определению теста. Мы хотим проверить, сколько времени занимает интернирование 10 000 строк, но по сути тестируется время создания (посредством конкатенации) 10 000 строк плюс время, затраченное на интернирование полученных строк. Диапазон значений этих строк также ограничен: они состоят из одинаковых 17 начальных символов, за которыми следует целое число. По аналогии с тем, как мы заранее создали входные данные для написанного вручную микробенчмарка с числами Фибоначчи, следует заранее создать входные данные и в этом случае.

На это можно возразить, что диапазон строк в этом тесте роли не играет, а объем работы по конкатенации незначителен, а следовательно, исходный тест абсолютно точен. Возможно, это и так, но чтобы доказать это, придется потрудиться. Вместо того чтобы строить предположения относительно происходящего, лучше просто написать тест, в котором эти вопросы не играют роли.

Также необходимо серьезно задуматься над тем, что же происходит в тесте. По сути таблица, в которой хранятся интернированные строки, представляет собой кэш: интернированная строка может в нем находиться (в таком случае она возвращается), а может и не находиться (в этом случае она вставляется в таблицу). А теперь при сравнении реализаций возникает проблема: хеш-карта с ручным управлением, предназначенная для параллельного доступа, никогда не очищается в ходе тестирования. Это означает, что в начальном цикле разогрева строки вставляются в карту, а в последующих циклах измерений строки уже находятся в ней: тест достигает 100%-ной частоты попаданий в кэш.

Таблица интернирования строк работает по другому принципу: ключи в таблице интернирования строк фактически являются слабыми ссылками. Тем не менее JVM может частично (или полностью) очистить данные в некоторый момент времени (так как интернированная строка выходит из области видимости сразу же после вставки в таблицу). Частота попаданий в кэш в этом случае не определена, но скорее всего, она будет далека от 100%. Следовательно, в текущем состоянии тесту интернирования строк придется выполнить больше работы, так

как ему придется чаще обновлять таблицу интернирования строк (для удаления и для последующего добавления записей).

Обеих проблем можно избежать, заранее создав строки в статическом массиве с их последующим интернированием (или вставкой в хеш-карту). Поскольку в статическом массиве хранятся ссылки на строки, ссылка в таблице строк не будет очищена. Тогда оба теста должны будут достигнуть 100%-ной частоты попадания в кэш в циклах измерений, и диапазон строк станет более полным.

Инициализация должна быть выполнена за пределами периода измерений, для чего используется аннотация `Setup`:

```
private static ConcurrentHashMap<String,String> map;
private static String[] strings;

@Setup(Level.Iteration)
public void setup() {
    strings = new String[nStrings];
    for (int i = 0; i < nStrings; i++) {
        strings[i] = makeRandomString();
    }
    map = new ConcurrentHashMap<>();
}

@Benchmark
public void testIntern(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String t = strings[i].intern();
        bh.consume(t);
    }
}
```

Аргумент `Level`, передаваемый аннотации `Setup`, управляет тем, когда должен выполняться заданны метод. `Level` принимает одно из трех значений:

- `Level.Trial` — подготовка выполняется однократно при инициализации кода хронометражного теста.
- `Level.Iteration` — подготовка выполняется перед каждой итерацией теста (каждым циклом измерения).
- `Level.Invocation` — подготовка выполняется перед каждым выполнением метода теста.

Аналогичная аннотация `Teardown` может использоваться в других случаях для очистки состояния в случае необходимости. jmh поддерживает много других возможностей, включая измерение отдельных вызовов метода или изменение среднего времени вместо пропускной способности, передачу дополнительных аргументов порожденной JVM, управление синхронизацией потоков и т. д.

Я не пытаюсь предоставить полное руководство по jmh; скорее, этот пример идеально демонстрирует сложность, задействованную в написании даже простого микробенчмарка.

КОД ПОДГОТОВКИ И НИЗКОУРОВНЕВЫЕ ТЕСТЫ

Некоторые тесты jmh в последующих главах могут сообщать, что некоторые операции различаются всего на несколько наносекунд. Эти тесты в действительности не измеряют конкретные операции на наносекундном уровне; они измеряют время выполнения множества операций и сообщают среднее значение (со статистическим отклонением). jmh делает это все за вас.

Отсюда следует одно важное предупреждение: если есть код подготовки, который должен выполняться при каждом вызове, jmh будет трудно выполнять этот анализ среднего времени. По этой причине (и некоторым другим) рекомендуется использовать аннотацию Level.Invocation как можно реже и только для тестовых методов, выполнение которых занимает относительно много времени.

Управление выполнением и воспроизводимость

После того как у вас появится правильный микробенчмарк, необходимо запустить его так, чтобы результаты имели статистический смысл для показателей, которые вы измеряете.

Как было показано ранее, по умолчанию jmh выполняет целевой метод в течение 10 секунд, выполняя его максимально возможное количество раз за этот интервал (так, в предыдущем примере он был выполнен в среднем 1772 раза за 10 секунд). Каждый 10-секундный тест является итерацией, по умолчанию используются пять итераций разогрева (их результаты отбрасываются) и пять итераций измерений при каждом ответвлении новой JVM. И все это повторяется для пяти испытаний.

Все это было сделано для того, чтобы фреймворк jmh мог провести статистический анализ для вычисления доверительного интервала в результате. В примерах, приводившихся ранее, доверительный интервал 99,9% имел диапазон около 10%, что может быть (а может и не быть) достаточно при сравнении с другими тестами.

Изменяя эти параметры, мы можем сократить или расширить доверительный интервал. Например, ниже приведены результаты для выполнения двух тестов с малым количеством итераций измерений и испытаний:

```
Benchmark                (nStrings)   Mode  Cnt    Score     Error  Units
MyBenchmark.testIntern        10000  thrpt    4  233.359 ±  95.304  ops/s
MyBenchmark.testMap           10000  thrpt    4  354.341 ± 166.491  ops/s
```

С этим результатом все выглядит так, словно метод `intern()` намного хуже использования карты, но взгляните на диапазон: возможно, реальный результат первого примера близок к 330 операций в секунду, тогда как реальный результат второго примера близок к 200 операций в секунду. Даже если это маловероятно, диапазоны слишком широки, для того чтобы окончательно решить, какой вариант лучше.

Такой результат получен из-за того, что мы использовали только два ответвленных испытания по две итерации каждое. Если увеличить это количество до 10 итераций, результат улучшится:

```
MyBenchmark.testIntern        10000  thrpt   20  172.738 ± 29.058  ops/s
MyBenchmark.testMap           10000  thrpt   20  305.618 ± 22.754  ops/s
```

Теперь диапазоны разделены, а мы можем с уверенностью заключить, что решение с картой лучше (по крайней мере для теста со 100%-ной частотой попадания в кэш и 10 000 неизменяемых строк).

Нет единственно верного правила относительно того, сколько итераций, сколько ответвленных испытаний или какой промежуток времени понадобится для получения данных, достаточных для получения очевидных результатов. Если сравнить эти два метода, которые мало отличаются друг от друга, потребуется намного больше итераций и испытаний. С другой стороны, если они настолько близки, возможно, лучше поискать другое решение с более заметным влиянием на производительность. И это еще один пример, в котором искусство влияет на науку; в какой-то момент вам придется самостоятельно решать, где проходят границы.

Для управления всеми этими переменными — количеством итераций, длиной каждого интервала и т. д. — используются аргументы командной строки. Самые важные из них:

- `-f 5` — количество ответвляемых испытаний (по умолчанию: 5).
- `-wi 5` — количество итераций на испытание (по умолчанию: 5).
- `-i 5` — количество измерительных итераций на испытание (по умолчанию: 5).
- `-r 10` — минимальная продолжительность каждой итерации (в секундах); итерация может выполняться и более длительное время в зависимости от фактической длины целевого метода.

Повышение этих параметров обычно снижает дисперсию результата, пока вы не получите желаемый доверительный диапазон. И наоборот, для более стабильных

тестов уменьшение этих параметров обычно сокращает время, необходимое для выполнения теста.

РЕЗЮМЕ

- jmh — фреймворк для написания микробенчмарков, который содержит средства, помогающие правильно выполнять требования к таким тестам.
- jmh не заменит серьезных размышлений о написании кода, характеристики которого вы собираетесь измерять; это всего лишь полезный инструмент для создания такого кода.

Примеры кода

Многие примеры в книге основаны на простом приложении для вычисления «исторических» максимумов и минимумов цен акций за заданный промежуток времени, а также стандартного отклонения за это время. Определение «исторических» заключено в кавычки, потому что в приложении все данные являются вымышленными; цены и биржевые обозначения генерируются случайным образом.

Основным объектом в приложении является объект `StockPrice`, представляющий диапазон цен на акции в заданный день вместе с коллекцией опционных цен на эти акции:

```
public interface StockPrice {
    String getSymbol();
    Date getDate();
    BigDecimal getClosingPrice();
    BigDecimal getHigh();
    BigDecimal getLow();
    BigDecimal getOpeningPrice();
    boolean isYearHigh();
    boolean isYearLow();
    Collection<? extends StockOptionPrice> getOptions();
}
```

В примерах обычно используется коллекция этих цен, представляющая историю их изменения за период времени (например, 1 год или 25 лет в зависимости от примера):

```
public interface StockPriceHistory {
    StockPrice getPrice(Date d);
    Collection<StockPrice> getPrices(Date startDate, Date endDate);
    Map<Date, StockPrice> getAllEntries();
```

```
    Map<BigDecimal,ArrayList<Date>> getHistogram();
    BigDecimal getAveragePrice();
    Date getFirstDate();
    BigDecimal getHighPrice();
    Date getLastDate();
    BigDecimal getLowPrice();
    BigDecimal getStdDev();
    String getSymbol();
}
```

Базовая реализация этого класса загружает набор цен из базы данных:

```
public class StockPriceHistoryImpl implements StockPriceHistory {
    ...
    public StockPriceHistoryImpl(String s, Date startDate,
        Date endDate, EntityManager em) {
        Date curDate = new Date(startDate.getTime());
        symbol = s;
        while (!curDate.after(endDate)) {
            StockPriceImpl sp = em.find(StockPriceImpl.class,
                        new StockPricePK(s, (Date) curDate.clone()));
            if (sp != null) {
                Date d = (Date) curDate.clone();
                if (firstDate == null) {
                    firstDate = d;
                }
                prices.put(d, sp);
                lastDate = d;
            }
            curDate.setTime(curDate.getTime() + msPerDay);
        }
    }
    ...
}
```

Предполагается, что архитектура выборки загружается из базы данных; эта функциональность будет использована в примерах главы 11. Для упрощения запуска примеров в большинстве случаев в них будет использоваться фиктивный диспетчер сущностей, генерирующий случайные данные. По сути многие примеры представляют собой мезобенчмарки уровня модуля, хорошо подходящие для демонстрации текущих проблем с быстродействием, — но представление о фактическом быстродействии приложения можно будет составить только при выполнении всего приложения (как в главе 11).

Еще один нюанс заключается в том, что многие примеры таким образом зависят от производительности используемого генератора случайных чисел. В отличие от примера микробенчмарка, это сделано намеренно для демонстрации различных проблем производительности в Java. (Если на то пошло, целью примеров

является измерение производительности произвольного кода, а производительность генератора случайных чисел соответствует этой цели. Ситуация сильно отличается от микробенчмарка, в котором включение времени генерирования случайных чисел повлияло бы на вычисления в целом.)

Примеры также сильно зависят от производительности класса `BigDecimal`, который используется для хранения всех точек данных. Это стандартный способ хранения денежных сумм; если денежные данные хранятся в виде примитивных объектов `double`, задача округления половин центов и меньших сумм создает немало проблем. С точки зрения написания примеров этот выбор также полезен, потому что он позволяет выполнять «бизнес-логику» или некоторые продолжительные вычисления — особенно при вычислении стандартного отклонения серии цен. Для вычисления стандартного отклонения необходимо знать квадратный корень числа `BigDecimal`. Стандартный Java API не предоставляет такой возможности, но в примерах используется следующий метод:

```
public static BigDecimal sqrtB(BigDecimal bd) {
    BigDecimal initial = bd;
    BigDecimal diff;
    do {
        BigDecimal sDivX = bd.divide(initial, 8, RoundingMode.FLOOR);
        BigDecimal sum = sDivX.add(initial);
        BigDecimal div = sum.divide(TWO, 8, RoundingMode.FLOOR);
        diff = div.subtract(initial).abs();
        diff.setScale(8, RoundingMode.FLOOR);
        initial = div;
    } while (diff.compareTo(error) > 0);
    return initial;
}
```

Он представляет собой реализацию вавилонского способа вычисления квадратного корня числа. Это не самая эффективная реализация; в частности, исходное предположение можно заметно улучшить, что позволило бы сэкономить часть итераций. Выбор был намеренным, потому что он позволяет вычислениям занять некоторое время (эмуляция бизнес-логики) и при этом демонстрирует основной принцип из главы 1: часто лучшим способом ускорения кода Java становится написание улучшенного алгоритма независимо от каких-либо оптимизаций или практик программирования Java.

Стандартное отклонение, средняя цена и гистограмма реализации интерфейса `StockPriceHistory` — все это производные значения. В разных примерах эти значения будут вычисляться немедленно (при загрузке данных из диспетчера сущностей) или в отложенном режиме (при вызове метода получения данных). Аналогичным образом интерфейс `StockPrice` ссылается на интерфейс `StockOptionPrice` — цену отдельных опционов для заданных акций в заданный день. Эти значения могут быть получены от диспетчера сущностей немедленно

или в отложенном режиме. В обоих случаях определение этих интерфейсов позволяет сравнивать эти подходы в разных ситуациях.

Эти интерфейсы также естественно вписываются в REST-приложения Java: пользователи могут выдать вызов с параметрами, определяющими биржевое обозначение и диапазон дат для интересующего вида акций. В стандартном случае запрос пройдет через стандартный механизм вызова, использующий вызов Java API for RESTful Web Services (JAX-RS), который разбирает входные параметры, вызывает встроенный компонент JPA для получения нижележащих данных и отправляет ответ:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public JsonObject getStockInfo(
    @DefaultValue("" + StockPriceHistory.STANDARD)
          @QueryParam("impl") int impl,
    @DefaultValue("true") @QueryParam("doMock") boolean doMock,
    @DefaultValue("") @QueryParam("symbol") String symbol,
    @DefaultValue("01/01/2019") @QueryParam("start") String start,
    @DefaultValue("01/01/2034") @QueryParam("end") String end,
    @DefaultValue("0") @QueryParam("save") int saveCount
    ) throws ParseException {

    StockPriceHistory sph;
    EntityManager em;
    DateFormat df = localDateFormatter.get();  // Потоково-локальный
    Date startDate = df.parse(start);
    Date endDate = df.parse(end);

    em = // ... Получить диспетчер сущностей
    sph = em.find(...на основании аргументов...);
    return JSON.createObjectBuilder()
            .add("symbol", sph.getSymbol())
            .add("high", sph.getHighPrice())
            .add("low", sph.getLowPrice())
            .add("average", sph.getAveragePrice())
            .add("stddev", sph.getStdDev())
            .build();
}
```

Этот класс может внедрять разные реализации компонента истории (для немедленной или отложенной инициализации, среди прочего); также он может кэшировать информацию, полученную из базы данных (или от фиктивного диспетчера сущностей). Все эти решения часто встречаются при настройке производительности корпоративных приложений (в частности, кэширование данных на среднем уровне иногда приносит значительный выигрыш производительности для серверов приложений). В примерах, приведенных в книге, также анализируются плюсы и минусы таких решений.

Итоги

Тестирование производительности требует компромиссов. Хороший выбор между несколькими альтернативами очень важен для успешного отслеживания характеристик производительности системы. Правильный выбор того, что именно должно тестироваться — первый этап, на котором практический опыт и интуиция окажут неоценимую помощь при настройке тестов производительности.

Микробенчмарки помогают задать ориентиры по производительности для некоторых операций. Помимо них, есть широкий континуум других тестов, от малых тестов уровня модуля до больших многоуровневых сред. Разные тесты на всей протяженности этого континуума обладают своими достоинствами, и правильный выбор тестов в континууме — еще один этап, в котором в игру вступают опыт и интуиция. В конечном итоге ничто не заменит тестирования полного приложения в том виде, в котором оно развертывается в рабочей среде; только тогда можно будет понять полный эффект всех проблем, связанных с производительностью.

Понимание того, что является или не является реальной регрессией в коде, тоже не всегда однозначно. Программы всегда проявляют случайное поведение, и после внедрения в формулу случайности никогда нельзя быть на 100% уверенными в том, что же означают данные. Применение статистического анализа к результатам поможет направить анализ по более объективному пути, но даже тогда некоторая субъективность неизбежно останется. Понимание нижележащих вероятностей и их смысла поможет сократить эту субъективность.

Наконец, после того как фундамент будет заложен, автоматизированная система тестирования может быть настроена для получения полной информации обо всем, что происходит во время теста. Зная, что происходит в системе и что означают результаты тестов, аналитик по производительности может применить свои навыки (как науку, так и искусство), чтобы программа могла продемонстрировать наилучшую возможную производительность.

ГЛАВА 3

Инструментарий производительности Java

В анализе производительности центральное место занимает видимость — то есть наличие информации о том, что происходит внутри приложения и в среде его выполнения. В области видимости центральное место занимают программные инструменты. Таким образом, вся суть оптимизации производительности заключается в инструментах.

В главе 2 я показал, насколько важно наличие данных для анализа производительности: вы должны измерить производительность приложения и понять, что означают полученные данные. Анализ производительности тоже должен базироваться на данных: вы должны располагать данными о том, что именно делает программа, чтобы улучшить ее производительность. Вопрос о том, как получить и интерпретировать эти данные, станет темой этой главы.

Существуют сотни программ для получения информации о том, что делает приложение Java. Рассматривать все эти программы было бы нереально. Многие важные инструменты включены в пакет JDK (Java Development Kit), и хотя у них есть конкуренты (как коммерческие, так и распространяемые с открытым кодом), в этой главе основное внимание уделяется инструментам из JDK как наиболее приоритетным.

Средства операционной системы и анализ

Отправная точка для анализа программ вообще не имеет прямого отношения к Java: это базовый набор инструментов, входящих в поставку операционной системы. В системах на базе Unix это `sar` (System Accounting Report) и программы, входящие в пакет, — `vmstat`, `iostat`, `prstat` и т. д. В поставку Windows

также входят графические мониторы ресурсов и программы командной строки (такие, как `typeperf`).

Каждый раз при выполнении тестов производительности данные должны запрашиваться у операционной системы. Как минимум необходимо собрать информацию об использовании процессора, памяти и диска; если программа использует сеть, то также должна быть получена информация об использовании сети. Если тесты производительности автоматизированы, это означает зависимость от средств командной строки. Но даже если тесты выполняются в интерактивном режиме, лучше иметь программу командной строки, которая сохраняет вывод, вместо того чтобы рассматривать график в графическом интерфейсе и гадать, что это все означает. Выходные данные всегда можно представить в графическом виде при проведении анализа.

Использование процессора

Начнем с мониторинга процессора и того, что он говорит нам о программах Java. Использование процессора обычно делится на две категории: пользовательское время и системное время (в Windows оно называется привилегированным временем). Пользовательское время — процент времени, в течение которого процессор выполняет код приложения, тогда как системное время — процент времени, в течение которого процессор выполняет код ядра. Системное время связано с операциями приложения; например, если приложение выполняет операции ввода/вывода, ядро выполняет код для чтения файла с диска, записывает буферизованные данные в сеть и т. д. Все, что использует системные ресурсы, заставляет приложение использовать больше системного времени.

Целью настройки производительности является повышение уровня использования процессора до максимально высокого уровня на как можно более короткий период времени. На первый взгляд это выглядит противоестественно; наверняка вам доводилось сидеть за своим настольным компьютером и следить за тем, как он отчаянно тормозит, потому что процессор загружен на 100%. Разберемся, что же фактически нам сообщают данные об использовании процессора.

Первое, о чем следует помнить — что показатель использования процессора является усредненным значением по интервалу времени: 5 секунд, 30 секунд и даже до 1 секунды (но меньшие значения на практике не встречаются). Допустим, средний уровень использования процессора в программе составляет 50% в течение тех 10 минут, в которых выполняется программа. Это означает, что процессор простаивает половину времени; если программу можно реструктуризовать так, чтобы в ней не было участков простоя (или других узких мест), производительность можно удвоить, чтобы программа выполнялась за 5 минут (со 100%-ной загрузкой процессора).

Если используемый программой алгоритм можно улучшить и снова удвоить производительность, процессор будет работать со 100%-ной загрузкой в течение 2,5 минуты, необходимых для завершения программы. Показатель использования процессора — признак того, насколько эффективно программа использует процессор, и чем больше число — тем лучше.

Выполнив команду `vmstat 1` на своем настольном компьютере с Linux, я получу серию строк (по одной в секунду), которая выглядит так:

```
% vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 2  0      0 1797836 1229068 1508276 0    0     0     9 2250 3634 41  3 55  0
 2  0      0 1801772 1229076 1508284 0    0     0     8 2304 3683 43  3 54  0
 1  0      0 1813552 1229084 1508284 0    0     3    22 2354 3896 42  3 55  0
 1  0      0 1819628 1229092 1508292 0    0     0    84 2418 3998 43  2 55  0
```

Пример был получен при запуске приложения с одним активным потоком — это упрощает анализ примера. Но концепции применимы даже в ситуациях с многопоточным выполнением.

В каждую секунду процессор занят на 450 мс (42% — время выполнения пользовательского кода, 3% — время выполнения системного кода). В течение 550 мс процессор простаивает. Бездействие процессора может объясняться несколькими причинами:

- Приложение может блокироваться по примитиву синхронизации, что приостанавливает его выполнение до освобождения блокировки.
- Приложение может чего-то ожидать — например, ответа на обращение к базе данных.
- Приложению нечего делать.

Первые две ситуации — это признак того, что можно что-то сделать. Если уровень конкуренции за блокировку может быть понижен, или базу данных можно настроить так, чтобы она быстрее возвращала ответ, программа будет работать быстрее, а средняя загрузка процессора приложением возрастет (конечно, если нет другой проблемы, которая продолжит блокировать работу приложения).

Недопонимание чаще всего возникает в третьем пункте. Если приложению нечего делать (а простой не обусловлен ожиданием блокировки или другого ресурса), то процессор будет тратить время на выполнение кода приложения. Это общий принцип, не относящийся напрямую к Java. Допустим, вы пишете простой сценарий, содержащий бесконечный цикл. При выполнении этот сценарий загрузит процессор на 100%. Следующее пакетное задание делает то же самое в Windows:

```
ECHO OFF
:BEGIN
ECHO LOOPING
GOTO BEGIN
REM В эту точку управление не передается...
ECHO DONE
```

Что означает, если этот сценарий не потребляет процессор на 100%? Это означает, что у операционной системы есть то, что она могла бы сделать (например, вывести еще одну строку с сообщением `LOOPING`), но вместо этого она решила бездействовать. Так как бездействие в данном случае не приносит никакой пользы и мы бы выполняли полезные (долгие) вычисления, принудительное периодическое бездействие процессора будет означать, что получение искомого ответа потребует больше времени.

ОГРАНИЧЕНИЕ ИСПОЛЬЗОВАНИЯ ПРОЦЕССОРА ДЛЯ ПРОГРАММ

Выполнение программы в любой момент, когда процессор доступен, обеспечивает максимальную производительность программы. В некоторых случаях такое поведение нежелательно. Например, если вы запускаете программу SETI@home, она будет потреблять все свободные такты процессора на вашей машине. Это нормально, если вы не заняты работой или же серфите в интернете либо пишете документы, но в других ситуациях оно может повредить эффективности вашей работы (например, если вы играете в компьютерную игру, создающую интенсивную нагрузку на процессор!).

Существуют некоторые механизмы операционной системы, позволяющие искусственно ограничивать долю ресурсов процессора, которые могут использоваться программой; фактически они заставляют процессор оставлять свободные такты просто на случай, что кто-то захочет ими воспользоваться. Также можно изменять приоритеты процессов, чтобы фоновые задания не мешали процессору делать то, что вам нужно, не оставляя при этом свободных тактов процессора. Эти методы выходят за рамки нашего обсуждения (и кстати говоря, программа SETI@home позволяет вам настроить их; она не занимает все свободные такты процессора на вашей машине, если только вы не разрешите ей это делать).

Если выполнить эту команду на однопроцессорной машине или контейнере, в большинстве случаев вы вряд ли заметите, как она выполняется. Но если вы попытаетесь запустить новую программу или измерить производительность другого приложения, эффект наверняка будет заметным. Операционные системы хорошо справляются с квантованием времени для программ, кон-

курирующих за ресурсы процессора, но новой программе будет доставаться меньше процессорного времени и она будет выполняться медленнее. Из-за этого люди иногда начинают думать, что будет хорошо оставить часть процессорного времени свободным просто на случай, если они понадобятся кому-то другому.

Но операционная система не может угадать, что вы собираетесь делать дальше; она (по умолчанию) делает все, что может, вместо того чтобы оставить процессор без нагрузки.

Java и однопроцессорное использование

Вернемся к обсуждению приложений Java — о чем свидетельствует периодическое бездействие процесса в этом случае? Это зависит от типа приложения. Если речь идет о приложении пакетного типа, которое выполняет фиксированный объем работы, вы никогда не должны столкнуться с бездействием процессора, потому что это означало бы, что процессору нечего делать. Повышение уровня использования процессора всегда является целью для пакетных заданий, потому что задание будет завершаться быстрее. Если процессор уже используется на 100%, вы всегда можете заняться поиском оптимизаций, которые позволяли бы завершить работу быстрее (стараясь удерживать при этом процессор на 100%-ном уровне использования).

Если в измерениях задействовано приложение в серверном стиле, которое получает запросы от источника, простои могут возникнуть из-за отсутствия работы: например, если веб-сервер обработал все входящие запросы HTTP и ожидает следующего запроса. Здесь в игру вступает среднее время. Пример вывода `vmstat` был получен во время работы сервера, обрабатывающего один запрос в секунду. Серверу приложения понадобилось 450 мс для обработки запроса — это означает, что процессор был занят на 100% в течение 450 мс и занят на 0% в течение 550 мс. Именно это имеет в виду программа, сообщая о том, что процессор занят на 45%.

Хотя это обычно происходит на уровне детализации, слишком малом для наглядного представления, при выполнении нагрузочных приложений процессор обычно работает короткими «импульсами». Тот же паттерн макроуровня проявится в отчетах, если процессор будет получать один запрос каждые полсекунды, а среднее время обработки запроса составляет 225 мс. Процессор занят 225 мс, бездействует 275 мс, снова занят 225 мс, снова бездействует 275 мс: в среднем занят на 45% и свободен на 55%.

Если приложение будет оптимизировано так, чтобы обработка каждого запроса занимала только 400 мс, общий уровень использования тоже сократится (до 40%). Это единственный случай, когда снижение загрузки процессора имеет смысл — в систему поступает фиксированный объем нагрузки, а приложение

не ограничивается по внешним ресурсам. С другой стороны, эта оптимизация также предоставляет возможность добавить в систему дополнительную нагрузку, что в конечном итоге приведет к повышению загрузки процессора. И на микроуровне оптимизация в данном случае все еще сводится к повышению уровня использования процессора до 100% на короткий промежуток времени (400 мс, необходимые для обработки запроса) — просто продолжительность импульса загрузки процессора слишком мала, чтобы большинство программ могло зарегистрировать 100%-ный уровень использования процессора.

Java и многопроцессорное использование

В рассмотренном примере предполагался один поток, выполняемый на одном процессоре, но для более общего случая с несколькими потоками, работающими на разных процессорах, концепции остаются неизменными. Множественные потоки могут исказить среднюю загрузку процессора разными интересными способами — один пример такого рода, приведенный в главе 5, демонстрирует влияние многопоточной уборки мусора на уровень использования процессора. Но в общем случае целью многопоточного выполнения на многопроцессорной машине остается повышение уровня использования процессора, для чего вы принимаете меры к устранению блокирования отдельных потоков или снижение уровня использования процессора (на продолжительном интервале), потому что потоки завершили свою работу и ожидают поступления новой работы.

В многопоточной многопроцессорной конфигурации есть еще одно важное дополнение относительно того, когда процессоры могут бездействовать: это возможно даже при наличии работы. Это может происходить, если в программе отсутствуют потоки для выполнения этой работы. Типичный случай — приложение с пулом потоков для выполнения различных задач, имеющим фиксированный размер. Задачи для потоков помещаются в очередь; если поток бездействует, а задача находится в очереди, поток берет задачу и выполняет ее. При этом каждый поток может выполнять только одну задачу в любой момент времени, а если эта конкретная задача блокируется (например, в ожидании ответа от базы данных), поток не может переключиться на новую задачу. Следовательно, в отдельные моменты возможны периоды, в которых имеются задачи для выполнения, но нет доступных потоков для их выполнения; результатом становится бездействие процессора.

В этом конкретном примере размер пула потоков следовало бы увеличить. Тем не менее не стоит полагать, что только из факта доступности бездействующего процессора следует, что размер потока пулов нужно увеличить для выполнения большего объема работы. Программа может не получать необходимые такты процессора по двум причинам, упоминавшимся ранее: из-за узких мест в блокировках или из-за внешних ресурсов. Важно понять, почему программа не полу-

чает доступа к процессору, перед выбором курса действий. (За дополнительной информацией по этой теме обращайтесь к главе 9.)

Просмотр данных об использовании процессора станет первым шагом для понимания производительности приложения, но не ждите от него слишком многого. Он только поможет вам понять, используется ли процессор в той степени, которую можно было бы ожидать, и не указывают ли данные на наличие проблем с синхронизацией или ресурсами.

Очередь выполнения

Системы Windows и Unix позволяют отслеживать количество потоков, которые могут выполняться в системе (то есть они не заблокированы по вводу/выводу, не приостановлены и т. д.). В системах Unix это называется *очередью выполнения*; многие служебные программы включают длину очереди выполнения в свои выходные данные. К их числу относится программа `vmstat`, вывод которой был приведен в предыдущем разделе: первое число в каждой строке — длина очереди выполнения. В Windows это число называется очередью процессора и выводится (среди прочего) программой `typeperf`:

```
C:> typeperf -si 1 "\System\Processor Queue Length"
"05/11/2019 19:09:42.678","0.000000"
"05/11/2019 19:09:43.678","0.000000"
```

В выходных данных присутствуют важные различия: длина очереди выполнения в системах Unix (1 или 2 в примере вывода `vmstat`) — количество всех выполняемых потоков, которые выполняются или могут выполняться при наличии доступных процессоров. В этом примере всегда присутствовал как минимум один поток, который должен был выполняться: один поток, выполняющий работу приложения. Тем не менее длина очереди выполнения не может быть менее 1. Помните, что очередь выполнения представляет все состояние машины, поэтому иногда могут существовать другие потоки (из других процессов), которые тоже хотят выполняться; поэтому длина очереди выполнения иногда достигала 2 в этом примере.

В Windows длина очереди процессора не включает количество потоков, выполняемых в настоящее время. По этой причине в примере вывода `typeperf` длина очереди процессора равна 0, хотя на машине работало все то же однопоточное приложение, в котором всегда выполнялся один поток.

Если количество потоков для выполнения превышает число доступных процессоров, производительность начинает снижаться. Таким образом, в общем случае длина очереди процессора должна быть равна в Windows или равна (или меньше) количеству процессоров в системах Unix. Это правило не является

непреложным; системные процессы и другие факторы периодически вступают в действие и ненадолго повышают это значение без сколько-нибудь существенных последствий для производительности. Но если длина очереди выполнения остается слишком большой в течение сколько-нибудь значительного периода времени, это признак того, что машина перегружена, и вам следует искать возможности сокращения объема работы, выполняемой на машине (либо перемещением заданий на другую машину, либо оптимизацией кода).

РЕЗЮМЕ

- Уровень использования процессора — первое, на что нужно обращать внимание при анализе производительности приложения.
- Цель оптимизации кода — повышение уровня использования процессора (на короткий период времени).
- Разберитесь, почему уровень использования процессора слишком низок, прежде чем браться за дело и пытаться настраивать приложение.

Уровень использования диска

Мониторинг использования диска имеет две важные цели. Первая относится к самому приложению: если приложение выполняет большой объем дискового ввода/вывода, то операции ввода/вывода легко могут стать узким местом.

Определить, в каком случае дисковый ввод/вывод создает проблемы с производительностью, непросто, потому что многое зависит от поведения приложения. Если приложение не обеспечивает эффективной буферизации данных, записываемых на диск (пример приводится в главе 12), статистика дискового ввода/вывода будет низкой. Но если объем ввода/вывода в приложении превышает возможности диска, статистика дискового ввода/вывода будет высокой. В любом случае производительность можно улучшить; будьте внимательны и учитывайте обе возможности. Базовые средства мониторинга ввода/вывода в некоторых системах лучше, чем в других. Ниже приведен частичный вывод `iostat` в системе Linux:

```
% iostat -xm 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          23.45    0.00   37.89    0.10    0.00   38.56

          Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s
          sda               0.00    11.60    0.60   24.20     0.02

          wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
          0.14    13.35     0.15    6.06    5.33    6.08   0.42   1.04
```

Приложение записывает данные на диск `sda`. На первый взгляд статистика операций с диском выглядит хорошо, `w_await` — время обслуживания каждой записи — относительно низко (6,08 мс), и диск задействован только на 1,04%. (Допустимые значения этого показателя зависят от физического диска, но диск на 5200 RPM в моей настольной системе работает нормально, когда время обслуживания ниже 15 мс.)

Однако есть признак, который указывает, что что-то идет не так: система проводит 37,89% своего времени в режиме ядра. Если система выполняла бы другие операции ввода/вывода (в других программах), это было бы объяснимо; но если все это системное время происходит из тестируемого приложения, в системе что-то работает неэффективно.

Тот факт, что система выполняет 24,2 операции записи в секунду — другой тревожный признак; это слишком большое значение при записи всего 0,14 Мбайт в секунду (`MBps`). Ввод/вывод стал узким местом, и на следующем шаге необходимо определить, как приложение выполняет операции записи.

Оборотная сторона проявляется тогда, когда диск не справляется с запросами ввода/вывода:

```
% iostat -xm 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          35.05    0.00    7.85   47.89    0.00    9.20

          Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s
          sda               0.00     0.20    1.00  163.40     0.00

          wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
          81.09  1010.19   142.74  866.47   97.60  871.17   6.08 100.00
```

У Linux есть одна приятная особенность: система немедленно сообщает о 100%-ном уровне использования диска; она также сообщает, что процессы проводят 47,89% своего времени в `iowait` (то есть в ожидании диска).

Даже в других системах, в которых доступны только низкоуровневые данные, по этим данным можно увидеть признаки возникших проблем: время завершения ввода/вывода (`w_await`) составляет 871 мс, размер очереди достаточно велик, а на диск записывается 81 Мбайт данных в секунду. Все это указывает на проблемы с дисковым вводом/выводом, а объем ввода/вывода в приложении (а возможно, в других местах системы) следует сократить.

Вторая причина для мониторинга использования диска (даже если не предполагается, что приложение будет выполнять значительный объем ввода/вывода) — необходимость отслеживания подкачки (swapping) в системе. Компьютеры оснащаются ограниченным объемом физической памяти, но на них могут выполняться приложения, использующие намного больший объем виртуальной

памяти. Приложения обычно резервируют больше памяти, чем им необходимо, и работают с подмножеством выделенной памяти. В обоих случаях операционная система может хранить неиспользуемые части памяти на диске и подгружать их в физическую память только по мере необходимости.

Такой механизм управления памятью обычно работает хорошо, особенно для интерактивных программ и приложений с графическим интерфейсом (и это хорошо, иначе вашему ноутбуку потребовалось бы намного больше памяти). Он несколько хуже работает для серверных приложений, потому что такие приложения обычно расходуют большие объемы памяти. И совсем плохо он работает в любых приложениях Java (включая приложения с графическим интерфейсом на базе Swing, работающие на вашем настольном компьютере) из-за кучи Java. Тема более подробно рассматривается в главе 5.

Системные программы также могут сообщить о том, выполняется ли подкачка в системе; например, вывод `vmstat` включает два столбца (`si` для подгрузки (Swap In) и `so` (Swap Out) для выгрузки), которые содержат информацию о том, происходит ли подкачка в системе. Активность диска — еще один индикатор, указывающий на то, что в системе происходит подкачка. Обращайте внимание на эти данные, потому что система, в которой происходит подкачка — перемещение страниц данных из основной памяти на диск и обратно, — будет иметь плохую производительность. Системы должны настраиваться таким образом, чтобы подкачка была сведена к минимуму.

РЕЗЮМЕ

- Мониторинг использования диска играет важную роль во всех приложениях. Даже в приложениях, которые не осуществляют запись на диск напрямую, системный механизм подкачки может повлиять на производительность.
- В приложениях, записывающих данные на диск, могут возникать узкие места как из-за того, что запись данных осуществляется неэффективно (слишком низкая пропускная способность), так и из-за того, что они записывают слишком много данных.

Уровень использования сети

Если ваше приложение использует сеть — например, REST-сервер, — вам также необходимо отслеживать сетевой трафик. Использование сети имеет много общего с обменом данными с диском: приложение может использовать сеть неэффективно, так что канал связи будет простаивать, или объем данных, записываемых в конкретный сетевой интерфейс, может превышать возможности этого интерфейса.

К сожалению, стандартные системные средства далеко не идеальны для решения проблем с мониторингом сетевого трафика, потому что обычно они сообщают только количество пакетов и количество байтов, передаваемых и получаемых через конкретный сетевой интерфейс. Эта информация полезна, но она ничего не говорит об излишней или недостаточной загрузке сети.

В системах Unix основным средством сетевого мониторинга является программа `netstat` (в большинство поставок Linux `netstat` не входит и загружается отдельно). В системе Windows для отслеживания использования сети в сценариях может использоваться программа `typeperf` — но в данном случае графический интерфейс предоставляет определенные преимущества: стандартный монитор ресурсов Windows отображает график уровня использования сети. К сожалению, в ситуации с автоматизированным тестированием производительности сети графический интерфейс особой пользы не принесет.

К счастью, существуют многочисленные программы для отслеживания сетевого трафика (как коммерческие, так и распространяемые с открытым кодом). В системах Unix популярная программа командной строки `nicstat` выдает сводку трафика для каждого сетевого интерфейса, включая степень загруженности интерфейса:

```
% nicstat 5
Time      Int       rKB/s   wKB/s   rPk/s   wPk/s   rAvs    wAvs   %Util  Sat
17:05:17  e1000g1   225.7   176.2   905.0   922.5   255.4   195.6  0.33   0.00
```

Интерфейс `e1000g1` является 1000-мегабитным; в данном примере он используется не особо интенсивно (`0.33%`). Эта программа (а также другие похожие программы) удобна тем, что она вычисляет уровень использования интерфейса. В этом выводе 225,7 Кбит/с записываются и 176,2 Кбит/с читаются через интерфейс. Деление на 1000 Мбит/с дает показатель загрузки сети 0,33%, причем программа `nicstat` смогла вычислить пропускную способность интерфейса автоматически.

Такие программы, как `typeperf` или `netstat`, сообщают объем прочитанных и записанных данных, но чтобы вычислить уровень использования сети, необходимо определить пропускную способность интерфейса и выполнить вычисления в ваших сценариях. Не забывайте, что скорость передач измеряется в битах в секунду (бит/с), тогда как программы обычно выдают результаты в байтах в секунду (байт/с). 1000-мегабитная сеть соответствует 125 мегабайт (Мбайт) в секунду. В данном примере читаются 0,22 Мбайт/с и записываются 0,16 Мбайт/с; сложив эти числа и разделив на 125, получаем уровень использования сети 0,33%. Таким образом, в `nicstat` (и других похожих программах) нет ничего волшебного; они просто более удобны в использовании.

Сети не могут поддерживать 100% уровня использования. Для локальных сетей Ethernet уровень использования, устойчиво держащийся свыше 40%, указывает

на насыщение интерфейса. В сетях с пакетной коммутацией или использующих другую технологию передачи данных максимально возможная установившаяся скорость передачи будет другой; чтобы определить подходящее значение для вашего случая, обратитесь к сетевому администратору. С другой стороны, Java просто использует сетевые параметры и интерфейсы операционной системы.

РЕЗЮМЕ

- В сетевых приложениях контролируйте состояние сети и следите за тем, чтобы оно не стало узким местом.
- В приложениях, передающих данные по сети, узкие места возникают либо из-за того, что они записывают данные неэффективно (недостаточный объем передаваемых данных), либо из-за того, что они записывают слишком много данных (избыточный объем передаваемых данных).

Средства мониторинга Java

Чтобы разобраться в состоянии самой JVM, нам понадобятся средства мониторинга Java. Эти средства входят в поставку JDK.

`jcmd` — выводит информацию о базовых классах, потоках и JVM для процессов Java. Программа подходит для использования в сценариях; она выполняется следующим образом:

```
% jcmd идентификатор_процесса команда необязательные_аргументы
```

Команда `help` выводит список всех возможных команд, а команда `help` <*команда*> выводит описание синтаксиса конкретной команды.

`jconsole` — выводит графическое представление параметров работы JVM, включая использование потоков и классов, а также деятельности уборщика мусора.

`jmap` — предоставляет дампы кучи и другую информацию об использовании памяти JVM. Подходит для использования в сценариях, хотя дампы кучи должны использоваться в программах последующей обработки данных.

`jinfo` — выдает информацию о системных свойствах JVM и позволяет задавать некоторые системные свойства динамически. Подходит для использования в сценариях.

`jstack` — выводит содержимое стеков процесса Java. Подходит для использования в сценариях.

`jstat` — предоставляет информацию об уборке мусора и загрузке классов. Подходит для использования в сценариях.

`jvisualvm` — программа с графическим интерфейсом для мониторинга JVM, профилирования выполняемых приложений и анализа дампов кучи JVM (что обычно делается в фазе последующей обработки, хотя `jvisualvm` также может получать дампы кучи от работающих программ).

Все эти программы легко запускаются на одной машине с JVM. Если JVM работает внутри контейнера Docker, то неграфические средства (то есть все, кроме `jconsole` и `jvisualvm`) можно запустить командой `docker exec` или же воспользоваться командой `nsenter` для входа в контейнер Docker. Однако оба варианта предполагают, что эти программы установлены в образе Docker, что определенно рекомендуется сделать. Образы Docker обычно урезаются до минимальных потребностей вашего приложения, поэтому в них включается только JRE, но рано или поздно в ходе разработки вам понадобится информация об этом приложении, поэтому лучше иметь необходимые инструменты (включенные в поставку JDK) наготове в образе Docker.

`jconsole` потребляет значительное количество системных ресурсов, поэтому запуск программы в рабочей системе может помешать ее работе. `jconsole` можно настроить так, чтобы программа запускалась локально и подключалась к удаленной системе; в этом случае она не повредит производительности удаленной системы. В рабочей среде для этого нужно будет установить сертификаты, позволяющие `jconsole` работать по SSL, и настроить систему безопасной аутентификации.

Все эти программы делятся на следующие широкие категории:

- Основная информация VM.
- Информация о потоках.
- Информация о классах.
- Оперативный анализ уборки мусора.
- Последующая обработка дампа кучи.
- Профилирование JVM.

Вероятно, вы заметили, что программы не относятся к одной конкретной категории — многие из них выполняют функции, относящиеся к разным категориям. Поэтому вместо того, чтобы исследовать каждую программу по отдельности, мы рассмотрим функциональные области видимости, важные для Java, и обсудим различные инструменты, предоставляющие эту информацию. Попутно будут рассмотрены другие программы (одни с открытым кодом, другие коммерческие), предоставляющие ту же базовую функциональность, но в чем-то превосходящие базовые средства JDK.

Основная информация VM

Средства этой категории предоставляют основную информацию о работающем процессе JVM: сколько времени работает, какие флаги JVM используются, значения системных свойств JVM и т. д.

Время работы — для определения промежутка времени, в течение которого работает JVM, можно воспользоваться следующей командой:

```
% jcmd идентификатор_процесса VM.uptime
```

Системные свойства — для вывода элементов `System.getProperties()` можно воспользоваться любой из следующих команд:

```
% jcmd идентификатор_процесса VM.system_properties
```

или

```
% jinfo -sysprops process_id
```

К их числу относятся все свойства, заданные в командной строке с ключом `-D`; все свойства, динамически добавленные приложением; и набор свойств по умолчанию для JVM.

Версия JVM — для получения версии JVM можно воспользоваться следующей командой:

```
% jcmd идентификатор_процесса VM.version
```

Командная строка JVM — командная строка выводится на вкладке сводной информации VM программы `jconsole` или при помощи программы `jcmd`:

```
% jcmd идентификатор_процесса VM.command_line
```

Флаги оптимизации JVM — флаги оптимизации JVM, действующие в приложении, выводятся следующей командой:

```
% jcmd идентификатор_процесса VM.flags [-all]
```

Работа с флагами оптимизации

JVM могут передаваться многочисленные флаги оптимизации, причем многие из них занимают центральное место в этой книге. Попытки запомнить все эти флаги и их значения по умолчанию могут быть довольно непростым делом; последние два примера `jcmd` помогут вам в этом. Команда `command_line` сообщает, какие флаги были заданы в командной строке, а также некоторые флаги, которые

устанавливаются непосредственно JVM (потому что их значение определялось эргономически). При включении параметра `-all` команда выводит все флаги в JVM.

Существуют сотни флагов оптимизации JVM, причем смысл многих неочевиден; как правило, их лучше никогда не изменять (см. врезку «Слишком много информации?» на с. 89). Задача определения флагов, действующих в настоящий момент, часто встречается при диагностике проблем производительности; команды `jcmd` позволяют получить эту информацию для работающей JVM. Часто вы предпочли бы узнать значения по умолчанию для конкретной платформы; в этом случае лучше воспользоваться параметром `-XX:+PrintFlagsFinal` командной строки. Проще всего выполнить следующую команду:

```
% java other_options -XX:+PrintFlagsFinal -version
...Сотни строк вывода, включая...
uintx InitialHeapSize                          := 4169431040     {product}
intx InlineSmallCode                           = 2000            {pd product}
```

Все остальные параметры, которые вы намереваетесь использовать, следует включить в командную строку, потому что определение некоторых параметров (особенно для флагов, относящихся к уборке мусора) может повлиять на итоговое значение других параметров. Команда выведет полный список флагов JVM и их значений (тот же список, который выводится с параметром `VM.flags -all` программы `jcmd` для действующей JVM).

Данные флагов в результатах этих команд выводятся одним из двух следующих способов. Двоеточие в первой строке приведенного вывода означает, что для флага используется значение, отличное от значения по умолчанию. Это может произойти по следующим причинам:

- Значение флага было указано непосредственно в командной строке.
- Другой параметр косвенно изменил значение этого параметра.
- JVM вычисляет значение по умолчанию эргономически.

Вторая строка (без двоеточия) показывает, что значение является значением по умолчанию для этой версии JVM. Значения по умолчанию для некоторых флагов могут различаться на разных платформах, что показано в последнем столбце вывода. Значение `product` означает, что значение флага по умолчанию является общим для всех платформ; `pd product` — что значение флага по умолчанию зависит от платформы.

Другие возможные значения последнего столбца — `manageable` (значение флага может изменяться динамически во время выполнения) и `C2 diagnostic` (флаг предоставляет диагностическую информацию для разработчиков компиляторов для понимания того, как функционирует компилятор).

Также для того, чтобы просмотреть эту информацию для работающего приложения, можно воспользоваться программой `jinfo`. Главное преимущество `jinfo` — возможность изменения значений некоторых флагов во время выполнения программы.

Получить значения всех флагов в процессе можно так:

```
% jinfo -flags идентификатор_процесса
```

С параметром `-flags` программа `jinfo` предоставляет информацию обо всех флагах; в противном случае выводятся только флаги, заданные в командной строке. Вывод этих команд читается не так легко, как с параметром `-XX:+PrintFlagsFinal`, но у `jinfo` имеются другие возможности, о которых следует помнить.

`jinfo` может проверить значение отдельного флага:

```
% jinfo -flag PrintGCDetails идентификатор_процесса
-XX:+PrintGCDetails
```

Хотя `jinfo` не указывает, является ли флаг управляемым (`manageable`), управляемые флаги (которые можно узнать при использовании фрагмента `PrintFlagsFinal`) можно включать и отключать при помощи `jinfo`:

```
% jinfo -flag -PrintGCDetails process_id  # Отключение PrintGCDetails
% jinfo -flag PrintGCDetails process_id
-XX:-PrintGCDetails
```

СЛИШКОМ МНОГО ИНФОРМАЦИИ?

Команда `PrintFlagsFinal` выводит сотни доступных флагов оптимизации для JVM (например, в JDK 8u202 поддерживаются 729 возможных флагов). Абсолютное большинство этих флагов предназначено для специалистов по поддержке для получения дополнительной информации от работающих (некорректно) приложений.

Когда вы узнаете о существовании флага `AllocatePrefetchLines` (со значением по умолчанию 3), появляется соблазн изменить это значение, чтобы упреждающая выборка команд лучше работала на конкретном процессоре. Но такая «оптимизация наугад» не оправдывается; никакие флаги не следует изменять без убедительных причин. В случае с флагом `AllocatePrefetchLines` для этого необходимо знать производительность упреждающей выборки приложения, характеристики процессора, на котором выполняется приложение, и эффект от изменения этого числа для кода JVM.

Учтите, что в JDK 8 `jinfo` может изменить значение любого флага, но это не означает, что JVM отреагирует на это изменение. Например, многие флаги, влияющие на поведение алгоритма уборки мусора, используются в момент запуска для определения различных параметров поведения уборщика. Изменение флага с помощью `jinfo` в более поздний момент не заставит JVM изменить ее поведение; виртуальная машина продолжит работать на основании тех параметров, с которыми алгоритм был инициализирован. Таким образом, этот способ работает только для флагов с пометкой `manageable` в выходных данных команды `PrintFlagsFinal`. В JDK 11 программа `jinfo` при попытке изменить значение флага, который невозможно изменить, выдаст сообщение об ошибке.

РЕЗЮМЕ

- Программа jcmd может использоваться для получения основной информации JVM, включая значения всех флагов оптимизации, для выполняемого приложения.
- Чтобы узнать значения флагов по умолчанию, включите параметр `-XX:+PrintFlagsFinal` в командную строку. Например, это может быть полезно для определения эргономических настроек флагов для конкретной платформы.
- Программа jinfo используется для просмотра (а в некоторых случаях и изменения) отдельных флагов.

Информация о потоках

`jconsole` и `jvisualvm` выводят информацию (в реальном времени) о количестве потоков, выполняемых в приложении. Просматривая стек выполняемых потоков, разработчик может определить, не заблокированы ли они. Информацию стеков можно получить программой `jstack`:

```
% jstack идентификатор_процесса
... Длинный вывод со стеками всех потоков ...
```

Информацию стеков также можно получить при помощи программы `jcmd`:

```
% jcmd идентификатор_процесса Thread.print
... Длинный вывод со стеками всех потоков ...
```

За дополнительной информацией о мониторинге стеков потоков обращайтесь к главе 9.

Информация о классах

Информация о количестве классов, используемых приложением, может быть получена при помощи программы `jconsole` или `jstat`. `jstat` также может предоставить информацию о компиляции классов.

За дополнительной информацией об использовании классов приложениями обращайтесь к главе 12. Получение информации о компиляции классов более подробно рассматривается в главе 4.

Оперативный анализ уборки мусора

Практически каждая программа мониторинга выводит хоть какую-то информацию об уборке мусора. `jconsole` выводит активный график использования кучи; `jcmd` позволяет выполнять операции уборки мусора; `jmap` выводит сводки использования кучи или информацию о неизменном (permanent) поколении либо создает дамп кучи; `jstat` предоставляет разнообразные представления текущей активности уборщика мусора. Примеры отслеживания процесса уборки мусора при помощи этих программ приведены в главе 5.

Последующая обработка дампов кучи

Дампы кучи могут сохраняться из графического интерфейса `jvisualvm` или из командной строки при помощи программ `jcmd` или `jmap`. *Дамп кучи* представляет собой мгновенный снимок кучи, который можно анализировать различными программами, включая `jvisualvm`. Обработка дампов кучи — одна из областей, в которых сторонние программы традиционно опережали возможности средств JDK, поэтому в примерах последующей обработки дампов кучи в главе 7 будет использоваться сторонняя программа — Eclipse Memory Analyzer Tool (`mat`).

Средства профилирования

Профилировщики — самое важное средство в инструментарии специалиста по анализу производительности. Для Java есть много профилировщиков, у каждого из которых имеются свои достоинства и недостатки. Профилирование относится к числу тех областей, в которых часто стоит использовать разные инструменты — особенно если речь идет о профилировщике с выборкой (sampling profiler). Профилировщики с выборкой обычно по-разному представляют информацию, так что один может лучше выявлять проблемы в одних приложениях и хуже в других.

Многие стандартные средства профилирования Java сами написаны на Java и работают за счет их «присоединения» к профилируемому приложению. Присоединение осуществляется через сокет или специальный интерфейс Java — JVM Tool Interface (JVMTI). Далее целевое приложение и программа профилирования обмениваются информацией о поведении целевого приложения.

Это означает, что оптимизации профилировщика необходимо уделить такое же внимание, как и настройке любого другого приложения Java. В частности, если профилируемое приложение велико, оно может передавать большие объемы данных профилировщику, а следовательно, профилировщик должен располагать кучей достаточно большого объема для обработки данных. Также часто бывает полезно запускать программу профилирования с параллельным алгоритмом уборки мусора; несвоевременные приостановки полной уборки мусора могут привести к переполнению буферов, содержащих данные.

Профилировщики с выборкой

Профилирование может происходить в одном из двух режимов: *режиме выборки* (sampling) или в *инструментальном режиме* (instrumented mode). Режим выборки — основной режим профилирования — сопряжен с минимальными затратами ресурсов. Это важно, поскольку одна из потенциальных проблем профилирования заключается в том, что включение измерений в приложение изменяет его характеристики производительности[1].

Ограничение воздействия профилирования приводит к результатам, которые более точно моделируют поведение приложения в обычных обстоятельствах.

К сожалению, при работе профилировщика с выборкой могут возникать самые разнообразные ошибки. Профилировщики с выборкой работают по периодическому срабатыванию таймера; профилировщик проверяет каждый поток и определяет, какой метод выполняется потоком. Предполагается, что этот метод выполняется с момента предыдущего срабатывания таймера.

Самая распространенная ошибка выборки изображена на рис. 3.1. Поток переключается между выполнением методов `methodA` (обозначается закрашенными прямоугольниками) и `methodB` (обозначается пустыми прямоугольниками). Если таймер срабатывает только при выполнении потоком `methodB`, в профиле будет указано, что поток провел все свое время за выполнением `methodB`, тогда как в действительности больше времени проводится в `methodA`.

Это самая распространенная, но далеко не единственная ошибка выборки. Чтобы свести эту ошибку к минимуму, следует проводить профилирование в течение

[1] Однако профилирование все равно необходимо: иначе как вы узнаете, жив ли кот внутри вашей программы?

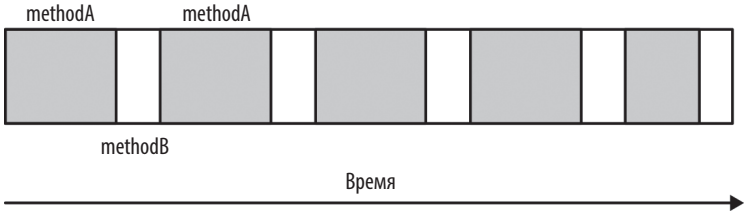


Рис. 3.1. Чередование выполняемых методов

более длительного времени и сокращать временные промежутки между точками выборки. Сокращение интервалов между точками выборки вступает в конфликт с другой целью — минимизацией эффекта профилирования на приложение; здесь необходимо найти баланс между двумя целями. Средства профилирования поразному решают проблему баланса, и это одна из причин, из-за которых две разные программы профилирования выдают заметно различающиеся данные.

Такая ошибка присуща всем профилировщикам с выборкой, но она усугубляется во многих программах Java (особенно старых). Это связано со *смещением безопасного состояния*. В стандартном интерфейсе Java для профилирования программа-профилировщик может получить трассировку стека потока только в тот момент, когда поток находится в безопасном состоянии. Потоки автоматически оказываются в безопасном состоянии, когда они:

- блокируются по синхронизируемой блокировке;
- блокируются в ожидании ввода/вывода;
- блокируются в ожидании монитора;
- находятся в запаркованном состоянии;
- выполняют код JNI (Java Native Interface) (если только они не выполняют функцию, блокирующую функцию уборки мусора).

Кроме того, JVM может установить флаг, приказывающий потокам перейти в безопасное состояние. Код проверки этого флага (и перехода в безопасное состояние при необходимости) вставляется в код JVM в ключевых позициях, включая некоторые операции выделения памяти, а также переходы в циклы или методы в компилируемом коде. Ни в одной спецификации не указано, где происходят проверки безопасного состояния, и местоположение этих проверок изменяется между версиями.

Последствия от смещения безопасного состояния на профилировщики с выборкой могут быть весьма значительными: так как стек проверяется только при нахождении потока в безопасном состоянии, выборка становится еще менее надежной. На рис. 3.1 маловероятно, чтобы в случайном профилировщике без смещения безопасного состояния проверка потоков осуществлялась только

в ходе выполнения `methodB`. Но со смещением безопасного состояния проще представить себе сценарий, при котором `methodA` никогда не оказывается в безопасном состоянии, поэтому вся работа приписывается `methodB`.

Java 8 предоставляет другой способ сбора трассировок стека (это одна из причин, по которым в старых программах присутствует смещение безопасного состояния, а в новых программах его обычно нет, хотя для этого новая программа должна быть переписана для использования нового механизма). На программном уровне это означает использование интерфейса `AsyncGetCallTrace`. Профилировщики, использующие этот интерфейс, часто называются *асинхронными профилировщиками*. «Асинхронность» относится к способу предоставления информации стека виртуальной машиной Java, а не к тому, как работает программа профилирования; профилировщик называется асинхронным, потому что JVM может предоставить стек в любой момент времени, не дожидаясь, пока поток окажется в (синхронном) безопасном состоянии.

В профилировщиках, использующих асинхронный интерфейс, артефакты выборки встречаются реже, чем у других профилировщиков с выборкой (хотя они все еще подвержены таким ошибкам, как на рис. 3.1). Асинхронный интерфейс был официально опубликован в Java 8, хотя существовал в виде приватного интерфейса задолго до появления этой версии.

На рис. 3.2 изображен простейший профиль, который был получен при измерении производительности REST-сервера, предоставляющего данные биржевых котировок в приложении из главы 2. REST-вызов настроен для возвращения потока байтов, содержащего сжатую сериализованную форму объекта акции (часть примера, который будет рассмотрен в главе 12). Мы воспользуемся этой программой в качестве основы для примеров, приведенных в этом разделе.

Снимок экрана был сделан в профилировщике Oracle Developer Studio. Эта программа использует асинхронный интерфейс профилирования, хотя обычно она не называется «асинхронным профилировщиком» (вероятно, по историческим

Total CPU Time		Name
EXCLUSIVE sec.	INCLUSIVE sec.	
781.206	781.206	*<Total>*
105.013	341.959	java.io.ObjectOutputStream.writeObject0(java.lang.Object, boolean)
39.047	98.979	sun.reflect.GeneratedMethodAccessor11.invoke(java.lang.Object, java.lang.Object[])
34.694	34.884	java.io.ObjectOutputStream$HandleTable.growSpine()
34.634	46.262	java.math.BigInteger.pow(int)
33.744	33.744	InstanceKlass::oop_push_contents(PSPromotionManager*, oopDesc*)
32.763	32.783	java.io.ObjectStreamClass.lookup(java.lang.Class, boolean)
30.201	30.221	java.math.MutableBigInteger.divideMagnitude(java.math.MutableBigInteger, java.math.MutableBigInteger, boolean)
29.671	341.409	java.io.ObjectOutputStream.defaultWriteFields(java.lang.Object, java.io.ObjectStreamClass)
25.308	61.883	PSPromotionManager::copy_to_survivor_space<false>(oopDesc*)
19.093	19.093	sun.misc.FloatingDecimal.readJavaFormatString(java.lang.String)
18.663	18.663	sun.misc.FDBigInteger.<init>(long, char[], int, int)
18.613	24.497	java.io.ObjectOutputStream$BlockDataOutputStream.setBlockDataMode(boolean)
15.801	63.655	java.math.BigInteger.toString(int)
15.311	26.308	ObjectSynchronizer::FastHashCode(Thread*, oopDesc*)
15.251	59.662	sun.misc.FloatingDecimal$ASCIIToBinaryBuffer.doubleValue()
15.211	341.449	java.io.ObjectOutputStream.writeSerialData(java.lang.Object, java.io.ObjectStreamClass)
14.920	14.920	jbyte_disjoint_arraycopy
14.460	21.265	java.io.ObjectOutputStream$BlockDataOutputStream.write(byte[], int, int, boolean)
12.899	83.228	java.math.BigDecimal.<init>(double, java.math.MathContext)
12.559	18.143	sun.misc.FDBigInteger.valueOfMulPow52(long, int, int)
11.198	24.487	java.math.BigInteger.pow(int)

Рис. 3.2. Профилирование с выборкой

причинам, так как она начала использовать этот интерфейс, когда он еще был приватным и популярный термин еще не появился). Программа поддерживает различные представления данных; в выбранном представлении показаны методы, потребляющие наибольшее количество процессорного времени.

Некоторые из этих методов относятся к сериализации объектов (например, метод `ObjectOutputStream.writeObject0()`), а многие относятся к обработке данных (например, `Math.pow()`)[1]. И все же сериализация объектов преобладает в этом профиле; чтобы улучшить производительность, необходимо повысить производительность сериализации.

Обратите внимание на последнее предложение: повысить необходимо производительность *сериализации*, а не самого метода `writeObject0()`. Обычно при просмотре профиля предполагается, что улучшения должны быть достигнуты за счет оптимизации верхнего метода в профиле. Такой подход слишком часто сталкивается с ограничениями. В данном случае метод `writeObject0()` является частью JDK; его производительность невозможно улучшить без переписывания JVM. Но из профиля известно, что узкое место производительности скрывается в процессе сериализации.

Следовательно, верхний метод(-ы) в профиле должен указать на область, в которой следует начать поиски оптимизаций. Специалист по производительности не станет пытаться ускорить методы JVM, но он постарается ускорить процесс сериализации объектов в целом.

Результат профилирования можно представить в двух дополнительных вариантах: в обоих случаях используется наглядное представление стека вызовов. Более новый метод называется *огненным графиком* — интерактивной диаграммой стека вызовов в приложении.

На рис. 3.3 изображена часть огненного графика, полученного с использованием асинхронного профилировщика с открытым кодом. Огненный график представляет собой перевернутую диаграмму методов, использующих наибольшую часть процессорного времени. В этой части графика метод `getStockObject()` занимает все время. Приблизительно 60% этого времени проходит в вызове `writeObject()`, а 40% времени — в конструкторе объекта `StockPriceHistoryImpl`. Также мы можем проверить стеки каждого из этих методов и выявить узкие места в производительности. Сам график интерактивен, по щелчку на линии можно просмотреть информацию о методе — включая полное имя, процессорное время и т. д.

Более старый (хотя и по-прежнему полезный) способ наглядного представления производительности основан на использовании нисходящей структуры, называемой *деревом вызовов*. Пример показан на рис. 3.4.

---

[1] Также встречаются ссылки на внутренний код C++ — например, `InstanceKlass::oop_push_contents`; об этом подробнее рассказано в следующем разделе.

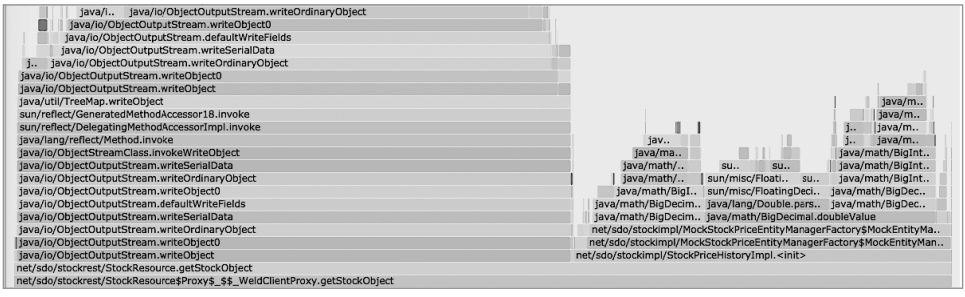


Рис. 3.3. Огненный график из профилировщика с выборкой

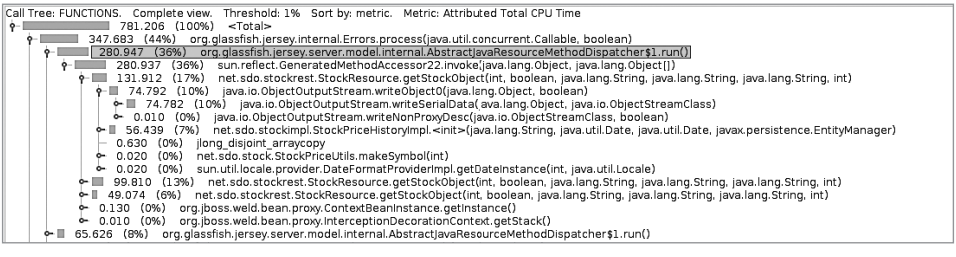


Рис. 3.4. Дерево вызовов из профилировщика с выборкой

В этом случае начальные позиции занимают сходные данные: из 100% времени 44% занимал метод `Errors.process()` и его потомки. Далее мы заходим в родительский метод и смотрим, где его потомки проводят время. Например, из 17% общего времени, проводимого в методе `getStockObject()`, 10% этого времени проводилось в `writeObject0`, а 7% — в конструкторе.

РЕЗЮМЕ

- Профилировщики с выборкой составляют самую распространенную разновидность профилировщиков.
- Из-за своего относительно низкого воздействия на производительность профилировщики с выборкой создают меньше артефактов измерений.
- Профилировщики с выборкой, которые могут использовать асинхронное чтение стеков, создают меньше артефактов измерений.
- Разные профилировщики с выборкой ведут себя по-разному; каждый профилировщик может лучше подходить для определенной категории приложений.

Инструментальные профилировщики

Инструментальные профилировщики намного сильнее воздействуют на систему, чем профилировщики с выборкой, но они также могут предоставить больше полезной информации о происходящем в программе.

Работа инструментальных профилировщиков основана на изменении байт-кода загружаемых классов (с вставкой кода для подсчета вызовов и т. д.). Они с гораздо большей вероятностью повлияют на производительность приложений, чем профилировщики с выборкой. Например, JVM использует встраивание небольших методов (см. главу 4), чтобы при выполнении кода малых методов никакие вызовы не совершались. Компилятор принимает это решение в зависимости от размера кода; в зависимости от того, как организован код, он может не подойти для встраивания. Это может привести к тому, что инструментальный профилировщик переоценит вклад некоторых методов в затраты времени; как правило, чем выше уровень инструментовки (изменения) кода, тем с большей вероятностью изменится его профиль выполнения.

Из-за изменений, вносимых в код посредством инструментовки, лучше ограничить ее использование отдельными классами. Это означает, что инструментовку лучше применять для анализа второго уровня: профилировщик с выборкой может указать на пакет или часть кода как на причину потерь производительности, после чего инструментальный профилировщик при необходимости используется для углубленного анализа этого кода.

На рис. 3.5 инструментальный профилировщик (не использующий асинхронные интерфейсы) анализирует REST-сервер из нашего примера.

Hot Spot	Self Time	Average Time	Invocations
java.io.ObjectOutputStream.writeObject	178 s (24 %)	29.328 us	6.089
net.sdo.stockimpl.MockStockPriceEntityManagerFactory$...	135 s (18 %)	4 us	33.367.720
java.math.BigDecimal.doubleValue	84.240 ms (11 %)	1 us	47.664.692
java.math.BigDecimal.<init>	56.846 ms (7 %)	0 us	71.503.127
net.sdo.stockimpl.StockPriceHistoryImpl.<init>	45.005 ms (6 %)	7.391 us	6.089
java.util.Random.nextDouble	29.423 ms (4 %)	0 us	166.832.511
net.sdo.stockimpl.MockStockPriceEntityManagerFactory$...	16.725 ms (2 %)	0 us	33.367.720
java.util.Calendar.get	11.849 ms (1 %)	0 us	61.967.753
javax.persistence.EntityManager.find	11.504 ms (1 %)	0 us	33.367.720

Рис. 3.5. Результат работы инструментального профилировщика

В результатах этого профилировщика присутствует ряд отличий. Во-первых, доминирующее время приписано методу `writeObject()`, а не методу `writeObject0()`. Дело в том, что приватные методы исключаются из инструментовки. Во-вторых, появился новый метод из диспетчера сущностей; он не отображался ранее, потому что в случае профилировщика с выборкой он был встроен в конструктор.

Но в этом виде профилирования более важную роль играет счетчик вызовов: метод диспетчера сущностей был вызван 33 миллиона раз, а количество вызовов генерирования случайного числа составило невероятные 166 миллионов раз. Намного большего выигрыша по быстродействию можно добиться за счет сокращения общего количества вызовов этих методов (вместо ускорения их реализаций), но мы бы не получили эти данные без инструментального профилирования.

Можно ли сказать, что этот результат профилирования лучше, чем у версии с выборкой? На этот вопрос трудно ответить однозначно; в конкретной ситуации даже невозможно определить, какой профиль более точен. Счетчик вызовов инструментального профилировщика безусловно точен, и эта дополнительная информация часто помогает определить, где код проводит больше времени и для каких операций оптимизация будет более эффективной.

В данном примере как инструментальные профилировщики, так и профилировщики с выборкой указывают на одну общую область кода: сериализацию объектов. На практике может оказаться, что разные профилировщики указывают на совершенно разные области кода. Профилировщики — хорошие средства оценки, но их результаты следует рассматривать именно как оценки: некоторые из них в отдельных случаях будут ошибочными.

СОВЕТ

- Инструментальные профилировщики предоставляют больше информации о приложении, но могут сильнее влиять на работу приложения, чем профилировщики с выборкой.
- Инструментальные профилировщики следует настроить на инструментовку небольших частей кода — отдельных классов или пакетов. Тем самым ограничивается их влияние на производительность приложения.

Блокирующие методы и временная шкала потоков

На рис. 3.6 представлены данные по REST-серверу при использовании другого инструментального профилировщика: это профилировщик, встроенный в `jvisualvm`. Теперь во времени выполнения доминируют методы `select()` (и в меньшей степени методы `run()` обработчика подключений `TCPTransport`).

Эти методы (и другие блокирующие методы) не потребляют процессорное время, поэтому они не вносят свой вклад в общий уровень использования процессора приложением. Их выполнение оптимизировать заведомо не удастся.

Потоки в приложении не проводят 673 секунды за выполнением кода в методе `select()`; они проводят 673 секунды за ожиданием события выбора.

Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
io.nettv.channel.nio.SelectedSelectionKeySetSelector.**select** ...		673,322 ms (44.7%)	673,346 ms	26,061
net.sdo.stockrest.StockResource.**getStockObject** (int, boole...		371,222 ms (24.7%)	575,867 ms	7,586
net.sdo.stockimpl.MockStockPriceEntityManagerFactory$MockEr		182,615 ms (12.1%)	184,148 ms	41,571,235
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.**run** ()		158,539 ms (10.5%)	158,539 ms	7
java.util.concurrent.ThreadPoolExecutor$Worker.**run** ()		60,000 ms (4%)	108,723 ms	4
net.sdo.stockimpl.StockPriceHistoryImpl.**<init>** (String, java.ut...		11,603 ms (0.8%)	204,321 ms	7,586
net.sdo.stockimpl.MockStockPriceEntityManagerFactory$MockEr		5,058 ms (0.3%)	190,745 ms	41,571,259
io.nettv.channel.socket.nio.NioSocketChannel.**doWrite** (io.ne...		3,471 ms (0.2%)	4,351 ms	15,172

Рис. 3.6. Профиль с блокирующими методами

По этой причине многие профилировщики не выдают информацию о блокируемых методах; эти потоки отображаются как бездействующие. В нашем конкретном примере это хорошо. Потоки ожидают в методе `select()`, потому что никакие данные серверу не передаются; неэффективности выполнения нет. Это их нормальное состояние.

В других случаях бывает нужно узнать время, проведенное в этих блокирующих вызовах. Время, проведенное потоком внутри метода `wait()`, — ожидание уведомления со стороны другого потока — является важным фактором, определяющим общее время выполнения многих приложений. Многие профилировщики на базе Java поддерживают фильтры и другие параметры, которые можно настроить так, чтобы эти блокирующие вызовы отображались или скрывались в результатах.

Кроме того, обычно более эффективно анализировать паттерны выполнения потоков вместо времени, которое профилировщик приписывает самому блокирующему методу. На рис. 3.7 показана временная шкала потоков из профилировщика Oracle Developer Studio.

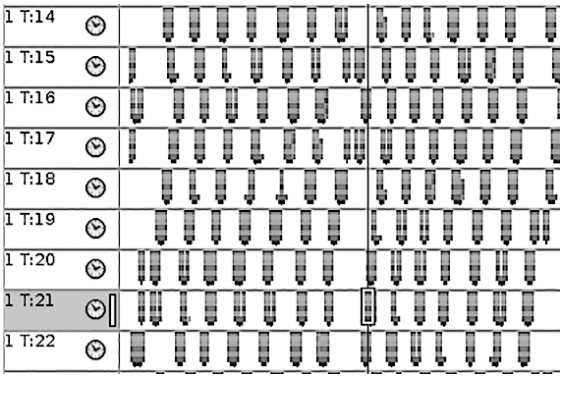


Рис. 3.7. Временная шкала потоков

Каждая горизонтальная область представляет отдельный поток (так что на диаграмме представлены девять потоков: от 1.14 до 1.22). Цветные (или изображенные в разных оттенках серого) прямоугольники представляют выполнение разных методов; пустые области представляют промежутки, в которых поток не выполнялся. На высоком уровне можно заметить, что поток 1.14 выполнил код, а потом чего-то ожидал.

Также обратите внимание на пустые области, в которых вроде бы не выполняются никакие потоки. На диаграмме изображены только девять из многих потоков приложения, поэтому нельзя исключать, что эти потоки ждут, пока один из других потоков что-то сделает, или что потоки выполняют `read()` (или другой блокирующий вызов).

РЕЗЮМЕ

- Заблокированные потоки не всегда являются источником проблем с производительностью; необходимо выяснить причину, по которой они блокируются.
- Заблокированные потоки можно идентифицировать по блокирующему методу или посредством анализа потока по временной шкале.

Профилировщики низкоуровневого кода

Такие средства, как асинхронные профилировщики и Oracle Developer Studio, предоставляют возможность профилирования низкоуровневого (native) кода (помимо кода Java). Данная возможность обладает двумя преимуществами.

Во-первых, в низкоуровневом коде может выполняться серьезная работа, включая операции с платформенными библиотеками и операции выделения памяти. В главе 8 мы воспользуемся профилировщиком внутреннего кода для анализа примера выделения памяти, создавшего проблему в реальной ситуации. Использование профилировщика внутреннего кода для отслеживания использования памяти позволило быстро выявить истинную причину происходящего.

Во-вторых, профилирование обычно проводится для поиска узких мест в коде приложения, но иногда низкоуровневый код неожиданно занимает доминирующее положение в затратах производительности. Мы бы предпочли узнать о том, что наш код проводит слишком много времени за уборкой мусора, анализируя журналы GC (как это будет сделано в главе 6), но если забыть этот путь, профилировщик, поддерживающий низкоуровневый код, быстро покажет, что приложение проводит слишком много времени за уборкой мусора. Аналогичным образом профилирование обычно желательно ограничивать периодом после разогрева программы, но если потоки компиляции (глава 4) работают и от-

нимают слишком много ресурсов процессора, профилировщик с поддержкой низкоуровневого кода сообщит вам об этом.

При просмотре огненного графика для нашего REST-сервера я привел только малую часть для удобочитаемости. На рис. 3.8 показан весь график.

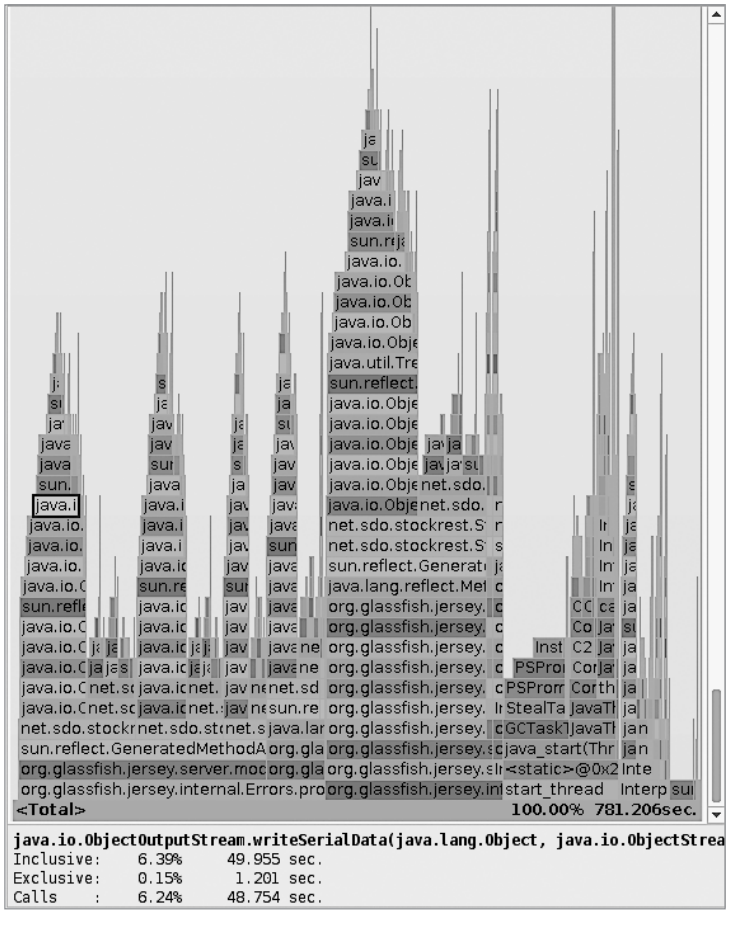


Рис. 3.8. Огненный график с включением низкоуровневого кода

В нижней части графика отображаются пять компонентов. Первые два (из кода JAX-RS) — потоки приложения и код Java. Однако третий представляет уборку мусора для процесса, а четвертый — компилятор[1].

[1] Напомню, что этот конкретный график был получен в Oracle Developer Studio, хотя асинхронный профилировщик выдает идентичный набор платформенных вызовов.

РЕЗЮМЕ

- Профилировщики низкоуровневого кода выдают информацию как по коду JVM, так и по коду приложения.
- Если профилировщик низкоуровневого кода показывает, что время уборки мусора доминирует в загрузке процессора, следует заняться настройкой уборщика мусора. Но если он показывает, что значительное время проводится в потоках компиляции, это обычно не влияет на производительность приложения.

Java Flight Recorder

Java Flight Recorder (JFR) — механизм JVM для выполнения облегченного анализа производительности выполняемых приложений. Данные JFR представляют историю событий в JVM, по которой можно диагностировать бывшую производительность и операции JVM.

Механизм JFR изначально был функцией JRockit JVM от BEA Systems. Со временем он был перенесен в Oracle HotSpot JVM; в JDK 8 только Oracle JVM поддерживает механизм JFR (и он лицензирован для использования только клиентами Oracle). Однако в JDK 11 механизм JFR стал доступен в JVM с открытым кодом, включая AdoptOpenJDK JVM. Так как в JDK 11 JFR распространяется с открытым кодом, имеется возможность обратного портирования в форме открытого кода для JDK 8, так что, возможно, AdoptOpenJDK и другие версии JDK 8 когда-нибудь будут включать JFR (хотя по крайней мере до 8u232 этого не произошло).

Основной принцип JFR заключается в том, что активизируется некоторый набор событий (например, одно из таких событий — блокирование потока в ожидании получения ресурса блокировки). Далее каждый раз при возникновении такого события данные об этом событии сохраняются (в памяти или в файле). Поток данных хранится в циклическом буфере, поэтому доступны только самые последние события. Затем данные таких событий — взятые от работающей JVM или прочитанные из сохраненного файла — выводятся соответствующей программой для проведения анализа или диагностики проблем с производительностью.

Всеми аспектами — типами событий, размером циклического буфера, местом хранения и т. д. — можно управлять при помощи различных аргументов JVM или специальных программ, включая команды `jcmd` при запуске программы. По умолчанию JFR настраивается на очень низкое потребление ресурсов: воздействие JFR составляет менее 1% от производительности программы. Это воздействие будет изменяться с включением новых событий, с изменением порога, при котором выдается информация о событиях, и т. д. Подробности настройки рассматриваются позднее в этом разделе, но сначала посмотрим, как выглядят выходные данные этих событий — это поможет понять, как работает JFR.

Java Mission Control

Для анализа сохраненных данных JFR чаще всего используется программа `jmc` (Java Mission Control), хотя существуют и другие средства; более того, вы даже можете воспользоваться инструментариями для написания собственных средств анализа. С переходом на JVM с полностью открытым кодом программа `jmc` вышла из кодовой базы OpenJDK и преобразовалась в отдельный проект. Это позволяет `jmc` развиваться по отдельному графику и пути выпуска, хотя на первый взгляд необходимость иметь дело с разными версиями усложняет дело.

JDK 8 `jmc` версии 5 входит в комплект Oracle JVM (единственной JVM, поддерживающей JFR). JDK 11 может использовать `jmc` версии 7, хотя в настоящее время двоичные файлы приходится загружать со страницы проекта OpenJDK. Предполагается, что со временем произойдет поглощение и сборки JDK будут включать соответствующие двоичные файлы `jmc`.

Программа `jmc` (Java Mission Control) открывает окно со списком процессов JVM на машине и предлагает выбрать один или несколько процессов для мониторинга. На рис. 3.9 изображена консоль JMX (Java Management Extensions) программы `jmc` с REST-сервером из нашего примера.

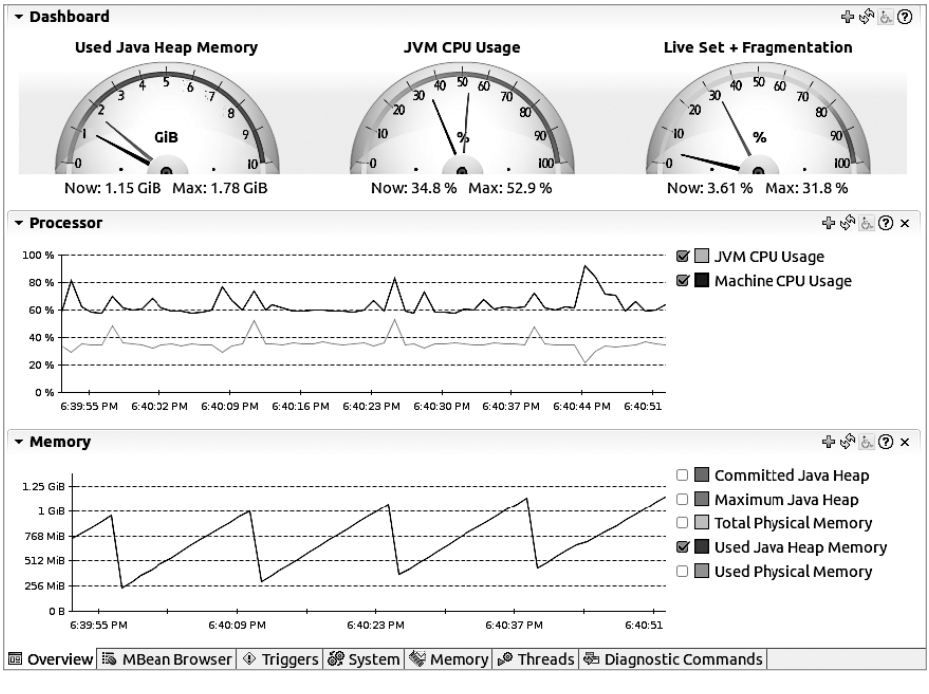


Рис. 3.9. Отображение данных в программе jmc

На этой панели отображается основная информация, отслеживаемая в `jmc`: уровень использования процессора, уровень использования кучи и время уборки мусора. Однако следует заметить, что график использования процессора включен в общие затраты процессорного времени на машине. Сама JVM использует около 38% процессора, хотя все процессы на машине потребляют около 60%. В этом заключается одна из ключевых особенностей мониторинга: через консоль JMX `jmc` может отслеживать всю систему, а не только JVM. Верхнюю часть панели можно настроить для вывода информации JVM (всевозможная статистика уборки мусора, загрузка классов, использование потоков, использование кучи и т. д.), а также специфической информации ОС (общий уровень использования процессоров и памяти, подкачка, средняя нагрузка и т. д.).

Как и другие средства мониторинга, `jmc` может обращаться с вызовами JMX к управляемым компонентам, предоставляемым приложением.

Краткий обзор JFR

Располагая необходимыми инструментами, можно разобраться в том, как работает JFR. В этом примере используются данные JFR, полученные для REST-сервера за 6-минутный период. При загрузке данных в `jmc` прежде всего выводится сводка основной информации мониторинга (рис. 3.10).

Эта информация сходна с той, которая отображается `jmc` при выводе основной информации. Над датчиками использования процессора и кучи располагается временная шкала событий (изображенная в виде ряда вертикальных полос). Временная шкала позволяет увеличить конкретную область, представляющую интерес; хотя в приведенном примере данные сохранялись за 6-минутный период, я сосредоточился на 38-секундном интервале ближе к концу записи.

График использования процессора более четко показывает суть происходящего: REST-сервер находится в нижней части графика (в среднем 20% использования), а машина работает на 38% использования процессора. В нижней части расположены корешки других вкладок с информацией о системных свойствах и параметрах сохранения данных JFR. Значки, расположенные по вертикали в левой части окна, более интересны: они дают представление о поведении приложения.

Представление Memory

Это окно содержит обширную информацию. На рис. 3.11 представлена только одна панель представления Memory.

График показывает, что потребление памяти довольно регулярно колеблется с очисткой молодого поколения (и мы увидим, что общий размер кучи растет

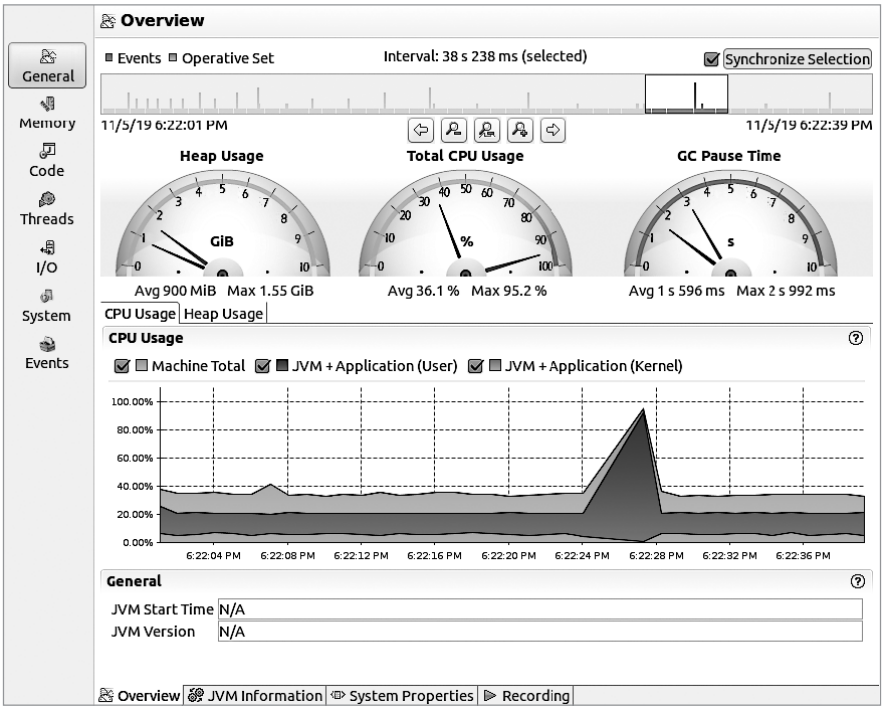


Рис. 3.10. Общая информация в JFR

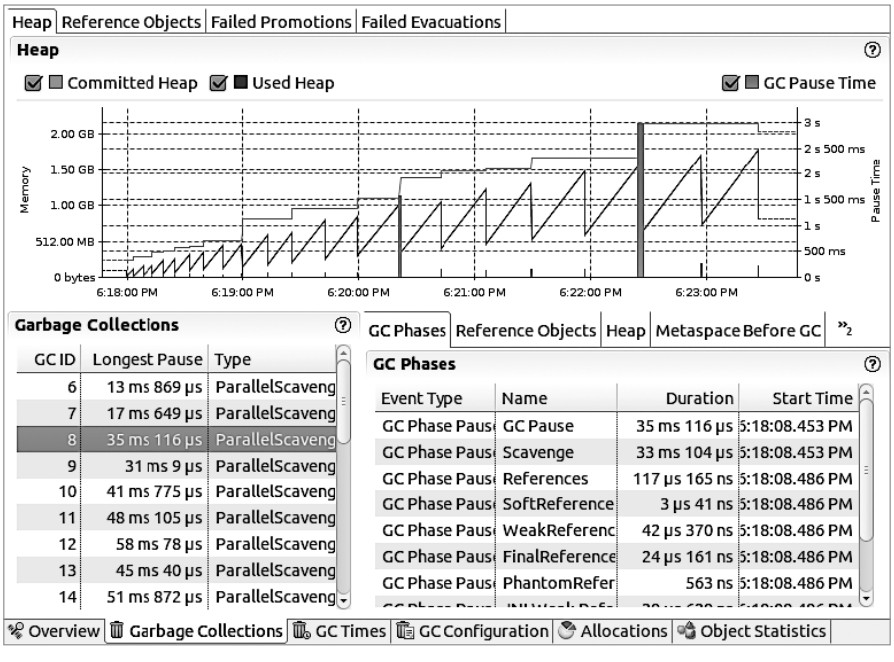


Рис. 3.11. Страница Memory в JFR

со временем: он начинается с 340 Мбайт, а в конце достигает 2 Гбайт). На левой нижней панели показаны все уборки мусора, произошедшие за время сохранения, включая их продолжительность и тип уборки (всегда `ParallelScavenge` в данном примере). При выборе одного из этих событий на правой нижней панели выводится еще более детализированная информация, со всеми конкретными фазами этой уборки и ее продолжительности.

На разных вкладках этой страницы содержится много другой информации: продолжительность и количество уничтоженных ссылочных объектов, наличие ошибок эвакуации (evacuation failure)[1] и повышений от параллельно работающих уборщиков, конфигурация самого алгоритма GC (включая размеры поколений и конфигурации пространства выживших (survivors)), и даже информация о конкретных видах объектов, для которых выделялась память. Читая главы 5 и 6, помните, что эта программа может диагностировать проблемы, которые обсуждаются здесь. Если вам потребуется понять, почему G1-уборщик прекратил работу и выполнил полную уборку мусора (было ли это вызвано ошибкой повышения?), как JVM отрегулировала порог хранения (tenuring threshold), или получить практически любые другие данные относительно того, почему уборщик мусора повел себя именно так, а не иначе, JFR сможет вам предоставить нужную информацию.

Представление Code

Страница Code в `jmc` выдает основную профильную информацию из сохраненных данных (рис. 3.12).

На первой вкладке этой страницы используется группировка по имени пакета — интересная возможность, отсутствующая во многих профилировщиках. Внизу на других вкладках представлены традиционные представления профилирования: горячие методы и дерево вызовов профилируемого кода.

В отличие от других профилировщиков, JFR поддерживает другие режимы вывода информации о коде. Вкладка `Exceptions` содержит информацию об обработке исключений в приложении (в главе 12 объясняется, почему избыточная обработка исключений может плохо отразиться на производительности). На других вкладках выводится информация о работе компилятора, включая вывод информации о программном кэше (см. главу 4).

С другой стороны, следует заметить, что приведенные здесь пакеты отсутствовали в профилях, которые рассматривались ранее; и наоборот, предыдущие проблемные места, которые мы видели, в данном случае отсутствуют. Так как

[1] См. https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm#JSGCT-GUID-0394E76A-1A8F-425E-A0D0-B48A3DC82B42. — *Примеч. ред.*

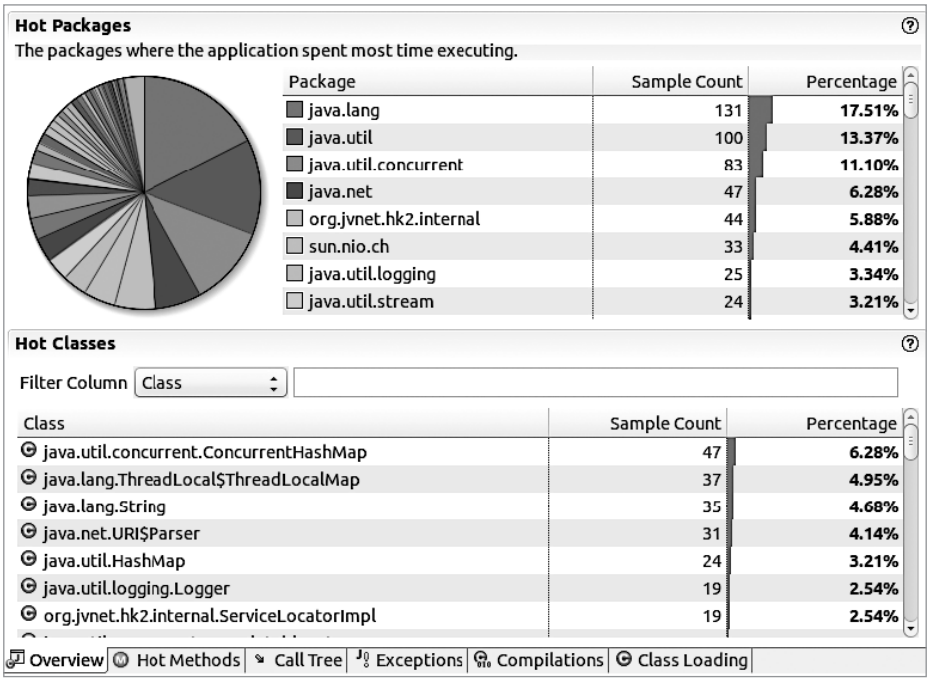


Рис. 3.12. Страница Code в JFR

процедура профилирования с выборкой в JFR проектировалась с расчетом на минимальное воздействие, частота выборки JFR (по крайней мере в конфигурации по умолчанию) достаточно низка, поэтому профили не настолько точны, как при более активной выборке.

Существуют и другие режимы вывода информации (для потоков, ввода/вывода и системных событий), но в основном они всего лишь обеспечивают удобное отображение реальных событий из сохраненных данных JFR.

События JFR

JFR генерирует поток событий, которые сохраняются для последующей обработки. Экраны, приводившиеся выше, обеспечивают различные представления для этих событий, но самые широкие возможности просмотра событий предоставляет панель Event (рис. 3.13).

Отображаемые события фильтруются на левой панели окна; здесь можно выбрать события уровня приложения и события уровня JVM. Учтите, что при сохранении данных включаются только определенные виды событий: в данный

момент нас интересует фильтрация при последующей обработке (в следующем разделе будет показано, как фильтровать события, которые должны включаться при сохранении).

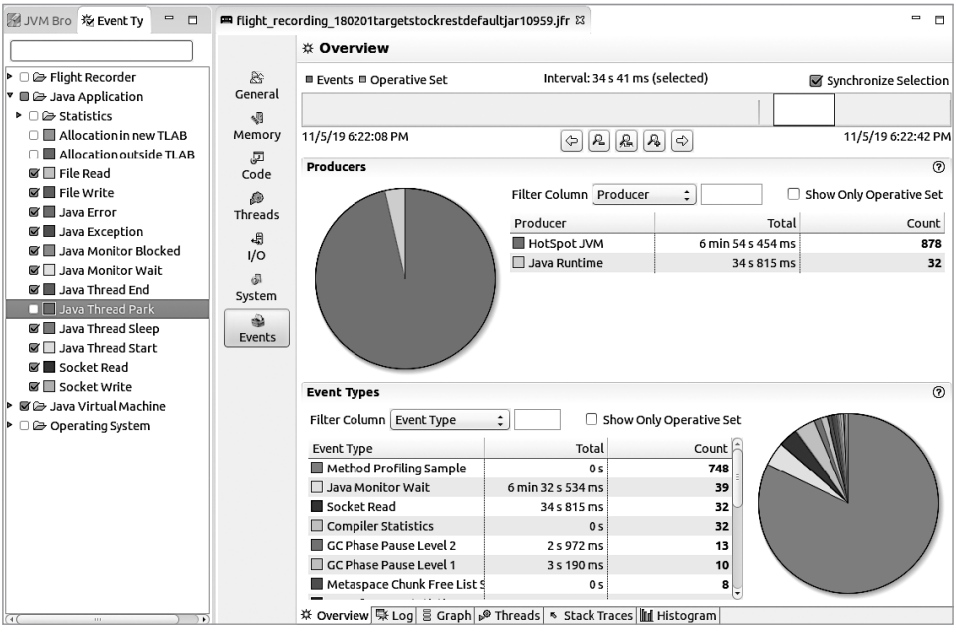


Рис. 3.13. Панель Event в JFR

На протяжении 34-секундного интервала в этом примере приложение произвело 878 событий от JVM и 32 события от библиотек JDK, а типы событий, сгенерированных за этот период, выводятся в нижней части окна. Рассматривая этот пример с профилировщиками, мы видели, почему доля событий парковки потоков и ожидания мониторов в этом примере будет достаточно высокой; эти события можно игнорировать (а события парковки потоков отфильтрованы на левой панели). Как насчет других событий?

За 34-секундный период потоки в приложении потратили 34 секунды на чтение из сокетов. Это число выглядит подозрительно — особенно из-за того, что оно появляется в JFR только в том случае, если чтение из сокета занимает более 10 миллисекунд. Проблема заслуживает более серьезного изучения, для чего можно перейти на вкладку Log (рис. 3.14).

Полезно просмотреть трассировку, связанную с этими событиями, но, как выясняется, несколько потоков используют блокирующий ввод/вывод для чтения административных запросов, которые должны поступать периодически. Между

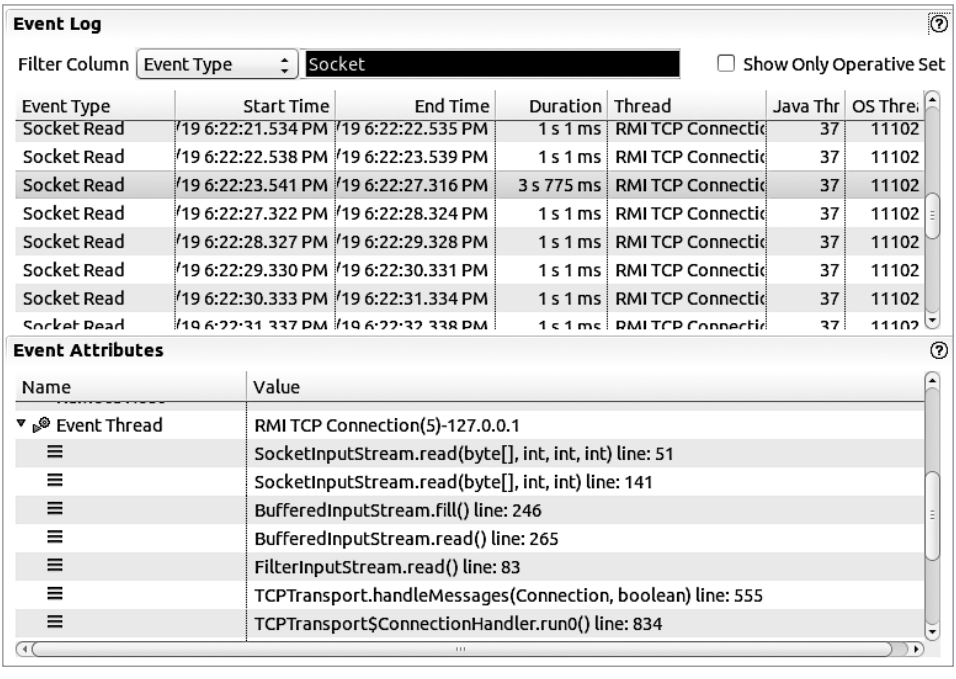


Рис. 3.14. Панель Log в JFR

этими запросами — на долгие периоды времени — потоки остаются заблокированными по методу `read()`. Таким образом, время чтения оказывается приемлемым: просто, как и при использовании профилировщика, вы должны определить, является ли блокировка многих потоков по вводу/выводу ожидаемой или же указывает на проблему производительности.

Остаются события, заблокированные по мониторам. Как показано в главе 9, конкуренция за блокировки проходит через два уровня: сначала поток находится в спин-блокировке, а затем использует (в процессе, называемом *расширением блокировки*) код, специфический для конкретного процессора и ОС, для ожидания. Стандартный профилировщик может подсказать о возникновении такой ситуации, потому что время активного ожидания включается в процессорное время, начисленное методу. Профилировщик низкоуровневого кода может предоставить информацию о блокировках, к которым применялось расширение, однако эта информация не является абсолютно надежной. С другой стороны, JVM может предоставить все эти данные непосредственно JFR.

Пример использования видимости блокировок приведен в главе 9, но относительно событий JFR можно сделать общий вывод: так как эти события поступают непосредственно от JVM, они предоставляют уровень получения информации

о приложении, которого невозможно достичь с помощью любого другого инструмента. В Java 11 JFR позволяет отслеживать около 131 типа событий. Точное количество и типы событий слегка различаются в зависимости от версии, но ниже перечислены наиболее полезные из них.

Для каждого типа события в следующем списке включены два пункта. События могут собирать базовую информацию, для получения которой могут использоваться другие средства — такие, как `jconsole` и `jcmd`; такая информация описывается в первом пункте. Во втором пункте описана информация, предоставляемая событием, которую трудно получить без использования JFR.

Загрузка классов

- Количество загруженных и выгруженных классов.
- Какой загрузчик загрузил класс; время, необходимое для загрузки отдельного класса.

Статистика потоков

- Количество созданных и уничтоженных потоков; дампы потоков.
- Какие потоки заблокированы по ожиданию (и конкретная блокировка, по которой они блокируются).

Ошибки и исключения

- Классы ошибок и исключений, используемые приложением.
- Количество выданных исключений и ошибок, трассировка стека при их создании.

Выделение памяти TLAB

- Количество выделений памяти в куче и размер потоково-локальных буферов (TLAB).
- Конкретные объекты, для которых в куче была выделена память, и трассировка стека при выделении памяти.

Файловый ввод/вывод и сокеты

- Время, проведенное за выполнением ввода/вывода.
- Время, потраченное на вызов чтения/записи; конкретный файл или сокет, операции чтения/записи с которым занимают много времени.

Блокирование по мониторам

- Потоки, ожидающие монитора.
- Конкретные потоки, заблокированные по конкретным мониторам, и продолжительность их блокирования.

Кэш команд

- Размер кэша команд и его заполнение.
- Методы, удаленные из кэша команд; конфигурация кэша команд.

Компиляция

- Какие методы компилируются, компиляция OSR (см. главу 4) и продолжительность компиляции.
- Ничего напрямую относящегося к JFR, но объединяет информацию из нескольких источников.

Уборка мусора

- Хронометраж уборки мусора, включая отдельные фазы; размеры поколений.
- Ничего напрямую относящегося к JFR, но объединяет информацию из нескольких источников.

Профилирование

- Инструментальные профили и профили с выборкой.
- Не так много информации, как можно было бы получить от полноценного профилировщика, но данные JFR предоставляют хорошую сводную информацию.

Включение JFR

Механизм JFR отключен в исходном состоянии. Чтобы включить его, добавьте флаг `-XX:+FlightRecorder` в командную строку приложения. Тем самым вы включаете возможность использования JFR, но данные начнут записываться только после того, как будет запущен процесс записи. Это можно сделать либо из графического интерфейса, либо из командной строки.

В Oracle JDK 8 необходимо задать следующий флаг (до появления флага `FlightRecorder`): `-XX:+UnlockCommercialFeatures` (по умолчанию `false`).

Если вы забудете включить эти флаги, помните, что изменить их значения и включить JFR можно при помощи `jinfo`. Если включить сохранение данных из `jmc`, при необходимости эти значения будут автоматически изменены в целевой JVM.

Включение JFR из jmc

Сохранение данных в локальном приложении проще всего включается из графического интерфейса `jmc` (Java Mission Control). При запуске `jmc` выводится

список всех процессов JVM, работающих в текущей системе. Процессы JVM отображаются в виде древовидной структуры: раскройте узел под меткой `Flight Recorder`, чтобы вызвать окно, изображенное на рис. 3.15.

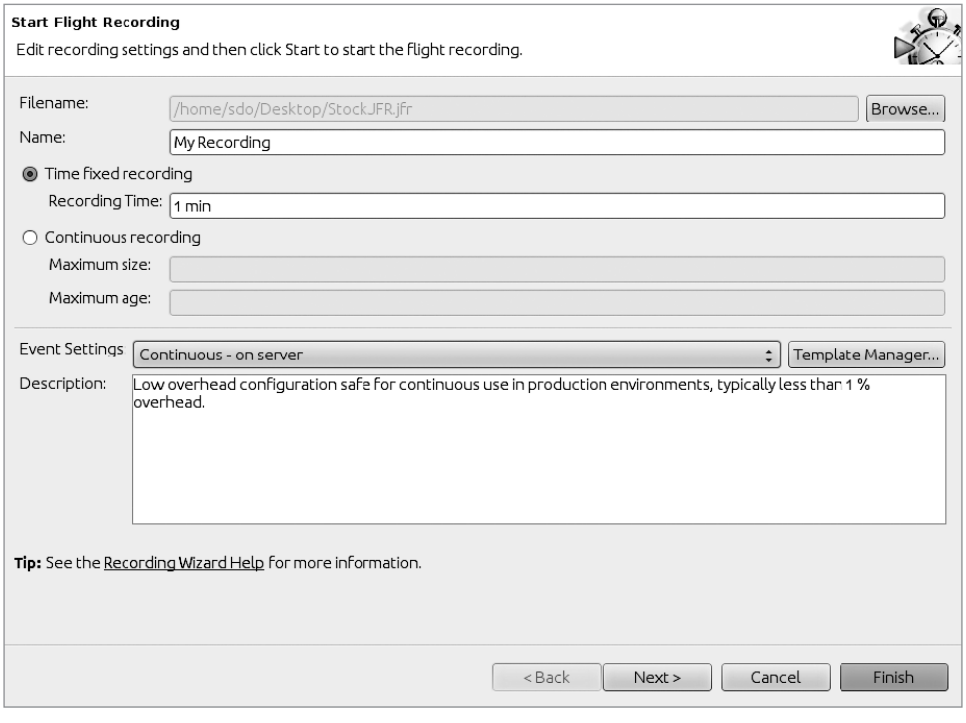


Рис. 3.15. Окно JFR Start Flight Recording

Сохранение данных производится в одном из двух режимов: либо в течение фиксированного времени (1 минута в данном случае), либо непрерывно. Для непрерывного сохранения используется циклический буфер; он содержит последние события в заданном промежутке.

Для выполнения упреждающего анализа — то есть когда вы сначала включаете сохранение, а затем генерируете работу или запускаете сохранение в эксперименте нагрузочного тестирования после разогрева JVM — должно использоваться сохранение фиксированной продолжительности. Такие сохраненные данные хорошо показывают реакцию JVM в ходе теста.

Непрерывное сохранение лучше подходит для *реактивного анализа*. Оно позволяет JVM сохранить самые последние события, а затем стереть сохраненные данные в ответ на событие. Например, сервер приложения WebLogic может

потребовать, чтобы сохраненные данные уничтожались в ответ на аномальное событие в сервере приложения (например, обработка запроса заняла более 5 минут). Вы можете настроить свои средства мониторинга так, чтобы сохраненные данные уничтожались в ответ на события любых типов.

Включение JFR из командной строки

После включения JFR (при помощи параметра `-XX:+FlightRecorder`) существуют несколько способов управления тем, когда и как должна выполняться фактическая запись.

В JDK 8 параметрами сохранения по умолчанию можно управлять при запуске JVM при помощи параметра `-XX:+FlightRecorderOptions=`*строка*; этот вариант наиболее удобен для реактивного сохранения. *Строка* в этом параметре содержит список пар «имя — значение», разделенных запятыми, из следующего списка:

`name=`*имя*

Имя, используемое для идентификации сохраненных данных.

`defaultrecording=<`*true*`|`*false*`>`

Признак запуска сохранения в исходном состоянии. По умолчанию используется значение `false`; для реактивного анализа следует выбрать значение `true`.

`settings=`*путь*

Имя файла с настройками JFR (см. следующий раздел).

`delay=`*время*

Период времени (например, `30s`, `1h`), по истечении которого включается сохранение.

`duration=`*время*

Период времени, в течение которого выполняется сохранение.

`filename=`*путь*

Имя файла, в который записываются сохраненные данные.

`compress=<`*true*`|`*false*`>`

Признак сжатия сохраненных данных (программой `gzip`); по умолчанию используется значение `false`.

`maxage=`*время*

Максимальное время хранения сохраненных данных в циклическом буфере.

`maxsize=`*размер*

Максимальный размер (например, `1024K`, `1M`) циклического буфера для сохраняемых данных.

`-XX:+FlightRecorderOptions` только настраивает значения по умолчанию для любых параметров; при конкретном сохранении эти настройки могут переопределяться.

И в JDK 8, и в JDK 11 можно запустить JFR в момент запуска программы, для чего используется флаг `-XX:+StartFlightRecording=`*строка* с аналогичным списком параметров, разделенных запятыми.

Такая настройка сохранения данных по умолчанию может принести пользу в некоторых ситуациях, но для достижения большей гибкости всеми параметрами можно управлять из `jcmd` во время выполнения.

Запуск сохранения данных:

```
% jcmd идентификатор_процесса JFR.start [список_параметров]
```

Список_параметров содержит серию пар «имя — значение», разделенных запятыми; они управляют тем, как осуществляется сохранение. Допустимые параметры полностью совпадают с теми, которые могут задаваться в командной строке с флагом `-XX:+FlightRecorderOptions=`*строка*.

Если было включено непрерывное сохранение, текущие данные в циклическом буфере можно в любой момент выгрузить в файл следующей командой:

```
% jcmd идентификатор_процесса JFR.dump [список_параметров]
```

Список_параметров может содержать следующие значения:

`name=`*имя*

Имя, с которым было запущено сохранение (см. следующий пример для `JFR.check`).

`filename=`*путь*

Имя файла для сохранения данных.

Может случиться так, что для заданного процесса были включены несколько сохранений данных JFR. Чтобы просмотреть все доступные сохранения, введите следующую команду:

```
% jcmd 21532 JFR.check [verbose]
21532:
Recording 1: name=1 maxsize=250.0MB (running)

Recording 2: name=2 maxsize=250.0MB (running)
```

В этом примере для процесса с идентификатором 21532 существуют два активных сохранения JFR с именами `1` и `2`. Эти имена могут использоваться для идентификации сохраненных данных в других командах `jcmd`.

Наконец, для отмены сохранения данных в процессе используется следующая команда:

```
% jcmd идентификатор_процесса JFR.stop [список_параметров]
```

Команда получает следующие параметры:

`name=`*имя*

Имя сохранения, которое нужно остановить.

`discard=<true|false>`

Если значение равно `true`, данные уничтожаются — вместо записи в файл с ранее предоставленным именем (если оно есть).

`filename=`*путь*

Имя файла для сохранения данных.

В автоматизированной системе тестирования производительности выполнение этих средств командной строки и сохранение данных пригодится тогда, когда потребуется проанализировать эти запуски на предмет регрессий.

Выбор событий JFR

Как упоминалось ранее, JFR поддерживает многие виды событий. Часто эти события являются периодическими: они происходят через каждые несколько миллисекунд (например, события профилирования работают на основе выборки). Другие события происходят только тогда, когда продолжительность события превышает некоторый порог (например, событие чтения файла срабатывает только в том случае, если время выполнения метода `read()` превысило заданный интервал).

Получение событий естественным образом требует затрат. Порог выявления событий также играет роль в затратах, происходящих из-за включения сохранения данных JFR (так как он увеличивает число событий). При сохранении по умолчанию отслеживаются не все события (шесть самых затратных событий не включаются), а порог временных событий достаточно высок. В результате затраты на сохранение данных по умолчанию не превышают 1%.

В некоторых ситуациях дополнительные затраты оправданны. Например, обращение к событиям TLAB позволит определить, создаются ли объекты непосредственно в старом поколении, но эти события не включаются по умолчанию. Аналогичным образом события профилирования включаются по умолчанию,

но только через каждые 20 мс — такая периодичность дает общую картину, но также может привести к ошибкам выборки[1].

ДРУГИЕ СОБЫТИЯ JFR

Механизм JFR является расширяемым: приложения могут определять собственные события. Следовательно, ваша реализация JFR может выводить намного больше доступных типов событий в зависимости от специфики приложения. Например, сервер приложения WebLogic включает несколько событий сервера приложения: операции JDBC, операции HTTP и т. д. Эти события обрабатываются так же, как и другие события JFR, описанные в тексте: их можно включать по отдельности, назначать пороги и т. д. Аналогичным образом в последних версиях JVM могут появиться дополнительные события, которые здесь не рассматриваются.

За самой подробной информацией обращайтесь к обновленной документации продукта.

События (и порог событий), сохраняемые JFR, определяются в шаблоне (который выбирается при помощи параметра `settings` в командной строке). JFR поставляется с двумя шаблонами: шаблоном по умолчанию (который ограничивает набор событий, чтобы затраты не превысили 1%) и профильным шаблоном (который настраивает большинство потоковых событий для срабатывания через каждые 10 мс). Оценка дополнительных затрат для профильного шаблона составляет 2% (хотя, как обычно, у вас все может быть по-другому, и, как правило, затраты ниже указанной величины).

Шаблонами управляет диспетчер шаблонов `jmc`; обратите внимание на кнопку запуска диспетчера шаблонов на рис. 3.15. Шаблоны хранятся в двух местах: в каталоге `$HOME/.jmc/<`*версия*`>` (локальный по отношению к пользователю) и в каталоге `$JAVA_HOME/jre/lib/jfr` (глобальный для JVM). Диспетчер шаблонов позволяет вам выбрать глобальный шаблон, выбрать локальный шаблон или определить новый шаблон. Чтобы определить шаблон, переберите все доступные события и включите (или отключите) их так, как требуется, возможно — с указанием порога срабатывания событий.

На рис. 3.16 показано, что событие `File Read` включается с порогом 15 мс; операции чтения файлов, занимающие больше времени, приведут к срабатыванию события. Это событие также было настроено для генерирования трассировки стека по событиям `File Read`. Это приводит к повышению затрат — а это объ-

[1] Вот почему рассмотренный профиль JFR не всегда совпадает с более активными профилями из предыдущих разделов.

ясняет, почему получение трассировки стека для событий может настраиваться при помощи параметра.

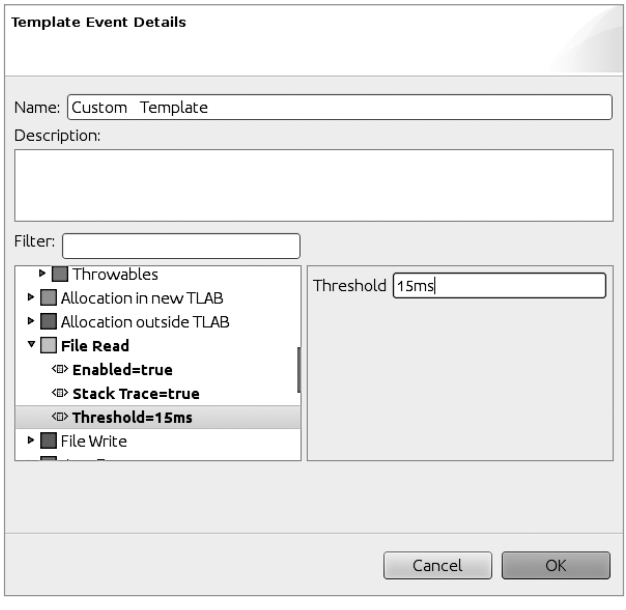


Рис. 3.16. Пример шаблона событий JFR

Шаблоны событий представляют собой простые файлы XML, поэтому для определения того, какие события включены в шаблоне (а также их порогов и конфигураций трассировки стека), лучше всего прочитать файл XML. Использование XML также позволяет определить файл локального шаблона на одной машине, а затем скопировать его в каталог глобальных шаблонов для использования другими участниками команды.

РЕЗЮМЕ

- Механизм Java Flight Recorder предоставляет лучшие возможности для получения информации о JVM, так как он встроен непосредственно в JVM.
- Как и все инструменты, JFR добавляет к приложению некоторые затраты. При повседневном использовании JFR можно настроить для сбора значительного объема информации при минимальных затратах.
- JFR используется при анализе производительности, но также может пригодиться в условиях реальной эксплуатации, чтобы вы могли проанализировать события, приводящие к ошибке.

Итоги

Хороший инструментарий играет ключевую роль в анализе производительности; в этой главе мы едва соприкоснулись с той информацией, которую можно получить при помощи специализированных программ. Некоторые ключевые обстоятельства, о которых необходимо помнить:

- Ни один инструмент не идеален, и у разных программ могут быть свои достоинства и недостатки. Профилировщик X может хорошо подходить для многих приложений, но в отдельных случаях он может упустить что-то такое, на что ясно укажет профилировщик Y. Всегда действуйте гибко.
- Средства мониторинга, работающие в режиме командной строки, могут собирать важные данные автоматически; обязательно включите сбор данных мониторинга в автоматизированное тестирование производительности.
- Программы быстро развиваются; возможно, некоторые программы, упомянутые в этой главе, уже устарели (или были заменены новыми, более совершенными средствами). Очень важно не отставать от жизни в этой области.

ГЛАВА 4

Работа с JIT-компилятором

JIT-компилятор (Just-in-time) занимает центральное место в виртуальной машине Java; ничто не влияет на производительность вашего приложения в большей степени, чем JIT-компилятор.

В этой главе компилятор рассматривается более подробно. Глава начинается с информации о том, как работает компилятор, а также обсуждения достоинств и недостатков JIT-компилятора. До выхода JDK 8 вам приходилось выбирать между двумя компиляторами Java. В наши дни эти два компилятора продолжают существовать, но они спокойно уживаются друг с другом, хотя в редких случаях приходится выбирать один из них. Наконец, мы рассмотрим некоторые настройки компилятора среднего и высокого уровня. Если приложение медленно работает без каких-либо очевидных причин, эти разделы помогут вам определить, виноват ли в этом компилятор.

JIT-компиляторы: общие сведения

Начнем с небольшого вводного курса; если вы уже разбираетесь в основах JIT-компиляции, спокойно переходите к следующему разделу.

Компьютеры — а конкретнее процессоры — могут выполнять только относительно немногочисленные низкоуровневые команды, называемые *машинным кодом*. Все программы, выполняемые процессором, должны быть преобразованы в эти команды.

Такие языки, как C++ и Fortran, назывались *компилируемыми языками*, потому что программы на этих языках распространялись в виде двоичного (откомпилированного) кода: вы пишете программу, после чего статический компилятор создает двоичный файл. Ассемблерный код в этом двоичном файле ориентирован на конкретный процессор. Процессоры с совместимым набором команд тоже смогут выполнить этот двоичный файл; например, процессоры AMD и Intel поддерживают общий базовый набор команд языка ассемблера, а более

новые версии процессоров почти всегда могут выполнять тот же набор команд, что и предыдущие версии этого процессора. Обратное не всегда справедливо; в новых версиях процессоров часто появляются команды, которые не работают на старых версиях этого процессора.

С другой стороны, такие языки, как PHP и Perl, являются *интерпретируемыми*. Исходный код программы может выполняться на любом процессоре, при условии что на машине имеется нужный интерпретатор (то есть программа `php` или `perl`). Интерпретатор преобразует каждую строку программы в двоичный код при выполнении этой строки.

У каждой системы есть свои плюсы и минусы. Программы, написанные на интерпретируемых языках, отличаются портируемостью: вы можете взять тот же код и перенести его на любую машину с подходящим интерпретатором, и этот код будет выполнен. Однако он может работать медленно. Простейший пример: подумайте, что происходит в цикле. Интерпретатору приходится заново транслировать каждую строку кода при выполнении цикла. Компилируемому коду не нужно снова и снова выполнять это преобразование.

Хороший компилятор учитывает ряд факторов при построении двоичного файла. Один из примеров — порядок двоичных команд; не все команды на языке ассемблера выполняются за одно и то же время. Команда суммирования значений, хранящихся в двух регистрах, может выполняться за один такт, но выборка слагаемых (из оперативной памяти) может потребовать нескольких тактов.

А значит, хороший компилятор построит двоичный файл, который выполняет команду для загрузки данных, выполняет другие команды, а затем — когда данные станут доступны — выполняет сложение. Интерпретатор, который просматривает исходный код по одной строке, не располагает достаточной информацией для генерирования такого кода; он запросит данные из памяти, дождется того момента, когда они станут доступными, а затем выполнит сложение. Кстати говоря, плохие компиляторы поступают так же, но даже самым лучшим компиляторам не всегда удается обойтись без ожидания завершения отдельных команд.

По этим (и другим) причинам интерпретируемый код почти всегда будет несравнимо медленнее откомпилированного: компилятор располагает достаточной информацией о программе и может предоставить такие оптимизации двоичного кода, на которые интерпретатор попросту не способен.

К достоинствам интерпретируемого кода относится портируемость. Двоичный файл, откомпилированный для процессора ARM, не сможет выполняться на процессоре Intel. Но двоичный файл, использующий новейшие команды AVX процессоров Intel Sandy Bridge, тоже не будет работать на старых процессорах Intel. По этой причине коммерческие программы обычно компилируются для относительно старой версии процессора и не используют новейшие команды. Существуют различные трюки, позволяющие обойти ограничение, например,

поставка двоичного файла с несколькими общими библиотеками, которые выполняют код, критичный по быстродействию, и поставляются в нескольких разновидностях для разных версий процессора.

Java пытается отыскать золотую середину. Приложения Java компилируются — но вместо компиляции в двоичный файл для конкретного типа процессора они компилируются в промежуточный низкоуровневый язык. Этот язык (он называется *байт-кодом* Java) затем выполняется двоичным файлом `java` (подобно тому, как интерпретируемый сценарий PHP выполняется двоичным файлом `php`). Таким образом Java достигает платформенной независимости, типичной для интерпретируемых языков. Так как программа `java` выполняет идеализированный двоичный код, она может откомпилировать код в двоичный код конкретной платформы при выполнении кода. Компиляция происходит во время выполнения программы, то есть как раз в нужный момент (Just In Time).

У JIT-компиляции по-прежнему существуют некоторые зависимости от платформ. Например, JDK 8 не может генерировать код для последнего набора команд процессоров Intel Skylake, хотя JDK 11 справляется с этой задачей. Я еще вернусь к этой теме в разделе «Флаги компилятора высокого уровня», с. 136. Механизм компиляции кода виртуальной машиной Java прямо во время его выполнения станет основной темой этой главы.

Компиляция HotSpot

Как упоминалось в главе 1, в этой книге рассматривается реализация Java Oracle Hotspot JVM. Название (HotSpot, то есть «горячая точка») происходит от подхода к компиляции кода. В типичной программе часто выполняется небольшое подмножество кода, и производительность приложения в целом зависит в основном от скорости выполнения этих частей кода. Такие критические секции называются горячими точками приложения.

Таким образом, в начале выполнения кода JVM не начинает немедленно компилировать этот код. Это происходит по двум основным причинам. Во-первых, если код будет выполнен только один раз, то его компиляция по сути ничего не даст; такие байт-коды Java быстрее интерпретировать, чем откомпилировать их и выполнить откомпилированный код (всего один раз).

Но если код, о котором идет речь, находится в часто вызываемом методе или в цикле с множеством итераций, компиляция оправдает затраченные усилия: время, потраченное на компиляцию кода, будет скомпенсировано экономией от повторных выполнений более быстрого откомпилированного кода. Именно по этой причине компилятор сначала выполняет интерпретируемый код — он может определить, какие методы вызываются достаточно часто для того, чтобы их компиляция была оправданна.

Вторая причина — оптимизация: чем больше раз JVM выполнит некоторый метод или цикл, тем больше информации у нее будет об этом коде. Это позволяет JVM применить различные оптимизации при компиляции этого кода.

РЕГИСТРЫ И ОСНОВНАЯ ПАМЯТЬ

Одна из самых важных оптимизаций, которую может применить компилятор, связана с тем, когда использовать значения из основной памяти и когда сохранять значения в регистре. Возьмем следующий код:

```
public class RegisterTest {
    private int sum;

    public void calculateSum(int n) {
        for (int i = 0; i < n; i++) {
            sum += i;
        }
    }
}
```

В какой-то момент переменная экземпляра `sum` должна находиться в основной памяти, но выборка значения из основной памяти является затратной операцией, выполнение которой занимает несколько тактов процессора. Если значение `sum` должно загружаться из основной памяти (и снова сохраняться в ней) при каждой итерации цикла, производительность будет ужасной. Вместо этого компилятор загружает в регистр исходное значение `sum`, выполняет цикл с этим значением в регистре, а затем (в неопределенный момент времени) сохраняет конечный результат из регистра в основной памяти.

Такая оптимизация очень эффективна, но это означает, что семантика синхронизации потоков (см. главу 9) критична для поведения приложения. Один поток не видит значение переменной, хранящейся в регистре, который используется другим потоком; синхронизация позволяет точно узнать, когда значение из регистра окажется в основной памяти и станет доступным для других потоков.

Использование регистров относится к общей оптимизации компилятора; обычно JIT-компилятор активно использует регистры. Эта тема подробно рассматривается в разделе «Анализ локальности» на с. 143.

Эти оптимизации (и возможности управлять ими) рассматриваются позднее в этой главе, а пока возьмем простой пример: метод `equals()`. Этот метод присутствует в каждом объекте Java (потому что он наследует от класса `Object`)

и часто переопределяется. Когда интерпретатор встречает команду `b = obj1.equals(obj2)`, он должен определить тип (класс) `obj1`, чтобы узнать, какой метод `equals()` следует выполнять. Динамический поиск информации о типе иногда занимает довольно много времени.

Допустим, со временем JVM замечает, что при каждом выполнении этой команды `obj1` относится к типу `java.lang.String`. Тогда JVM может сгенерировать откомпилированный код, который напрямую вызывает метод `String.equals()`. Теперь код работает быстрее не только потому, что он откомпилирован, но и потому, что он может пропустить процедуру определения вызываемого метода.

Впрочем, все не так просто: может оказаться, что при следующем выполнении кода `obj1` будет ссылаться не на `String`, а на что-то другое. JVM создает откомпилированный код, который допускает такую возможность, но это потребует деоптимизации с последующей реоптимизацией кода (пример приведен в разделе «Деоптимизация» на с. 133). В целом откомпилированный код будет работать быстрее (по крайней мере пока `obj1` продолжает ссылаться на `String`), потому что он обходит процедуру поиска выполняемого метода. Такие оптимизации могут применяться только после того, как код отработает в течение какого-то времени, и по результатам наблюдений за ним: это вторая причина, по которой JIT-компиляторы какое-то время ожидают перед компилированием части кода.

РЕЗЮМЕ

- Одной из целей проектирования Java было совмещение платформенной независимости сценарных языков с быстродействием компилируемых языков.
- Файл класса Java компилируется на промежуточный язык (байт-код Java), после чего компилируется далее на язык ассемблера, используемый JVM.
- При компиляции байт-кода на язык ассемблера применяются оптимизации, значительно повышающие производительность.

Многоуровневая компиляция

Давным-давно JIT-компилятор существовал в двух разновидностях, и вам приходилось устанавливать разные версии JDK в зависимости от того, какой компилятор вы хотели использовать. Эти компиляторы назывались *клиентским* и *серверным* компилятором. В 1996 году это различие было важным; в 2020-м ситуация изменилась. В наши дни в поставку всех JVM включаются оба компилятора (хотя в повседневной практике они обычно называются серверными JVM).

ФЛАГИ КОМПИЛЯТОРА

В старых версиях Java используемый компилятор выбирался при помощи флага, который не следовал стандартным правилам флагов JVM: для клиентского компилятора указывался флаг `-client`, а для серверного — флаг `-server` или `-d64`.

Так как разработчики не изменяют сценарии без необходимости, вы наверняка столкнетесь со сценариями и другими командными строками, в которых присутствует флаг `-client` или `-server`. Просто запомните, что начиная с JDK 8 эти флаги не делают ничего. Это также относится ко многим более ранним версиям JDK: если указать флаг `-client` для JVM, поддерживающей только серверный компилятор, все равно будет использован серверный компилятор.

С другой стороны, следует учитывать, что старый аргумент `-d64` (который фактически был синонимом для `-server`) был исключен из JDK 11, а попытка его использования приведет к ошибке. В JDK 8 этот аргумент игнорировался.

Хотя JVM теперь называются серверными, различия между клиентским и серверным компиляторами сохраняются; оба компилятора доступны и используются JVM. Важно знать о различиях между ними, чтобы понять, как работает компилятор.

Традиционно разработчики JVM (и даже некоторые программы) обозначали компиляторы именами `C1` (компилятор 1, клиентский компилятор) и `C2` (компилятор 2, серверный компилятор). В наши дни эти имена стали более уместными, так как различия между клиентскими и серверными компьютерами стерлись, поэтому я буду использовать именно эти обозначения.

Главное отличие двух компиляторов связано с их активностью при компилировании кода. Компилятор `C1` начинает компилировать на более ранней стадии, чем компилятор `C2`. Это означает, что в начале выполнения кода компилятор `C1` будет работать быстрее, потому что он соответственно откомпилирует больше кода, чем компилятор `C2`.

С другой стороны, информация, собранная во время ожидания компилятором `C2`, позволяет ему применить более эффективные оптимизации в откомпилированном коде. В конечном итоге код, сгенерированный компилятором `C2`, будет быстрее кода, сгенерированного компилятором `C1`. С точки зрения пользователя плюсы и минусы такого решения зависят от того, как долго будет выполняться программа и настолько важную роль играет время ее запуска.

Пока компиляторы существовали по отдельности, возникал очевидный вопрос: зачем вообще нужно выбирать? Почему JVM не может начать с компилятора `C1`, а потом использовать компилятор `C2` с получением нужной информации? Такой подход называется *многоуровневой компиляцией*; сейчас он применяется всеми JVM. Его можно явно отключить при помощи флага `-XX:-TieredCompilation` (значение по умолчанию `true`); последствия рассматриваются в разделе «Флаги компилятора высокого уровня» на с. 136.

Распространенные флаги компилятора

В этом разделе рассматриваются два часто используемых флага, влияющих на работу JIT-компилятора.

Настройка кэша команд

В процессе компиляции JVM хранит набор команд на языке ассемблера в кэше команд. Кэш команд имеет фиксированный размер, и после его заполнения JVM не сможет компилировать дополнительный код.

Как нетрудно понять, слишком малый размер кэша команд создает потенциальные проблемы. Некоторые горячие методы будут откомпилированы, но другие останутся неоткомпилированными; в итоге в приложении будет выполняться большое количество (очень медленного) интерпретируемого кода.

При заполнении кэша команд JVM выдает следующее предупреждение:

```
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full.
         Compiler has been disabled.
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the
         code cache size using -XX:ReservedCodeCacheSize=
```

Это сообщение легко упустить. Определить, не перестал ли компилятор компилировать код, также можно по выходным данным журнала компиляции (см. далее в этом разделе).

В реальности нет хорошего механизма для определения объема кэша команд, необходимого конкретному приложению. Таким образом, при увеличении размера кода обычно приходится действовать наугад; как правило, значение по умолчанию просто увеличивается в 2 или 4 раза. Максимальный размер кэша команд задается флагом `-XX:ReservedCodeCacheSize=`N (где N — значение по умолчанию для конкретного компилятора). Управление кэшем команд осуществляется примерно так же, как и управление большей частью памяти в JVM: есть исходный

размер (который задается флагом `-XX:InitialCodeCacheSize=`*N*). Кэш команд создается с исходным размером и расширяется по мере заполнения. Исходный размер кэша команд равен 2496 Кбайт, а максимальный размер по умолчанию составляет 240 Мбайт. Изменение размера кэша происходит в фоновом режиме и не влияет на производительность, поэтому в общем случае не требуется ничего, кроме настройки размера `ReservedCodeCacheSize`.

Почему бы не задать очень большой максимальный размер кэша команд, чтобы память в нем никогда не кончалась? Это зависит от объема ресурсов, доступных на целевой машине. Если указать размер кэша команд 1 Гбайт, то JVM зарезервирует 1 Гбайт системной памяти. Эта память не будет выделена до того, как в ней возникнет необходимость, но она остается зарезервированной; это означает, что на вашей машине должна быть доступна виртуальная память, достаточная для зарезервированного объема.

ЗАРЕЗЕРВИРОВАННАЯ И ВЫДЕЛЕННАЯ ПАМЯТЬ

Важно понимать различия между резервированием и выделением памяти JVM. Эти различия относятся к кэшу команд, куче Java и другим системным структурам памяти JVM.

За дополнительной информацией по теме обращайтесь к разделу «Потребление памяти» на с. 300.

Кроме того, на старых машинах Windows с 32-разрядной JVM общий размер процесса не может превышать 4 Гбайт. Сюда включается куча Java, память всего кода JVM (включая платформенные библиотеки и стеки потоков), а также вся системная память, выделяемая приложением (либо напрямую, либо при помощи библиотек New I/O [NIO]), и, разумеется, кэш команд.

Существуют причины, по которым размер кэша команд ограничивается, и в больших приложениях его иногда приходится ограничивать. На 64-разрядных машинах с достаточным объемом памяти завышенный порог вряд ли будет иметь практические последствия для приложения: приложение не исчерпает пространство памяти процесса, а резервируемая дополнительная память обычно предоставляется операционной системой.

В Java 11 кэш команд сегментирован на три части:

- Код, не принадлежащий методам.
- Профилированный код.
- Непрофилированный код.

По умолчанию кэш команд создается с постоянным размером (240 Мбайт), а для настройки общего размера кэша команд используется флаг `ReservedCodeCacheSize`. В этом случае для сегмента кода, не принадлежащего методам, память выделяется по количеству потоков компилятора (см. раздел «Потоки компиляции», с. 138); на машине с четырьмя процессорами он составляет около 5,5 Мбайт. Два других сегмента в равных долях делят оставшийся кэш кода команд — например, около 117,2 Мбайт для каждого на машине с четырьмя процессорами (в сумме 240 Мбайт).

Вам вряд ли когда-нибудь потребуется настраивать эти сегменты по отдельности, но если такая необходимость все же возникнет, используйте следующие флаги:

- `-XX:NonNMethodCodeHeapSize=`*N* для кода, не принадлежащего методам;
- `-XX:ProfiledCodeHapSize=`*N* для профилированного кода;
- `-XX:NonProfiledCodeHapSize=`*N* для непрофилированного кода.

Размер кэша команд (и сегментов JDK 11) можно отслеживать в реальном времени: используйте программу `jconsole` и выберите диаграмму Memory Pool Code Cache на панели Memory. Также можно включить механизм Java Native Memory Tracking (глава 8).

РЕЗЮМЕ

- Кэш команд — ресурс с определенным максимальным размером, который влияет на общий объем компилируемого кода, который может выполняться JVM.
- Очень большие приложения в конфигурации по умолчанию могут заполнить весь кэш команд; следите за состоянием кэша команд и увеличивайте его размер при необходимости.

Анализ процесса компиляции

Второй флаг не является оптимизацией как таковой: он не улучшает производительность приложения. Вместо этого флаг `-XX:+PrintCompilation` (по умолчанию `false`) дает представление о внутренней работе компилятора (хотя мы также рассмотрим другие инструменты, предоставляющие аналогичную информацию).

Если флаг `PrintCompilation` установлен, то при каждой компиляции метода (или цикла) JVM выводит строку с информацией о только что откомпилированном коде. Большинство строк в журнале компиляции имеет следующий формат:

временная_метка идентификатор_компиляции атрибуты (уровень) имя_метода размер деопт

Здесь *временная_метка* — время после завершения компиляции (относительно нуля, которым считается момент запуска JVM).

Идентификатор_компиляции — внутренний идентификатор задачи. Обычно это монотонно растущее число, но иногда встречаются идентификаторы компиляции с нарушением числового порядка. Обычно это происходит при существовании нескольких потоков компиляции, которые работают быстрее или медленнее относительно друг друга. Однако не стоит делать вывод, что одна конкретная задача компиляции по какой-то причине выполняется слишком медленно: обычно такие расхождения возникают из-за особенностей планирования потоков.

Поле *атрибуты* содержит набор из пяти символов, обозначающих состояние компилируемого кода. Если некоторый атрибут применяется к заданной компиляции, выводится символ из следующего списка; в противном случае для этого атрибута выводится пробел. Следовательно, строка атрибутов из пяти символов может выглядеть как два и более элемента, разделенных пробелами. Поддерживаются следующие атрибуты:

- `%` Компиляция с замещением в стеке (OSR).
- `s` Метод синхронизирован.
- `!` У метода имеется обработчик исключений.
- `b` Компиляция происходит в блокирующем режиме.
- `n` Компиляция выполнялась с оберткой платформенного метода.

Первый из этих атрибутов относится к механизму OSR (On-Stack Replacement). JIT-компиляция является асинхронным процессом: когда JVM решает, что некоторый метод должен компилироваться, этот метод помещается в очередь. Вместо того чтобы дожидаться компиляции, JVM продолжает интерпретировать метод, и при следующем его вызове JVM выполняет откомпилированную версию метода (конечно, если компиляция завершилась).

Но представьте себе долгий цикл. JVM замечает, что сам цикл должен компилироваться, и ставит код в очередь на компиляцию. Но этого недостаточно: JVM должна иметь возможность начать выполнять откомпилированную версию цикла в то время, пока цикл еще выполняется, — было бы неэффективно ожидать, пока завершится сам цикл и вмещающий метод (чего может и не произойти). Поэтому когда компиляция кода цикла будет завершена, JVM заменит код (в стеке), и следующая итерация цикла будет выполнять намного более быструю откомпилированную версию кода. Этот механизм замещения в стеке называется *OSR*.

Следующие два атрибута должны быть понятны без объяснений. Флаг блокирования никогда не выводится по умолчанию в текущих версиях Java; он указывает, что компиляция не выполняется в фоновом режиме (за дополнительной информацией обращайтесь к разделу «Потоки компиляции», с. 138). Наконец,

атрибут платформенного метода означает, что JVM генерирует компилируемый код для упрощения вызова платформенного метода.

Если многоуровневая компиляция была отключена, то следующее поле (*уровень*) будет пустым. В противном случае здесь выводится число, указывающее, какой уровень завершил компиляцию. Затем следует имя компилируемого метода (или метода, содержащего цикл, компилируемый для OSR), который выводится в формате *ИмяКласса*::*метод*.

Далее идет *размер* (в байтах) компилируемого кода. Указывается размер байт-кода Java, а не размер компилируемого кода (а значит, по нему, к сожалению, не удастся спрогнозировать размер кэша команд).

АНАЛИЗ КОМПИЛЯЦИИ ПРОГРАММОЙ JSTAT

Для просмотра журнала компиляции программа должна быть запущена с флагом `-XX:+PrintCompilation`. Если программа была запущена без этого флага, можно получить ограниченную информацию о работе компилятора при помощи программы jstat.

Программа jstat поддерживает два параметра для получения информации о компиляторе. Параметр `-compiler` поставляет сводную информацию о количестве компилируемых методов (здесь 5003 — идентификатор процесса анализируемой программы):

```
% jstat -compiler 5003
Compiled Failed Invalid   Time   FailedType FailedMethod
     206      0       0     1.97          0
```

Обратите внимание: при этом также выводится количество методов, которые не откомпилировались, и имя последнего метода, который не откомпилировался; если из профилей и другой информации возникает предположение, что метод работает медленно, потому что он не был откомпилирован, эту гипотезу можно легко проверить.

Также можно воспользоваться параметром `-printcompilation` для получения информации о последнем откомпилированном методе. Так как `jstat` может получать необязательный аргумент для повторения своей операции, вы можете проследить за тем, какие методы компилируются, с течением времени. В этом примере jstat повторяет информацию для процесса с идентификатором 5003 каждую секунду (1000 мс):

```
% jstat -printcompilation 5003 1000
Compiled  Size  Type Method
     207     64    1 java/lang/CharacterDataLatin1 toUpperCase
     208      5    1 java/math/BigDecimal$StringBuilderHelper getCharArray
```

Наконец, в некоторых случаях сообщение в конце строки компиляции указывает на то, что произошла некоторая разновидность деоптимизации; обычно это свидетельствует о том, что код стал недействительным (non-entrant) или превратился в зомби. За дополнительной информацией обращайтесь к разделу «Деоптимизация», с. 133.

Журнал компиляции также может включать строку, которая выглядит примерно так:

```
timestamp идентификатор_компиляции COMPILE SKIPPED: причина
```

Строка (с текстом `COMPILE SKIPPED`) указывает, что при компиляции заданного метода что-то пошло не так. Два распространенных случая с выводимым сообщением *причина*:

`Code cache filled`

> Размер кэша команд необходимо увеличить при помощи флага `ReservedCodeCache`.

`Concurrent classloading`

> Класс был изменен во время компиляции. JVM снова откомпилирует его позднее; в журнале должно появиться сообщение о перекомпиляции метода.

Во всех случаях (кроме заполнения кэша) должна быть выполнена повторная попытка компиляции. Если этого не сделать, ошибка помешает компиляции кода. Ошибки в компиляторе исключать нельзя, но обычной мерой во всех случаях становится рефакторинг кода и приведение его к более простому виду, с которым сможет справиться компилятор.

Несколько строк вывода при включении флага `PrintCompilation` для REST-приложения с акциями:

```
  28015  850       4     net.sdo.StockPrice::getClosingPrice (5 bytes)
  28179  905  s    3     net.sdo.StockPriceHistoryImpl::process (248 bytes)
  28226   25 %     3     net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
  28244  935       3     net.sdo.MockStockPriceEntityManagerFactory$\
                             MockStockPriceEntityManager::find (507 bytes)
  29929  939       3     net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
 106805 1568   !   4     net.sdo.StockServlet::processRequest (197 bytes)
```

В этот вывод включены только некоторые методы, относящиеся к акциям (и не обязательно приводятся все строки, относящиеся к конкретному методу). Несколько интересных моментов, на которые стоит обратить внимание: первый такой метод был откомпилирован только через 28 секунд после запуска сервера, а перед ним были откомпилированы 849 других методов. В данном случае все эти остальные методы были методами сервера или JDK (отфильтрованы в при-

веденном выводе). Серверу понадобилось всего около 2 секунд для запуска; следующие 26 секунд до того, как что-то было откомпилировано, приложение фактически бездействовало, так как сервер ожидал запросов.

Остальные строки приведены для того, чтобы подчеркнуть некоторые интересные особенности. Метод `process()` синхронизирован, поэтому в атрибутах присутствует символ `s`. Внутренние классы компилируются так же, как и любые другие, и встречаются в выводе в обычной записи Java: *внешний_класс*$*внутренний класс*. Метод `processRequest()` выводится с обработчиком исключения, как и предполагалось.

Наконец, вспомните реализацию конструктора `StockPriceHistoryImpl`, который содержит большой цикл:

```
public StockPriceHistoryImpl(String s, Date startDate, Date endDate) {
    EntityManager em = emf.createEntityManager();
    Date curDate = new Date(startDate.getTime());
    symbol = s;
    while (!curDate.after(endDate)) {
         StockPrice sp = em.find(StockPrice.class, new StockPricePK(s,
curDate));
         if (sp != null) {
            if (firstDate == null) {
                firstDate = (Date) curDate.clone();
            }
            prices.put((Date) curDate.clone(), sp);
            lastDate = (Date) curDate.clone();
        }
        curDate.setTime(curDate.getTime() + msPerDay);
    }
}
```

Цикл выполняется чаще, чем сам конструктор, поэтому к циклу применяется OSR-компиляция. Учтите, что компиляция этого метода заняла много времени; ее идентификатор компиляции равен 25, но он появляется только после того, как будут откомпилированы другие методы в диапазоне 900 (такие строки OSR легко прочитать как 25% и ломать голову над тем, куда делись остальные 75%, но помните, что число — идентификатор компиляции, а `%` — признак компиляции). Такая картина типична для компиляции OSR.

Уровни многоуровневой компиляции

В журнале компиляции для программы, использующей многоуровневую компиляцию, выводится уровень, на котором компилируется каждый метод. В нашем примере код компилировался на уровнях 3 или 4, хотя до настоящего момента мы обсудили только два компилятора (плюс интерпретатор). Оказывается,

существуют целых 5 уровней компиляции, потому что компилятор `C1` поддерживает три уровня:

- `0` Интерпретируемый код.
- `1` Простой компилируемый код `C1`.
- `2` Ограниченный компилируемый код `C1`.
- `3` Полный компилируемый код `C1`.
- `4` Компилируемый код `C2`.

Типичный журнал компиляции показывает, что большинство методов сначала компилируется на уровне 3: полная компиляция `C1`. (Конечно, все методы начинают с уровня 0, но в журнале это не показано.) Если метод выполняется достаточно часто, он будет откомпилирован на уровне 4 (а код уровня 3 будет недействительным). Это самый частый путь: компилятор `C1` откладывает компиляцию до того момента, когда у него появится информация об использовании кода, чтобы воспользоваться ею для применения оптимизации.

Если очередь компилятора `C2` заполнена, методы будут извлекаться из очереди `C2` и компилироваться на уровне 2. Таким образом обеспечивается более быстрая компиляция метода; позднее метод будет откомпилирован на уровне 3, когда компилятор `C1` соберет профильную информацию, и наконец, откомпилирован на уровне 4, когда очередь компилятора `C2` станет менее занятой.

С другой стороны, если очередь компилятора `C1` заполнена, то метод, запланированный для компиляции на уровне 3, может стать кандидатом для компиляции уровня 4, при этом все еще ожидая компиляции на уровне 3. В таком случае он быстро компилируется на уровне 2, а затем переводится на уровень 4.

Тривиальные методы могут начинать с уровня 2 или 3, но затем переходят на уровень 1 из-за своей тривиальности. Если компилятор `C2` по какой-либо причине не сможет откомпилировать код, он также переходит на уровень 1. И конечно, при деоптимизации код переходит на уровень 0.

Флаги частично управляют этим поведением, но ожидать заметных результатов при настройке на этом уровне оптимистично. Лучшая ситуация для производительности встречается тогда, когда методы компилируются так, как ожидается: уровень 0 → уровень 3 → уровень 4. Если методы часто компилируются на уровне 2, и при этом доступны свободные ресурсы процессора, рассмотрите возможность увеличения количества потоков компилятора; это приводит к сокращению размера очереди компилятора `C2`. Если свободные ресурсы процессора недоступны, то единственное, что можно попытаться сделать, — сократить размер приложения.

Деоптимизация

При обсуждении вывода флага `PrintCompilation` упоминались два случая деоптимизации кода компилятором. Термин «*деоптимизация*» означает, что компилятор должен «отменить» предыдущую компиляцию. В результате производительность приложения сократится — по крайней мере до того, как компилятор сможет перекомпилировать проблемный код.

Деоптимизация происходит в двух случаях: когда код становится недействительным и когда код превращается в зомби.

Недействительный код

Код может стать недействительным по двум причинам. Одна обусловлена особенностями работы классов и интерфейсов, а другая является подробностью реализации многоуровневой компиляции.

Начнем с первого случая. Вспомните, что в биржевом приложении есть интерфейс `StockPriceHistory`. В примере этот интерфейс имеет две реализации: базовую (`StockPriceHistoryImpl`) и с добавлением протоколирования (`StockPriceHistoryLogger`) к каждой операции. В коде REST используемая реализация зависит от параметра `log` в URL:

```
StockPriceHistory sph;
String log = request.getParameter("log");
if (log != null && log.equals("true")) {
    sph = new StockPriceHistoryLogger(...);
}
else {
    sph = new StockPriceHistoryImpl(...);
}
// Затем JSP обращается с вызовами:
sph.getHighPrice();
sph.getStdDev();
// И т. д.
```

При нескольких вызовах к адресу http://localhost:8080/StockServlet (без параметра `log`) компилятор видит, что реальным типом объекта `sph` является `StockPriceHistoryImpl`. Тогда он применяет встраивание кода и проводит другие оптимизации на основании этой информации.

Допустим, позднее делается вызов к http://localhost:8080/StockServlet?log=true. Теперь предположение, сделанное компилятором относительно типа объекта `sph`, становится неверным; предыдущие оптимизации более недействительны. Происходит деоптимизация, и предыдущие оптимизации отбрасываются. При

большом количестве дополнительных вызовов с включенным протоколированием JVM быстро откомпилирует этот код и создаст новые оптимизации.

В журнале компиляции для этого сценария появляются строки следующего вида:

```
 841113   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                made not entrant
 841113  937  s          net.sdo.StockPriceHistoryImpl::process (248 bytes)
                                made not entrant
1322722   25 %          net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                                made zombie
1322722  937  s          net.sdo.StockPriceHistoryImpl::process (248 bytes)
                                made zombie
```

Обратите внимание: как OSR-откомпилированный конструктор, так и методы со стандартной компиляцией стали недействительными (not entrant), а позднее они превратились в зомби.

Деоптимизация кажется чем-то скверным (по крайней мере в отношении производительности), но это не всегда так. В табл. 4.1 приведены данные о количестве операций в секунду, выполняемых REST-сервером в ситуациях с деоптимизацией.

Таблица 4.1. Производительность сервера с деоптимизацией

Сценарий	OPS
Стандартная реализация	24,4
Стандартная реализация после деоптимизации	24,4
Реализация с протоколированием	24,1
Смешанная реализация	24,3

Стандартная реализация дает 24,4 OPS. Предположим, сразу же после этого теста запускается тест, активизирующий путь `StockPriceHistoryLogger`, — ситуация, которая произвела приведенные выше примеры деоптимизации. Полный вывод `PrintCompilation` показывает, что все методы класса `StockPriceHistoryImpl` будут деоптимизированы с началом запросов к реализации с протоколированием. Но если после деоптимизации будет снова активизирован путь, использующий реализацию `StockPriceHistoryImpl`, этот код будет перекомпилирован (с несколько измененными предположениями), и вы по-прежнему увидите около 24,4 OPS (после очередного периода разогрева).

Конечно, это лучший случай. Что произойдет, если вызовы будут чередоваться таким образом, что компилятор не сможет правильно предположить, по какому

пути пойдет выполнение кода? Из-за дополнительного протоколирования путь, включающий его, достигает приблизительно 24,1 OPS на сервере. Если операции чередуются, мы получаем около 24,3 OPS: почти та величина, которую можно ожидать от сервера. Таким образом, если не считать кратковременной точки, в которой обрабатывается *ловушка* (trap), деоптимизация не оказала сколько-нибудь заметного влияния на производительность.

Второе, что может сделать код недействительным, — особенности механизма многоуровневой компиляции. При компиляции кода компилятором `C2` JVM должна заменить код, уже откомпилированный компилятором `C1`. Для этого старый код помечается как недействительный, а тот же механизм деоптимизации используется для замены вновь откомпилированного (и более эффективного) кода. Таким образом, при выполнении программы с многоуровневой компиляцией в журнале компиляции появляются методы, которые стали недействительными. Без паники: эта «деоптимизация» на самом деле значительно ускоряет выполнение кода.

Чтобы убедиться в этом, обратите внимание на значения уровня в журнале:

```
40915   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ 48
                             (156 bytes)
40923 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41418   87 %     4       net.sdo.StockPriceHistoryImpl::<init> @ 48
                             (156 bytes)
41434   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2
                             (156 bytes)
                                    made not entrant
41458 3749       4       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41469 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                    made not entrant
42772 3697       3       net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                                    made zombie
42861   84 %     3       net.sdo.StockPriceHistoryImpl::<init> @ -2
                             (156 bytes)
                                    made zombie
```

Здесь конструктор сначала OSR-компилируется на уровне 3, а затем полностью компилируется также на уровне 3. Через секунду код OSR становится пригодным для компиляции 4-го уровня, поэтому он компилируется на уровне 4, а код OSR уровня 3 стал недействительным. Тот же процесс затем применяется для стандартной компиляции, и наконец, код уровня 3 становится зомби-кодом.

Деоптимизация зомби-кода

Когда в журнале компиляции сообщается о создании зомби-кода, это означает возврат предыдущего кода, который стал недействительным. В предыдущем

примере после выполнения теста с реализацией `StockPriceHistoryLogger` код класса `StockPriceHistoryImpl` стал недействительным. При этом объекты класса `StockPriceHistoryImpl` остались. Со временем все эти объекты были уничтожены уборщиком мусора. Когда это произошло, компилятор заметил, что методы этого класса теперь могут быть помечены как зомби-код.

С точки зрения производительности это хорошо. Вспомните, что откомпилированный код хранится в кэше команд фиксированного размера; при обнаружении зомби-методов соответствующий код может быть удален из кэша команд, освобождая место для других компилируемых классов (или ограничивает объем памяти, которую JVM потребуется выделить позднее).

Возможный недостаток такого решения заключается в том, что если код класса стал зомби-кодом, а затем был заново загружен и стал интенсивно использоваться, JVM придется перекомпилировать и заново оптимизировать код. Однако именно это произошло в предыдущем сценарии, когда тест был выполнен без протоколирования, затем с протоколированием и снова без него; это не привело к заметному влиянию на производительность. В общем случае незначительные перекомпиляции зомби-кода не оказывают заметного эффекта на большинство приложений.

РЕЗЮМЕ

- Для получения информации о том, как компилируется код, установите флаг `PrintCompilation`.
- По выводу при включенном флаге `PrintCompilation` можно убедиться в том, что компиляция продолжается так, как ожидалось.
- Многоуровневая компиляция может происходить на пяти разных уровнях двух компиляторов.
- Деоптимизация — процесс, в котором JVM заменяет ранее откомпилированный код. Обычно это происходит в контексте замены кода C1 кодом C2, но также может происходить из-за изменений в профиле выполнения приложения.

Флаги компилятора высокого уровня

В этом разделе описаны другие флаги, влияющие на работу компилятора. В основном этот материал дает вам возможность еще лучше понять, как работает компилятор; обычно использовать их не рекомендуется. С другой стороны, есть еще одна причина, по которой я привожу эти описания: когда-то они применялись достаточно широко. Так что если вы сталкивались с ними и пытались понять, что они делают, этот раздел ответит на ваши вопросы.

Пороги компиляции

В этой главе несколько неопределенно описывались условия, инициировавшие компиляцию кода. Главным фактором является частота выполнения кода; после того как код будет выполнен некоторое количество раз, достигается его порог компиляции, и компилятор считает, что у него достаточно информации для компиляции кода.

Оптимизации влияют на эти пороги. И этот раздел написан для того, чтобы вы лучше поняли, как работает компилятор (а также чтобы познакомить вас с некоторыми терминами); в текущих версиях JVM оптимизация порога никогда не имеет особого смысла.

Компиляция зависит от двух счетчиков в JVM: количества вызовов метода и количества возвратов в любых циклах в методе. *Возврат* можно рассматривать как количество завершений итераций цикла — либо из-за достижения конца цикла, либо из-за выполнения такой команды, как `continue`.

При выполнении метода Java JVM проверяет сумму этих двух счетчиков и решает, подходит ли метод для компиляции. В таком случае метод ставится в очередь на компиляцию (за дополнительной информацией об очередях обращайтесь к разделу «Потоки компиляции», с. 138). Такая разновидность компиляции не имеет официального названия, но часто называется *стандартной компиляцией*.

Аналогичным образом при завершении выполнения цикла счетчик возвратов увеличивается и проверяется. Если счетчик возвратов превысил свой индивидуальный порог, цикл (а не весь метод) становится подходящим для компиляции.

Настройки влияют на эти пороги. При отключении многоуровневой компиляции стандартная компиляция инициируется значением флага `-XX:CompileThreshold=`*N*. Значение *N* по умолчанию равно 10 000. Изменение значения флага `CompileThreshold` заставит компилятор принять решение о компиляции кода ранее (или позднее) обычного. Однако следует заметить, что хотя флаг только один, порог вычисляется суммированием счетчика возвратов и счетчика входов в метод.

Рекомендации по настройке флага `CompileThreshold` встречаются достаточно часто; этот флаг используется в нескольких публикациях хронометражных тестов Java (например, часто после 8000 итераций). Некоторые приложения все еще поставляются с этим флагом, установленным по умолчанию.

Вспомните, о чем говорилось ранее: я говорил, что этот флаг работает только при отключенной многоуровневой компиляции, — а это означает, что при включенной многоуровневой компиляции (как это обычно бывает) этот флаг не делает вообще ничего. Использование этого флага в действительности является пережитком JDK 7 и более ранних времен.

Когда-то этот флаг рекомендовалось использовать по двум причинам: во-первых, его снижение улучшит время запуска для приложений, использующих компилятор `C2`, потому что код будет компилироваться быстрее (и обычно с той же эффективностью). Во-вторых, это может привести к тому, что будут откомпилированы некоторые методы, которые в противном случае никогда бы не были откомпилированы.

Последний момент интересен: если программа будет выполняться бесконечно, не означает ли это, что весь ее код будет рано или поздно откомпилирован? Нет, система работает иначе: счетчики, используемые компиляторами, увеличиваются с выполнением методов и циклов, но они также уменьшаются со временем. Периодически (а именно при достижении JVM безопасного состояния) значение каждого счетчика уменьшается.

С практической точки зрения это означает, что счетчики могут рассматриваться как относительная метрика недавнего уровня активности метода или цикла. Побочный эффект заключается в том, что часто выполняемый код может быть никогда не откомпилирован компилятором `C2`, даже если программа выполняется бесконечно. Такие методы иногда называются *теплыми* (вместо «горячих»). До многоуровневой компиляции это был один из случаев, в которых снижение порога компиляции приносило реальную пользу.

Однако в наши дни компилироваться будут даже теплые методы, хотя, вероятно, можно было бы добиться незначительного повышения эффективности, если бы методы удалось откомпилировать компилятором `C2` вместо `C1`. Практическая польза невелика, но если вас действительно интересует эта тема, попробуйте изменить флаги `-XX:Tier3InvocationThreshold=`*N* (по умолчанию 200), чтобы метод быстрее компилировался компилятором `C1`, и флаг `-XX:Tier4InvocationThreshold=`*N* (по умолчанию 5000), чтобы метод быстрее компилировался компилятором `C2`. Аналогичные флаги доступны и для порога возвратов.

РЕЗЮМЕ

- Пороги, при которых компилируются методы (или циклы), определяются настраиваемыми параметрами.
- Без многоуровневой компиляции настройка порогов имела смысл, но с многоуровневой компиляцией изменять их не рекомендуется.

Потоки компиляции

В разделе «Пороги компиляции» на с. 137 упоминалось о том, что когда метод (или цикл) становится пригодным для компиляции, он ставится в очередь на

компиляцию. Очередь обрабатывается одним или несколькими фоновыми потоками.

Эти очереди не работают по принципу «первым пришел, первым обслужен»; методы с более высокими счетчиками активизации являются приоритетными. Даже когда программа начинает выполнение с большим количеством кода, который нуждается в компиляции, приоритеты гарантируют, что самый важный код будет откомпилирован в первую очередь. (Это еще одна причина, по которой идентификаторы в выводе `PrintCompilation` могут следовать с нарушением порядка.)

Компиляторы `C1` и `C2` используют разные очереди, каждая из которых обрабатывается разными потоками (возможно, несколькими). Количество потоков определяется по сложной логарифмической формуле, подробности представлены в табл. 4.2.

Таблица 4.2. Количество потоков компиляторов C1 и C2 по умолчанию для многоуровневой компиляции

Процессоры	Потоки C1	Потоки C2
1	1	1
2	1	1
4	1	2
8	1	2
16	2	6
32	3	7
64	4	8
128	4	10

Количество потоков компилятора можно отрегулировать при помощи флага `-XX:CICompilerCount=`*N*. Он определяет общее количество потоков, которые будут использоваться JVM для обработки очереди(-ей); для многоуровневой компиляции одна треть потоков (минимум один) будет использоваться для обработки очереди компилятора `C1`, а остальные потоки (но тоже не менее одного) будут использованы для обработки очереди компилятора `C2`. Значение этого флага по умолчанию равно сумме двух столбцов из табл. 4.2. Если многоуровневая компиляция отключена, то запускается только заданное количество потоков компилятора `C2`.

Когда стоит рассматривать возможность регулировки этого значения? Поскольку значение по умолчанию определяется по количеству процессоров, это одна из тех ситуаций, при которых при выполнении со старой версией JDK 8 в контейнере Docker автоматическая оптимизация может пойти насмарку. В таких обстоятельствах можно вручную присвоить этому флагу нужное значение (используя цели из табл. 4.2 как ориентир в зависимости от количества процессоров, выделенных контейнеру Docker).

Если программа выполняется на однопроцессорной виртуальной машине, наличие только одного потока компилятора может принести незначительную пользу: доступные ресурсы процессора ограничены, а сокращение количества потоков, конкурирующих за ресурс, во многих случаях улучшит быстродействие. Тем не менее это преимущество ограничено только исходным периодом разогрева; после этого количество методов, пригодных для компиляции, не создаст конкуренции за процессор. Когда пакетное приложение с акциями было запущено на однопроцессорной машине, а количество потоков компилятора было ограничено 1, исходные вычисления выполнялись приблизительно на 10% быстрее (потому что им не приходилось так часто конкурировать за процессор). Чем больше итераций выполнялось, тем меньшим становился общий эффект этого начального ускорения, пока все горячие методы не были откомпилированы, а преимущество не исчезло полностью.

С другой стороны, большое количество потоков может легко перегрузить систему, особенно если в ней одновременно работают несколько JVM (каждая из которых запускает много потоков компиляции). Сокращение количества потоков в этом случае может повысить общую эффективность (хотя, как обычно, за это приходится расплачиваться более долгим периодом разогрева).

Аналогичным образом, при большом количестве доступных ресурсов процессора теоретически программа может выиграть — по крайней мере во время периода разогрева — от повышения количества потоков процессора. На практике добиться какого-то выигрыша будет исключительно сложно. Кроме того, при доступных ресурсах процессора лучше использовать меры, которые бы задействовали эти ресурсы на протяжении всего времени выполнения приложения (не ограничиваясь ускорением компиляции в начале).

Еще один параметр, относящийся к потокам компиляции, — значение флага `-XX:+BackgroundCompilation`, который по умолчанию равен `true`. Этот параметр означает, что очередь обрабатывается асинхронно, как было описано выше. Но этому флагу также можно присвоить значение `false`; в этом случае если метод будет пригодным для компиляции, то код, который захочет выполнить его, должен будет ожидать, пока он будет откомпилирован (вместо того, чтобы продолжать его выполнение в интерпретаторе). Фоновая компиляция также отключается при установке флага `-Xbatch`.

РЕЗЮМЕ

- Компиляция выполняется асинхронно для методов, помещенных в очередь компиляции.
- Очередь не имеет строгой упорядоченности; горячие методы компилируются ранее других методов в очереди. Это еще одна причина, по которой может быть нарушен порядок идентификаторов компиляции в журнале.

Встраивание

Одной из важнейших оптимизаций, применяемых кодом, становится встраивание (inlining) методов. Код, следующий практике объектно-ориентированного проектирования, часто содержит атрибуты, доступ к которым осуществляется через get-методы (и, возможно, set-методы):

```
public class Point {
    private int x, y;

    public void getX() { return x; }
    public void setX(int i)  { x = i; }
}
```

Затраты на вызовы таких методов достаточно велики (особенно относительно объема кода в методе). Более того, на заре существования Java в советах по повышению производительности часто не рекомендовалось применять инкапсуляцию такого рода именно из-за влияния всех этих вызовов методов на производительность. К счастью, JVM теперь применяет встраивание для таких методов. А значит, вы можете написать следующий код:

```
Point p = getPoint();
p.setX(p.getX() * 2);
```

По сути компилируемый код выполняется так:

```
Point p = getPoint();
p.x = p.x * 2;
```

Встраивание включено по умолчанию. Его можно отключить при помощи флага `-XX:-Inline`, хотя эта оптимизация настолько важна, что делать это почти никогда не следует (например, отключение встраивания сокращает производительность пакетного теста с акциями более чем на 50%). Поскольку встраивание настолько важно — а также из-за того, что в нашем распоряжении еще много

разных средств настройки, — часто встречаются рекомендации относительно настройки поведения механизма встраивания в JVM.

К сожалению, у нас почти нет информации о том, как JVM выполняет встраивание кода. Если откомпилировать JVM из исходного кода, можно построить отладочную версию с флагом `-XX:+PrintInlining`. Этот флаг предоставляет разнообразную информацию о решениях относительно встраивания, которые принимаются компилятором. Лучшее, что вы можете сделать — просмотреть данные профилирования. Если в верхней части профильных данных окажутся простые методы, которые могли бы выиграть от встраивания, поэкспериментируйте с флагами встраивания.

Главное решение относительно того, стоит ли встраивать метод, зависит от его размера и от того, насколько он горяч. JVM определяет, является ли метод горячим (то есть часто вызываемым), на основании своих внутренних вычислений; никакие настраиваемые параметры не управляют напрямую этим процессом. Если метод признается пригодным для встраивания, потому что он часто вызывается, он будет встроен только в том случае, если размер его байт-кода менее 325 байт (или размера, заданного флагом `-XX:MaxFreqInlineSize=`*N*). В противном случае он будет признан пригодным для встраивания только в том случае, если его размер менее 35 байт (или другого значения, заданного флагом `-XX:MaxInlineSize=`*N*).

Иногда встречаются рекомендации, которые предлагают увеличить значение флага `MaxInlineSize`, чтобы встраивание применялось к большему количеству методов. При этом часто упускается из виду один аспект: присваивание `MaxInlineSize` значения, превышающего 35, означает, что метод будет встроен при первом вызове. Но если метод вызывается часто — а в этом случае его производительность начинает играть намного более важную роль, — он все равно будет встроен со временем (если его размер менее 325 байт). В противном случае изменение флага `MaxInlineSize` может сократить время разогрева, необходимое для теста, но вряд ли оно окажет значительное влияние на приложение, которое выполняется достаточно долго.

РЕЗЮМЕ

- Встраивание — самая полезная оптимизация, которую может применить компилятор, особенно для объектно-ориентированного кода с хорошо инкапсулированными атрибутами.
- Настраивать флаги встраивания обычно не нужно. Рекомендации, которые предлагают это делать, часто упускают из виду связь между обычным встраиванием и встраиванием часто выполняемого кода. При анализе эффектов встраивания необходимо учитывать оба случая.

Анализ локальности

Компилятор C2 применяет агрессивные оптимизации при включенном анализе локальности (-XX:+DoEscapeAnalysis, по умолчанию true). Например, возьмем следующий класс для работы с факториалами:

```
public class Factorial {
    private BigInteger factorial;
    private int n;
    public Factorial(int n) {
        this.n = n;
    }
    public synchronized BigInteger getFactorial() {
        if (factorial == null)
            factorial = ...;
        return factorial;
    }
}
```

Для сохранения первых 100 факториалов в массиве может использоваться следующий код:

```
ArrayList<BigInteger> list = new ArrayList<BigInteger>();
for (int i = 0; i < 100; i++) {
    Factorial factorial = new Factorial(i);
    list.add(factorial.getFactorial());
}
```

Обращение к объекту factorial встречается только внутри этого цикла; никакой другой код не сможет обратиться к нему. Следовательно, JVM может выполнить с этим объектом некоторые оптимизации:

- При вызове метода getFactorial() не обязательно получать блокировку синхронизации.
- Поле n не нужно хранить в памяти; его можно хранить в регистре. Ссылка на объект factorial тоже может храниться в регистре.
- Собственно, создавать объект factorial вообще не обязательно; с таким же успехом можно отслеживать отдельные поля объекта.

Подобные оптимизации весьма нетривиальны. В данном примере они просты, но они могут встречаться и в более сложном коде. В зависимости от использования кода некоторые из этих оптимизаций применяться не будут. Анализ локальности может определить, какие из этих оптимизаций возможны, и внести необходимые изменения в откомпилированный код.

Анализ локальности включен по умолчанию. В редких случаях он может неверно оценить ситуацию. Обычно это маловероятно, и в современных JVM такие

ошибки встречаются редко. В прошлом обнаруживались ошибки с серьезными последствиями, поэтому иногда можно видеть рекомендации, предлагающие отключить анализ локальности. Вероятно, такие советы уже не актуальны, хотя с учетом всех агрессивных оптимизаций компилятора может оказаться, что отключение этого механизма сделает код более надежным. Если вы столкнетесь с такой ситуацией, лучшим вариантом будет упрощение кода: простой код лучше компилируется. (При этом сама ситуация является ошибкой, о которой следует сообщить.)

РЕЗЮМЕ

- Анализ локальности — самая сложная из всех оптимизаций, которые могут выполняться компилятором. Такие оптимизации нередко становятся причиной некорректной работы микробенчмарков.

Код для конкретного процессора

Как уже упоминалось ранее, одно из преимуществ JIT-компилятора заключается в том, что он может генерировать код для разных процессоров в зависимости от того, на какой машине он работает. Конечно, предполагается, что JVM построена с информацией о более новых процессорах.

Именно так действует компилятор для микросхем Intel. В 2011 году компания Intel представила расширение Advanced Vector Extensions (AVX2) для Sandy Bridge (и более поздних микросхем). Вскоре поддержка этих команд была реализована в JVM. Затем в 2016 году компания Intel расширила ее для включения команд AVX-512; они присутствуют в Knights Landing и более поздних микросхемах. Эти команды не поддерживаются в JDK 8, но поддерживаются в JDK 11.

Обычно вам вообще не приходится беспокоиться об этой возможности; JVM распознает процессор, на котором она работает, и выбирает подходящий набор команд. Тем не менее при таком количестве новой функциональности иногда возникают проблемы.

Поддержка команд AVX-512 впервые появилась в JDK 9, хотя и была отключена по умолчанию (вернее, сначала она была включена по умолчанию, а затем отключена). В JDK 11 эти команды были включены по умолчанию. Однако начиная с JDK 11.0.6 эти команды снова были отключены по умолчанию. Таким образом, даже в JDK 11 эта возможность все еще находится в разработке. (Кстати говоря, ситуация не уникальна для Java; разработчикам многих программ пришлось основательно потрудиться над поддержкой команд AVX-512.)

Итак, даже на новом оборудовании Intel при выполнении некоторых программ может оказаться, что старый набор команд работает намного лучше. Новый

набор команд обычно приносит пользу в научных вычислениях, которые часто выполняются в приложениях Java.

Наборы команд выбираются при помощи аргумента `-XX:UseAVX=`*N*, где *N* определяется следующим образом:

`0` Команды AVX не используются.

`1` Использование команд Intel AVX уровня 1 (для Sandy Bridge и других процессоров).

`2` Использование команд Intel AVX уровня 2 (для Haswell и других процессоров).

`3` Использование команд Intel AVX-512 (для Knights Landing и других процессоров).

Значение этого флага по умолчанию зависит от процессора, на котором работает JVM. JVM идентифицирует процессор и выбирает наибольшее поддерживаемое значение из возможных. В Java 8 уровень 3 не поддерживается, поэтому на большинстве процессоров используется значение 2. В Java 11 на новых процессорах Intel по умолчанию используется значение 3 в версиях до 11.0.5 и значение 2 в более поздних версиях.

Это одна из причин, по которым желательно использовать последние версии Java 8 и Java 11 (см. главу 1): в новейших версиях доступны важные исправления. Если вы будете использовать более новую версию Java 11 на последних процессорах Intel, попробуйте установить флаг `-XX:UseAVX=2`; во многих случаях это приведет к приросту производительности.

Для полноты стоит упомянуть, что флаг `-XX:UseSSE=`*N* поддерживает расширения Intel SSE (Streaming SIMD Extensions) с 1 по 4. Эти расширения предназначены для линейки процессоров Pentium. Настройка этого флага в 2010 году имела смысл, потому что все варианты его использования были проанализированы. В наши дни обычно можно просто положиться на то, что флаг работает надежно.

Плюсы и минусы многоуровневой компиляции

Я несколько раз упоминал о том, что при отключенной многоуровневой компиляции JVM работает несколько иначе. Стоит ли вообще отключать ее при всех ее преимуществах по производительности?

Одна из возможных причин — выполнение в условиях с ограничениями по памяти. Конечно, ваша 64-разрядная машина, вероятно, оснащена большим объемом памяти, но при этом приложение может выполняться в контейнере Docker с небольшим лимитом памяти или на облачной виртуальной машине,

на которой просто не хватает памяти. А может быть, на вашей большой машине работают десятки JVM. В таких ситуациях стоит задуматься о сокращении потребления памяти вашим приложением.

В главе 8 приводятся общие рекомендации по этому поводу, но в этом разделе будет рассмотрено влияние многоуровневой компиляции на кэш команд.

В табл. 4.3 показан результат запуска NetBeans в моей системе с парой десятков проектов, которые открываются при запуске.

Таблица 4.3. Влияние многоуровневой компиляции на кэш команд

Режим компилятора	Откомпилированные классы	Закрепленный размер кэша команд	Время запуска
`+TieredCompilation`	22 733	46,5 Мбайт	50,1 секунды
`-TieredCompilation`	5609	10,7 Мбайт	68,5 секунды

Компилятор `C1` откомпилировал вчетверо больше классов; как и следовало ожидать, ему потребовалось примерно вчетверо больше памяти для кэша команд. В абсолютном выражении экономия 34 Мбайт вряд ли на что-нибудь повлияет. С другой стороны, экономия 300 Мбайт в программе, компилирующей 200 000 классов, может стать важным фактором на некоторых платформах.

Чего мы лишаемся при отключении многоуровневой компиляции? Как видно из таблицы, тратится больше времени на запуск приложения и загрузки всех классов проекта. Но как насчет программ с длительным временем выполнения, когда можно ожидать, что все горячие точки будут откомпилированы?

В этом случае при достаточно длительном периоде разогрева с отключенной многоуровневой компиляцией время выполнения должно остаться примерно тем же. В табл. 4.4 приведены данные по нашему REST-серверу с акциями после периодов разогрева 0, 60 и 300 секунд.

Таблица 4.4. Производительность серверных приложений с многоуровневой компиляцией

Период разогрева	-XX:-TieredCompilation	-XX:+TieredCompilation
0 секунд	23,72	24,23
60 секунд	23,73	24,26
300 секунд	24,42	24,43

Период измерений составляет 60 секунд, поэтому даже при отсутствии периода разогрева у компилятора будет возможность получить достаточно информации для компиляции горячих точек; следовательно, даже при отсутствии периода разогрева особых различий не будет. (Кроме того, значительный объем кода был откомпилирован при запуске сервера.) Учтите, что в конечном итоге многоуровневая компиляция все равно сможет обеспечить небольшой выигрыш (хотя вряд ли он будет заметен). Причины для этого обсуждались при рассмотрении порогов компиляции; всегда будет небольшое количество методов, компилируемых компилятором `C1` при использовании многоуровневой компиляции, которые не будут компилироваться компилятором `C2`.

КОМПИЛЯТОР JAVAC

В контексте производительности процесс компиляции в основном определяется JIT-компилятором, встроенным в JVM. Вспомните, что код Java сначала компилируется в байт-код; для этого используется компилятор javac. В завершение этого раздела стоит сказать о нем несколько слов.

Самый важный момент: компилятор javac — за одним исключением — вообще не влияет на производительность. В частности:

- Флаг `-g` для включения дополнительной отладочной информации не влияет на производительность.
- Использование ключевого слова `final` в программах Java не приводит к созданию более быстрого компилируемого кода.
- Перекомпиляция более новой версией javac (обычно) не ускоряет работу программы.

Эти три пункта годами встречались в общих рекомендациях по повышению производительности, а потом вышел JDK 11. В JDK 11 появился новый способ выполнения конкатенации строк, который может работать быстрее предыдущих версий, но для его использования требуется перекомпиляция кода. Именно здесь скрывается исключение из правила; в общем случае использование новых возможностей никогда не должно требовать перекомпиляции в байт-коды. За дополнительной информацией обращайтесь к разделу «Строки», с. 428.

GraalVM

GraalVM — новая виртуальная машина. Конечно, она представляет средства для выполнения кода Java, но также позволяет выполнять код многих других языков. Универсальная виртуальная машина также может выполнять JavaScript,

Python, Ruby, R и традиционный байт-код JVM из Java и других языков, компилируемых в байт-коды JVM (например, Scala, Kotlin и т. д.). Graal существует в двух вариантах выпуска: полное издание с открытым кодом Community Edition (CE) и коммерческое издание Enterprise Edition (EE). Каждое издание содержит двоичные файлы, поддерживающие Java 8 или Java 11.

GraalVM вносит свой вклад в производительность JVM по двум важным направлениям. Во-первых, технология дополнений позволяет GraalVM строить полностью платформенные двоичные файлы; об этом будет рассказано позднее в этом разделе.

Во-вторых, GraalVM может работать в режиме обычной JVM, но с новой реализацией компилятора `C2`. Этот компилятор написан на Java (в отличие от традиционного компилятора `C2`, написанного на C++).

Традиционная JVM содержит версию GraalVM JIT-компилятора в зависимости от того, когда была построена JVM. Эти JIT-компиляторы взяты из CE-версии GraalVM и работают медленнее EE-версии; они также обычно отстают от версий GraalVM, которые можно загрузить напрямую.

Использование компилятора GraalVM в JVM считается экспериментальным, поэтому для того, чтобы включить его, необходимо передать флаги: `-XX:+UnlockExperimentalVMOptions`, `-XX:+EnableJVMCI` и `-XX:+UseJVMCICompiler`. По умолчанию все эти флаги равны `false`.

В табл. 4.5 указана производительность стандартного компилятора Java 11, компилятора Graal из EE-версии 19.2.1 и версии GraalVM, встроенной в Java 11 и 13.

Таблица 4.5. Производительность компилятора Graal

JVM/компилятор	OPS
JDK 11/Standard C2	20,558
JDK 11/Graal JIT	14,733
Graal 1.0.0b16	16,3
Graal 19.2.1	26,7
JDK 13/Standard C2	21,9
JDK 13/Graal JIT	26,4

Данные, как и прежде, приводятся для производительности нашего REST-сервера (хотя и на другом оборудовании, так что базовое значение составляет всего 20,5 OPS вместо 24,4).

Интересно заметить прогрессию: JDK 11 строился с относительно ранней версией компилятора Graal, так что по производительности компилятор отстает от компилятора `C2`. Компилятор Graal был усовершенствован по сравнению с ранними сборками, хотя даже последняя версия раннего доступа (1.0) по скорости уступала стандартной VM. Версии Graal в конце 2019 года (выпущенные в виде рабочей версии 19.2.1) стали работать значительно быстрее. Выпуск JDK 13 для раннего доступа содержал одну из этих поздних сборок и по производительности близок к компилятору Graal, несмотря на то что компилятор `C2` был лишь незначительно усовершенствован по сравнению с JDK 11.

Предварительная компиляция

Эта глава началась с изложения философии, лежащей в основе JIT-компиляторов. Несмотря на все преимущества, код по-прежнему требует периода разогрева перед выполнением. А если в нашей ситуации будет лучше работать модель традиционной компиляции: скажем, встроенная система без дополнительной памяти, необходимой JIT, или программа, которая завершается еще до того, как у нее будет возможность пройти период разогрева?

В этом разделе мы рассмотрим две экспериментальные функции для таких ситуаций. *Статическая компиляция* — экспериментальная возможность стандартного JDK 11, а возможность получения двоичного файла, состоящего исключительно из низкоуровневого кода, — экспериментальная функция Graal VM.

Статическая компиляция

Статическая компиляция (AOT, Ahead-of-time) появилась в JDK 9 только для Linux, но в JDK 11 она стала доступной на всех платформах. С точки зрения производительности работа все еще продолжается, но в этом разделе я расскажу о ней в общих чертах[1].

AOT-компиляция позволяет откомпилировать часть кода приложения (или весь код) перед его выполнением. Откомпилированный код превращается в общую библиотеку, которую JVM затем использует при запуске приложения. Теоретически это означает, что участие JIT не требуется, особенно при запуске приложения: ваш код должен с самого начала работать как минимум не хуже кода, откомпилированного компилятором `C1`, и при этом вам не придется дожидаться компиляции этого кода.

[1] Одним из преимуществ AOT-компиляции становится ускорение запуска, но совместное использование данных класса приложения дает — по крайней мере пока — более значительные преимущества в отношении скорости запуска, и к тому же поддерживается без ограничений; см. раздел «Совместное использование данных классов», с. 444.

На практике дело обстоит иначе: время запуска приложения сильно зависит от размера общей библиотеки (а следовательно, времени загрузки этой общей библиотеки в JVM). Это означает, что простое приложение типа «Hello, world» при использовании AOT-компиляции не будет работать быстрее (более того, оно может работать даже медленнее в зависимости от решений, принимаемых при предварительной компиляции общей библиотеки). AOT-компиляция ориентирована на программы вроде REST-сервера с относительно долгим временем запуска. В этом случае время загрузки общей библиотеки компенсируется долгим временем запуска, и AOT-компиляция окупается. Но также следует помнить, что AOT-компиляция относится к числу экспериментальных функций, и, возможно, небольшие программы тоже смогут извлекать из нее пользу.

Чтобы воспользоваться AOT-компиляцией, создайте общую библиотеку с откомпилированными классами, выбранными вами, при помощи программы `jaotc`. Затем общая библиотека загружается в JVM при помощи аргумента командной строки.

Программа `jaotc` поддерживает несколько параметров, но командная строка для построения лучшей библиотеки выглядит примерно так:

```
$ jaotc --compile-commands=/tmp/methods.txt \
    --output JavaBaseFilteredMethods.so \
    --compile-for-tiered \
    --module java.base
```

Эта команда строит откомпилированную версию модуля java.base в заданном выходном файле. AOT-компиляция может применяться к модулям, как это сделали мы, или к наборам классов.

Время загрузки общей библиотеки зависит от ее размера, который в свою очередь зависит от количества методов в библиотеке. Также можно загрузить несколько общих библиотек, полученных в результате предварительной компиляции разных частей кода. Возможно, этот способ проще в управлении, но обеспечивает ту же производительность, поэтому мы сосредоточимся на одной библиотеке.

Возможно, вам захочется предварительно компилировать все подряд, однако хорошо продуманная предварительная компиляция отдельных частей кода только улучшит общую производительность. Именно поэтому в этой рекомендации компилируется только модуль java.base.

Команды компиляции (в файле /tmp/methods.txt в данном примере) также ограничивают данные, компилируемые в общую библиотеку. Файл содержит строки следующего вида:

```
compileOnly java.net.URI.getHost()Ljava/lang/String;
```

Эта строка сообщает `jaotc`, что при компиляции класса `java.net.URI` следует включить только метод `getHost()`. Другие строки могут ссылаться на другие методы этого класса, чтобы они были включены в компиляцию; в конечном итоге в общую библиотеку будут включены только методы, перечисленные в файле.

Для построения списка команд компиляции нам понадобится список всех методов, фактически используемых в приложении. Для этого следует запустить приложение командой следующего вида:

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+LogTouchedMethods \
      -XX:+PrintTouchedMethodsAtExit <другие аргументы>
```

При завершении программы выводятся строки с именами всех методов, используемых программой, в следующем формате:

```
java/net/URI.getHost:()Ljava/lang/String;
```

Чтобы получить файл methods.txt, сохраните эти строки, вставьте в начало каждой строки директиву `compileOnly` и удалите двоеточия перед аргументами.

Классы, предварительно откомпилированные `jaotc`, используют разновидность компилятора `C1`, поэтому в программе с длительным временем выполнения они будут откомпилированы неоптимально. Последний параметр, который вам может понадобиться, — `--compile-for-tiered`. Он строит общую библиотеку так, чтобы ее методы были пригодны для компилятора `C2`.

Если вы используете AOT-компиляцию для кратковременной программы, этот аргумент можно опустить, но помните, что в нашем случае целевой областью применения является сервер. Если не разрешить предварительно откомпилированным методам становиться пригодными для компиляции `C2`, разогретая производительность сервера будет ниже возможной.

Не приходится удивляться тому, что если запустить приложение с библиотекой с включенной многоуровневой компиляцией и использовать флаг `-XX:+PrintCompilation`, вы увидите тот же прием замещения кода, который вы уже видели ранее: AOT-компиляция выглядит как еще один уровень в выводе, и вы увидите, как AOT-методы будут становиться недействительными и будут замещаться при обработке их JIT-компилятором.

После того как библиотека будет создана, ее можно будет использовать с приложением командой следующего вида:

```
$ java -XX:AOTLibrary=/path/to/JavaBaseFilteredMethods.so <другие аргументы>
```

Если вы хотите убедиться в том, что библиотека используется, включите флаг `-XX:+PrintAOT` в аргументы JVM; по умолчанию этот флаг равен `false`. По

аналогии с флагом `-XX:+PrintCompilation`, флаг `-XX:+PrintAOT` будет выдавать результат каждый раз, когда предварительно откомпилированный метод будет использоваться JVM. Типичная строка выглядит так:

```
373  105     aot[ 1]   java.util.HashSet.<init>(I)V
```

В первом столбце выводится время с момента запуска программы в миллисекундах; таким образом, до того как конструктор класса `HashSet` был загружен из общей библиотеки и начал выполняться, прошло 373 миллисекунды. Во втором столбце выводится идентификатор, присвоенный методу, а в третьем — библиотека, из которой был загружен метод. Индекс (1 в данном примере) также выводится с этим флагом:

```
18    1     loaded    /path/to/JavaBaseFilteredMethods.so  aot library
```

`JavaBaseFilteredMethods.so` — первая (и единственная) библиотека, загруженная в этом примере, поэтому ее индекс равен 1 (второй столбец), а последующие обращения к `aot` с этим индексом относятся к этой библиотеке.

Компиляция в низкоуровневый код в GraalVM

AOT-компиляция приносила пользу в относительно больших программах, но не помогала (и даже могла тормозить) в малых, быстро выполняемых программах. Дело в том, что эта функция остается экспериментальной, а ее архитектура заставляет JVM загружать общую библиотеку.

GraalVM, с другой стороны, может генерировать исполняемые файлы, состоящие из низкоуровневого кода, которые могут выполняться без JVM. Эти исполняемые файлы идеально подходят для программ с короткой продолжительностью выполнения. Запустив примеры, вы можете заметить ссылки (например, проигнорированные ошибки) на классы GraalVM: AOT-компиляция использует в качестве основы GraalVM. Эта возможность GraalVM предназначена для ранних последователей (Early Adopter); она может использоваться в рабочем коде с соответствующей лицензией, но гарантия на нее не распространяется.

GraalVM строит двоичные файлы, которые запускаются достаточно быстро, особенно в сравнении с выполняемыми программами в JVM. Однако в этом режиме GraalVM не оптимизирует код так агрессивно, как компилятор `C2`, так что при достаточно длительном выполнении приложения традиционная JVM в конечном итоге победит. В отличие от AOT-компиляции, платформенный двоичный файл GraalVM не компилирует классы с использованием компилятора `C2` во время выполнения.

Кроме того, потребление памяти программы с низкоуровневым кодом, сгенерированным GraalVM, изначально существенно ниже, чем у традиционной

JVM. Однако к тому моменту, когда программа запустится и расширит кучу, это преимущество исчезает.

Также устанавливаются ограничения для функциональности Java, которая может использоваться в программе, откомпилированной в низкоуровневый код. Некоторые из этих ограничений:

- Динамическая загрузка классов (например, вызовом `Class.forName()`).
- Завершители (finalizers).
- Java Security Manager.
- JMX и JVMTI (включая профилирование JVMTI).
- Применение отражения (reflection) часто требует специального программирования или настройки.
- Использование динамических заместителей (proxies) часто требует специальной настройки.
- Использование JNI часто требует специального программирования или настройки.

Все плюсы и минусы продемонстрированы в демонстрационной программе из проекта GraalVM, которая рекурсивно подсчитывает файлы в каталоге. При небольшом количестве файлов программа с низкоуровневым кодом, произведенная GraalVM, относительно мала и работает быстро, но когда объем работы увеличивается, а в игру вступает JIT, традиционный компилятор JVM генерирует более эффективные оптимизации и работает быстрее, как видно из табл. 4.6.

Таблица 4.6. Время подсчета файлов с низкоуровневым и JIT-компилируемым кодом

Количество файлов	Java 11.0.5	Приложение с низкоуровневым кодом
7	217 мс (36 Кбайт)	4 мс (3 Кбайт)
217	279 мс (37 Кбайт)	20 мс (6 Кбайт)
169 000	2,3 с (171 Кбайт)	2,1 с (249 Кбайт)
1,3 миллиона	19,2 с (212 Кбайт)	25,4 с (269 Кбайт)

В таблице указано время подсчета файлов; в круглых скобках указывается общее потребление памяти при запуске программы (измеренное по завершении).

Конечно, сам проект GraalVM стремительно развивается, и можно ожидать, что оптимизации в низкоуровневом коде также улучшатся со временем.

Итоги

В этой главе приведено много информации о том, как работает компилятор. Она поможет вам понять некоторые общие рекомендации, приведенные в главе 1 по поводу малых методов и простого кода, а также влияния компилятора на микробенчмарки из главы 2. Напомню некоторые из них:

- Не бойтесь малых методов (в частности, get- и set-методов), потому что они легко встраиваются. Если вам кажется, что затраты на вызов метода могут оказаться слишком высокими, теоретически вы правы (мы показали, что отказ от встраивания приводит к значительному ухудшению производительности). Но на практике это не так, потому что компилятор решает проблему.
- Код, который должен быть откомпилирован, помещается в очередь компиляции. Чем больше кода находится в очереди, тем дольше придется ожидать программе для достижения оптимальной производительности.
- Хотя размер кэша команд можно (и нужно) изменять, он по-прежнему остается ресурсом конечного размера.
- Чем проще код, тем больше оптимизаций можно с ним выполнить. Данные профилирования и анализ локальности позволят ускорить выполнение кода, но сложные циклические структуры и большие методы ограничивают их эффективность.

Наконец, если вы профилируете свой код и обнаружите, что какие-то методы неожиданно занимают начальные позиции в профиле (методы, которых там быть не должно), воспользуйтесь полученной информацией, разберитесь, что происходит в компиляторе, и позаботьтесь о том, чтобы он успешно справлялся со специфическими особенностями вашего кода.

ГЛАВА 5

Знакомство с уборкой мусора

В этой главе рассматриваются основы уборки мусора в JVM. Пожалуй, настройка уборки мусора — самая важная вещь, которую можно выполнить для повышения производительности приложений Java (не считая изменения исходного кода).

Так как производительность приложений Java сильно зависит от технологии уборки мусора, не приходится удивляться тому, что есть немало разных уборщиков. В OpenJDK входят три уборщика, пригодных для использования в рабочем коде; еще один уборщик, который считается устаревшим в JDK 11, но все еще сохраняет популярность в JDK 8; а также экспериментальные уборщики мусора, которые (в идеале) будут пригодными для использования в рабочем коде в будущих версиях. В других реализациях Java (таких, как Open J9 или Azul JVM) также существуют свои уборщики.

Характеристики производительности всех этих уборщиков мусора сильно различаются; мы сосредоточимся только на тех, которые входят в поставку OpenJDK. Каждый уборщик подробно рассматривается в следующей главе. Все они используют одни и те же базовые концепции, поэтому в этой главе будут описаны основные принципы работы уборщиков мусора.

Общие сведения об уборке мусора

Одна из самых привлекательных особенностей программирования на Java заключается в том, что разработчикам не нужно явно управлять жизненным циклом объектов: объекты создаются тогда, когда потребуется, а когда они перестают использоваться, JVM автоматически освобождает объект. Если вы, как и я, проводите много времени за оптимизацией использования памяти программами на Java, вся эта схема может показаться скорее недостатком, чем чем-то положительным (а время, которое я трачу на описания схем уборки мусора, вроде бы только подкрепляет эту позицию). Конечно, уборка мусора не является безусловным благом, но я еще помню, сколько проблем было с отслеживанием

null-указателей и висячих ссылок в других языках. Я категорически утверждаю, что настройка уборщиков мусора — задача намного более простая (и отнимающая меньше времени), чем выслеживание ошибок с указателями.

На базовом уровне задача уборщика мусора сводится к нахождению используемых объектов и освобождению памяти, связанной с остальными объектами (неиспользуемыми). Иногда этот процесс описывается как поиск объектов, на которые больше не осталось ссылок (предполагается, что ссылки отслеживаются по счетчику). Однако подсчет ссылок такого рода неэффективен. Для заданного списка объектов на каждый объект в списке (кроме первого) существует указатель из другого объекта в списке — но если ничто не указывает на начальный элемент списка, то есть список не используется, а занимаемая им память может быть освобождена. А если список является циклическим (например, последний элемент списка указывает на начальный), то на каждый объект в списке существует ссылка — при этом ни один объект в списке использоваться не может, потому что никакие объекты не содержат ссылок на сам список.

Таким образом, динамическое отслеживание ссылок по счетчикам не подходит; вместо этого JVM должна периодически искать в куче неиспользуемые объекты. Для этого она начинает с корневых объектов GC — то есть объектов, достижимых за пределами кучи. Прежде всего это стеки потоков и системные классы. Эти объекты достижимы всегда, поэтому алгоритм GC перебирает все объекты, доступные из одного из корневых объектов. Объекты, достижимые из корневых объектов GC, являются *живыми*; остальные недостижимые объекты относятся к мусору (даже если они содержат ссылки на живые объекты или друг на друга).

Когда алгоритм GC находит неиспользуемые объекты, JVM может освободить память, занимаемую этими объектами, и использовать ее для создания дополнительных объектов. Тем не менее обычно оказывается неэффективно просто отслеживать всю свободную память и использовать ее для будущего выделения памяти; в какой-то момент память нужно будет уплотнить для предотвращения фрагментации.

Представьте программу, которая создает массив на 1000 байт, затем другой массив на 24 байта и повторяет этот процесс в цикле. Когда этот процесс заполнит кучу, структура памяти будет выглядеть так, как показано в верхней строке рис. 5.1; куча заполнена, а размеры блоков памяти с массивами чередуются.

Когда куча заполнится, JVM освободит неиспользуемые массивы. Допустим, все 24-байтовые массивы не используются, а 1000-байтовые массивы все еще нужны; мы приходим ко второй строке на рис. 5.1. Куча содержит свободные области, но не сможет выделить блок памяти размером более 24 байт — если только JVM не сместит все 1000-байтовые массивы так, чтобы они размещались вплотную друг к другу, а вся свободная память образует один блок, из которого она сможет выделяться по мере надобности (третья строка на рис. 5.1).

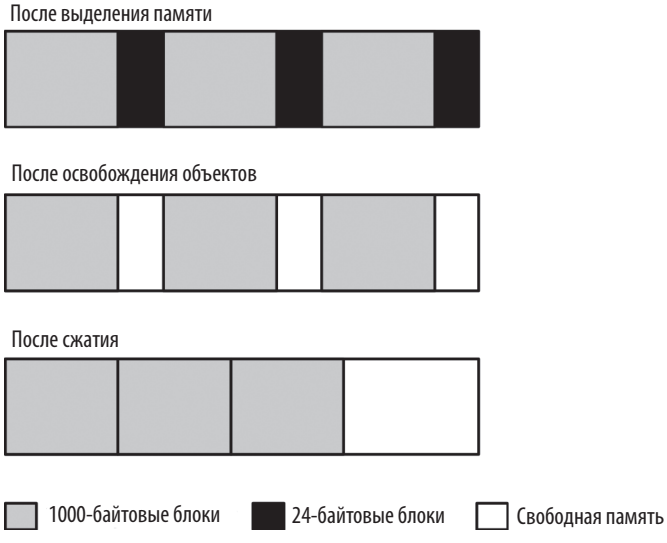


Рис. 5.1. Идеализированное представление кучи в ходе уборки мусора

Конкретные реализации устроены сложнее, но производительность уборщика мусора определяется следующими базовыми операциями: поиск неиспользуемых объектов, освобождение занимаемой ими памяти и сжатие (дефрагментация) кучи. Разные уборщики мусора по-разному подходят к выполнению этих операций, особенно сжатия: одни алгоритмы откладывают сжатие до того момента, когда оно станет абсолютно необходимым, другие сжимают большие блоки кучи, третьи сжимают кучу перемещением небольших блоков памяти. Различия в подходах объясняют, почему разные алгоритмы обладают разными характеристиками производительности.

Эти операции проще выполнять, если никакие потоки приложения не выполняются во время работы уборщика мусора. Программы Java обычно имеют многопоточную структуру, а уборщик мусора часто запускает несколько потоков. В этом обсуждении рассматриваются две логические группы потоков: потоки, выполняющие логику приложения (часто называемые *модификаторами*, потому что они модифицируют объекты как часть логики приложения), и потоки, выполняющие уборку мусора (потоки GC). Когда потоки GC отслеживают ссылки на объекты или перемещают объекты в памяти, они должны позаботиться о том, чтобы эти объекты не использовались потоками приложений. Это относится в первую очередь к перемещению объектов в памяти уборщиком мусора: положение объекта в памяти изменяется в ходе этой операции, а следовательно, никакие потоки приложений не должны обращаться к этим объектам.

Паузы, возникающие при приостановке потоков приложения, называются *глобальными паузами*. Эти паузы обычно оказывают наибольшее влияние на производительность приложения, и минимизация таких пауз становится одним из важных факторов оптимизации GC.

Уборщики мусора с учетом поколений

Несмотря на возможные расхождения в реализациях, работа большинства уборщиков мусора основана на разбиении кучи на *поколения*: *старое* и *молодое*. Молодое поколение далее делится на части: *Эдем* и *область выживших* (хотя иногда термином «Эдем» ошибочно обозначают все молодое поколение).

Определение разных поколений обосновывается тем, что многие объекты используются в течение очень коротких периодов времени. Возьмем, например, цикл в вычислении цен акций, который суммирует квадраты разности текущей цены от средней (часть вычисления стандартного отклонения):

```
sum = new BigDecimal(0);
for (StockPrice sp : prices.values()) {
    BigDecimal diff = sp.getClosingPrice().subtract(averagePrice);
    diff = diff.multiply(diff);
    sum = sum.add(diff);
}
```

Как и многие классы Java, класс `BigDecimal` является неизменяемым: объект представляет конкретное число, и изменить его не удастся. При выполнении арифметических операций с объектом создается новый объект (и часто предыдущий объект с предыдущим значением теряется). При выполнении простого цикла для данных биржевых котировок за год (приблизительно 250 итераций) создаются 750 объектов `BigDecimal` для хранения промежуточных значений только в этом цикле. Эти объекты теряются при следующей итерации цикла. Внутри `add()` и других методов код библиотеки JDK создает еще больше промежуточных объектов `BigDecimal` (и других). В конечном итоге малый объем кода создает и быстро уничтожает множество объектов.

Подобные операции типичны для Java, поэтому уборщик мусора проектировался с учетом того обстоятельства, что многие объекты (а иногда их большинство) используются только временно. Именно здесь в игру вступает архитектура с поколениями. Объекты сначала создаются в молодом поколении, который составляет подмножество всей кучи. Когда молодое поколение заполняется, уборщик мусора останавливает все потоки приложения и уничтожает молодое поколение. Неиспользуемые объекты уничтожаются, а объекты, которые продолжают использоваться, перемещаются в другое место. Эта операция называется *уборкой мусора в молодом поколении*, или *малой уборкой мусора*.

Такая архитектура обладает двумя преимуществами с точки зрения производительности. Во-первых, поскольку молодое поколение составляет только часть всей кучи, оно обрабатывается быстрее, чем вся куча. Потоки приложения останавливаются на гораздо более короткий период времени, чем они бы останавливались при одновременной обработке всей кучи. С другой стороны, это также означает, что потоки приложения останавливаются чаще, чем они бы приостанавливались, если бы JVM выполняла уборку мусора только при заполнении всей кучи; этот компромисс будет более подробно рассмотрен позднее в этой главе. А пока скажу, что более короткие паузы почти всегда оказываются большим преимуществом, даже если они происходят чаще.

Второе преимущество следует из особенностей создания объектов в молодом поколении. Объекты создаются в Эдеме (который включает большую часть молодого поколения). Когда молодое поколение очищается в ходе уборки мусора, все объекты в Эдеме либо перемещаются, либо уничтожаются: неиспользуемые объекты могут быть уничтожены, а используемые объекты либо перемещаются в одну из областей для выживших, либо в старое поколение. Так как все выжившие объекты переместились, молодое поколение автоматически сжимается при уборке: в конце уборки Эдем и одна из областей выживших объектов пусты, а объекты, оставшиеся в молодом поколении, сжимаются в другой области выживших.

Распространенные алгоритмы уборки мусора делают глобальные паузы в ходе уборки молодого поколения.

При перемещении объектов в старое поколение со временем оно также будет заполнено, и JVM нужно будет найти в старом поколении любые объекты, которые более не используются, и уничтожить их. Здесь проявляются самые значительные различия между алгоритмами уборки мусора. Самые простые алгоритмы останавливают все потоки приложения, находят неиспользуемые объекты, освобождают их память, а затем сжимают кучу. Этот процесс называется *полной уборкой мусора*, и обычно он создает относительно долгую паузу в работе потоков приложения.

С другой стороны, возможно — хотя и сложнее с вычислительной точки зрения — найти неиспользуемые объекты во время работы потоков приложения. Так как фаза, в которой сканируются неиспользуемые объекты, может отрабатывать без остановки потоков приложения, эти алгоритмы называются *конкурентными* (concurrent) *уборщиками мусора*. Также они называются *непрерывными* (что, строго говоря, неверно), потому что они сводят к минимуму необходимость приостановки всех потоков приложения. Конкурентные уборщики мусора также по-разному подходят к сжатию старого поколения.

При использовании конкурентного уборщика приложение обычно сталкивается с меньшим количеством пауз (которые к тому же становятся более короткими). С другой стороны, при этом приложение в целом использует больше ресурсов

процессора. Конкурентных уборщиков также труднее настраивать для обеспечения оптимального быстродействия (хотя в JDK 11 настраивать конкурентных уборщиков — таких, как G1 GC, — намного проще, чем в предыдущих версиях, что объясняется техническим прогрессом с момента появления первых конкурентных уборщиков мусора).

Размышляя о том, какой уборщик мусора лучше подойдет для вашей ситуации, подумайте об общих целях производительности, к соблюдению которых вы стремитесь. В любой ситуации существуют компромиссы. При измерении времени отклика для отдельных запросов в приложении (таком, как REST-сервер) следует учитывать следующие обстоятельства:

- На отдельные запросы будут влиять паузы — и что еще важнее, более долгие паузы для полной уборки мусора. Если вашей целью является минимизация последствий пауз на время отклика, конкурентный уборщик мусора может оказаться более подходящим.
- Если среднее время отклика важнее выбросов (то есть 90%-ного времени отклика), возможно, с неконкурентным уборщиком мусора вы добьетесь лучших результатов.
- Преимущества от предотвращения долгих пауз с конкурентным уборщиком мусора сопряжены с затратами на повышенную загрузку процессора. Если на вашей машине не хватает свободных ресурсов процессора, необходимых для конкурентного уборщика мусора, возможно, неконкурентный уборщик будет лучшим вариантом.

Для выбора уборщика мусора в пакетном приложении следует руководствоваться следующими соображениями:

- Если на машине имеется достаточное количество ресурсов процессора, использование конкурентного уборщика мусора для предотвращения полных пауз GC позволит заданию завершиться быстрее.
- Если ресурсы процессора ограничены, из-за дополнительного потребления процессора конкурентным уборщиком мусора пакетное задание завершится быстрее.

РЕЗЮМЕ

- Алгоритмы GC обычно делят кучу на старое и молодое поколение.
- Алгоритмы GC обычно используют глобальные паузы для уничтожения объектов в молодом поколении; обычно эта операция выполняется достаточно быстро.
- Минимизация последствий от выполнения уборки мусора в старом поколении подразумевает поиск баланса между продолжительностью пауз и загрузкой процессора.

Алгоритмы уборки мусора

OpenJDK 12 предоставляет различные алгоритмы уборки мусора с разной степенью поддержки в более ранних версиях. В табл. 5.1 перечислены эти алгоритмы с указанием их статуса в выпусках OpenJDK и Oracle.

Таблица 5.1. Уровень поддержки различных алгоритмов уборки мусора

Алгоритм уборки мусора	Поддержка в JDK 8	Поддержка в JDK 11	Поддержка в JDK 12
Последовательная уборка мусора	S	S	S
Параллельная уборка мусора	S	S	S
Уборщик мусора G1	S	S	S
Конкурентный алгоритм пометки с уничтожением (CMS)	S	D	D
ZGC	–	E	E
Shenandoah	E2	E2	E2
Epsilon GC	–	E	E

S — полная поддержка; D — считается устаревшим; E — экспериментальная поддержка; E2 — экспериментальная поддержка в сборках OpenJDK, но не в сборках Oracle.

Ниже приводятся краткие описания всех алгоритмов; в главе 6 приведена более подробная информация об их настройке.

Последовательный уборщик мусора

Последовательный уборщик мусора — простейший из всех. Он используется по умолчанию, если приложение работает на машине клиентского уровня (32-разрядные JVM для Windows) или на однопроцессорной машине. Когда-то казалось, что последовательному уборщику мусора пора на свалку, но контейнеризация изменила ситуацию: с появлением одноядерных контейнеров Docker и виртуальных машин (даже с гиперпоточным ядром, которое воспринимается как два процессора) этот алгоритм снова стал актуальным.

Последовательный уборщик мусора использует один поток для обработки кучи. Он останавливает все потоки приложений на время обработки кучи (для малой или полной уборки мусора). Во время полной уборки мусора происходит полное сжатие старого поколения.

Последовательный уборщик мусора включается при помощи флага `-XX:+UseSerialGC` (хотя обычно он используется по умолчанию в тех случаях, в которых он может использоваться). Учтите, что в отличие от многих флагов JVM последовательный уборщик мусора не отключается заменой знака + на знак - (то есть с флагом `-XX:-UseSerialGC`). В тех системах, в которых последовательный уборщик мусора используется по умолчанию, он отключается выбором другого алгоритма уборки мусора.

Параллельный уборщик мусора

В JDK 8 параллельный уборщик мусора используется по умолчанию на всех 64-разрядных машинах с двумя и более процессорами. Параллельный уборщик мусора использует несколько потоков для уничтожения молодого поколения, вследствие чего малая уборка мусора работает намного быстрее при использовании последовательного уборщика мусора. Алгоритм также использует несколько потоков для обработки старого поколения — собственно, именно по этой причине он называется *параллельным*.

Параллельный уборщик мусора останавливает все потоки приложения на время малой и полной уборки мусора и полностью сжимает старое поколение во время полной уборки. Так как он используется по умолчанию во многих ситуациях, в которых уместно его использование, явно включать его не нужно. Чтобы включить его тогда, когда это потребуется, используйте флаг `-XX:+UseParallelGC`.

Учтите, что в старых версиях JVM параллельная уборка мусора включалась для старых и новых поколений по отдельности, поэтому иногда встречаются упоминания флага `-XX:+UseParallelOldGC`. Этот флаг считается устаревшим (хотя и продолжает функционировать, и его можно отключить для параллельной уборки только молодого поколения, если по какой-то причине возникнет такая необходимость).

Уборщик мусора G1

Уборщик мусора G1 использует стратегию конкурентной уборки мусора для очистки кучи с минимальными паузами. Этот уборщик используется по умолчанию в JDK 11 и более поздних версиях для 64-разрядных JVM на машинах, оснащенных двумя и более процессорами.

Уборщик мусора G1 делит кучу на области, но при этом рассматривает кучу как разделенную на два поколения. Некоторые области формируют молодое поколение, при уборке которого приостанавливаются все потоки приложения, а все живые объекты перемещаются в старое поколение или области выживших объектов (для чего используются множественные потоки).

В уборщике мусора G1 старое поколение обрабатывается фоновыми потоками, которым не нужно останавливать потоки приложения для выполнения большей части своей работы. Так как старое поколение делится на области, уборщик мусора G1 может освобождать объекты из старого поколения, копируя их из одной области в другую; это означает, что он (по крайней мере частично) сжимает кучу в ходе нормальной обработки. Это способствует предотвращению фрагментации куч G1, хотя полностью исключить ее нельзя.

За предотвращение полных циклов уборки мусора приходится расплачиваться процессорным временем: (множественные) фоновые потоки, которые уборщик мусора G1 использует для обработки старого поколения, должны располагать процессорным временем на время выполнения потоков приложения.

Уборщик мусора G1 включается при помощи флага `-XX:+UseG1GC`. В большинстве случаев он используется по умолчанию в JDK 11, но также сохраняет функциональность в JDK 8 — особенно в поздних сборках JDK 8, содержащих многие важные исправления ошибок и улучшения производительности, которые были перенесены из более поздних выпусков. Как можно увидеть при углубленном анализе уборщика G1, в JDK 8 не поддерживается один важный аспект производительности, из-за которого данный механизм может стать непригодным для этого выпуска.

Уборщик мусора CMS

Уборщик мусора CMS стал первым конкурентным уборщиком. Как и другие алгоритмы, CMS останавливает все потоки приложений в ходе малой уборки мусора, которую он проводит в нескольких потоках.

CMS официально считается устаревшим в JDK 11 и выше, а использовать его в JDK 8 не рекомендуется. С практической точки зрения главный недостаток CMS заключается в том, что он не может сжимать кучу в ходе фоновой обработки. Если куча фрагментируется (что с большой вероятностью произойдет в какой-то момент), CMS приходится остановить все потоки приложения и сжать кучу, что противоречит самой цели конкурентного уборщика. С учетом этого обстоятельства и появлением уборщика G1 использовать CMS более не рекомендуется.

CMS включается флагом `-XX:+UseConcMarkSweepGC`, который по умолчанию равен `false`. Традиционно CMS также требовал установки флага `-XX:+UseParNewGC` (в противном случае уборка в молодом поколении будет выполняться одним потоком), хотя этот флаг считается устаревшим.

Экспериментальные уборщики мусора

Уборка мусора остается благодатной почвой для разработчиков JVM, а более новые версии Java поставляются с тремя экспериментальными алгоритмами, упоминавшимися ранее. Я подробнее расскажу о них в следующей главе; а пока

выясним, как выбрать между тремя уборщиками мусора, поддерживаемыми в средах реальной эксплуатации.

ВКЛЮЧЕНИЕ И ВЫКЛЮЧЕНИЕ ПРИНУДИТЕЛЬНОЙ УБОРКИ МУСОРА

Уборка мусора обычно происходит тогда, когда JVM считает, что она необходима: малая уборка мусора инициируется при заполнении нового поколения, полная — при заполнении старого поколения, а конкурентная (если она применима) — когда куча близка к заполнению.

Java предоставляет механизм, при помощи которого приложения могут инициировать принудительную уборку мусора: метод `System.gc()`. Вызывать этот метод почти всегда нежелательно. Вызов всегда запускает полную уборку мусора (даже если JVM работает с уборщиком G1 или CMS), поэтому потоки приложения будут остановлены на относительно длительный период времени. Вызов этого метода не сделает приложение более эффективным; он заставит уборку мусора выполниться ранее, чем она могла бы произойти, но в действительности проблема не исчезает, а всего лишь перемещается в другое место.

У каждого правила есть исключения, особенно при мониторинге производительности или хронометражных тестах. Для небольших хронометражных тестов, которые выполняют код для разогрева JVM, выполнение принудительной уборки мусора перед циклом измерений может иметь смысл. (Так поступает jmh, хотя обычно это не обязательно.) Аналогичным образом, при анализе кучи обычно стоит провести полную уборку мусора перед получением дампа. Многие способы получения дампа кучи все равно проводят полную уборку мусора, но ее также можно инициировать другими способами: выполнить команду `jcmd <идентификатор_процесса> GC.run` или подключиться к JVM при помощи jconsole и щелкнуть на кнопке Perform GC на панели Memory.

Другим исключением является механизм RMI (Remote Method Invocation), который вызывает `System.gc()` каждый час в процессе работы распределенного уборщика мусора. Периодичность вызова можно изменить, присвоив другие значения двум системным свойствам: `-Dsun.rmi.dgc.server.gcInterval=N` и `-Dsun.rmi.dgc.client.gcInterval=N`. Значения `N` задаются в миллисекундах, а значение по умолчанию равно 3 600 000 (один час).

Если вы выполняете сторонний код, который вызывает метод `System.gc()`, а для вас это нежелательно, уборку мусора можно предотвратить включением флага `-XX:+DisableExplicitGC` в аргументы JVM; по умолчанию этот флаг равен false. Такие приложения, как серверы Java EE, часто включают этот аргумент, для того чтобы вызовы RMI GC не мешали их работе.

РЕЗЮМЕ

- Поддерживаемые алгоритмы уборки мусора принимают разные меры для достижения своей цели — минимизации воздействия уборки мусора на приложение.
- Последовательный уборщик мусора имеет смысл (и используется по умолчанию), когда на машине доступен только один процессор, а дополнительные потоки уборки мусора помешают работе приложения.
- Параллельный уборщик мусора используется по умолчанию в JDK 8; он максимизирует общую эффективность приложения, но может стать причиной долгих пауз в отдельных операциях.
- Уборщик мусора G1 используется по умолчанию в JDK 11 и выше; он проводит конкурентную чистку старого поколения во время выполнения потоков приложения, теоретически избегая полной уборки мусора. Такая архитектура снижает вероятность полной уборки мусора по сравнению с CMS.
- Уборщик мусора CMS может одновременно чистить старое поколение во время выполнения потоков приложения. Если у процессора хватает ресурсов для фонового выполнения, алгоритм может избежать полных циклов уборки мусора в приложении. Сейчас он считается устаревшим, а вместо него рекомендуется использовать G1.

Выбор алгоритма уборки мусора

Выбор алгоритма уборки мусора зависит от нескольких факторов: от имеющегося оборудования, от специфики приложения, от целей приложения по производительности. В JDK 11 уборщик мусора G1 часто оказывается наилучшим вариантом; в JDK 8 выбор зависит от приложения.

Для начала будем условно считать, что алгоритм G1 является более подходящим вариантом, но у каждого правила есть свои исключения. В случае уборки мусора эти исключения связаны с объемом ресурсов процессора, необходимых приложению, относительно доступного оборудования и вычислительными ресурсами, необходимыми потокам G1 для фоновой обработки.

Если вы используете JDK 8, способность алгоритма G1 избегать полной уборки мусора также станет ключевым фактором. Если уборщик мусора G1 не является лучшим вариантом, выбор между параллельными и последовательными уборщиками мусора зависит от количества процессоров на машине.

Когда используется (и не используется) последовательный уборщик мусора

На машине с одним процессором JVM по умолчанию использует последовательный алгоритм уборки мусора. В эту категорию входят виртуальные машины с одним процессором и контейнеры Docker, ограниченные одним процессором. Если ограничить контейнер Docker одним процессором в ранних версиях JDK 8, по умолчанию все равно будет использоваться параллельный уборщик мусора. В такой среде стоит подумать об использовании последовательного уборщика (даже при том, что вам придется включать его вручную).

В таких средах последовательный уборщик мусора обычно является хорошим вариантом, но в отдельных случаях уборщик G1 обеспечивает лучшие результаты. Этот пример также станет хорошей отправной точкой для понимания общих компромиссов, связанных с выбором алгоритма уборки мусора.

Решение о выборе G1 и других уборщиков зависит от наличия свободных ресурсов процессора для фоновых потоков G1, поэтому начнем с пакетного задания, создающего интенсивную нагрузку на процессор. В пакетном задании процессор будет занят на 100% в течение долгого времени, и в таких случаях последовательный уборщик имеет явное преимущество.

В табл. 5.2 приведено время, необходимое одному потоку для вычисления истории котировок для 100 000 котировок за трехлетний период.

Таблица 5.2. Время выполнения разных алгоритмов уборки мусора на одном процессоре

Алгоритм	Затраченное время	Время приостановки для уборки мусора
Последовательный	434 секунды	79 секунд
Параллельный	503 секунды	144 секунды
G1	501 секунда	97 секунд

Преимущество однопоточной уборки мусора наиболее заметно проявляется при сравнении последовательного уборщика с параллельным. Время, потраченное на фактические вычисления, равно общему затраченному времени за вычетом времени уборки мусора. С последовательным и параллельным уборщиком это время приблизительно одинаково (около 355 секунд), но последовательный уборщик побеждает, потому что он проводит намного меньше времени в состоянии остановки для уборки мусора. В частности, последовательный уборщик тратит в среднем 505 мс на полную уборку, тогда как параллельному уборщику

хватает 1392 мс. Алгоритм параллельного уборщика сопряжен с немалым объемом дополнительных затрат — эти затраты оправданны, если куча обрабатывается одним или несколькими потоками, но если доступен только один поток, они только снижают эффективность алгоритма.

Теперь сравним последовательного уборщика мусора с уборщиком G1. Если устранить паузу при работе с уборщиком G1, приложению потребуются 404 секунды на вычисления — но из других примеров мы знаем, что оно должно занимать только 355 секунд. Откуда взялись еще 49 секунд?

Поток вычислений может использовать все доступное процессорное время. В то же время фоновым потокам G1 необходимы ресурсы процесса для их работы. Так как ресурсов процессора не хватает для удовлетворения обеих потребностей, они будут совместно использовать процессор: какое-то время работает поток вычислений, затем какое-то время работает фоновый поток G1. Общий результат заключается в том, что поток вычислений не может работать в течение 49 секунд, потому что процессор занят «фоновым» потоком уборки мусора G1.

Вот что я имею в виду, говоря, что при выборе уборщика G1 необходимо иметь достаточно ресурсов процессора для работы его фоновых потоков. Когда подолгу работающий поток приложения занимает единственный доступный процессор, алгоритм G1 не может считаться хорошим вариантом. Но как насчет другой конфигурации — скажем, микросервисы, выполняющие простые REST-запросы на оборудовании с ограниченной мощностью? В табл. 5.3 приведено время выполнения для веб-сервера, обрабатывающего приблизительно 11 запросов в секунду на своем единственном процессоре, что занимает приблизительно 50% доступных ресурсов процессора.

Таблица 5.3. Время отклика для одного процессора с разными алгоритмами уборки мусора

Алгоритм уборки мусора	Среднее время отклика	90%-ное время отклика	99%-ное время отклика	Загрузка процессора
Последовательный	0,10 секунды	0,18 секунды	0,69 секунды	53%
Параллельный	0,16 секунды	0,18 секунды	1,40 секунды	49%
G1	0,13 секунды	0,28 секунды	0,40 секунды	48%

Алгоритм по умолчанию (последовательный) все еще обеспечивает наилучшее среднее время на 30%. И снова дело в том, что уборка молодого поколения последовательным уборщиком обычно происходит быстрее, чем у любых других алгоритмов, так что средний запрос сталкивается с меньшими задержками.

Некоторые невезучие запросы прерываются полной уборкой мусора, выполняемой последовательным уборщиком. В нашем эксперименте среднее время полной уборки мусора последовательным уборщиком составило 592 мс, а в отдельных случаях оно достигало 730 мс. В результате для 1% запросов время отклика составило почти 700 мс.

И все равно это лучше того, на что способен параллельный уборщик мусора. Полная уборка мусора параллельным уборщиком в среднем занимала 1192 мс, а максимум достигал 1510 мс. Таким образом, 99%-ное время отклика параллельного уборщика вдвое превышало соответствующий показатель для последовательного уборщика. Эти выбросы также искажали среднее время.

Уборщик мусора G1 находится где-то в середине. В отношении среднего времени отклика он уступает последовательному уборщику, потому что более простой последовательный алгоритм работает быстрее. В данном случае это относится прежде всего к малой уборке мусора, которая занимала в среднем 86 мс для последовательного уборщика, но достигала 141 мс для G1. Таким образом, средний запрос будет сталкиваться с большими задержками в случае G1.

При этом уборщик G1 обладает 99%-ным временем отклика, существенно меньшим, чем у последовательного уборщика. В нашем примере уборщик G1 смог избежать полной уборки мусора, поэтому он не сталкивался с более чем 500-миллисекундными задержками последовательного уборщика.

Здесь необходимо выбрать то, что же вы собираетесь оптимизировать: если среднее время отклика является самой важной целью, последовательный (используемый по умолчанию) уборщик мусора становится лучшим вариантом. Если вы хотите провести оптимизацию для 99%-ного времени отклика, уборщик G1 побеждает. В принципе это субъективно, но для меня 30-миллисекундные различия в среднем времени не настолько важны, как 300-миллисекундные различия в 99% времени — таким образом, в данном случае уборщик мусора G1 становится предпочтительным перед уборщиком по умолчанию для платформы.

В этом примере выполняется интенсивная уборка мусора; в частности, неконкурентным уборщикам приходится выполнять значительный объем операций полной уборки мусора. Если настроить тест так, чтобы все объекты могли уничтожаться без необходимости полной уборки мусора, последовательный алгоритм может конкурировать с G1, как видно из табл. 5.4.

При отсутствии полной уборки мусора преимущества последовательного уборщика перед G1 исчезают. При незначительной активности уборки мусора все показатели лежат в одном диапазоне, а все уборщики работают с приблизительно равной эффективностью. С другой стороны, отсутствие полной уборки мусора довольно маловероятно. При наличии достаточных ресурсов процессора

уборщик G1 обычно будет давать лучший результат даже в том случае, если последовательный уборщик используется по умолчанию.

Таблица 5.4. Время отклика для одного процессора с разными алгоритмами уборки мусора (без полной уборки мусора)

Алгоритм уборки мусора	Среднее время отклика	90%-ное время отклика	99%-ное время отклика	Загрузка процессора
Последовательный	0,05 секунды	0,08 секунды	0,11 секунды	53%
Параллельный	0,08 секунды	0,09 секунды	0,13 секунды	49%
G1	0,05 секунды	0,08 секунды	0,11 секунды	52%

Один гиперпоточный процессор

Как насчет одноядерной машины или контейнера Docker с гиперпоточным процессором (который выглядит для JVM как двухпроцессорная машина)? В этом случае JVM не будет использовать последовательную уборку мусора по умолчанию — она считает, что на машине два процессора, поэтому по умолчанию будет использоваться параллельный уборщик в JDK 8 и уборщик G1 в JDK 11. Но, как оказалось, последовательный уборщик мусора часто обладает преимуществами и на таком оборудовании. В табл. 5.5 показано, что происходит при проведении описанного выше пакетного эксперимента на одном гиперпоточном процессоре.

Таблица 5.5. Время выполнения разных алгоритмов уборки мусора на одном гиперпоточном процессоре

Алгоритм	Затраченное время	Время приостановки для уборки мусора
Последовательный	432 секунды	82 секунды
Параллельный	478 секунд	117 секунд
G1	476 секунд	72 секунды

Последовательный уборщик не работает в нескольких потоках, поэтому его временные показатели практически не изменились по сравнению с предыдущим тестом. У других алгоритмов показатели улучшились, но не настолько, насколько можно было бы надеяться, — параллельный уборщик запускает два потока, но вместо сокращения наполовину время пауз сократилось приблизительно на 20%. Аналогичным образом уборщик G1 все еще не может получить достаточных ресурсов процессора для своих фоновых потоков.

Итак, по крайней мере в этом случае — пакетное задание, выполняемое относительно долго и с частой уборкой мусора, — выбор JVM по умолчанию будет неправильным, а приложению лучше использовать последовательного уборщика, несмотря на присутствие «двух» процессоров. Если бы на машине были доступны два реальных процессора (то есть два ядра), ситуация была бы иной. Работа параллельного уборщика заняла бы всего 72 секунды, что меньше времени, необходимого последовательному уборщику. На этой стадии эффективность последовательного уборщика исчезает, поэтому мы исключим его из будущих примеров.

Другое обстоятельство, касающееся последовательного уборщика: приложение с очень малым размером кучи (скажем, 100 Мбайт) с последовательным уборщиком все равно будет работать эффективнее, несмотря на количество доступных ядер.

Когда используется параллельный уборщик мусора

Если на машине доступно несколько процессоров, между алгоритмами уборки мусора могут возникать более сложные взаимодействия, но на простейшем уровне плюсы и минусы G1 и параллельного уборщика остаются теми же, которые вам уже известны. Например, табл. 5.6 показывает, как наше приложение работает при выполнении двух или четырех потоков приложения на машине с четырьмя ядрами (когда ядра не являются гиперпоточными).

Таблица 5.6. Время пакетной обработки с разными алгоритмами уборки мусора

Потоки приложения	Алгоритм G1	Параллельный алгоритм
Два	410 секунд (60,8%)	446 секунд (59,7%)
Четыре	513 секунд (99,5%)	536 секунд (99,%)

В таблице приведено количество секунд, необходимое для выполнения теста, и загрузка процессора (в круглых скобках). С двумя потоками приложения алгоритм G1 работает значительно быстрее параллельного алгоритма. Главная причина заключается в том, что с параллельным уборщиком приложение проводит 35 секунд приостановленным для полной уборки мусора. Алгоритм G1 может избежать этой уборки мусора ценой (относительно небольшого) увеличения затрат процессорного времени.

Даже с четырьмя потоками приложения G1 в данном примере побеждает. Здесь уборщик мусора приостанавливает потоки приложения суммарно на 176 секунд. Алгоритм G1 приостанавливает потоки приложения всего на 88 секунд. Фоновым потокам G1 пришлось конкурировать с потоками приложения за ресурсы

процессора, что отняло у потоков приложения около 65 секунд. Это все еще означало, что уборщик G1 отработал на 23 секунды быстрее.

Когда ключевым фактором являются затраты времени, параллельный уборщик будет иметь преимущество, если он проводит за приостановкой потоков приложения меньше времени, чем G1. Это происходит при выполнении одного и более из следующих условий:

- Полные уборки мусора отсутствуют (или их немного). Паузы от полной уборки мусора могут легко перевесить время приостановки приложения, но если их вообще не будет, то параллельный уборщик уже не уступает конкуренту.
- Старое поколение обычно заполнено, вследствие чего фоновым потокам уборщика G1 приходится выполнять больше работы.
- Потокам G1 не хватает ресурсов процессора.

В следующей главе работа различных алгоритмов описана более подробно, и вы поймете причины, лежащие в основе пунктов этого списка (а также способы обойти их на уровне настройки уборщиков мусора). А пока рассмотрим несколько примеров, подтверждающих сказанное.

Начнем с данных в табл. 5.7. Этот тест содержит тот же код, который использовался ранее для пакетных заданий с долгими вычислениями, хотя и с некоторыми изменениями: вычисления производятся в нескольких потоках приложения (двух в данном случае); старое поколение инициализируется объектами, чтобы оно было заполнено на 65%; и почти все объекты могут быть освобождены непосредственно из молодого поколения. Тест выполнялся в системе с четырьмя процессорами (без гиперпоточности), чтобы фоновым потокам G1 хватало ресурсов процессора для выполнения.

Таблица 5.7. Пакетная обработка с долгоживущими объектами

Метрика	Уборщик G1	Параллельный уборщик
Затраченное время	212 секунд	193 секунды
Использование процессора	67%	51%
Паузы на уборку мусора в молодом поколении	30 секунд	13,5 секунды
Паузы на полную уборку мусора	0 секунд	1,5 секунды

Из-за малого количества объектов, переводимых в старое поколение, параллельный уборщик мусора приостановил потоки приложения всего на 15 секунд, и только 1,5 секунды было потрачено на уборку старого поколения.

Количество новых объектов, переводимых в старое поколение, невелико, но тест инициализирует старое поколение так, чтобы потоки уборщика G1 сканировали его на наличие мусора. Это повышает объем работы для фоновых потоков уборки мусора и заставляет G1 выполнять больше работы по уборке молодого поколения в попытке компенсировать заполнение старого поколения. В итоге уборщик G1 приостановил приложение на 30 секунд во время теста с двумя потоками — больше, чем с параллельным уборщиком.

СРЕДНИЙ УРОВЕНЬ ИСПОЛЬЗОВАНИЯ ПРОЦЕССОРА И УБОРКА МУСОРА

Анализ только среднего уровня загрузки процессора при проведении тестирования упускает интересную картину того, что происходит во время уборки мусора. Параллельный уборщик (по умолчанию) во время своей работы потребляет 100% процессорного времени, доступного на машине, поэтому более точное представление загрузки процессора в ходе теста с двумя потоками приложения изображено на рис. 5.2. В большинстве случаев работают только потоки приложения, потребляющие 50% процессорного времени. Когда активизируется уборка мусора, потребляется 100% процессора. По этой причине график фактического уровня использования процессора имеет вид «пилы», хотя средний уровень за время теста изображается прямой пунктирной линией.

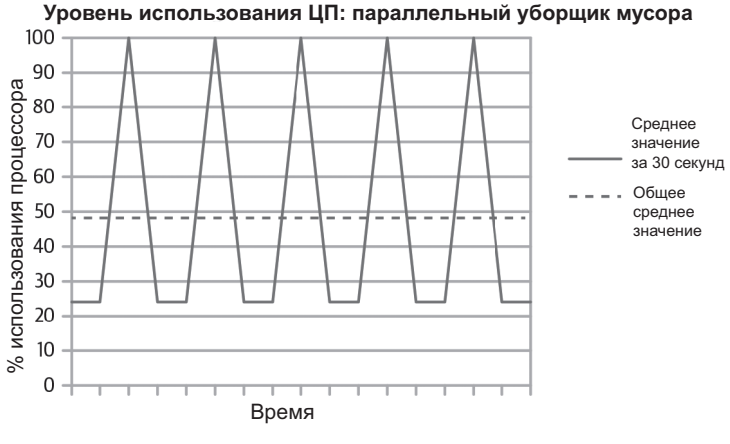


Рис. 5.2. Фактический и средний уровень использования процессора (параллельная уборка мусора)

Эффект будет другим для конкурентного уборщика, в котором фоновые потоки работают одновременно с потоками приложения. В этом случае примерный вид графика загрузки показан на рис. 5.3.

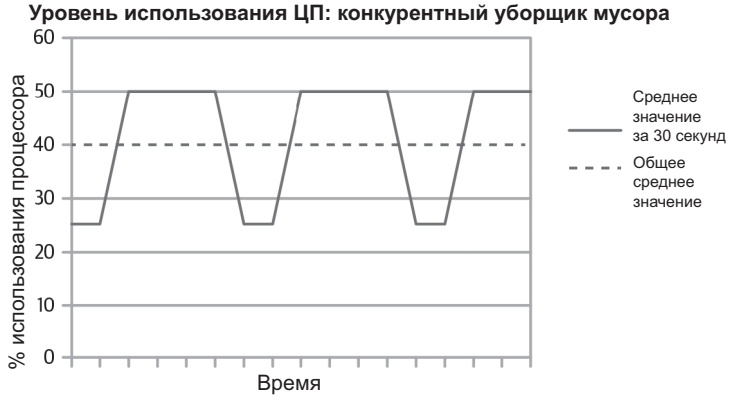


Рис. 5.3. Фактический и средний уровень использования процессора (уборка мусора G1)

Поток приложения изначально использует 50% ресурсов процессора. Со временем он создает достаточно мусора, чтобы фоновый поток уборщика G1 начал работать; этот поток также потребляет весь процессор, доводя общую загрузку до 75%. Когда поток завершает работу, уровень использования процессора падает до 50%, и т. д. Обратите внимание на отсутствие пиковых периодов со 100%-ной загрузкой; такое представление несколько упрощено. В процессе уборки мусора в молодом поколении алгоритмом G1 будут наблюдаться очень короткие выбросы до 100%, но они достаточно коротки, чтобы их можно было игнорировать в контексте нашего обсуждения. (Эти очень короткие пики также встречаются в параллельном уборщике.)

В конкурентном уборщике мусора могут существовать множественные фоновые потоки, но эффект будет похожим: при выполнении фоновые потоки будут потреблять ресурсы процессора и повышать средний уровень загрузки процессора на длительном промежутке времени.

Это может быть важно в системе мониторинга, активизируемой по уровню использования процессора: вы хотите, чтобы сигналы процессора не срабатывали по 100%-ным пикам использования процессора в полной уборке мусора или более продолжительным (но менее выраженным) пикам из-за фоновых конкурентных потоков. Такие пики — нормальное явление в программах Java.

Другой пример: если фоновым потокам уборщика G1 не хватает ресурсов процессора, параллельный уборщик покажет лучшие результаты, как видно из табл. 5.8.

Таблица 5.8. Пакетная обработка с занятыми процессорами

Метрика	Уборщик G1	Параллельный уборщик
Затраченное время	287 секунд	267 секунд
Использование процессора	99%	99%
Паузы на уборку мусора в молодом поколении	80 секунд	63 секунды
Паузы на полную уборку мусора	0 секунд	37 секунд

На самом деле ситуация не отличается от однопроцессорной конфигурации: конкуренция за ресурсы процессора между фоновыми потоками G1 и потоками приложения означает, что потоки приложения фактически приостанавливались даже при отсутствии пауз на уборку мусора.

Если вас в большей степени интересует интерактивная обработка и время отклика, то параллельному уборщику будет непросто превзойти G1. Если же вашему серверу не хватает ресурсов процессора в такой степени, что уборщик G1 и потоки приложения начинают конкурировать за процессор, то алгоритм G1 будет обладать худшим временем отклика (как в тех случаях, которые были показаны выше). Если сервер был настроен так, что полная уборка мусора практически отсутствует, то уборщик G1 и параллельный уборщик будут выдавать сходные результаты. Но чем больше полных уборок мусора будет происходить в работе параллельного уборщика, тем лучше будет среднее, 90%-ное и 99%-ное время отклика у уборщика G1.

РЕЗЮМЕ

- Уборщик мусора G1 в настоящее время является лучшим алгоритмом, который выбирается в большинстве приложений.
- Выбор последовательного уборщика мусора оправдан при выполнении приложений, интенсивно использующих процессор, на машине с одним процессором, даже если этот процессор является гиперпоточным. Уборщик мусора G1 будет лучше работать на таком оборудовании для задач, не создающих интенсивной нагрузки на процессор.
- Выбор параллельного уборщика мусора оправдан на многопроцессорных машинах, на которых выполняются задания, интенсивно использующие процессор. Даже для заданий, не связанных с высокой загрузкой процессора, параллельный уборщик может оказаться более удачным вариантом, если полная уборка мусора выполняется относительно редко или если старое поколение обычно заполнено.

Основная настройка уборщика мусора

Хотя алгоритмы уборки мусора различаются по особенностям работы с кучей, они используют общие основные параметры конфигурации. Во многих случаях эти основные параметры — все, что необходимо для запуска приложения.

Определение размера кучи

Первый основной параметр уборщика мусора — размер кучи приложения. Расширенные настройки влияют на размер поколений кучи; в этом разделе мы начнем с обсуждения настройки общего размера кучи.

Выбор размера кучи, как и большинство аспектов производительности, сводится к поиску баланса. Если куча слишком мала, то программа будет проводить слишком много времени за уборкой мусора и недостаточно — за выполнением логики приложения. Но простой выбор очень большого размера кучи тоже не всегда решает проблему. Время, проведенное в паузах уборки мусора, зависит от размера кучи, поэтому с увеличением размера кучи продолжительность этих пауз также возрастет. Паузы происходят реже, но из-за их продолжительности общая производительность ухудшается.

Вторая опасность возникает при использовании очень большой кучи. Компьютерные операционные системы используют виртуальную память для управления физической памятью машины. На машине может быть установлено 8 Гбайт физической памяти, но благодаря ОС все выглядит так, словно доступно намного больше памяти. Объем виртуальной памяти определяется конфигурацией ОС, но допустим, что ОС создает впечатление доступности 16 Гбайт памяти. Для управления виртуальной памятью ОС использует процесс, называемый *подкачкой* (swapping). Вы можете загружать программы, использующие до 16 Гбайт памяти; ОС копирует неактивные части этих программ на диск. Когда эти области памяти понадобятся приложению, ОС скопирует их с диска в оперативную память (обычно для этого ей сначала приходится скопировать что-то из памяти на диск, чтобы освободить место).

Процесс хорошо работает в системах, в которых работает множество приложений, потому что приложения редко находятся в активном состоянии конкурентно. С приложениями Java все не так просто. Если программа Java с 12-гигабайтной кучей работает в такой системе, ОС может разместить 8 Гбайт кучи в оперативной памяти, и еще 4 Гбайт на диске (описание несколько упрощено, потому что другие программы тоже используют часть памяти). JVM об этом не знает; подкачка реализуется ОС прозрачно для приложений. А значит, JVM радостно заполнит все 12 Гбайт кучи, которые ей разрешено использовать. Это создаст серьезные проблемы с производительностью, когда ОС загружает данные с диска в память (что само по себе является довольно затратной операцией).

Что еще хуже, подкачка гарантированно произойдет во время полной уборки мусора, когда JVM потребуется обратиться ко всей куче. Если система выполняет подкачку во время полной уборки мусора, то паузы будут на порядок длиннее, чем в обычном случае. Аналогичным образом, когда вы используете алгоритм G1, а фоновый поток очищает данные в куче, почти наверняка ему придется подолгу дожидаться копирования данных с диска в основную память — а это приведет к сбою затратного режима конкурентного выполнения.

Итак, первое правило при выборе размера кучи — никогда не указывать размер кучи, превышающий объем физической памяти на машине. А если в системе работают несколько JVM, это относится к сумме всех куч. Также необходимо оставить место для служебной памяти JVM и для памяти других приложений: для большинства типичных профилей ОС не менее 1 Гбайт памяти.

Размер кучи определяется двумя значениями: исходным (задается флагом `-Xms`*N*) и максимальным (`-Xmx`*N*). Значения по умолчанию зависят от операционной системы, объема оперативной памяти в системе и используемой JVM. На значения по умолчанию также могут влиять другие флаги командной строки; определение размера кучи — одна из базовых эргономических настроек JVM.

Задача JVM — найти «разумное» исходное значение по умолчанию для кучи на основании доступных системных ресурсов и расширять кучу до «разумного» максимума в том (и только в том) случае, когда приложению потребуется больше памяти (в зависимости от того, сколько времени оно проводит за выполнением уборки мусора). Значения исходного и максимального размера по умолчанию приведены в табл. 5.9 (без учета некоторых расширенных флагов оптимизации и технических подробностей, изложенных в этой и в следующей главах). JVM слегка округляет эти значения в меньшую сторону для выравнивания; из журналов уборки мусора, в которых выводятся фактические размеры, видно, что значения не полностью совпадают с числами из этой таблицы.

Таблица 5.9. Размеры кучи по умолчанию

Операционная система и JVM	Исходный размер кучи (Xms)	Максимальный размер кучи (Xmx)
Linux	Минимум (512 Мбайт, 1/64 физической памяти)	Минимум (32 Гбайт, 1/4 физической памяти)
macOS	64 Мбайт	Минимум (1 Гбайт, 1/4 физической памяти)
Windows — 32-разрядные клиентские JVM	16 Мбайт	256 Мбайт
Windows — 64-разрядные серверные JVM	64 Мбайт	Минимум (1 Гбайт, 1/4 физической памяти)

На машине с менее чем 192 Мбайт физической памяти максимальный размер кучи составляет половину физической памяти (96 Мбайт и менее).

Учтите, что значения из табл. 5.9 — одна из тех настроек, которые будут ошибочными для контейнеров Docker с ограничениями памяти в версиях JDK 8 до обновления 192: для вычисления размеров по умолчанию JVM использует общий объем памяти на машине. В более поздних версиях JDK 8 и JDK 11 JVM использует ограничение памяти контейнера.

Определение исходного и максимального размера кучи позволяет JVM настраивать свое поведение в зависимости от текущей нагрузки. Если JVM видит, что с исходным размером кучи выполняется слишком большой объем уборки мусора, она постепенно расширяет кучу, пока JVM не будет выполнять «правильный» объем уборки мусора или пока куча не достигнет максимально возможного размера.

Для приложений, которым большая куча не нужна, это означает, что размер кучи задавать вообще не нужно. Вместо этого вы задаете цели алгоритма уборки мусора по производительности: продолжительность пауз, которую вы считаете приемлемой, процент времени, проводимого за уборкой мусора, и т. д. Подробности зависят от используемого алгоритма уборки мусора и рассматриваются в следующей главе (хотя даже при этом значения по умолчанию выбираются так, что для широкого спектра приложений они не нуждаются в настройке).

В мире, в котором JVM часто работают в изолированных контейнерах, обычно бывает достаточно указать максимальный размер кучи. На виртуальной машине, в которой работает в основном одна JVM, исходная куча по умолчанию составляет только четверть памяти, выделенной виртуальной машине. Аналогичным образом в JDK 11 для контейнера Docker с ограничениями памяти куча обычно должна потреблять бóльшую часть памяти (за вычетом служебной памяти, о чем говорилось ранее). Значения по умолчанию лучше подходят для систем, в которых выполняются разные приложения (в отличие от контейнеров, выделенных под конкретную JVM).

Нет однозначного правила для определения максимального размера кучи (кроме того, что он не должен превышать максимума, поддерживаемого машиной). Хорошее эмпирическое правило — выбирать размер кучи так, чтобы она была занятой на 30% после полной уборки мусора. Чтобы вычислить это значение, выполняйте приложение до того, как оно достигнет устойчивого состояния: точки, в которой будут загружены все кэшируемые данные, будет создано максимальное количество клиентских подключений и т. д. Затем подключитесь к приложению при помощи `jconsole`, инициируйте полную уборку мусора и проследите за тем, сколько памяти используется после завершения полной уборки мусора. (Для параллельной уборки мусора также можно обратиться к журналу,

если он доступен.) Если вы пойдете по этому пути, обязательно задайте размер своего контейнера, чтобы он содержал дополнительные 0,5–1 Гбайт памяти для потребностей JVM, не относящихся к куче.

Учтите, что автоматическое изменение размера кучи происходит даже при явном задании максимального размера: куча начинается с исходного размера по умолчанию, а JVM будет наращивать кучу для соблюдения целей алгоритма уборки мусора по производительности. Если куче будет назначен размер, превышающий необходимый, это не обязательно приводит к потерям памяти: куча увеличивается настолько, насколько потребуется для выполнения целей GC по производительности.

С другой стороны, если вам точно известен размер кучи, необходимой приложению, вы с таким же успехом можете присвоить это значение как исходный и максимальный размер кучи (например, `-Xms4096m -Xmx4096m`). Это немного повысит эффективность уборки мусора, потому что алгоритму никогда не придется определять, нужно ли изменять размер кучи.

РЕЗЮМЕ

- JVM пытается найти разумный минимальный и максимальный размер кучи на основании машины, на которой она работает.
- Если только вашему приложению не нужна куча с размером больше размера по умолчанию, рассмотрите возможность настройки целей производительности алгоритма уборки мусора (см. главу 6) вместо точной настройки размеров кучи.

Определение размеров поколений

Когда размер кучи будет определен, JVM должна решить, какие части кучи выделить под молодое и старое поколение. JVM обычно делает это автоматически и неплохо справляется с определением оптимального отношения между молодым и старым поколением. В некоторых случаях эти значения приходится настраивать вручную, хотя в основном я включил этот раздел, чтобы вы лучше поняли, как работает уборка мусора.

Последствия изменения размеров поколений для производительности должны быть очевидны: если молодое поколение относительно велико, время паузы уборки мусора в молодом поколении возрастет, но молодое поколение будет очищаться реже и меньшее количество объектов будет переведено в старшее поколение. Но с другой стороны, поскольку старое поколение относительно малочисленно, оно будет заполняться чаще и инициировать больше полных уборок мусора. Главное — правильно выдержать баланс.

Разные алгоритмы уборки мусора пытаются выдерживать этот баланс разными способами. Но все алгоритмы уборки мусора для определения размеров поколений используют один набор флагов, описанных в этом разделе.

Все флаги командной строки для настройки размеров поколений регулируют размер молодого поколения; старому поколению достается все остальное. Для определения размера молодого поколения используются следующие флаги:

`-XX:NewRatio=`*N*

Задает отношение размера молодого поколения к старому.

`-XX:NewSize=`*N*

Задает исходный размер молодого поколения.

`-XX:MaxNewSize=`*N*

Задает максимальный размер молодого поколения.

`-Xmn`*N*

Сокращенная форма для присваивания `NewSize` и `MaxNewSize` одного значения.

Размер молодого поколения сначала определяется значением `NewRatio`, которое по умолчанию равно 2. Параметры, влияющие на размеры областей кучи, обычно задаются в форме отношений; значение используется в формуле для определения доли задействованной памяти. Значение `NewRatio` используется в следующей формуле:

```
Исходный размер молодого поколения = Исходный размер кучи / (1 + NewRatio)
```

При подстановке исходного размера кучи и `NewRatio` вы получаете значение, которое определяет размер молодого поколения. Таким образом, исходный размер молодого поколения по умолчанию составляет 33% от исходного размера кучи.

Также размер молодого поколения можно задать явно при помощи флага `NewSize`. Если этот флаг установлен, он обладает более высоким приоритетом, чем значение, вычисленное при помощи `NewRatio`. У флага нет значения по умолчанию, потому что оно вычисляется по значению `NewRatio`.

С увеличением кучи также увеличивается размер молодого поколения до максимума, заданного флагом `MaxNewSize`. По умолчанию этот максимум также определяется по значению `NewRatio`, хотя при вычислении используется максимальный (а не исходный) размер кучи.

В итоге настройка молодого поколения, основанная на определении диапазона минимального и максимального размера, оказывается довольно сложной задачей. При фиксированном размере кучи (для чего `-Xms` назначается равным `-Xmx`)

обычно лучше использовать `-Xmn` для определения фиксированного размера и для молодого поколения. Если приложению нужна куча с динамически изменяемым размером и оно требует большего (или меньшего) молодого поколения, сосредоточьтесь на определении значения `NewRatio`.

Адаптивное определение размеров

Размеры кучи, поколений и областей выживших объектов могут изменяться в ходе выполнения, так как JVM пытается найти оптимальную производительность в соответствии со своей политикой и настройками. Это решение с максимально доступной эффективностью зависит от прошлой производительности: предполагается, что будущий процесс уборки мусора будет похож на уборку мусора в недалеком прошлом. Такое предположение оказывается разумным для многих вариантов нагрузки, и даже при внезапном изменении темпов выделения памяти JVM адаптирует размеры областей к новой информации.

Адаптивное определение размеров имеет два важных преимущества. Во-первых, оно означает, что небольшим приложениям не нужно беспокоиться об определении завышенного размера кучи. Возьмем административные программы командной строки, используемые для регулировки работы таких подсистем, как сервер NoSQL, — обычно такие программы живут недолго и расходуют минимум ресурсов памяти. Такие приложения используют 64 (или 16) Мбайт кучи, несмотря на то что куча по умолчанию может увеличиться до 1 Гбайт. Из-за адаптивного определения размеров такие приложения не нуждаются в специальной настройке; значения по умолчанию для платформы гарантируют, что они не будут тратить много памяти.

Кроме того, многим приложениям вообще не нужно беспокоиться о настройке размера кучи — или если им нужна куча большего размера, чем выбирается по умолчанию для платформы, они просто указывают больший размер и забывают обо всем остальном. JVM может автоматически настраивать размеры кучи и поколений для использования оптимального объема памяти с учетом целей алгоритма GC по производительности. Механизм адаптивного определения размеров обеспечивает работу автоматизированной настройки.

Тем не менее регулировка размеров занимает небольшой промежуток времени — в основном во время паузы при уборке мусора. Если вы потратили время за точной настройкой параметров уборки мусора и ограничений размеров кучи приложения, адаптивное определение размеров можно отключить. Отключение адаптивного определения размеров также может пригодиться в приложениях, которые проходят через сильно различающиеся фазы, и вы хотите оптимально настроить уборку мусора в одной из таких фаз.

На глобальном уровне адаптивное определение размеров отключается флагом `-XX:-UseAdaptiveSizePolicy` (по умолчанию `true`). За исключением областей

выживших (которые подробно рассматриваются в следующей главе), адаптивное определение размеров также фактически отключается, если минимальному и максимальному размеру кучи присвоены одинаковые значения — как и исходному и максимальному размеру нового поколения.

Чтобы понаблюдать за тем, как JVM изменяет размеры областей, установите флаг `-XX:+PrintAdaptiveSizePolicy`. При выполнении уборки мусора в журнал будет занесена информация о том, как изменялись размеры поколений в ходе уборки.

РЕЗЮМЕ

- В пределах общего размера кучи размеры поколений определяются объемом памяти, выделенным под новое поколение.
- Молодое поколение увеличивается вместе с общим размером кучи, но также может колебаться в пределах нескольких процентов от общей кучи (на основании исходного и максимального размера молодого поколения).
- Адаптивное определение размеров управляет тем, как JVM изменяет отношение между размерами молодого и старого поколения в куче.
- Адаптивное определение размеров обычно должно оставаться включенным, так как регулировка размеров поколений позволяет алгоритмам уборки мусора попытаться выполнить свои цели по продолжительности пауз.
- Для куч с точной настройкой адаптивное определение размера можно отключить ради небольшого прироста производительности.

Определение размера метапространства

Когда JVM загружает классы, она должна отслеживать некоторые метаданные, относящиеся к этим классам. Они занимают отдельную область кучи, которая называется метапространством (metaspace). В старых JVM использовалась другая реализация, называемая *постоянным поколением* (permgen).

Для конечных пользователей метапространство непрозрачно: мы знаем, что в нем хранится набор данных, относящихся к классам, и в некоторых обстоятельствах размер этой области необходимо настраивать.

Следует учесть, что в метапространстве не хранятся ни фактические экземпляры классов (объекты `Class`), ни объекты отражения (например, объекты `Method`);

они хранятся в обычной куче. Информация из метапространства используется только компилятором и исполнительной средой JVM, а хранящиеся в метапространстве данные называются *метаданными классов*.

Нет хорошего способа заранее вычислить объем памяти, которая понадобится конкретной программе для ее метапространства. Он будет пропорционален количеству используемых классов, так что большим приложениям понадобятся большие области. Это еще одна область, в которой изменения в технологии JDK упростили жизнь разработчика: настройка постоянного поколения была вполне распространенным делом, но в наши дни настраивать метапространство приходится относительно редко. В основном это объясняется тем, что значения по умолчанию для размера метапространства задаются с большим запасом. В табл. 5.10 перечислены исходные и максимальные размеры по умолчанию.

Таблица 5.10. Размеры метапространства по умолчанию

JVM	Исходный размер по умолчанию	Максимальный размер по умолчанию
32-разрядная клиентская JVM	12 Мбайт	Не ограничивается
32-разрядная серверная JVM	16 Мбайт	Не ограничивается
64-разрядная JVM	20,75 Мбайт	Не ограничивается

Метапространство своим поведением напоминает отдельный экземпляр обычной кучи. Его размер определяется динамически в зависимости от исходного размера (`-XX:MetaspaceSize=`*N*) и увеличивается по мере необходимости до максимального размера (`-XX:MaxMetaspaceSize=`*N*).

МЕТАПРОСТРАНСТВО СЛИШКОМ ВЕЛИКО

Так как размер метапространства по умолчанию не ограничен, приложение (особенно в 32-разрядной JVM) может столкнуться с нехваткой памяти из-за заполнения метапространства. Программа NMT (Native Memory Tracking), описанная в главе 8, поможет диагностировать подобные ситуации. Если метапространство растет слишком сильно, вы можете присвоить `MaxMetaspaceSize` меньшее значение — но тогда приложение рано или поздно столкнется с ошибкой `OutOfMemoryError` при заполнении метапространства. В такой ситуации единственное реальное решение — определить, почему метаданные класса достигли такого размера.

Изменение размера метапространства требует полной уборки мусора и поэтому является дорогостоящей операцией. Если в начале программы (при загрузке классов) выполняется значительный объем уборки мусора, это часто происходит из-за изменения размеров постоянного поколения или метапространства, так что для улучшения времени запуска стоит поэкспериментировать с увеличением исходного размера. Например, серверы обычно задают исходный размер метапространства 128 Мбайт, 192 Мбайт и выше.

Классы Java могут подвергаться уборке мусора, как и все остальное. Это типичное явление в серверах приложений, которые создают новые загрузчики классов при каждом развертывании (или повторном развертывании) приложения. Старые загрузчики классов остаются без ссылок и считаются пригодными для уборки мусора, как и все классы, определенные ими. При этом новые классы приложения будут иметь новые метаданные, поэтому в метапространстве должно хватать места для этого. Часто при этом происходит полная уборка мусора, потому что метапространство должно вырасти (или уничтожить старые метаданные).

Одна из причин для ограничения размера метапространства — защита от *утечки* загрузчиков классов: сервер приложения (или другая программа — скажем, IDE) постоянно определяет новые загрузчики классов и классы без освобождения ссылок на старые загрузчики. В такой ситуации появляется риск заполнения метапространства и захвата большого объема памяти на машине. С другой стороны, фактический загрузчик классов и объекты классов в этом случае все еще остаются в основной куче — эта куча с большой вероятностью заполнится и вызовет ошибку `OutOfMemoryError`, прежде чем память, занимаемая метапространством, начнет создавать проблемы.

Дампы кучи (см. главу 7) могут использоваться для диагностики существующих загрузчиков классов. В свою очередь, эта информация поможет определить, не заполняется ли метапространство из-за утечки загрузчиков классов. Также можно воспользоваться программой `jmap` с аргументом `-clstats` для вывода информации о загрузчиках классов.

РЕЗЮМЕ

- Метапространство содержит метаданные классов (не объекты классов) и ведет себя как отдельная куча.
- Исходный размер этой области может определяться на основании ее использования, после того как будут загружены все классы. Это несколько ускорит запуск.
- Приложения, которые определяют и уничтожают большое количество классов, время от времени сталкиваются с полной уборкой мусора при заполнении метапространства и уничтожении старых классов. Такая ситуация особенно типична для среды разработки.

Управление параллелизмом

Все алгоритмы уборки мусора, кроме последовательного уборщика, используют несколько потоков. Количеством этих потоков управляет флаг `-XX:ParallelGCThreads=`*N*. Значение флага влияет на количество потоков, используемых для следующих операций:

- Уборка молодого поколения с флагом `-XX:+UseParallelGC`.
- Уборка старого поколения с флагом `-XX:+UseParallelGC`.
- Уборка молодого поколения с флагом `-XX:+UseG1GC`.
- Глобальные паузы уборщика G1 (но не полные уборки мусора).

Так как эти операции уборки мусора останавливают все потоки приложения, JVM пытается использовать максимально возможные ресурсы процессора для минимизации времени паузы. По умолчанию это означает, что JVM будет выполнять один поток для каждого процессора на машине, не более восьми. После того как этот поток будет достигнут, JVM добавляет новый поток только для каждых 1,6 процессора. Таким образом, общее количество потоков (где *N* — количество процессоров) на машине с более чем 8 процессорами определяется по формуле:

```
ПараллельныеПотокиУборкиМусора = 8 + ((N - 8) * 5 / 8)
```

Иногда это число оказывается слишком большим. Приложение, использующее небольшую кучу (допустим, 1 Гбайт) на машине с восемью процессорами, будет чуть более эффективным с четырьмя или шестью потоками, работающими с кучей. На 128-процессорной машине 83 потока будет слишком много для любых куч, кроме самых больших.

Если JVM запускается в контейнере Docker с ограничением по количеству процессоров, то в формуле используется это ограничение.

Кроме того, если на машине работает более одной JVM, желательно ограничить общее количество потоков уборки мусора по всем JVM. Потоки уборки мусора работают достаточно эффективно, и каждый будет потреблять 100% одного процессора (вот почему средний уровень использования процессора для параллельного уборщика был выше, чем ожидалось в предыдущем примере). На машинах с восемью и менее процессорами уборка мусора будет потреблять 100% процессорных ресурсов на машине. На машинах с большим количеством процессоров и несколькими JVM слишком много потоков уборки мусора будут работать параллельно.

Для примера возьмем 16-процессорную машину, на которой работают 4 JVM; каждая JVM по умолчанию получает 13 потоков уборки мусора. Если все четыре JVM будут выполнять GC конкурентно, на машине появятся 52 потока, конкурирующих за процессорное время. Уровень конкуренции слишком велик; ограничение каждой JVM четырьмя потоками уборки мусора будет более эффективным. И хотя маловероятно, чтобы все четыре JVM выполняли операцию уборки мусора конкурентно, даже одна JVM, проводящая уборку мусора с 13 потоками, означает, что потокам приложений других JVM теперь придется конкурировать за ресурсы процессора на машине, где 13 из 16 процессоров на 100% заняты выполнением задач уборки мусора. Выделение каждой JVM четырех потоков уборки мусора обеспечит лучший баланс в данном случае.

Обратите внимание: этот флаг не задает количество фоновых потоков, используемых уборщиком G1 (хотя и влияет на него). Подробности приводятся в следующей главе.

РЕЗЮМЕ

- Базовое количество потоков, используемых всеми алгоритмами уборки мусора, зависит от количества процессоров на машине.
- При выполнении нескольких JVM на одной машине это количество будет слишком большим, и его следует сократить.

Инструменты уборки мусора

Так как уборка мусора играет важнейшую роль в производительности Java, есть много программных средств для отслеживания ее производительности. Чтобы увидеть влияние уборки мусора на производительность приложения, лучше всего ознакомиться с журналом уборки мусора — протоколом всех операций уборки мусора во время выполнения программы.

Информация в журнале уборки мусора зависит от алгоритма уборки мусора, но основные средства управления журналом остаются неизменными для всех алгоритмов. При этом управление журналом отличается у JDK 8 и последующих версий: в JDK 11 используется другой набор аргументов командной строки для включения и управления журналом уборки мусора. Основные принципы управления журналом уборки мусора приводятся в этом разделе, а более подробная информация о содержимом журнала находится в разделах, посвященных оптимизации конкретных алгоритмов, в следующей главе.

Включение протоколирования уборки мусора в JDK 8

В JDK 8 предусмотрено несколько способов включения протоколирования (ведения журнала) уборки мусора. Любой из флагов `-verbose:gc` или `-XX:+PrintGC` создает простой журнал уборки мусора (эти флаги являются синонимами, и по умолчанию журнал отключен). Флаг `-XX:+PrintGCDetails` создает журнал с гораздо более подробной информацией. Этот флаг рекомендуется использовать (он равен `false` по умолчанию); по простому журналу часто бывает слишком сложно провести диагностику событий, происходящих при уборке мусора.

В сочетании с подробным журналом рекомендуется включать флаг `-XX:+PrintGCTimeStamps` или `-XX:+PrintGCDateStamps` для определения времени между операциями уборки мусора. Эти два аргумента отличаются тем, что в первом случае временные метки задаются относительно 0 (время запуска JVM), тогда как во втором используется фактическая строка даты. Это слегка снижает производительность меток при форматировании дат, хотя эта операция выполняется достаточно редко и ее эффект вряд ли будет заметен.

Журнал уборки мусора записывается в стандартный вывод, хотя приемник можно (и обычно нужно) изменить при помощи флага `-Xloggc:`*имя_файла*. Флаг `-Xloggc` автоматически включает ведение простого журнала, если только при этом также не был установлен флаг `PrintGCDetails`.

Объем данных, хранящихся в журнале уборки мусора, может быть ограничен при помощи *ротации журнала*; данная возможность может быть полезной для сервера, работающего в течение долгого времени, который мог бы забить весь диск журналами за несколько месяцев работы. Ротацией журналов управляют следующие флаги: `-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=`*N* `-XX:GCLogFileSize=`*N*. По умолчанию флаг `UseGCLogFileRotation` отключен. При установке этого флага количество файлов по умолчанию равно 0 (не ограничено), а размер файла журнала по умолчанию равен 0 (не ограничен). Следовательно, вы должны задать значения всех этих параметров, чтобы ротация журнала работала так, как ожидалось. Следует заметить, что размер файла журнала будет округляться до 8 Кбайт для меньших значений.

Объединяя все сказанное, полезный набор флагов для ведения журнала выглядит так:

```
-Xloggc:gc.log -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFile=8 -XX:GCLogFileSize=8m
```

С этими флагами события уборки мусора будут регистрироваться с временными метками для сопоставления с другими журналами, а сохраненные журналы будут

ограничены 64 Мбайт в восьми файлах. Активность ведения журнала достаточно мала, что позволяет включать ее даже в рабочих системах.

Включение протоколирования уборки мусора в JDK 11

JDK 11 и более поздние версии используют новый механизм *унифицированного протоколирования*. Это означает, что все операции ведения журнала — относящиеся к уборке мусора или нет — включаются флагом `-Xlog`. Затем к этому флагу присоединяются различные параметры, управляющие протоколированием. Чтобы включить режим протоколирования, сходный с длинным примером для JDK 8, используйте следующий флаг:

```
-Xlog:gc*:file=gc.log:time:filecount=7,filesize=8M
```

Двоеточия делят команду на четыре части. Для получения дополнительной информации о доступных параметрах можно выполнить команду `-Xlog:help:`, но я покажу, как они соответствуют структуре строки.

Первая часть (`gc*`) сообщает, для каких модулей должно быть включено протоколирование; в нашем примере оно включается для всех модулей уборки мусора. Существуют параметры для протоколирования только отдельных частей (например, `gc+age` будет включать в журнал информацию о хранении объектов — эта тема рассматривается в следующей главе). Эти конкретные модули часто предоставляют ограниченный вывод на уровне протоколирования по умолчанию, так что вы можете использовать конструкцию вида `gc*,gc+age=debug` для протоколирования базовых сообщений (информационного уровня) от всех модулей `gc` и сообщений отладочного уровня из кода хранения. Как правило, протоколирования всех модулей на информационном уровне достаточно.

Вторая часть задает приемник для хранения файла журнала.

Третья часть (`time`) является декоратором: этот декоратор приказывает сохранять в журнале сообщения с временной меткой, как это делалось ранее для JDK 8. Можно указать сразу несколько декораторов.

Наконец, в четвертом разделе указываются параметры вывода; в данном случае включается ротация журналов при достижении 8 Мбайт, с созданием до восьми журналов.

Учтите, что ротация журнала несколько различается между JDK 8 и JDK 11. Допустим, задается имя файла журнала gc.log, и сохраняться должны три файла. В JDK 8 запись будет вестись следующим образом.

1. Начать запись в gc.log.0.current.
2. При заполнении переименовать файл в gc.log.0 и начать запись в gc.log.1.current.
3. При заполнении переименовать файл в gc.log.1 и начать запись в gc.log.2.current.
4. При заполнении переименовать файл в gc.log.2, удалить gc.log.0 и начать запись в новый файл gc.log.0.current.
5. Повторить цикл.

В JDK 11 запись ведется по следующей схеме.

1. Начать запись в gc.log.
2. При заполнении переименовать файл в gc.log.0 и начать запись в новый файл gc.log.
3. При заполнении переименовать файл в gc.log.1 и начать запись в новый файл gc.log.
4. При заполнении переименовать файл в gc.log.2 и начать запись в новый файл gc.log.
5. При заполнении переименовать файл в gc.log.0, удалить старый файл gc.log.0 и начать запись в новый файл gc.log.

Вас интересует, почему в приведенной выше команде для JDK 11 мы потребовали хранить семь журналов? Потому что в этом случае будут существовать восемь активных файлов. Также следует заметить, что в обоих случаях число, присоединяемое к файлу, ничего не говорит о порядке создания файлов. Числа повторно используются в цикле, так что некоторый порядок все же есть, но самым старым журнальным файлом может оказаться любой файл в наборе.

В журнале хранится большой объем информации, относящейся к каждому уборщику; подробности будут изложены в следующей главе. Разбор журналов в поисках полезной информации, относящейся к вашему приложению, также может сообщить, сколько пауз возникло во время работы, сколько времени в среднем они занимали и т. д.

К сожалению, хороших программ для разбора журналов с открытым кодом не так много. Как и в случае с профилировщиками, появились коммерческие предложения — например, предложения от jClarity (Censum) и GCeasy. Последний продукт предоставляет бесплатный сервис разбора журналов.

Для мониторинга кучи в реальном времени используйте `jvisualvm` или `jconsole`. На панели Memory программы `jconsole` отображается график размера кучи в реальном времени (рис. 5.4).

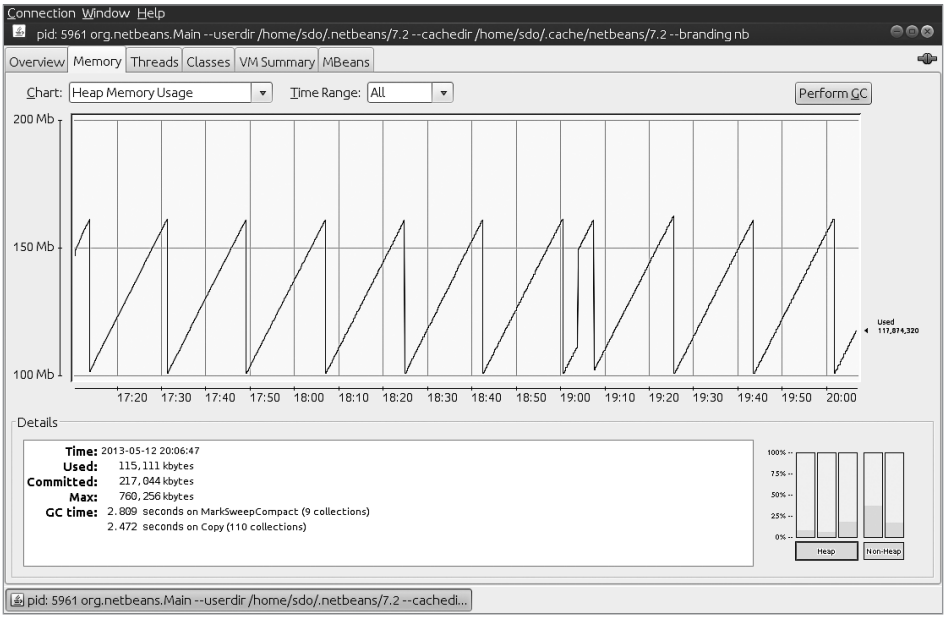


Рис. 5.4. График размера кучи в реальном времени

В этом конкретном представлении отображается вся куча, которая периодически переключается между использованием 100 Мбайт и 160 Мбайт. `jconsole` может отображать данные только по Эдему, областям выживших, старому поколению или постоянному поколению. Если выбрать Эдем для отображения информации, то график имел бы сходную структуру, так как размер Эдема колебался бы между 0 Мбайт и 60 Мбайт (и, как можно предположить, график старого поколения представлял бы собой практически горизонтальную линию на уровне 100 Мбайт).

Для использования в сценариях следует выбирать `jstat`. Программа `jstat` предоставляет девять параметров для вывода различной информации о куче; команда `jstat -options` выводит полный список. Также заслуживает внимания параметр `-gcutil`, который выводит время, проведенное за уборкой мусора, а также текущий процент заполнения каждой области уборки мусора. Другие параметры `jstat` выводят размеры в килобайтах.

Помните, что `jstat` получает необходимый аргумент (периодичность повторения команды в миллисекундах), чтобы вы могли отслеживать эффект уборки мусора с течением времени. Пример вывода, повторяющегося через каждую секунду:

```
% jstat -gcutil 23461 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
 51.71   0.00  99.12  60.00  99.93     98    1.985     8    2.397    4.382
  0.00  42.08   5.55  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08   6.32  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  68.06  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  82.27  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  96.67  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  99.30  60.98  99.93     99    2.016     8    2.397    4.413
 44.54   0.00   1.38  60.98  99.93    100    2.042     8    2.397    4.439
 44.54   0.00   1.91  60.98  99.93    100    2.042     8    2.397    4.439
```

Когда начался мониторинг идентификатора процесса 23461, программа уже выполнила 98 уборок мусора в молодом поколении (`YGC`), что в сумме заняло 1,985 секунды (`YGCT`). Также были выполнены восемь полных уборок мусора (`FGC`), которые заняли 2,397 секунды (`FGCT`); таким образом, общее время уборки мусора (`GCT`) составило 4,382 секунды.

Отображаются данные по всем трем разделам молодого поколения: двум областям выживших (`S0` и `S1`) и Эдему (`E`). Мониторинг начался с заполнения Эдема (99,12%), поэтому в следующую секунду появилось молодое поколение: Эдем сократился до 5,55%, области выживших поменялись местами, а небольшой объем памяти был переведен в старое поколение (`O`), уровень использования которого повысился до 60,98%. Как обычно, в постоянном поколении (`P`) изменений почти не было, потому что все необходимые классы уже были загружены приложением.

Если вы забыли включить ведение журнала уборки мусора, эти данные станут хорошей альтернативой для наблюдения за поведением уборки мусора во времени.

РЕЗЮМЕ

- Журналы уборки мусора — ключевая часть данных, необходимых для диагностики проблем с уборкой мусора; они должны регулярно собираться (даже на рабочих серверах).
- Для получения расширенного журнала уборки мусора используйте флаг `PrintGCDetails`.
- Существуют различные программы для разбора и анализа журналов уборки мусора; они помогают обобщать данные в журналах.
- jstat предоставляет информацию о ходе уборки мусора в работающей программе.

Итоги

Производительность уборщика мусора — одна из ключевых составляющих общей производительности любого приложения Java. Однако для многих приложений настройка сводится к выбору подходящего алгоритма уборки мусора и при необходимости — увеличению размера кучи приложения. Тогда механизм адаптивного определения размеров позволит JVM автоматически настроить ее поведение, чтобы обеспечить хорошую производительность.

Более сложные приложения требуют дополнительной настройки (особенно для конкретных алгоритмов уборки мусора). Если простые настройки уборки мусора, описанные в этой главе, не обеспечивают производительности, необходимой приложению, обратитесь к описанию параметров.

ГЛАВА 6

Алгоритмы уборки мусора

В главе 5 рассматривается общее поведение всех уборщиков мусора, включая флаги JVM, действующие для всех алгоритмов уборки мусора: способ выбора размеров кучи, размеры поколений, журналы и т. д. Во многих ситуациях основных настроек уборки мусора оказывается достаточно. В остальных случаях вам приходится анализировать конкретные операции алгоритма уборки мусора, чтобы понять, как изменить их параметры для минимизации последствий уборки мусора для приложения.

При настройке конкретного уборщика мусора важнейшую роль играют данные из журнала уборки мусора при включении этого уборщика. В начале этой главы все алгоритмы рассматриваются в контексте вывода журнала; это помогает понять, как работает алгоритм уборки мусора и как настроить его для повышения эффективности. Затем в каждом разделе приводится информация о настройке, требуемой для достижения этой повышенной эффективности.

В этой главе также рассматриваются подробности некоторых новых, экспериментальных уборщиков. Возможно, такие уборщики не обладают 100%-ной стабильностью на момент написания книги, но, скорее всего, к моменту выхода следующей LTS-версии Java они станут полноценными и пригодными для реальной разработки (ведь уборщик мусора G1 начинался как экспериментальный, а теперь используется по умолчанию в JDK 11).

На производительность всех алгоритмов уборки мусора влияют некоторые нетипичные ситуации: создание очень больших объектов, объекты со средним сроком жизни (не очень коротким и не очень большим) и т. д. Эти случаи рассмотрены в конце этой главы.

Параллельный уборщик мусора

Начнем с параллельного уборщика мусора. Хотя ранее было показано, что он обычно уступает уборщику G1, параллельный уборщик устроен проще. На его примере будет проще понять, как происходит уборка мусора.

Как упоминалось в главе 5, уборщики мусора должны выполнять три основные операции: поиск неиспользуемых объектов, освобождение их памяти и сжатие кучи. Параллельный уборщик мусора выполняет все эти операции в одном цикле уборки мусора. За одну операцию такие уборщики могут убирать либо молодое, либо старое поколение.

На рис. 6.1 изображено состояние кучи до и после уборки в молодом поколении.

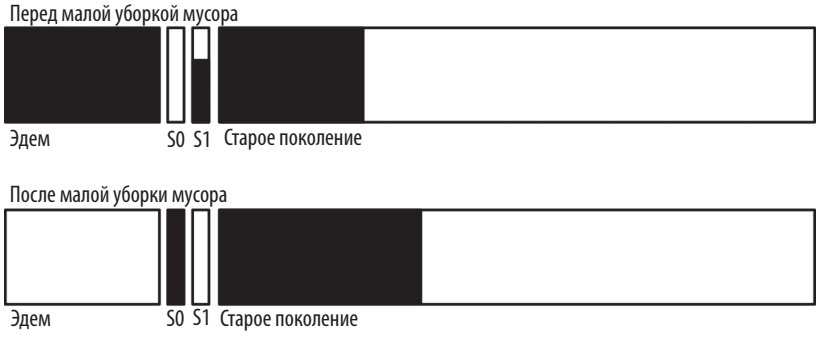


Рис. 6.1. Уборка в молодом поколении параллельным уборщиком мусора

Уборка в молодом поколении происходит при заполнении Эдема. При уборке в молодом поколении все объекты перемещаются из Эдема; одни перемещаются в одну из областей выживших (`S0` на этой диаграмме), другие перемещаются в старое поколение, которое теперь содержит больше объектов. Конечно, многие объекты уничтожаются, потому что на них не осталось ни одной ссылки.

Так как Эдем обычно остается пустым после этой операции, рассматривать происходящее как его сжатие несколько необычно, но суть именно такова.

В журнале уборки мусора JDK 8 с флагом `PrintGCDetails` малая уборка мусора параллельным уборщиком выглядит так:

```
17.806: [GC (Allocation Failure) [PSYoungGen: 227983K->14463K(264128K)]
            280122K->66610K(613696K), 0.0169320 secs]
            [Times: user=0.05 sys=0.00, real=0.02 secs]
```

Уборка мусора произошла через 17,806 секунды после начала программы. Объекты в молодом поколении теперь занимают 14 463 Кбайт (14 Мбайт в пространстве выживших); до уборки мусора они занимали 227 983 Кбайт (227 Мбайт[1]). Общий размер молодого поколения на этой стадии равен 264 Мбайт.

[1] В действительности 227 893 Кбайт составляют всего 222 Мбайт. Для простоты обсуждения в этой главе я буду округлять килобайты до 1000, как это делают производители дисков.

При этом общее заполнение кучи (как молодое, так и старое поколение) сократилось с 280 Мбайт до 66 Мбайт, а размер всей кучи на этот момент составлял 613 Мбайт. Операция заняла менее 0,02 секунды (0,02 секунды реального времени в конце вывода — округленное фактическое время 0,0169320 секунды). Программа заняла больше процессорного времени, чем реального, потому что уборка в молодом поколении выполнялась несколькими потоками (в данной конфигурации — четырьмя).

Тот же журнал в JDK 11 выглядел бы примерно так:

```
[17.805s][info][gc,start       ] GC(4) Pause Young (Allocation Failure)
[17.806s][info][gc,heap        ] GC(4) PSYoungGen: 227983K->14463K(264128K)
[17.806s][info][gc,heap        ] GC(4) ParOldGen: 280122K->66610K(613696K)
[17.806s][info][gc,metaspace   ] GC(4) Metaspace: 3743K->3743K(1056768K)
[17.806s][info][gc             ] GC(4) Pause Young (Allocation Failure)
                                           496M->79M(857M) 16.932ms
[17.086s][info][gc,cpu         ] GC(4) User=0.05s Sys=0.00s Real=0.02s
```

Информация не изменилась; она просто представлена в другом формате. Эта запись журнала состоит из нескольких строк; предыдущая запись на самом деле представляла собой одну строку (которую невозможно воспроизвести в формате книги). В этом журнале также выводятся размеры метапространства, но они не изменяются во время уборки в молодом поколении. Метапространство также не включается в общий размер кучи, выведенный в пятой строке примера.

На рис. 6.2 изображено состояние кучи до и после полной уборки мусора.

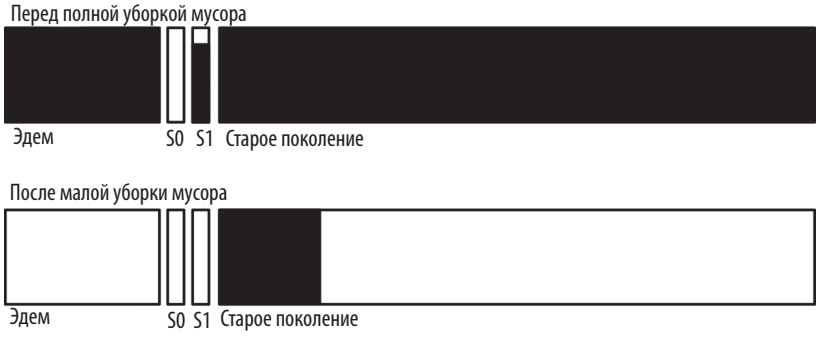


Рис. 6.2. Параллельная полная уборка мусора

Уборка в старом поколении освобождает все в молодом поколении. В старом поколении остаются только те объекты, на которые существуют активные ссылки, и все эти объекты подвергаются сжатию, так что начало старого поколения заполнено, а остаток свободен.

В журнале уборки мусора запись об этой операции выглядит так:

```
64.546: [Full GC (Ergonomics) [PSYoungGen: 15808K->0K(339456K)]
          [ParOldGen: 457753K->392528K(554432K)] 473561K->392528K(893888K)
          [Metaspace: 56728K->56728K(115392K)], 1.3367080 secs]
          [Times: user=4.44 sys=0.01, real=1.34 secs]
```

Молодое поколение теперь занимает 0 байт (а его размер составляет 339 Мбайт). Это означает, что области выживших также были очищены. Данные в старом поколении сократились с 457 Мбайт до 392 Мбайт, а следовательно, общее использование кучи сократилось с 473 Мбайт до 392 Мбайт. Размер метапространства не изменился; оно не обрабатывается в большинстве полных уборок мусора. (Если в метапространстве заканчивается место, то JVM проведет полную уборку мусора и вы увидите, что размер метапространства изменился; я продемонстрирую это чуть позже.) Так как полная уборка мусора требует значительно большего объема работы, она заняла 1,3 секунды реального времени и 4,4 секунды процессорного времени (снова для четырех параллельных потоков).

В JDK 11 аналогичный журнал для JDK выглядит так:

```
[63.205s][info][gc,start       ] GC(13) Pause Full (Ergonomics)
[63.205s][info][gc,phases,start] GC(13) Marking Phase
[63.314s][info][gc,phases      ] GC(13) Marking Phase 109.273ms
[63.314s][info][gc,phases,start] GC(13) Summary Phase
[63.316s][info][gc,phases      ] GC(13) Summary Phase 1.470ms
[63.316s][info][gc,phases,start] GC(13) Adjust Roots
[63.331s][info][gc,phases      ] GC(13) Adjust Roots 14.642ms
[63.331s][info][gc,phases,start] GC(13) Compaction Phase
[63.482s][info][gc,phases      ] GC(13) Compaction Phase 1150.792ms
[64.482s][info][gc,phases,start] GC(13) Post Compact
[64.546s][info][gc,phases      ] GC(13) Post Compact 63.812ms
[64.546s][info][gc,heap        ] GC(13) PSYoungGen: 15808K->0K(339456K)
[64.546s][info][gc,heap        ] GC(13) ParOldGen: 457753K->392528K(554432K)
[64.546s][info][gc,metaspace   ] GC(13) Metaspace: 56728K->56728K(115392K)
[64.546s][info][gc             ] GC(13) Pause Full (Ergonomics)
                                            462M->383M(823M) 1336.708ms
[64.546s][info][gc,cpu         ] GC(13) User=4.446s Sys=0.01s Real=1.34s
```

РЕЗЮМЕ

- Параллельный уборщик мусора выполняет две операции: малую и полную уборку мусора. Каждая из них выполняет пометку, освобождение и сжатие целевого поколения.
- Хронометражные данные из журнала уборки мусора позволяют определить общее влияние уборки мусора на приложение при использовании этих уборщиков.

Настройка адаптивного и статического определения размеров кучи

Настройка параллельного уборщика мусора сводится к времени паузы и соблюдению баланса между общим размером кучи и размерами старого/молодого поколения.

При этом приходится учитывать два компромиссных решения. Во-первых, действует классический компромисс между затратами времени и затратами памяти. Большая куча потребляет больше памяти на машине, но благодаря потреблению этой памяти (по крайней мере до определенной степени) приложение начинает работать более эффективно.

Второй компромисс относится к времени выполнения уборки мусора. Число полных пауз можно сократить, увеличивая размер кучи, но это может негативно отразиться на среднем времени отклика, потому что уборка мусора занимает больше времени. Аналогичным образом паузы полной уборки мусора можно сократить, выделяя для нового поколения бóльшую часть кучи, чем для старого, но в свою очередь это приведет к повышению частоты уборки мусора в старом поколении.

Последствия этих компромиссов представлены на рис. 6.3. На графике изображена максимальная производительность REST-сервера при разных размерах кучи. При малом размере кучи — 256 Мбайт — сервер проводит значительное время за уборкой мусора (36% общего времени); в результате производительность ограничивается. С увеличением размера кучи производительность быстро возрастает — пока размер кучи не достигнет 1500 Мбайт. После этого скорость роста производительности замедляется: приложение на этой стадии уже не привязано к уборке мусора (за которой проводится всего 6%). Начинает действовать закон убывающей отдачи: приложение может использовать дополнительную память для повышения производительности, но выигрыш становится менее заметным.

После достижения порога 4500 Мбайт производительность начинает слегка убывать. На этой стадии приложение достигло второй балансной точки: дополнительная память удлиняет циклы уборки мусора, а более долгие циклы — несмотря на их меньшую частоту — могут сократить общую производительность.

Данные на графике были получены при отключении адаптивного определения размеров в JVM; минимальному и максимальному размеру кучи были присвоены одинаковые значения. Вы можете провести эксперименты в любом приложении и определить оптимальные размеры для кучи и поколений, но чаще бывает проще поручить эти решения JVM (что обычно и происходит, так как адаптивное определение размеров включено по умолчанию).

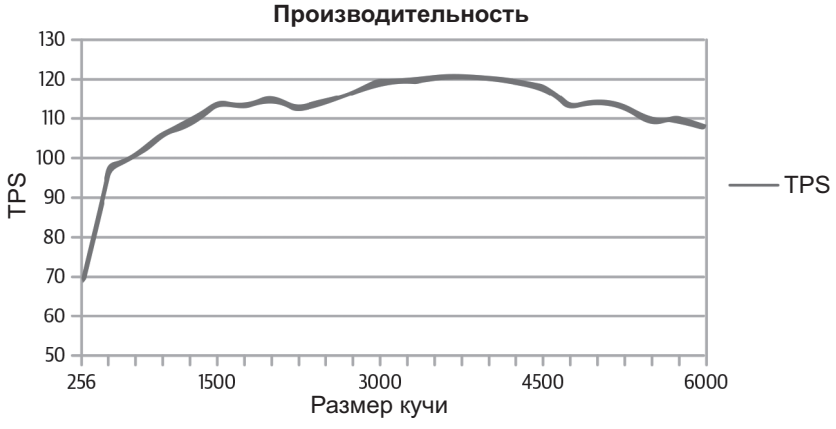


Рис. 6.3. Производительность при разных размерах кучи

Адаптивное определение размеров в параллельном уборщике изменяет размер кучи (и поколений) для соблюдения целей по продолжительности пауз. Эти цели задаются следующими флагами: `-XX:MaxGCPauseMillis=`*N* и `-XX:GCTimeRatio=`*N*.

Флаг `MaxGCPauseMillis` задает максимальную продолжительность паузы, приемлемую для приложения. Появляется искушение присвоить ему значение 0 или какую-нибудь малую величину (скажем, 50 мс). Учтите, что эта цель относится как к малой, так и к полной уборке мусора. При выборе очень малого значения у приложения будет очень маленькое старое поколение: такое, которое может быть очищено за 50 мс. В результате JVM будет очень, очень часто проводить полную уборку мусора, и производительность будет просто катастрофической. Будьте реалистом: используйте величину, которая может быть достигнута на практике. По умолчанию флаг не устанавливается.

Флаг `GCTimeRatio` задает время, которое приложение может проводить за уборкой мусора (относительно времени выполнения потоков приложения). Это соотношение, так что над значением *N* стоит немного поразмыслить. Оно используется в следующей формуле для определения процента времени, в течение которого (в идеале) должны выполняться потоки приложения:

$$ЦельПоПроизводительности = 1 - \frac{1}{(1 + GCTimeRatio)}.$$

Значение `GCTimeRatio` по умолчанию равно 99. Подставляя это значение в формулу, мы получаем 0,99 — это означает, что в соответствии с целью 99% времени приложение должно проводить за своими вычислениями, только 1% — за уборкой мусора. Но вас не должно сбивать с толку сочетание чисел в этом конкретном

случае. Если значение `GCTimeRatio` равно 95, это не означает, что уборка мусора должна занимать до 5% времени — на нее должно расходоваться до 1,94% времени.

Проще сначала определить минимальный процент времени, в течение которого приложение должно выполнять свою работу (скажем, 95%), а затем вычислить значение `GCTimeRatio` по следующей формуле:

$$GCTimeRatio = \frac{Производительность}{(1 - Производительность)}.$$

Если цель по производительности составляет 95% (0,95), то согласно формуле значение `GCTimeRatio` равно 19.

JVM использует эти два флага для задания размера кучи в границах, установленных исходным (`-Xms`) и максимальным (`-Xmx`) размером кучи. Флаг `MaxGCPauseMillis` обладает более высоким приоритетом: если он установлен, то размеры молодого и старого поколения регулируются до тех пор, пока не будет выполнена цель по продолжительности пауз. Когда это произойдет, общий размер кучи увеличивается до выполнения цели по соотношению времени. После достижения обеих целей JVM пытается сократить размер кучи для достижения наименьшего возможного размера, удовлетворяющего обеим целям.

Так как цель по продолжительности пауз по умолчанию не устанавливается, автоматическое определение размера кучи обычно приводит к тому, что размер кучи (и поколений) увеличивается до достижения цели `GCTimeRatio`. Однако на практике значение этого флага по умолчанию оказывается слишком оптимистичным. Конечно, ваш опыт может быть другим, но мне обычно попадались приложения, которые проводят за уборкой мусора от 3% до 6% и при этом работают вполне прилично. Иногда мне приходится работать над приложениями в средах с серьезной нехваткой памяти; такие приложения проводят от 10 до 15% времени за уборкой мусора. Уборка мусора серьезно влияет на производительность таких приложений, но общие цели производительности при этом выполняются. Таким образом, оптимальные настройки изменяются в зависимости от целей приложений. При отсутствии других целей я начинаю с соотношения 19 (5% времени расходуется на уборку мусора).

В табл. 6.1 перечислены эффекты динамической настройки в приложении с небольшим размером кучи и незначительной уборкой мусора (на примере REST-сервера с акциями, использующего малое количество объектов с долгим сроком жизни).

По умолчанию минимальный размер кучи составляет 64 Мбайт, а максимальный — 2 Гбайт (так как машина оснащена 8 Гбайт физической памяти). В этом случае `GCTimeRatio` работает так, как ожидалось: размер кучи динамически меняется до 649 Мбайт — точки, в которой приложение проводит около 1% общего времени за уборкой мусора.

Таблица 6.1. Эффект динамической настройки уборки мусора

Параметры уборки мусора	Конечный размер кучи	Процент времени на уборку мусора	OPS
По умолчанию	649 Мбайт	0,9%	9,2
`MaxGCPauseMillis=50ms`	560 Мбайт	1,0%	9,2
`Xms=Xmx=2048m`	2 Гбайт	0,04%	9,2

Настройка флага `MaxGCPauseMillis` в данном случае начинает сокращать размер кучи для соблюдения цели по продолжительности пауз. Так как объем работы у уборщика мусора в этом примере невелик, приложение проводит всего 1% общего времени за уборкой мусора при сохранении той же производительности 9,2 OPS.

Наконец, следует заметить, что больше — не всегда значит лучше. Полная куча на 2 Гбайт означает, что приложение сможет тратить меньше времени за уборкой мусора, но в данном случае уборка мусора не является доминирующим фактором производительности, так что производительность не возрастает. Как обычно, время, потраченное на оптимизацию не той части приложения, не окупается.

Если изменить то же приложение так, чтобы последние 50 запросов каждого пользователя сохранялись в глобальном кэше (например, в кэше JPA), уборщику мусора придется выполнять больше работы. В табл. 6.2 представлены некоторые соотношения для такой ситуации.

Таблица 6.2. Влияние заполнения кучи на динамическую настройку уборки мусора

Параметры уборки мусора	Конечный размер кучи	Процент времени на уборку мусора	OPS
По умолчанию	1,7 Гбайт	9,3%	8,4
`MaxGCPauseMillis=50ms`	588 Мбайт	15,1%	7,9
`Xms=Xmx=2048m`	2 Гбайт	5,1%	9,0
`Xmx=3560M; MaxGCRatio=19`	2,1 Гбайт	8,8%	9,0

В тесте, который проводит значительное время за уборкой мусора, поведение уборки мусора отличается. JVM никогда не сможет достичь цели по производительности 1% в этом тесте; она пытается по возможности приблизиться к цели по умолчанию и неплохо справляется со своей задачей, используя 1,7 Гбайт памяти.

Нереалистичная цель по продолжительности паузы ухудшает поведение приложения. Чтобы достичь времени уборки мусора в 50 мс, куча удерживается на уровне 588 Мбайт, но это означает, что уборки мусора становятся слишком частыми. Как следствие, производительность значительно ухудшается. В этом сценарии для достижения лучшей производительности следует приказать JVM использовать всю кучу, установив исходный и максимальный размеры равными 2 Гбайт.

Наконец, последняя строка таблицы показывает, что происходит, если для кучи определены разумные размеры, а для цели по продолжительности пауз выбран реалистичный уровень 5%. Сама JVM определила, что оптимальный размер кучи составляет приблизительно 2 Гбайт, и достигла такой же производительности, как в варианте с ручной настройкой.

РЕЗЮМЕ

- Динамическая настройка кучи — первый шаг определения ее размеров. В широком спектре применений ничего больше не понадобится, а динамические настройки минимизируют потребление памяти JVM.
- Статическая настройка размера кучи позволяет добиться максимальной возможной производительности. Размеры, определяемые JVM для разумного набора целей производительности, станут хорошей отправной точкой для такой настройки.

Уборщик мусора G1

Уборщик мусора G1 работает с отдельными областями кучи. Каждая область (по умолчанию их около 2048) может принадлежать старому или новому поколению, и области поколений не обязаны занимать смежную область в памяти. Идея определения областей в старом поколении основана на том, что когда конкурентно работающие фоновые потоки ищут объекты, на которые нет ссылок, некоторые области будут содержать больше мусора, чем другие. Фактическая уборка мусора все равно требует остановки потоков приложений, но алгоритм G1 может сосредоточиться на областях, которые в основном состоят из мусора, и проводить лишь незначительное время за очисткой этих областей. Именно от такого подхода — очистки только областей, занятых в основном мусором, — происходит название алгоритма G1: «Garbage First», то есть «сначала мусор».

Это не относится к областям молодого поколения: в ходе уборки мусора в молодом поколении все молодое поколение либо освобождается, либо повышается (переводится в область выживших или в старое поколение). При этом молодое

поколение определяется в областях — отчасти из-за того, что заранее определенные области существенно упрощают изменение размеров поколений.

Алгоритм G1 называется *конкурентным*, потому что пометка свободных объектов в старом поколении происходит конкурентно с работой потоков приложения (то есть они продолжают работать). Тем не менее речь не идет о полной конкурентности, потому что пометка и сжатие молодого поколения требуют остановки всех потоков приложения, а сжатие старого поколения также происходит во время остановки потоков приложения.

Уборка мусора G1 состоит из четырех логических операций:

- Уборка в молодом поколении.
- Фоновый конкурентный цикл пометки.
- Смешанная уборка.
- При необходимости — полная уборка мусора.

Рассмотрим все эти операции поочередно, начиная с уборки G1 в молодом поколении (рис. 6.4).

Каждый квадратик на диаграмме представляет область уборки мусора G1. Данные в каждой области представляются черным цветом, а буква обозначает поколение, к которому принадлежит область: [E] — Эдем (Eden), [O] — старое поколение (Old generation), [S] — область выживших (Survivor space). Пустые области не принадлежат поколениям; алгоритм G1 произвольно использует их для того поколения, которое сочтет нужным.

Рис. 6.4. Уборка мусора G1 в молодом поколении

Уборка G1 в молодом поколении инициируется при заполнении Эдема (в данном примере после заполнения четырех областей). После уборки мусора Эдем

остается пустым (хотя для него выделены области, которые начнут заполняться данными при дальнейшей работе приложения). По крайней мере одна область выделена под область выживших (частично заполненная в данном примере), и часть данных была перемещена в старое поколение.

В журнале процесса данные уборки мусора G1 несколько отличаются от других уборщиков. Пример для JDK 8 был получен при помощи `PrintGCDetails`, но для G1 выводится более подробная информация. В приведенных примерах представлены лишь самые важные строки.

Стандартная уборка мусора в молодом поколении:

```
23.430: [GC pause (young), 0.23094400 secs]
...
   [Eden: 1286M(1286M)->0B(1212M)
    Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)]
   [Times: user=0.85 sys=0.05, real=0.23 secs]
```

Уборка в молодом поколении заняла 0,23 секунды реального времени, за которые потоки уборки мусора потребили 0,85 секунды процессорного времени. 1286 Мбайт объектов были перемещены из Эдема (которому был адаптивно назначен размер 1212 Мбайт); 74 Мбайт из них были перемещены в область выживших (она увеличилась в размерах с 78 Мбайт до 152 Мбайт), а остальные были освобождены. Мы знаем, что они были освобождены, наблюдая за тем, что общее заполнение кучи снизилось до 1212 Мбайт. В общем случае некоторые объекты из области выживших могли быть перемещены в старое поколение, а если область выживших была переполнена, то некоторые объекты из Эдема могли быть переведены непосредственно в старое поколение — в этих случаях размер старого поколения увеличится.

В JDK 11 аналогичный журнал выглядит так:

```
[23.200s][info   ][gc,start     ] GC(10) Pause Young (Normal)
                                           (G1 Evacuation Pause)
[23.200s][info   ][gc,task      ] GC(10) Using 4 workers of 4 for evacuation
[23.430s][info   ][gc,phases    ] GC(10)   Pre Evacuate Collection Set: 0.0ms
[23.430s][info   ][gc,phases    ] GC(10)   Evacuate Collection Set: 230.3ms
[23.430s][info   ][gc,phases    ] GC(10)   Post Evacuate Collection Set: 0.5ms
[23.430s][info   ][gc,phases    ] GC(10)   Other: 0.1ms
[23.430s][info   ][gc,heap      ] GC(10) Eden regions: 643->606(606)
[23.430s][info   ][gc,heap      ] GC(10) Survivor regions: 39->76(76)
[23.430s][info   ][gc,heap      ] GC(10) Old regions: 67->75
[23.430s][info   ][gc,heap      ] GC(10) Humongous regions: 0->0
[23.430s][info   ][gc,metaspace ] GC(10) Metaspace: 18407K->18407K(1067008K)
[23.430s][info   ][gc           ] GC(10) Pause Young (Normal)
                                           (G1 Evacuation Pause)
                                           1454M(4096M)->242M(4096M) 230.104ms
[23.430s][info   ][gc,cpu       ] GC(10) User=0.85s Sys=0.05s Real=0.23s
```

Конкурентный цикл уборки мусора G1 начинается и завершается так, как показано на рис. 6.5.

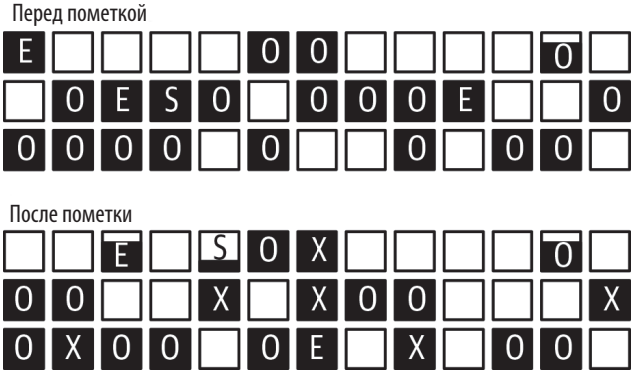


Рис. 6.5. Конкурентная уборка мусора G1

На диаграмме представлены три важных момента, на которые следует обратить внимание. Во-первых, молодое поколение изменило заполнение; в конкурентном цикле будет по крайней мере одна (а возможно, и более) уборка мусора в молодом поколении. А следовательно, области Эдема перед циклом пометки были полностью освобождены и началось выделение новых областей Эдема.

Во-вторых, некоторые области теперь снабжены пометкой X. Эти области принадлежат старому поколению (обратите внимание: они все еще содержат данные) — области, которые, как было определено в ходе пометки, состоят в основном из мусора.

Наконец, заметим, что старое поколение (состоящее из областей с пометкой O или X) в действительности обладает большим заполнением после завершения цикла. Это связано с тем, что уборки в молодом поколении, происходящие в цикле пометки, переместили данные в старое поколение. Кроме того, цикл пометки не освобождает никаких данных в старом поколении; он просто идентифицирует области, которые состоят в основном из мусора. Данные из этих областей будут освобождены в более позднем цикле.

Конкурентный цикл уборки мусора G1 состоит из нескольких фаз. В одних фазах все потоки приложения приостанавливаются, в других нет. Первая фаза называется *исходной пометкой* (в JDK 8) или *запуском конкурентной обработки* (в JDK 11). В этой фазе останавливаются все потоки приложений — отчасти потому, что в ней также выполняется уборка мусора в молодом поколении и она подготавливает следующие фазы цикла.

В JDK 8 это выглядит так:

```
50.541: [GC pause (G1 Evacuation pause) (young) (initial-mark), 0.27767100 secs]
    ... много других данных ...
    [Eden: 1220M(1220M)->0B(1220M)
     Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)]
    [Times: user=1.02 sys=0.04, real=0.28 secs]
```

То же в JDK 11:

```
[50.261s][info   ][gc,start      ] GC(11) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
[50.261s][info   ][gc,task       ] GC(11) Using 4 workers of 4 for evacuation
[50.541s][info   ][gc,phases     ] GC(11)   Pre Evacuate Collection Set: 0.1ms
[50.541s][info   ][gc,phases     ] GC(11)   Evacuate Collection Set: 25.9ms
[50.541s][info   ][gc,phases     ] GC(11)   Post Evacuate Collection Set: 1.7ms
[50.541s][info   ][gc,phases     ] GC(11)   Other: 0.2ms
[50.541s][info   ][gc,heap       ] GC(11) Eden regions: 1220->0(1220)
[50.541s][info   ][gc,heap       ] GC(11) Survivor regions: 144->144(144)
[50.541s][info   ][gc,heap       ] GC(11) Old regions: 1875->1946
[50.541s][info   ][gc,heap       ] GC(11) Humongous regions: 3->3
[50.541s][info   ][gc,metaspace  ] GC(11) Metaspace: 52261K->52261K(1099776K)
[50.541s][info   ][gc            ] GC(11) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
                                          1220M->0B(1220M) 280.055ms
[50.541s][info   ][gc,cpu        ] GC(11) User=1.02s Sys=0.04s Real=0.28s
```

Как и при обычной уборке мусора в молодом поколении, потоки приложения были остановлены (на 0,28 секунды), а молодое поколение было очищено (так что Эдем начинается с размера 0). 71 Мбайт данных был перемещен из молодого поколения в старое. В JDK 8 понять это не так просто (приходится вычислять результат 2093 – 3242 + 1220); в выводе JDK 11 этот факт лучше виден.

С другой стороны, в выводе JDK 11 присутствует ряд моментов, которые еще не упоминались. Во-первых, размеры областей задаются в областях, а не в мегабайтах. О размерах мы еще поговорим в этой главе, но в этом примере размер одной области составляет 1 Мбайт. Кроме того, в выводе JDK 11 упоминается новое пространство: *огромные* (humongous) области. Они являются частью старого поколения и также будут более подробно рассмотрены в этой главе.

Сообщение исходной пометки (или запуска конкурентной обработки) указывает, что цикл фоновой конкурентной обработки начался. Так как исходная пометка в фазе цикла пометки также требует остановки всех потоков приложения, уборщик G1 пользуется циклом уборки в молодом поколении для выполнения этой работы. Последствия от добавления фазы исходной пометки в уборку мусора в молодом поколении не столь значительны: она увеличивает затраты процессорного времени на 20% по сравнению с предыдущей уборкой (то есть обычной уборкой в молодом поколении), хотя пауза увеличилась совсем незначительно.

(К счастью, на машине хватало свободных ресурсов процессора для параллельных потоков G1, иначе пауза получилась бы более продолжительной.)

Затем уборщик G1 сканирует корневую область:

```
50.819: [GC concurrent-root-region-scan-start]
51.408: [GC concurrent-root-region-scan-end, 0.5890230]

[50.819s][info ][gc             ] GC(20) Concurrent Cycle
[50.819s][info ][gc,marking     ] GC(20) Concurrent Clear Claimed Marks
[50.828s][info ][gc,marking     ] GC(20) Concurrent Clear Claimed Marks 0.008ms
[50.828s][info ][gc,marking     ] GC(20) Concurrent Scan Root Regions
[51.408s][info ][gc,marking     ] GC(20) Concurrent Scan Root Regions 589.023ms
```

Процесс занимает 0,58 секунды, но он не останавливает потоки приложения; используются только фоновые потоки. Тем не менее эта фаза не может быть прервана уборкой мусора в молодом поколении, поэтому наличие доступных ресурсов процессора для этих фоновых потоков критично. Если молодое поколение окажется заполненным во время сканирования корневой области, уборка в молодом поколении (которая останавливает все потоки приложения) должна дождаться завершения сканирования корневой области. По сути это означает, что для уборки в молодом поколении потребуется пауза большей длительности, чем обычно. Эта ситуация представлена в журнале уборки мусора так:

```
350.994: [GC pause (young)
  351.093: [GC concurrent-root-region-scan-end, 0.6100090]
  351.093: [GC concurrent-mark-start],
  0.37559600 secs]

[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions
[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions 610.364ms
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark (350.994s)
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark From Roots
[350.994s][info][gc,task      ] GC(50) Using 1 workers of 1 for marking
[350.994s][info][gc,start     ] GC(51) Pause Young (Normal) (G1 Evacuation
                                       Pause)
```

Пауза уборки мусора начинается до конца сканирования корневой области. В JDK 8 чередование вывода в журнале указывает на то, что уборке в молодом поколении пришлось сделать паузу для завершения сканирования корневой области, прежде чем она смогла продолжить работу. В JDK 11 обнаружить этот факт немного сложнее: для этого нужно заметить, что временная метка конца сканирования корневой области в точности совпадает с меткой начала следующей уборки мусора в молодом поколении.

В любом случае невозможно точно определить, на какое время была отложена уборка в молодом поколении. Величина задержки не обязательно составила все 610 мс в данном примере; какое-то время (до того, как было заполнено моло-

дое поколение) работа продолжалась. Но в этом случае по временным меткам видно, что потоки приложения ожидали лишние 100 мс — вот почему продолжительность паузы при уборке молодого поколения приблизительно на 100 мс больше средней продолжительности других пауз в этом журнале. (Если такое происходит часто, это признак того, что уборку мусора G1 необходимо лучше настроить, как описано в следующем разделе.)

После сканирования корневой области уборщик G1 входит в фазу конкурентной пометки. Это происходит полностью в фоновом режиме; в начале и в конце выводятся сообщения:

```
111.382: [GC concurrent-mark-start]
....
120.905: [GC concurrent-mark-end, 9.5225160 sec]
```

```
[111.382s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s)
[111.382s][info][gc,marking   ] GC(20) Concurrent Mark From Roots
...
[120.905s][info][gc,marking   ] GC(20) Concurrent Mark From Roots 9521.994ms
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean 0.522ms
[120.910s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s, 120.910s)
                                       9522.516ms
```

Конкурентная пометка может быть прервана, так что уборки в молодом поколении могут происходить в этой фазе (на месте многоточий будет длинный вывод уборки мусора).

Также обратите внимание на то, что в примере JDK 11 вывод включает тот же идентификатор уборки мусора (20), как у записи, в которой произошло сканирование корневой области. Операция разбивается с большей детализацией, чем система ведения журнала JDK: в JDK все фоновое сканирование считается одной операцией. Мы разбиваем обсуждение на более мелкие логические операции, потому что, например, сканирование корневой области может создать паузу, тогда как конкурентная пометка — нет.

За фазой пометки следует фаза повторной пометки и нормальная фаза очистки:

```
120.910: [GC remark 120.959:
        [GC ref-PRC, 0.0000890 secs], 0.0718990 secs]
  [Times: user=0.23 sys=0.01, real=0.08 secs]
120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs]
  [Times: user=0.04 sys=0.00, real=0.01 secs]
```

```
[120.909s][info][gc,start     ] GC(20) Pause Remark
[120.909s][info][gc,stringtable] GC(20) Cleaned string and symbol table,
                                        strings: 1369 processed, 0 removed,
                                        symbols: 17173 processed, 0 removed
[120.985s][info][gc            ] GC(20) Pause Remark 2283M->862M(3666M) 80.412ms
[120.985s][info][gc,cpu        ] GC(20) User=0.23s Sys=0.01s Real=0.08s
```

В этих фазах потоки приложений останавливаются, хотя обычно на непродолжительное время. Затем конкурентно происходит дополнительная фаза очистки:

```
120.996: [GC concurrent-cleanup-start]
120.996: [GC concurrent-cleanup-end, 0.0004520]

[120.878s][info][gc,start      ] GC(20) Pause Cleanup
[120.879s][info][gc            ] GC(20) Pause Cleanup 1313M->1313M(3666M) 1.192ms
[120.879s][info][gc,cpu        ] GC(20) User=0.00s Sys=0.00s Real=0.00s
[120.879s][info][gc,marking    ] GC(20) Concurrent Cleanup for Next Mark
[120.996s][info][gc,marking    ] GC(20) Concurrent Cleanup for Next Mark
                                        117.168ms
[120.996s][info][gc            ] GC(20) Concurrent Cycle 70,177.506ms
```

В этом случае нормальный фоновый цикл пометки G1 завершен — по крайней мере в том, что касается уборки мусора. Однако пока еще почти ничего не было реально освобождено. Небольшой объем памяти был освобожден в фазе очистки, но к настоящему моменту уборщик мусора G1 всего лишь выявил старые области, которые состоят в основном из мусора и могут быть освобождены (на рис. 6.5 они помечены знаком X).

Теперь уборщик мусора G1 выполняет серию смешанных уборок мусора. Они называются «смешанными» (mixed), потому что выполняется нормальная уборка мусора в молодом поколении, но также в ней задействованы некоторые помеченные области из фонового сканирования. Эффект смешанной уборки мусора показан на рис. 6.6.

Рис. 6.6. Смешанная уборка, выполняемая уборщиком мусора G1

Как обычно в молодом поколении, уборка мусора G1 полностью опустошила Эдем и отрегулировала области выживших. Кроме того, была проведена уборка в двух помеченных регионах. Было заранее известно, что эти области в основ-

ном содержат мусор, так что большая их часть была освобождена. Все «живые» данные в этих областях были перемещены в другую область (по аналогии с тем, как «живые» данные перемещались из молодого поколения в области старого поколения). Так уборка мусора G1 производит сжатие старого поколения — такое перемещение объектов фактически сжимает кучу по ходу уборки мусора G1.

Смешанная операция уборки мусора обычно выглядит в журнале примерно так:

```
79.826: [GC pause (mixed), 0.26161600 secs]
....
   [Eden: 1222M(1222M)->0B(1220M)
    Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)]
   [Times: user=1.01 sys=0.00, real=0.26 secs]

[3.800s][info][gc,start      ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
[3.800s][info][gc,task       ] GC(24) Using 4 workers of 4 for evacuation
[3.800s][info][gc,phases     ] GC(24)   Pre Evacuate Collection Set: 0.2ms
[3.825s][info][gc,phases     ] GC(24)   Evacuate Collection Set: 250.3ms
[3.826s][info][gc,phases     ] GC(24)   Post Evacuate Collection Set: 0.3ms
[3.826s][info][gc,phases     ] GC(24)   Other: 0.4ms
[3.826s][info][gc,heap       ] GC(24) Eden regions: 1222->0(1220)
[3.826s][info][gc,heap       ] GC(24) Survivor regions: 142->144(144)
[3.826s][info][gc,heap       ] GC(24) Old regions: 1834->1820
[3.826s][info][gc,heap       ] GC(24) Humongous regions: 4->4
[3.826s][info][gc,metaspace  ] GC(24) Metaspace: 3750K->3750K(1056768K)
[3.826s][info][gc            ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
                                          3791M->3791M(3983M) 124.390ms
[3.826s][info][gc,cpu        ] GC(24) User=1.01s Sys=0.00s Real=0.26s
[3.826s][info][gc,start      ] GC(25) Pause Young (Mixed) (G1 Evacuation Pause)
```

Следует заметить, что использование кучи сократилось на величину, превышающую 1,222 Мбайт, удаленных из Эдема. Разность (16 Мбайт) кажется небольшой, но я напомню, что часть пространства выживших была конкурентно перемещена в старое поколение; кроме того, каждая смешанная уборка мусора освобождает только часть целевых областей старого поколения.

В дальнейшем вы увидите, что очень важно позаботиться о том, что смешанные уборки мусора очищают достаточно памяти для предотвращения будущих конфликтов конкурентного доступа.

В JDK 11 первая смешанная уборка мусора снабжается пометкой `Prepared Mixed` и немедленно следует за конкурентной очисткой.

Смешанные циклы уборки мусора продолжаются до тех пор, пока (почти) во всех помеченных областях не будет проведена уборка мусора; в этой точке уборщик G1 возобновляет обычные циклы уборки молодого поколения. Со временем уборщик G1 запустит еще один конкурентный цикл для определения того, какие области в старом поколении должны быть освобождены следующими.

Хотя для смешанных циклов уборки мусора обычно указывается причина (`Mixed`), уборки в молодом поколении иногда помечаются обычным образом по схеме конкурентного цикла (например, `G1 Evacuation Pause`). Если конкурентный цикл найдет в старом поколении области, которые могут быть полностью освобождены, эти области освобождаются в ходе обычной паузы перемещения молодого поколения. С технической точки зрения это не является смешанным циклом в реализации уборщика мусора. Однако с логической точки зрения это так и есть: объекты освобождаются из молодого поколения или перемещаются в старое поколение, и в то же время мусорные объекты (вернее, области) освобождаются из старого поколения.

Если все проходит нормально, то это весь набор операций уборки мусора, которые будут присутствовать в журнале. Но нужно рассматривать некоторые аномальные ситуации.

Иногда в журнале отображается полная уборка мусора, это говорит о том, что дополнительная настройка (возможно, с включением дополнительной памяти кучи) положительно скажется на производительности приложения. Она инициируется в четырех случаях:

- *Сбой конкурентного режима* — уборщик мусора G1 запускает цикл пометки, но старое поколение заполняется до завершения цикла. В этом случае уборщик G1 отменяет цикл пометки:

```
51.408: [GC concurrent-mark-start]
65.473: [Full GC 4095M->1395M(4096M), 6.1963770 secs]
 [Times: user=7.87 sys=0.00, real=6.20 secs]
71.669: [GC concurrent-mark-abort]
```

```
[51.408][info][gc,marking     ] GC(30) Concurrent Mark From Roots
...
[65.473][info][gc             ] GC(32) Pause Full (G1 Evacuation Pause)
                                       4095M->1305M(4096M) 60,196.377
...
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark From Roots 191ms
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark Abort
```

  Этот сбой означает, что размер кучи следует увеличить, что фоновая обработка G1 должна начаться быстрее или что цикл необходимо настроить для ускорения выполнения (например, за счет добавления фоновых потоков). Ниже рассказано о том, как это сделать.

- *Сбой повышения* — уборщик мусора G1 завершил цикл пометки и начал выполнение смешанных уборок мусора для очистки старых областей. Прежде чем он смог очистить достаточно места, слишком большое количество объектов было переведено из молодого поколения, и в старом поколении все равно не хватает места. В журнале за смешанной уборкой мусора немедленно следует полная уборка:

```
2226.224: [GC pause (mixed)
        2226.440: [SoftReference, 0 refs, 0.0000060 secs]
        2226.441: [WeakReference, 0 refs, 0.0000020 secs]
        2226.441: [FinalReference, 0 refs, 0.0000010 secs]
        2226.441: [PhantomReference, 0 refs, 0.0000010 secs]
        2226.441: [JNI Weak Reference, 0.0000030 secs]
                (to-space exhausted), 0.2390040 secs]
....
    [Eden: 0.0B(400.0M)->0.0B(400.0M)
     Survivors: 0.0B->0.0B Heap: 2006.4M(2048.0M)->2006.4M(2048.0M)]
    [Times: user=1.70 sys=0.04, real=0.26 secs]
2226.510: [Full GC (Allocation Failure)
        2227.519: [SoftReference, 4329 refs, 0.0005520 secs]
        2227.520: [WeakReference, 12646 refs, 0.0010510 secs]
        2227.521: [FinalReference, 7538 refs, 0.0005660 secs]
        2227.521: [PhantomReference, 168 refs, 0.0000120 secs]
        2227.521: [JNI Weak Reference, 0.0000020 secs]
                2006M->907M(2048M), 4.1615450 secs]
    [Times: user=6.76 sys=0.01, real=4.16 secs]
```

```
[2226.224s][info][gc            ] GC(26) Pause Young (Mixed)
                                           (G1 Evacuation Pause)
                                           2048M->2006M(2048M) 26.129ms
...
[2226.510s][info][gc,start      ] GC(27) Pause Full (G1 Evacuation Pause)
```

Этот сбой означает, что смешанные уборки мусора должны выполняться быстрее; каждая уборка мусора в молодом поколении должна обрабатывать больше областей в старом поколении.

- *Сбой эвакуации* — при выполнении уборки в молодом поколении не хватает места для хранения всех выживших объектов в областях выживших и в старом поколении. В журналах уборки мусора это выглядит как особая разновидность уборки мусора в молодом поколении:

```
60.238: [GC pause (young) (to-space overflow), 0.41546900 secs]
```

```
[60.238s][info][gc,start       ] GC(28) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
[60.238s][info][gc,task        ] GC(28) Using 4 workers of 4
                                          for evacuation
[60.238s][info][gc             ] GC(28) To-space exhausted
```

Это признак того, что куча в основном заполнена или фрагментирована. Уборщик G1 пытается компенсировать этот факт, но, скорее всего, это кончится плохо: JVM прибегнет к полной уборке мусора. Проблема проще всего решается повышением размера кучи, хотя возможные решения приводятся в разделе «Расширенная настройка», с. 225.

- *Сбой создания огромных объектов* — приложения, создающие очень большие объекты, могут инициировать другую разновидность полной уборки

мусора в уборщике G1; за дополнительной информацией (в том числе и той, как ее избежать) обращайтесь к разделу «Создание огромных объектов уборщиком мусора G1», с. 236. В JDK 8 диагностировать эту ситуацию невозможно без использования специальных параметров управления журналом, но в JDK 11 эта информация отображается в следующей записи:

```
[3023.091s][info][gc,start     ] GC(54) Pause Full (G1 Humongous Allocation)
```

- *Порог уборки мусора в метаданных* — как я уже упоминал, метапространство фактически представляет собой отдельную кучу, а уборка мусора в нем осуществляется независимо от основной кучи. Оно не чистится уборщиком мусора G1, но когда возникает необходимость в его уборке в JDK 8, уборщик G1 выполняет полную уборку мусора (непосредственно после уборки мусора в молодом поколении) с основной кучей:

```
0.0535: [GC (Metadata GC Threshold) [PSYoungGen: 34113K->20388K(291328K)]
    73838K->60121K(794112K), 0.0282912 secs]
    [Times: user=0.05 sys=0.01, real=0.03 secs]
0.0566: [Full GC (Metadata GC Threshold) [PSYoungGen: 20388K->0K(291328K)]
    [ParOldGen: 39732K->46178K(584192K)] 60121K->46178K(875520K),
    [Metaspace: 59040K->59036K(1101824K)], 0.1121237 secs]
    [Times: user=0.28 sys=0.01, real=0.11 secs]
```

В JDK 11 уборка/изменение размеров метапространства могут выполняться без полной уборки мусора.

РЕЗЮМЕ

- Уборщик мусора G1 использует несколько циклов (и фаз в конкурентном цикле). На хорошо настроенной JVM, на которой работает уборщик G1, должны происходить только циклы уборки в молодом поколении, смешанной и конкурентной уборки мусора.
- В работе алгоритма происходят небольшие паузы для конкурентных фаз G1.
- Уборщика G1 следует настроить так, чтобы по возможности предотвратить циклы полной уборки мусора.

Настройка уборщика мусора G1

Главная цель настройки уборщика G1 — гарантировать, что сбои конкурентного режима или сбоев эвакуации не потребуют полной уборки мусора. Методы, используемые для предотвращения полной уборки мусора, также могут ис-

пользоваться в ситуациях, когда частые уборки мусора в молодом поколении должны дожидаться завершения сканирования корневой области.

Настройка для предотвращения полной уборки мусора критична в JDK 8, потому что при выполнении полной уборки мусора в JDK 8 уборщик G1 использует только один поток. В результате продолжительность паузы оказывается больше обычного. В JDK 11 полная уборка мусора выполняется несколькими потоками, что способствует сокращению пауз (фактически продолжительность паузы получается такой же, как при полной уборке мусора параллельным уборщиком). Это отличие — одна из причин, по которым стоит обновиться до JDK 11, если вы используете уборщика G1 (хотя приложение JDK 8, избегающее полных уборок мусора, будет работать нормально). Во-вторых, настройка позволяет минимизировать паузы, возникающие в ходе работы.

Для предотвращения полной уборки мусора возможны следующие варианты:

- Увеличить размер старого поколения либо за счет увеличения общего размера кучи, либо регулировкой соотношения между поколениями.
- Увеличить количество фоновых потоков (при наличии достаточных ресурсов процессора).
- Чаще выполнять фоновые операции уборщика мусора G1.
- Увеличить объем работы, выполняемой в циклах смешанной уборки.

Здесь может применяться множество разных настроек, но одна из целей уборщика G1 заключается в том, что он не должен нуждаться в такой настройке. Собственно, основная настройка уборщика G1 осуществляется всего одним флагом: тем же флагом `-XX:MaxGCPauseMillis=`*N*, который использовался для настройки параллельного уборщика.

Для уборщика G1 (в отличие от параллельного уборщика) у этого флага есть значение по умолчанию: 200 мс. Если паузы для глобальных фаз уборщика G1 начинают превышать это значение, уборщик G1 пытается исправить ситуацию — отрегулировать соотношение между молодым и старым поколением, корректировать размер кучи, начинать фоновую обработку на более ранней стадии, изменять порог хранения и (что самое важное) обрабатывать больше или меньше областей старого поколения во время цикла смешанной уборки мусора.

Здесь приходится учитывать некоторые компромиссы: при уменьшении значения размер молодого поколения сократится для соблюдения цели по продолжительности паузы, но при этом уборка мусора в молодом поколении будет выполняться чаще. Кроме того, количество областей старого поколения, которые могут быть обработаны в ходе смешанной уборки мусора, уменьшится для соблюдения цели по продолжительности паузы, а это повышает риск сбоя конкурентного режима.

Если настройка цели по продолжительности пауз не препятствует полной уборке мусора, эти аспекты можно настраивать по отдельности. Настройка размеров кучи для уборщика G1 осуществляется так же, как и для других алгоритмов уборки мусора.

Настройка фоновых потоков G1

В каком-то отношении конкурентная пометка при уборке мусора G1 вступает в гонку с потоками приложения: уборщик G1 должен очищать старое поколение быстрее, чем приложение переводит в него новые данные. Чтобы это условие выполнялось, попробуйте увеличить количество потоков фоновой пометки (предполагается, что на машине хватает ресурсов процессора).

Уборщик G1 использует два набора потоков. Первым набором управляет флаг `-XX:ParallelGCThreads=`*N*, впервые упомянутый в главе 5. Этот флаг влияет на количество потоков, используемых в фазах с приостановкой потоков приложения: уборки в молодом поколении и смешанной уборки, а также фаз конкурентного цикла повторной пометки, когда потоки должны быть остановлены. Второй флаг `-XX:ConcGCThreads=`*N* влияет на количество потоков, используемых для конкурентной повторной пометки.

Значение флага `ConcGCThreads` определяется следующим образом:

```
ConcGCThreads = (ParallelGCThreads + 2) / 4
```

В операции выполняется целочисленное деление, поэтому в конфигурации с пятью и менее параллельными потоками будет использоваться один фоновый поток сканирования, от шести до девяти параллельных потоков — два фоновых потока сканирования, и т. д.

Повышение количества фоновых потоков сканирования сокращает конкурентный цикл, благодаря чему уборщикам мусора G1 будет проще завершить освобождение старого поколения во время циклов смешанной уборки мусора, прежде чем другие потоки смогут снова заполнить его. Как обычно, предполагается, что на машине имеются доступные ресурсы процессора: в противном случае сканирующие потоки будут отбирать процессор у приложения и фактически создавать паузы в его выполнении, как было показано при сравнении последовательного уборщика мусора с G1 в главе 5.

Настройка уборщика G1 для изменения частоты выполнения

Уборщик G1 также может выиграть свою гонку, если цикл фоновой пометки запустится в более ранний момент. Этот цикл запускается при достижении кучей соотношения заполнения, определяемого флагом `-XX:InitiatingHeapOccupancy`

`Percent=N`, который имеет значение по умолчанию 45. Этот процент относится ко всей куче, а не только к старому поколению.

Значение `InitiatingHeapOccupancyPercent` остается постоянным; уборщик G1 никогда не изменяет его в попытках соблюдения целей по продолжительности пауз. Если задать слишком высокое значение, приложение в итоге начнет выполнять полную уборку мусора, потому что конкурентные фазы не успевают завершиться, пока заполняется остаток кучи. Если значение будет слишком малым, то приложение выполнит больше фоновой уборки мусора, чем могло бы в противном случае.

Конечно, в какой-то момент эти фоновые потоки все равно должны отработать, поэтому можно считать, что на машине хватает ресурсов процессора для их выполнения. Но слишком частое их выполнение приведет к значительным потерям для быстродействия, потому что в работе приложения будут чаще возникать мелкие паузы для конкурентных фаз, приостанавливающих потоки приложения. Эти паузы быстро накапливаются, поэтому слишком частого выполнения фоновой очистки G1 следует избегать. Проверьте размер кучи после конкурентного цикла и убедитесь в том, что параметру `InitiatingHeapOccupancyPercent` присвоено более высокое значение.

Настройка циклов смешанной уборки мусора G1

После конкурентного цикла уборщик мусора G1 не может начать новый конкурентный цикл, пока не будет завершена уборка во всех ранее помеченных областях старого поколения. Следовательно, для того чтобы уборщик G1 начинал цикл пометки в более ранний момент времени, также можно обрабатывать больше областей в цикле смешанной уборки мусора (что в конечном счете приведет к уменьшению количества циклов смешанной уборки).

Объем работы, выполняемой при смешанной уборке мусора, зависит от трех факторов. Первый — количество областей, которые изначально состояли в основном из мусора. Напрямую влиять на него невозможно: область объявляется пригодной для обработки в ходе смешанной уборки мусора, если она состоит из мусора на 85%.

Второй фактор — максимальное количество циклов смешанной уборки мусора, в которых уборщик G1 обрабатывает эти регионы; оно задается значением флага `-XX:G1MixedGCCountTarget=N`. Значение по умолчанию равно 8; уменьшение этого значения помогает избежать сбоев повышения (за счет удлинения пауз между циклами смешанной уборки).

С другой стороны, если паузы смешанной уборки мусора занимают слишком много времени, это значение можно увеличить, чтобы во время смешанной уборки выполнялось меньше работы. Только обязательно следите за тем,

чтобы увеличнеие этого числа не задержало следующий конкурентный цикл G1 на слишком долгое время, иначе может произойти сбой конкурентного режима.

Наконец, третьим фактором является максимально допустимая длина паузы уборки мусора (то есть значение `MaxGCPauseMillis`). Количество смешанных циклов, задаваемое флагом `G1MixedGCCountTarget`, определяет верхнюю границу; если в пределах целевой продолжительности паузы остается время, уборщик G1 обработает более 1/8 (или другого заданного значения) помеченных областей старого поколения. Увеличение значения флага `MaxGCPauseMillis` позволяет обрабатывать больше областей старого поколения в ходе одной смешанной уборки мусора, что в свою очередь может позволить уборщику G1 скорее начать следующий конкурентный цикл.

РЕЗЮМЕ

- Настройку уборки мусора G1 следует начать с выбора разумного целевого времени продолжительности паузы.
- Если после этого полная уборка мусора все еще создает проблемы, а размер кучи не может быть увеличен, применяются конкретные настройки в зависимости от типа сбоев:
  - чтобы фоновые потоки выполнялись чаще, измените `InitiatingHeapOccupancyPercent`;
  - если доступны дополнительные процессоры, отрегулируйте количество потоков при помощи флага `ConcGCThreads flag`;
  - чтобы предотвратить сбои поведения, уменьшите значение `G1MixedGCCountTarget`.

Уборщик мусора CMS

Хотя уборщик мусора CMS считается устаревшим, он все еще доступен в текущих сборках JDK. Из-за этого я расскажу, как его настроить и почему он считается устаревшим.

Алгоритм CMS состоит из трех основных операций:

- Уборка в молодом поколении (с остановкой всех потоков приложения).
- Выполнение конкурентного цикла для стирания данных из старого поколения.
- Выполнение полной уборки мусора для сжатия старого поколения в случае необходимости.

Уборка молодого поколения в алгоритме CMS изображена на рис. 6.7.

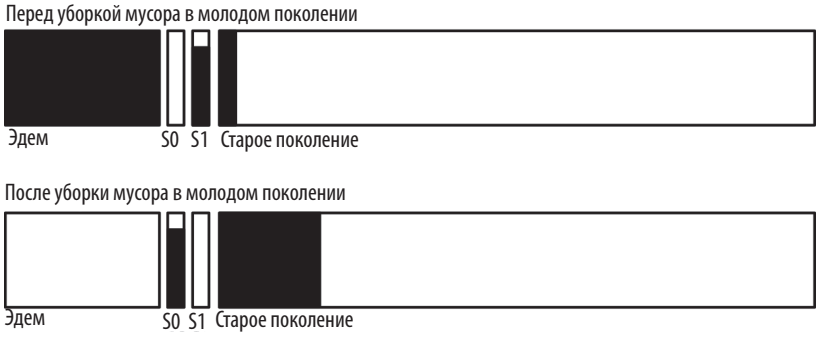


Рис. 6.7. Уборка в молодом поколении, выполняемая уборщиком мусора CMS

Уборка мусора в молодом поколении CMS напоминает параллельный алгоритм: данные перемещаются из Эдема в одну область выживших (и в старое поколение, если область выживших заполнена).

Вывод в журнале уборки мусора для CMS тоже выглядит похоже (приведу только формат журнала для JDK 8):

```
89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs]
               1303940K->772142K(2027264K), 0.1220090 secs]
               [Times: user=0.42 sys=0.02, real=0.12 secs]
```

Размер молодого поколения в настоящее время составляет 629 Мбайт; после уборки мусора от него остается 69 Мбайт (в области выживших). Аналогичным образом размер всей кучи составляет 2027 Мбайт — 772 Мбайт из которых остаются занятыми после уборки. Весь процесс занял 0,12 секунды, хотя параллельные потоки уборки мусора набрали 0,42 секунды использования процессора.

Конкурентный цикл изображен на рис. 6.8.

CMS запускает конкурентный цикл на основании степени заполнения кучи. Когда заполнение кучи достигнет достаточного порога, запускаются фоновые потоки, которые перебирают кучу и удаляют объекты. В конце цикла куча выглядит так, как показано в нижней части диаграммы. Обратите внимание: старое поколение не сжимается — в одних областях существуют объекты, другие области пусты. Когда уборка мусора в молодом поколении перемещает объекты из Эдема в старое поколение, JVM попытается использовать эти свободные области для хранения объектов. Часто такие объекты не помещаются в одну из свободных областей; из-за этого после цикла CMS максимальный уровень (high water mark) кучи возрастает.

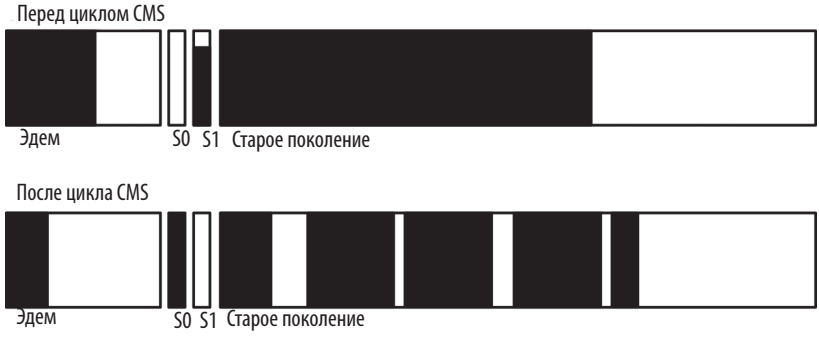


Рис. 6.8. Конкурентная уборка, выполняемая CMS

В журнале уборка мусора выглядит как последовательность фаз. Хотя большая часть конкурентного цикла использует фоновые потоки, некоторые фазы создают короткие паузы при остановке всех потоков приложения.

Конкурентный цикл начинается с фазы исходной пометки, в которой останавливаются все потоки приложения:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
               772530K(2027264K), 0.0830120 secs]
               [Times: user=0.08 sys=0.00, real=0.08 secs]
```

Эта фаза отвечает за поиск всех корневых объектов в куче. Первый набор чисел показывает, что объекты в настоящее время занимают 702 Мбайт из 1398 Мбайт старого поколения, а из второго набора следует, что из всей кучи 2027 Мбайт заняты 772 Мбайт. В этой фазе цикла CMS потоки приложения были остановлены на 0,08 секунды.

Затем идет фаза пометки, и в ней потоки приложения не останавливаются. Фаза представлена в журнале уборки мусора следующими строками:

```
90.059: [CMS-concurrent-mark-start]
90.887: [CMS-concurrent-mark: 0.823/0.828 secs]
               [Times: user=1.11 sys=0.00, real=0.83 secs]
```

Фаза пометки заняла 0,83 секунды (и 1,11 секунды процессорного времени). Так как в этой фазе выполняется только пометка, с заполнением кучи ничего не происходит, поэтому данные здесь не приводятся. Если бы данные выводились, то, скорее всего, в них бы отразился рост кучи из-за объектов, созданных в молодом поколении за эти 0,83 секунды, так как потоки приложения продолжали выполняться.

Затем следует фаза предварительной очистки, которая также выполняется конкурентно с потоками приложения:

```
90.887: [CMS-concurrent-preclean-start]
90.892: [CMS-concurrent-preclean: 0.005/0.005 secs]
                [Times: user=0.01 sys=0.00, real=0.01 secs]
```

Далее идет фаза повторной пометки, состоящая из нескольких операций:

```
90.892: [CMS-concurrent-abortable-preclean-start]
92.392: [GC 92.393: [ParNew: 629120K->69888K(629120K), 0.1289040 secs]
                1331374K->803967K(2027264K), 0.1290200 secs]
                [Times: user=0.44 sys=0.01, real=0.12 secs]
94.473: [CMS-concurrent-abortable-preclean: 3.451/3.581 secs]
                [Times: user=5.03 sys=0.03, real=3.58 secs]

94.474: [GC[YG occupancy: 466937 K (629120 K)]
        94.474: [Rescan (parallel) , 0.1850000 secs]
        94.659: [weak refs processing, 0.0000370 secs]
        94.659: [scrub string table, 0.0011530 secs]
                [1 CMS-remark: 734079K(1398144K)]
                1201017K(2027264K), 0.1863430 secs]
        [Times: user=0.60 sys=0.01, real=0.18 secs]
```

Постойте, разве CMS не выполняет фазу предварительной очистки? Почему в выводе упоминается отменяемая (abortable) фаза предварительной очистки?

Отменяемая фаза предварительной очистки используется из-за того, что фаза повторной пометки (которая, строго говоря, представлена последней записью в этом выводе) не является конкурентной — все потоки приложения в ней останавливаются. CMS хочет избежать ситуации, в которой за уборкой в молодом поколении немедленно следует фаза повторной пометки; в этом случае потоки приложения будут остановлены для двух пауз, следующих сразу же друг за другом. В данном случае целью является минимизация пауз за счет предотвращения смежных пауз.

Таким образом, отменяемая фаза предварительной очистки ожидает, пока молодое поколение не заполнится приблизительно на 50%. Теоретически это половина пути между уборками в молодом поколении, что дает CMS лучшие шансы для предотвращения смежных пауз. В этом примере отменяемая фаза предварительной очистки стартует в 90,8 секунды и около 1,5 секунды ожидает возникновения обычной уборки мусора (92,392 секунды в журнале). CMS использует прошлое поведение для вычисления момента, когда вероятна следующая уборка мусора в молодом поколении — в данном случае алгоритм CMS рассчитал, что она произойдет через 4,2 секунды. Итак, через 2,1 секунды (в 94,4 секунды) уборщик CMS завершает фазу предварительной очистки (что он называет *отменой* фазы, хотя это единственный способ завершения этой фазы). Наконец, CMS выполняет фазу повторной пометки, которая приостанавливает потоки приложения на 0,18 секунды (потоки приложения не были приостановлены во время отменяемой фазы предварительной очистки).

Затем следует другая конкурентная фаза — фаза стирания:

```
94.661: [CMS-concurrent-sweep-start]
95.223: [GC 95.223: [ParNew: 629120K->69888K(629120K), 0.1322530 secs]
                999428K->472094K(2027264K), 0.1323690 secs]
                [Times: user=0.43 sys=0.00, real=0.13 secs]
95.474: [CMS-concurrent-sweep: 0.680/0.813 secs]
                [Times: user=1.45 sys=0.00, real=0.82 secs]
```

Эта фаза заняла 0,82 секунды и выполнялась конкурентно с потоками приложения. Также она была прервана уборкой в молодом поколении. Эта уборка в молодом поколении не имела никакого отношения к фазе стирания, но я оставил ее здесь как пример того, что уборки в молодом поколении могут происходить конкурентно с фазами уборки в старом поколении. На рис. 6.8 обратите внимание на изменение состояния уборки в молодом поколении во время конкурентной уборки мусора — в фазе очистки могло произойти произвольное количество уборок в молодом поколении (а также будет как минимум одна уборка мусора в молодом поколении из-за отменяемой фазы предварительной очистки).

Далее следует конкурентная фаза сброса:

```
95.474: [CMS-concurrent-reset-start]
95.479: [CMS-concurrent-reset: 0.005/0.005 secs]
        [Times: user=0.00 sys=0.00, real=0.00 secs]
```

Это последняя из конкурентных фаз; цикл CMS завершен, а объекты без ссылок в старом поколении теперь освобождены (что приводит к состоянию кучи, показанному на рис. 6.8). К сожалению, журнал не предоставляет информации о количестве освобожденных объектов; строка `reset` не содержит информации о заполнении кучи. Чтобы получить представление о ней, взгляните на следующую уборку мусора в молодом поколении:

```
98.049: [GC 98.049: [ParNew: 629120K->69888K(629120K), 0.1487040 secs]
                1031326K->504955K(2027264K), 0.1488730 secs]
```

Сравните с заполнением старого поколения при 89,853 секунды (до начала цикла CMS), которое составляло приблизительно 703 Мбайт (вся куча в этой точке занимала 772 Мбайт, что включало 69 Мбайт в области выживших, так что старое поколение занимало оставшиеся 703 Мбайт). В уборке при 98,049 секунды старое поколение занимает около 504 Мбайт; следовательно, цикл CMS очистил около 199 Мбайт памяти.

Если все идет хорошо, то алгоритм CMS выполняет только эти циклы, а в журнале уборки мусора CMS будут выведены только эти сообщения. Однако существуют еще три сообщения, которые говорят о том, что у CMS возникли проблемы. Первое — сообщение о сбое конкурентного режима:

```
267.006: [GC 267.006: [ParNew: 629120K->629120K(629120K), 0.0000200 secs]
        267.006: [CMS267.350: [CMS-concurrent-mark: 2.683/2.804 secs]
        [Times: user=4.81 sys=0.02, real=2.80 secs]
  (concurrent mode failure):
        1378132K->1366755K(1398144K), 5.6213320 secs]
        2007252K->1366755K(2027264K),
        [CMS Perm : 57231K->57222K(95548K)], 5.6215150 secs]
        [Times: user=5.63 sys=0.00, real=5.62 secs]
```

Когда происходит уборка мусора в молодом поколении, а в старом поколении не хватает места для хранения всех объектов, которые должны быть повышены, CMS выполняет то, что по сути является полной уборкой мусора. Все потоки приложения останавливаются, а старое поколение чистится от всех мертвых объектов, в результате чего его заполнение сокращается до 1366 Мбайт — операция, которая остановила потоки приложения на целые 5,6 секунды. Операция была однопоточной, и это одна из причин, по которым она заняла столько времени (и одна из причин, по которым сбои конкурентного режима усугубляются с ростом кучи).

Этот сбой конкурентного режима — главная причина, по которой уборщик мусора CMS считается устаревшим. В работе уборщика G1 тоже может возникать сбой конкурентного режима, но при возврате к полной уборке мусора последняя в JDK 11 (но не в JDK 8) выполняется параллельно. Полная уборка CMS занимает во много раз больше времени, потому что она должна выполняться в одном потоке[1].

Вторая проблема возникает тогда, когда в старом поколении хватает памяти для хранения переводимых объектов, но свободное пространство фрагментировано, поэтому попытка повышения завершается неудачей:

```
6043.903: [GC 6043.903:
        [ParNew (promotion failed): 614254K->629120K(629120K), 0.1619839 secs]
        6044.217: [CMS: 1342523K->1336533K(2027264K), 30.7884210 secs]
        2004251K->1336533K(1398144K),
        [CMS Perm : 57231K->57231K(95548K)], 28.1361340 secs]
        [Times: user=28.13 sys=0.38, real=28.13 secs]
```

Здесь уборщик CMS запустил уборку в молодом поколении и предположил, что существует память для хранения всех повышенных объектов (в противном случае он бы объявил сбой конкурентного режима). Это предположение оказалось ошибочным: уборщик CMS не смог перевести объекты, потому что старое поколение было фрагментировано (или, что намного менее вероятно, потому что объем памяти для повышения оказался больше, чем ожидал алгоритм CMS).

[1] В принципе можно было бы провести аналогичную работу, чтобы полная уборка CMS также выполнялась параллельными потоками, но проект G1 был более приоритетным.

В результате в середине уборки мусора (когда все потоки уже были остановлены) алгоритм CMS провел уборку и сжатие всего старого поколения. К счастью, после сжатия кучи проблемы фрагментации были решены (по крайней мере временно). Но за это пришлось расплачиваться огромной 28-секундной паузой. Пауза получается намного длиннее, чем при сбое конкурентного режима CMS, потому что сжатию подвергалась вся куча; сбой конкурентного режима просто освободил объекты в куче. На этой стадии куча выглядит так, как она выглядела в конце полной уборки мусора параллельным уборщиком (см. рис. 6.2): молодое поколение полностью пусто, а старое поколение сжато.

Наконец, в журнале CMS может отображаться полная уборка мусора без обычных сообщений конкурентной уборки мусора:

```
279.803: [Full GC 279.803:
                [CMS: 88569K->68870K(1398144K), 0.6714090 secs]
                558070K->68870K(2027264K),
                [CMS Perm : 81919K->77654K(81920K)],
                0.6716570 secs]
```

Это происходит тогда, когда метапространство заполнилось и нуждается в уборке. CMS не выполняет уборку в метапространстве, поэтому в случае его заполнения необходимо провести полную уборку мусора, для того чтобы все классы без ссылок были удалены. В разделе «Расширенная настройка», с. 225, показано, как решить эту проблему.

РЕЗЮМЕ

- Работа алгоритма CMS состоит из нескольких операций уборки мусора, но ожидаемыми являются операции малой уборки мусора и конкурентные циклы.
- Сбои конкурентного режима и сбои повышения в CMS обходятся очень дорого; алгоритм CMS должен быть настроен так, чтобы предотвратить их по мере возможности.
- По умолчанию CMS не выполняет уборку мусора в метапространстве.

Настройка для предотвращения сбоев конкурентного режима

Главная цель настройки CMS — предотвращение сбоев конкурентного режима или повышения. Как показывает журнал уборки мусора CMS, сбои конкурентного режима происходят из-за того, что CMS недостаточно быстро чистит старое поколение: когда наступает время убирать мусор в молодом поколении, CMS

определяет, что для перевода этих объектов в старое поколение не хватит памяти, и вместо этого уборка сначала выполняется в старом поколении.

Старое поколение изначально заполняется с размещением объектов вплотную друг к другу. При заполнении определенной части старого поколения (по умолчанию 70%) начинается конкурентный цикл, и фоновый поток(-и) CMS начинает сканировать старое поколение в поисках мусора. В этот момент начинается гонка: алгоритм CMS должен завершить сканирование старого поколения и освобождение объектов до того, как заполнится остаток (30%) старого поколения. Если конкурентный цикл проигрывает гонку, CMS столкнется со сбоем конкурентного режима.

Чтобы попытаться предотвратить этот сбой, можно действовать несколькими способами:

- Увеличить старое поколение — либо за счет изменения пропорций между новым и старым поколением, либо простым увеличением размера кучи.
- Чаще запускать фоновый поток.
- Использовать больше фоновых потоков.

АДАПТИВНОЕ ОПРЕДЕЛЕНИЕ РАЗМЕРОВ И CMS

CMS использует параметры `MaxGCPauseMllis=N` и `GCTimeRatio=N` для определения размеров кучи и поколений.

Одно значительное отличие в подходе CMS — то, что молодое поколение никогда не изменяет размеров до момента полной уборки мусора. Так как основная цель CMS заключается в том, чтобы избежать полной уборки мусора, это означает, что хорошо настроенное приложение CMS никогда не изменяет размеров своего молодого поколения.

Сбои конкурентного режима часто возникают в ходе запуска программы, так как CMS адаптивно изменяет размер кучи и метапространства. Возможно, CMS стоит запускать с увеличенным исходным размером кучи (и метапространством) — это частный случай расширения кучи для предотвращения этих сбоев.

Если в системе имеется доступная память, лучшим решением будет увеличение размера кучи. Если нет — измените параметры работы фоновых потоков.

Более частый запуск фонового потока

Один из способов выиграть гонку CMS — начинать конкурентный цикл на более ранней стадии. Если конкурентный цикл начнется заполнением 60% старого

поколения, у CMS будет больше шансов завершить его, чем при запуске цикла при достижении 70% старого поколения. Проще всего это делается установкой двух флагов:

```
-XX:CMSInitiatingOccupancyFraction=N
-XX:+UseCMSInitiatingOccupancyOnly
```

Использование обоих флагов также помогает понять логику CMS: если оба флага установлены, то CMS определяет момент запуска фонового потока только на основании процента заполнения старого поколения. (Причем в отличие от уборщика G1, процент заполнения касается только старого поколения, а не всей кучи.)

По умолчанию флаг `UseCMSInitiatingOccupancyOnly` равен `false`, и CMS использует более сложный алгоритм для определения того, где должен начаться фоновый поток. Если фоновый поток должен начинаться на более ранней стадии, лучше запустить его самым простым способом и присвоить флагу `UseCMSInitiatingOccupancyOnly` значение `true`.

Настройка значения `CMSInitiatingOccupancyFraction` может потребовать нескольких итераций. Если флаг `UseCMSInitiatingOccupancyOnly` установлен, то для `CMSInitiatingOccupancyFraction` по умолчанию используется значение 70: цикл CMS запускается в тот момент, когда старое поколение заполнено на 70%.

Улучшенное значение этого флага для заданного приложения можно определить по журналу уборки мусора; для этого нужно определить, когда начался сбойный цикл CMS. Найдите сбой конкурентного режима в журнале, а затем вернитесь к началу последнего предшествующего цикла CMS. В строке `CMS-initial-mark` указана степень заполнения старого поколения при начале цикла CMS:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
               772530K(2027264K), 0.0830120 secs]
               [Times: user=0.08 sys=0.00, real=0.08 secs]
```

В данном примере она составляет около 50% (702 Мбайт из 1398 Мбайт). Этот порог не обеспечивал достаточной быстроты, так что `CMSInitiatingOccupancyFraction` необходимо присвоить значение, меньшее 50. (И хотя значение по умолчанию этого флага равно 70, в этом примере потоки CMS запускались при заполнении старого поколения на 50%, потому что флаг `UseCMSInitiatingOccupancyOnly` не был установлен.)

Здесь появляется искушение просто присвоить значение 0 или другое малое число, чтобы фоновый цикл CMS выполнялся все время. Обычно так поступать не рекомендуется, но если вы понимаете суть компромиссов, на которые вы идете, такое решение может сработать.

Первый компромисс относится к процессорному времени: фоновый поток CMS будет выполняться непрерывно и поглощать заметную часть процессорного времени — каждый фоновый поток CMS будет занимать процессор на 100%. Также будут очень короткие всплески активности, когда выполняются несколько потоков CMS и общая загрузка процессора на машине в результате увеличивается. Если эти потоки выполняются без необходимости, это приводит к напрасной потере ресурсов процессора.

С другой стороны, использование этих ресурсов процессора не обязательно создает проблему. Фоновые потоки CMS должны когда-то выполняться даже в лучшем случае. Однако машина всегда должна располагать ресурсами для выполнения этих потоков. Таким образом, при выборе размеров машины необходимо планировать такое использование процессора.

Второй компромисс — намного более важный — относится к паузам. Как было показано в журнале уборки мусора, некоторые фазы цикла CMS останавливают все потоки приложения. Главная причина для использования CMS — ограничение пауз уборки мусора, так что выполнение CMS чаще, чем это необходимо, нерационально. Паузы CMS обычно намного короче пауз уборки мусора в молодом поколении, и для конкретного приложения эти паузы могут быть незаметны — приходится выбирать баланс между дополнительными паузами и снижением вероятности сбоев конкурентного режима. Но постоянная работа фоновых операций уборки мусора с большой вероятностью приведет к избыточным общим паузам, что в конечном итоге приведет к снижению производительности приложения.

Если подобные компромиссы для вас неприемлемы, то не присваивайте `CMSInitiatingOccupancyFraction` значение, которое выше объема живых данных в куче как минимум на 10–20%.

Настройка фоновых потоков CMS

Каждый фоновый поток CMS потребит 100% процессора на машине. Если приложение сталкивается со сбоем конкурентного режима, а на машине доступны ресурсы процессора, можно увеличить количество фоновых потоков при помощи флага `-XX:ConcGCThreads=`*N*. В CMS этот флаг настраивается не так, как в уборщике мусора G1; используется следующая формула:

```
ConcGCThreads = (3 + ParallelGCThreads) / 4
```

Таким образом, CMS повышает значение `ConcGCThreads` на один шаг ранее, чем G1.

СОВЕТ

- Предотвращение сбоев конкурентного режима — ключ к достижению наилучшей возможной производительности с CMS.

- Простейшим способом предотвращения этих сбоев (когда это возможно) является увеличение размера кучи.
- Если этот вариант невозможен, следующим шагом должен стать более ранний запуск конкурентных фоновых потоков регулировкой `CMSInitiatingOccupancyFraction`.
- Настройка количества фоновых потоков также может принести пользу.

Расширенная настройка

В этом разделе описаны некоторые довольно необычные ситуации. И хотя они возникают нечасто, вы найдете здесь описания многих низкоуровневых аспектов алгоритмов уборки мусора.

Порог хранения и области выживших

После уборки мусора в молодом поколении останутся живые объекты. В их число входят не только вновь созданные объекты, которым суждено существовать в течение долгого времени, но и объекты, которые в остальных случаях имели бы короткий срок жизни. Вспомните цикл вычислений с `BigDecimal` в начале главы 5. Если JVM выполнит уборку мусора в середине этого цикла, некоторым из этих краткосрочных объектов `BigDecimal` не повезет: они были только что созданы и используются, поэтому освободить их нельзя — но они не проживут настолько долго, чтобы это оправдало их перемещение в старое поколение.

Именно по этой причине молодое поколение делится на две области: выживших и Эдем. Такая конфигурация позволяет объектам получить дополнительные шансы на уборку, пока они находятся в молодом поколении, вместо их повышения в старое поколение (с заполнением последнего).

Когда в молодом поколении выполняется уборка мусора, а JVM находит объект, который все еще жив, этот объект перемещается в область выживших вместо старого поколения. Во время первой уборки мусора в молодом поколении объекты перемещаются из Эдема в область выживших 0. При следующей уборке мусора живые объекты перемещаются из области выживших 0 из Эдема в область выживших 1. В этот момент Эдем и область выживших 0 полностью пусты. Следующая уборка мусора переводит живые объекты из области выживших 1 и Эдема в область выживших 0 и т. д. (Области выживших также называются входной и выходной; при каждой уборке мусора объекты перемещаются из выходной области во входную. В данном случае входная и выходная область — всего лишь указатели, которые переключаются между двумя областями выживших при каждой уборке мусора.)

Конечно, это не может продолжаться вечно, иначе никакие объекты бы никогда не были перемещены в старое поколение. Объекты перемещаются в старое поколение в двух ситуациях. Во-первых, области выживших относительно малы. Когда целевая область выживших заполняется во время уборки в молодом поколении, все оставшиеся живые объекты в Эдеме перемещаются прямо в старое поколение. Во-вторых, количество циклов уборки мусора, во время которых объекты могут оставаться в области выживших, ограничено. Ограничение называется *порогом хранения* (tenuring threshold).

Настройки могут повлиять на каждую из этих ситуаций. Области выживших занимают часть памяти молодого поколения, и, как и другие области кучи, JVM изменяет их размеры динамически. Исходный размер областей выживших определяется флагом `-XX:InitialSurvivorRatio=`*N*, который используется в следующей формуле:

```
размер_области_выживших = новый_размер / (InitialSurvivorRatio + 2)
```

Если значение `InitialSurvivorRatio` равно 8, каждая область выживших будет занимать 10% молодого поколения.

JVM может увеличить размер областей выживших до максимума, определяемого значением флага `-XX:MinSurvivorRatio=`*N*. Этот флаг используется в следующей формуле:

```
максимальный_размер_области_выживших  = новый_размер / (MinSurvivorRatio + 2)
```

По умолчанию это значение равно 3, оно означает, что максимальный размер области выживших составляет 20% от молодого поколения. Также учтите, что значение представляет соотношение, так что *минимальное* значение отношения определяет *максимальный* размер области выживших. Возможно, название выбрано не совсем удачно.

Чтобы сохранить фиксированный размер областей выживших, присвойте `SurvivorRatio` нужное значение и сбросьте флаг `UseAdaptiveSizePolicy` (хотя помните, что отключение адаптивного определения размеров будет также применяться к старому и новому поколению).

JVM определяет, нужно ли увеличить или уменьшить размер областей выживших (согласно определенным отношениям), на основании заполнения области выживших после уборки мусора. Размеры областей выживших будут изменяться так, чтобы они по умолчанию были заполнены на 50% после уборки. Это значение можно изменить при помощи флага `-XX:TargetSurvivorRatio=`*N*.

Наконец, остается вопрос том, сколько циклов уборки мусора объект будет перебрасываться между областями выживших, после чего он будет перемещен в старое поколение. Ответ определяется порогом хранения. JVM постоянно вычисляет

порог хранения, который, по ее мнению, будет лучшим. Изначальное значение порога определяется значением, заданным флагом `-XX:InitialTenuringThreshold=N` (по умолчанию равен 7 для параллельного уборщика и G1 и 6 для CMS). JVM в конечном итоге определяет порог в диапазоне от 1 до значения, заданного флагом `-XX:MaxTenuringThreshold=N`; для параллельного уборщика и G1 максимальный порог по умолчанию равен 15, а для CMS он равен 6.

ALWAYSTENURE И NEVERTENURE

Порог хранения всегда находится в диапазоне от 1 до `MaxTenuringThreshold`. Даже если JVM начинается с исходного порога хранения, равного максимальному порогу хранения, JVM может уменьшить это значение.

Два флага могут обойти это поведение в двух крайних случаях. Если вы знаете, что объекты, которые пережили уборку мусора в молодом поколении, всегда будут существовать в течение долгого времени, вы можете установить флаг `-XX:+AlwaysTenure` (по умолчанию false), что по сути эквивалентно присваиванию `MaxTenuringThreshold` значения 0. Это редкая ситуация; она означает, что объекты всегда будут повышаться в старое поколение, вместо того чтобы храниться в области выживших.

Второй флаг `-XX:+NeverTenure` также равен false по умолчанию. Он влияет на два аспекта: он работает так, словно исходный и максимальный пороги бесконечны, что не позволяет JVM понизить эти потоки. Другими словами, если в области выживших остается место, никакой объект не будет повышен до старого поколения.

Если учесть все сказанное, какие значения могут настраиваться в тех или иных обстоятельствах? Полезно проанализировать статистику хранения; эти данные не выводятся командами журнального вывода уборки мусора, которые мы использовали до сих пор.

В JDK 8 для вывода данных в журнале следует установить флаг `-XX:+PrintTenuringDistribution` (по умолчанию `false`). В JDK 11 он включается добавлением команд `age*=debug` или `age*=trace` в аргумент `Xlog`.

Самое важное, на что следует обращать внимание, — если области выживших настолько малы, что во время малой уборки мусора объекты повышаются прямо из Эдема в старое поколение. Такой ситуации следует избегать, потому что краткосрочные объекты в конечном итоге заполнят старое поколение, что приведет к слишком частым полным уборкам мусора.

В журналах уборки мусора для параллельного уборщика на это условие указывает только следующая строка:

```
Desired survivor size 39059456 bytes, new threshold 1 (max 15)
         [PSYoungGen: 657856K->35712K(660864K)]
         1659879K->1073807K(2059008K), 0.0950040 secs]
         [Times: user=0.32 sys=0.00, real=0.09 secs]
```

Журнал с JDK 11 с параметром `age*=debug` выглядит аналогично; в нем тоже будет выведен желательный размер области выживших во время уборки мусора.

Желательный размер одной области выживших составит 39 Мбайт молодого поколения, состоящего из 660 Мбайт: JVM вычислила, что две области выживших должны занимать около 11% молодого поколения. Однако вопрос о том, хватит ли этого для предотвращения переполнения, остается открытым. Журнал не дает определенного ответа, но тот факт, что JVM отрегулировала порог хранения до 1, означает, что большинство объектов все равно напрямую переводится в старое поколение, поэтому порог хранения был понижен до минимума. Вероятно, приложение напрямую повышает объекты без полноценного использования областей выживших.

При использовании уборщика мусора G1 в журнале JDK 8 выводится более содержательная информация:

```
 Desired survivor size 35782656 bytes, new threshold 2 (max 6)
 - age   1:   33291392 bytes,   33291392 total
 - age   2:    4098176 bytes,   37389568 total
```

В JDK 11 для получения этой информации в конфигурацию журнала включается команда `age*=trace`.

Желательный размер области выживших близок к предыдущему примеру (35 Мбайт), но в выводе также приводится размер всех объектов в области выживших. При 37 Мбайт данных для повышения в области выживших действительно происходит переполнение.

Возможно ли исправить ситуацию? Это зависит от приложения. Если ожидаемый срок жизни объектов превышает несколько циклов уборки мусора, они со временем все равно окажутся в старом поколении, так что регулировка областей выживших и порога хранения особой пользы не принесет. Но если объекты будут пропадать после всего нескольких циклов уборки мусора, часть производительности может быть достигнута за счет более эффективной организации областей выживших.

Если размер областей выживших увеличивается (за счет снижения соотношения выживших), память берется из раздела Эдема молодого поколения. А поскольку именно здесь создаются сами объекты, это означает, что меньшее количество объектов будет создаваться перед запуском малой уборки мусора. Из-за этого использовать этот вариант обычно не рекомендуется.

Другая возможность — увеличение размера молодого поколения. В данной ситуации это может быть нерационально: возможно, объекты будут реже повышаться до старого поколения, но поскольку само старое поколение уменьшается, приложению придется чаще выполнять полную уборку мусора.

Если возможно увеличить размер кучи в целом, то как молодому поколению, так и областям выживших достанется больше памяти — что может оказаться лучшим решением. Обычно желательно увеличить размер кучи (или по крайней мере молодого поколения) и сократить соотношение выживших. В результате размер областей выживших будет увеличен в большей степени, чем размер Эдема. В итоге приложение должно выполнять примерно столько же уборок в молодом поколении, как и прежде. При этом количество полных уборок мусора должно сократиться, потому что меньше объектов будет повышено до старого поколения (напомню: при этом предполагается, что срок жизни объектов не превышает нескольких циклов уборки мусора).

Если размеры областей выживших были отрегулированы так, что в них никогда не происходит переполнение, объекты будут повышаться до старого поколения только после достижения порога `MaxTenuringThreshold`. Значение можно увеличить, чтобы объекты оставались в областях выживших в течение еще нескольких циклов уборки мусора. Однако следует учитывать, что при увеличении порога хранения и более долгого пребывания объектов в области выживших будет оставаться меньше места в областях выживших при будущих уборках в молодом поколении: повышается вероятность того, что область выживших переполнится, и объекты снова начнут переводиться прямо в старое поколение.

РЕЗЮМЕ

- Области выживших предназначены для того, чтобы объекты (особенно только что созданные) могли оставаться в молодом поколении в течение нескольких циклов уборки мусора. Тем самым повышается вероятность того, что объект будет освобожден до того, как он будет переведен в старое поколение.
- Если области выживших слишком малы, объекты будут переводиться прямо в старое поколение — а это, в свою очередь, увеличит число уборок мусора в старом поколении.
- Лучшее решение в такой ситуации — увеличить размер кучи (или хотя бы молодого поколения) и предоставить JVM управление областями выживших.
- В отдельных случаях настройка порога хранения или размеров областей выживших может предотвратить повышение объектов в старое поколение.

Создание больших объектов

В этом разделе подробно описан процесс выделения памяти JVM при создании объектов. Это довольно интересная техническая информация, и она важна для приложений, которые часто создают множество больших объектов. В этом контексте термин «большие» относителен; как вы увидите, его смысл зависит от размера особого буфера в JVM.

Этот буфер называется *потоково-локальным буфером*, или *TLAB* (Thread-Local Allocation Buffer). Определение размера TLAB является важным фактором для всех алгоритмов уборки мусора, а алгоритму G1 приходится учитывать еще одно обстоятельство для очень больших объектов (и снова термин является относительным — но для 2-гигабайтной кучи речь идет об объектах, размер которых превышает 512 Мбайт). Последствия от использования очень больших объектов для уборщика G1 могут быть очень серьезными — изменение размеров TLAB (для решения проблемы больших объектов при использовании любого уборщика мусора) относительно нетипично, но изменение размеров областей G1 (для решения проблемы очень больших объектов при использовании G1) происходит гораздо чаще.

TLAB

В главе 5 описан процесс создания объектов в Эдеме; это позволяет ускорить создание объектов (особенно с малым сроком жизни).

Одна из причин, по которым создание объектов в Эдеме происходит так быстро, заключается в том, что каждый поток имеет выделенную область, которой он создает объекты — буфер TLAB. При непосредственном создании объектов в общей области (такой, как Эдем) потребовалась бы некоторая форма синхронизации для управления указателями на свободную память в этом пространстве. Если у каждого потока имеется собственная область для выделения памяти, потоку не придется использовать синхронизацию при создании объектов[1].

Обычно использование TLAB полностью прозрачно для разработчиков и конечных пользователей: буферы TLAB включены по умолчанию, JVM управляет их размерами и использованием. Важно понимать, что буферы TLAB имеют небольшие размеры, так что большие объекты в TLAB создать невозможно. Большие объекты приходится создавать непосредственно в куче, а это требует дополнительного времени из-за синхронизации.

С заполнением TLAB объекты определенного размера уже не могут в ней создаваться. В этот момент у JVM появляется выбор. Первый вариант — «списать»TLAB и выделить новый буфер для потока. Так как TLAB представля-

[1] Данная схема может рассматриваться как разновидность предотвращения конкуренции за счет использования потоково-локальных переменных (см. главу 9).

ет собой всего лишь часть Эдема, «списанный» буфер TLAB будет очищен при следующей уборке в молодом поколении и может быть использован повторно в будущем. Также JVM может создать объект прямо в куче и оставить существующий буфер TLAB (по крайней мере до того, как поток создаст в TLAB дополнительные объекты). Допустим, размер TLAB составляет 100 Кбайт, и 75 Кбайт уже были выделены. Если потребуется выделить память под новый объект размером 30 Кбайт, TLAB можно «списать», что приведет к потере 25 Кбайт памяти Эдема. Также можно создать 30-килобайтный объект прямо в куче в надежде на то, что следующий созданный объект поместится в 25 Кбайт памяти, остающейся свободной в TLAB. Параметры могут управлять этим поведением (см. далее в этом разделе), но принципиально здесь то, что выбор размера TLAB очень важен. По умолчанию размер TLAB определяется тремя факторами: количеством потоков в приложении, Эдеме и скоростью выделения памяти потоками.

Таким образом, настройка параметров TLAB может принести пользу в приложениях двух типов: создающих множество больших объектов и имеющих большое количество потоков относительно размера Эдема. По умолчанию буферы TLAB активны; их можно отключить при помощи флага `-XX:-UseTLAB`, хотя они обеспечивают такой прирост производительности, что отключать их почти никогда не следует.

Так как вычисление размера TLAB отчасти базируется на скорости выделения памяти потоками, невозможно определенно спрогнозировать оптимальный размер TLAB для приложения. Вместо этого можно отслеживать выделение памяти в TLAB, чтобы увидеть, создаются ли объекты вне TLAB. Если таких объектов достаточно много, возможны два варианта: сократить размер создаваемого объекта или отрегулировать параметры определения размеров TLAB.

Мониторинг создания объектов в TLAB — одна из областей, в которых Java Flight Recorder намного мощнее других инструментов. На рис. 6.9 показан пример экрана мониторинга выделения памяти из данных JFR.

За 5 секунд, выбранных для сохранения данных, 49 объектов были вне TLAB; максимальный размер этих объектов составлял 48 байт. Так как минимальный размер TLAB равен 1,35 Мбайт, мы знаем, что эти объекты были созданы в куче только потому, что буфер TLAB был заполнен на момент создания; они не были созданы в куче из-за своего размера. Как правило, это происходит непосредственно перед возникновением уборки мусора в молодом поколении (с заполнением Эдема — а следовательно, и буферов TLAB, выделенных из Эдема).

Общее выделение памяти за этот период составляет 1,59 Кбайт; ни количество создаваемых объектов, ни их размер не создают проблем. Некоторые объекты всегда будут создаваться вне TLAB, особенно когда Эдем приближается к уборке в молодом поколении. Сравните с рис. 6.10, из которого видно, что значительная часть выделения памяти происходит за пределами TLAB.

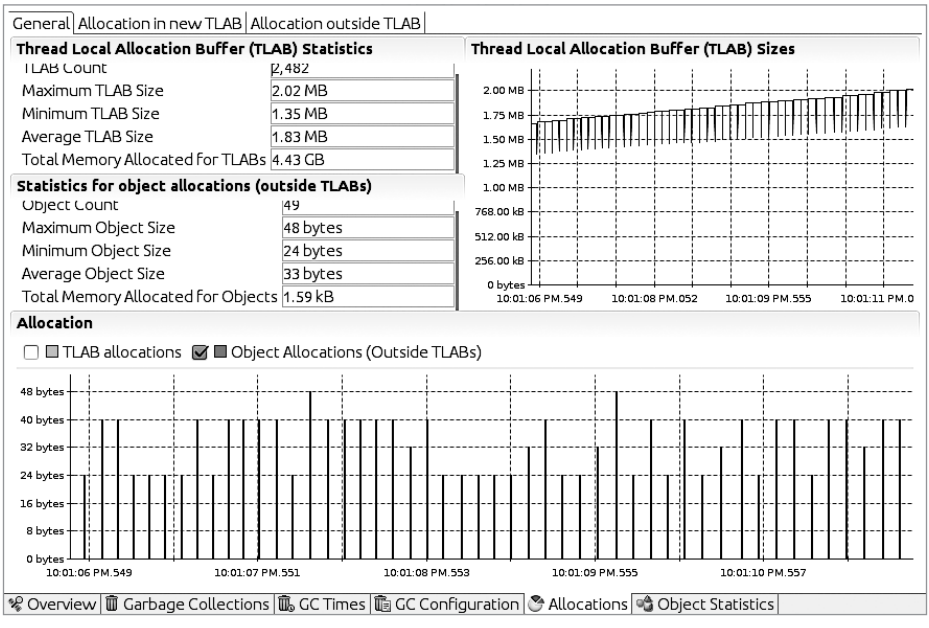


Рис. 6.9. Буферы TLAB в Java Flight Recorder

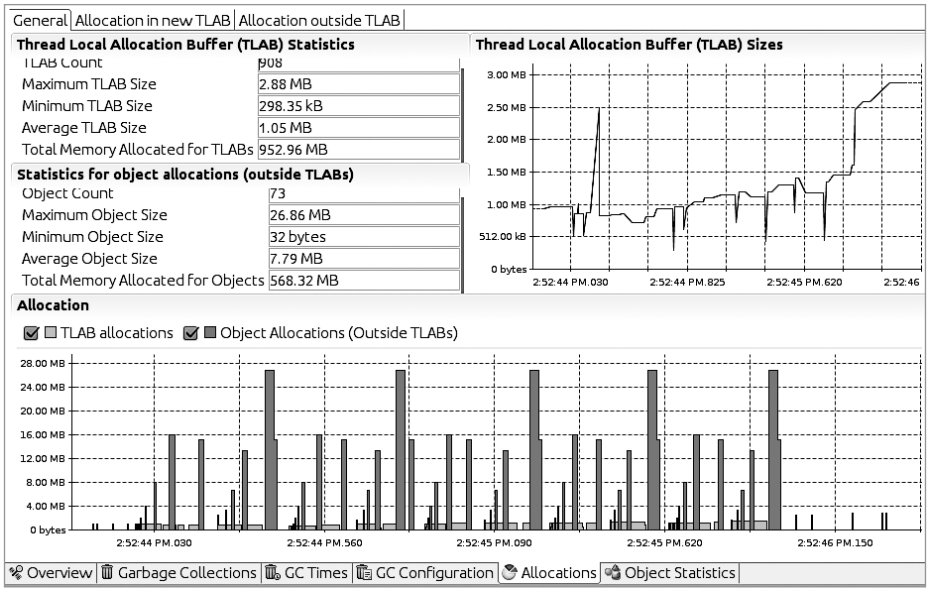


Рис. 6.10. Избыточное выделение памяти за пределами TLAB

Общая память, выделенная за пределами TLAB, за время сохранения данных составила 952,32 Мбайт. Это одна из тех ситуаций, в которых переход на использование объектов меньшего размера или настройка JVM для выделения этих объектов в TLAB большего размера могут оказывать положительный эффект. При этом на других вкладках могут отображаться реальные объекты, созданные в TLAB; можно даже получить содержимое стеков, из которых были созданы эти объекты. Если при выделении памяти TLAB возникнут проблемы, JFR поможет быстро обнаружить их.

Без использования JFR для отслеживания создания объектов TLAB лучше всего добавить флаг `-XX:+PrintTLAB` в командную строку (JDK 8) или включить команду `tlab*=trace` в конфигурацию журнала для JDK 11 (для получения приведенной ниже информации, и не только). Тогда при каждой уборке мусора в молодом поколении журнал уборки мусора будет содержать записи двух видов: строку для каждого потока, описывающую использование TLAB для этого потока, и сводную строку с описанием общего использования TLAB в JVM.

Запись уровня потока выглядит так:

```
TLAB: gc thread: 0x00007f3c10b8f800 [id: 18519] desired_size: 221KB
    slow allocs: 8  refill waste: 3536B alloc: 0.01613    11058KB
    refills: 73 waste  0.1% gc: 10368B slow: 2112B fast: 0B
```

`gc` в этом выводе означает, что строка была выведена во время уборки мусора; сам поток является обычным потоком приложения. Размер TLAB этого потока составляет 221 Кбайт. С момента последней уборки в молодом поколении в куче были созданы восемь объектов (`slow allocs`); они составили 1,6% (0,01613) общего объема памяти, выделенной этим потоком, что в сумме дает 11 058 Кбайт. 0,1% TLAB теряется (`waste`), что объясняется тремя причинами: 10 336 байт были свободны в TLAB при запуске текущего цикла уборки мусора; 2112 байт были свободны в других («списанных») TLAB, и 0 байт были выделены специальным «быстрым» диспетчером распределения памяти.

После вывода данных TLAB каждого потока JVM выводит строку сводных данных (в JDK 11 вывод этих данных включается командой `tlab*=debug`):

```
TLAB totals: thrds: 66  refills: 3234 max: 105
        slow allocs: 406 max 14 waste:  1.1% gc: 7519856B
        max: 211464B slow: 120016B max: 4808B fast: 0B max: 0B
```

В данном случае 66 потоков выделяли память с момента последней уборки мусора в молодом поколении. Среди этих потоков буферы TLAB были повторно заполнены 3234 раза; максимальное количество повторных заполнений одним конкретным потоком было равно 105. В целом в куче память выделялась 406 раз (максимум 14 раз на один поток), и 1,1% TLAB было потеряно из свободной памяти в списанных TLAB.

Если в данных уровня потока часто говорится о выделении памяти за пределами TLAB, возможно, вам стоит изменить их размеры.

Определение размеров TLAB

Если приложение проводит много времени за созданием объектов за пределами TLAB, оно выиграет от изменений, переводящих создание объектов в TLAB. Если несколько конкретных разновидностей объектов всегда выделяются за пределами TLAB, лучшим решением станут изменения в программном коде.

В других случаях — или если программные изменения невозможны — вы можете попытаться изменить размер TLAB под особенности приложения. Так как размер TLAB основан на размере Эдема, настройка некоторых параметров его размера приведет к автоматическому увеличению размера TLAB.

Размер TLAB можно задать явно при помощи флага `-XX:TLABSize=`*N* (значение по умолчанию 0 означает, что размер должен вычисляться динамически, как описано выше). Этот флаг определят только исходный размер TLAB; чтобы предотвратить изменение размеров при каждой уборке мусора, добавьте флаг `-XX:-ResizeTLAB` (по умолчанию флаг равен `true`). Это самый простой (и откровенно говоря, единственный полезный) вариант для анализа производительности настройки TLAB.

Если новый объект не помещается в текущем буфере TLAB (но поместится в новом пустом буфере TLAB), JVM приходится принимать решение: создать объект в куче или списать текущий буфер TLAB и выделить новый. Это решение зависит от нескольких параметров. В журнальном выводе TLAB значение `refill waste` сообщает текущий порог для этого решения: если TLAB не может создать новый объект, превышающий это значение, новый объект будет создан в куче. Если создаваемый объект меньше этого значения, буфер TLAB будет списан.

Значение определяется динамически, но по умолчанию оно начинается с 1% размера TLAB — или, говоря конкретнее, значения, заданного флагом `-XX:TLABWasteTargetPercent=`*N*. С каждым созданием объекта за пределами кучи это значение увеличивается с приращением `-XX:TLABWasteIncrement=`*N* (по умолчанию 4). Таким образом предотвращается достижение потоком порога из TLAB и постоянное создание объектов в куче: с ростом целевого процента вероятность списания TLAB также увеличивается. Настройка значения `TLABWasteTargetPercent` также регулирует размер TLAB, поэтому хотя с этим значением можно экспериментировать, эффект не всегда предсказуем.

Наконец, при использовании изменения размера TLAB минимальный размер TLAB может быть задан флагом `-XX:MinTLABSize=`*N* (по умолчанию 2 Кбайт). Максимальный размер TLAB немногим менее 1 Гбайт (максимальная область,

которую может занимать массив целых чисел, округленная для выравнивания объектов в памяти), и изменить его невозможно.

РЕЗЮМЕ

- В приложениях, создающих множество больших объектов, может принести пользу настройка TLAB (хотя использование меньших объектов в приложении может быть более эффективным).

Огромные объекты

Объекты, создаваемые за пределами TLAB, создаются в Эдеме, если это возможно. Если объект не помещается в Эдеме, он должен быть создан непосредственно в старом поколении. При этом нормальный жизненный цикл уборки мусора для этого объекта нарушается, а если объект имеет короткий срок жизни, это отрицательно отражается на уборке мусора. В таком случае сделать почти ничего нельзя — разве что изменить приложение так, чтобы в нем не использовались большие объекты с коротким сроком жизни.

Однако уборщик мусора G1 обрабатывает огромные объекты особым образом: G1 создает их в старом поколении, если они превышают область G1. Если приложение использует большое количество огромных объектов, уборщику мусора G1 может потребоваться специальная настройка для компенсации этого факта.

Размеры областей уборщика мусора G1

Уборщик мусора G1 делит кучу на области, каждая из которых имеет фиксированный размер. Размер области не изменяется динамически; он определяется при запуске на основании минимального размера кучи (значение `Xms`). Минимальный размер области равен 1 Мбайт. Если минимальный размер кучи больше 2 Гбайт, то размеры областей задаются по следующей формуле (логарифм по основанию 2):

```
размер_области = 1 << log(Исходный Размер Кучи / 2048);
```

Короче говоря, размер области равен наименьшей степени 2, ближайшей к размеру 2048 областей, на которые делится исходная куча. Также при этом используются некоторые минимальные и максимальные ограничения: размер области всегда превышает 1 Мбайт и никогда не бывает больше 32 Мбайт. В табл. 6.3 перечислены все возможные варианты.

Размер области G1 может задаваться флагом `-XX:G1HeapRegionSize=`*N* (по умолчанию 0 — означает, что должно использоваться динамическое значение). Присвоенное значение должно представлять собой степень 2 (например, 1 Мбайт или 2 Мбайт); в противном случае оно округляется до ближайшей степени 2.

Таблица 6.3. Размеры областей по умолчанию для G1

Размер кучи	Размер области по умолчанию для G1
Менее 4 Гбайт	1 Мбайт
От 4 Гбайт до 8 Гбайт	2 Мбайт
От 8 Гбайт до 16 Гбайт	4 Мбайт
От 16 Гбайт до 32 Гбайт	8 Мбайт
От 32 Гбайт до 64 Гбайт	16 Мбайт
Более 64 Гбайт	32 Мбайт

РАЗМЕРЫ ОБЛАСТЕЙ G1 И БОЛЬШИЕ КУЧИ

Обычно размер области уборщика мусора G1 настраивается только для решения проблем с созданием огромных объектов, но он также может настраиваться еще в одном случае.

Возьмем приложение, для которого задан очень большой диапазон кучи: например, `-Xms2G -Xmx32G`. В этом случае размер области составит 1 Мбайт. Полностью развернутая куча будет содержать 32 000 области G1. Областей для обработки получается слишком много; алгоритм G1 проектировался на представлении о том, что количество областей ближе к 2048. Увеличение размера области G1 несколько повысит эффективность уборки мусора G1; значение выбирается таким образом, чтобы ожидаемый размер кучи был ближе к 2048 областям.

Создание огромных объектов уборщиком мусора G1

Если размер области G1 составляет 1 Мбайт, а программа выделяет память для массива на 2 миллиона байт, массив не поместится в одной области G1. Но эти огромные объекты должны создаваться в смежных областях G1. Если размер области G1 составляет 1 Мбайт, то для выделения массива на 3,1 Мбайт уборщик мусора G1 должен найти четыре области в старом поколении для выделения массива. (Остаток последней области останется пустым с потерей 0,9 Мбайт памяти.) Это противоречит обычной стратегии уборщика мусора G1 по выполнению сжатия, которая заключается в освобождении произвольных областей на основании их заполнения.

На самом деле уборщик мусора G1 определяет огромный объект как объект, размер которого достигает половины размера области, так что выделение массива

на 512 Кбайт (плюс 1 байт) в этом случае запустит рассматриваемую процедуру выделения памяти для огромных объектов.

Так как огромный объект создается напрямую в старом поколении, он не может быть освобожден в ходе уборки мусора в молодом поколении. Таким образом, если объект имеет короткий срок жизни, это тоже противоречит архитектуре уборщика мусора, основанной на поколениях. Огромный объект будет освобожден в ходе конкурентного цикла уборки G1. К счастью, огромный объект может быть быстро освобожден, потому что это единственный объект в занимаемых им областях. Огромные объекты освобождаются в ходе фазы очистки конкурентного цикла (а не при смешанной уборке мусора).

Повышение размера области уборки мусора G1, чтобы все объекты, создаваемые программой, могли поместиться в одной области уборки мусора, может сделать уборку G1 более эффективной. Это означает, что размер области G1 вдвое больше размера самого большого объекта (плюс 1 байт).

Когда-то огромные объекты создавали гораздо большие проблемы в G1, потому что поиск необходимых областей для создания объекта обычно требовал полной уборки мусора (и потому что полные уборки мусора не были параллелизованы). Усовершенствования G1 в JDK 8u60 (и во всех сборках JDK 11) сводят эту проблему к минимуму, так что сейчас эта проблема уже не столь критична.

РЕЗЮМЕ

- Размер областей G1 определяется как степень 2, начиная с 1 Мбайт.
- Кучи, максимальный размер которых сильно отличается от исходного, будет содержать слишком много областей G1; в этом случае размер области G1 должен быть увеличен.
- Приложения, создающие объекты, размер которых превышает половину размера области G1, должны увеличить размер области G1, чтобы объекты могли поместиться в области G1. Для выполнения этого условия приложение должно создать объект с размером не менее 512 Кбайт (так как наименьший размер области G1 составляет 1 Мбайт).

Флаг AggressiveHeap

Флаг `AggressiveHeap` (по умолчанию `false`) появился в ранней версии Java как попытка упростить настройку разных аргументов командной строки — аргументов, подходящих для очень большой машины с большим объемом памяти, на которой работает одна JVM. Хотя флаг продолжает поддерживаться с тех

версий, использовать его не рекомендуется (хотя официально он не признан устаревшим).

Проблема в том, что этот флаг скрывает фактически выбранные настройки, из-за чего становится труднее разобраться, какие параметры задает JVM. Некоторые из устанавливаемых значений теперь задаются эргономически на основании более достоверной информации о машине, на которой работает JVM, так что в некоторых случаях этот флаг вредит быстродействию. Я часто видел командные строки, которые включают этот флаг, а позднее переопределяют заданные им значения. (Кстати говоря, этот способ работает: в настоящее время более поздние значения в командной строке переопределяют более ранние значения. Впрочем, такое поведение не гарантировано.)

В табл. 6.4 перечислены все настройки, автоматически задаваемые при установке флага `AggressiveHeap`.

Таблица 6.4. Настройки, включаемые флагом AggressiveHeap

Флаг	Значение
`Xmx`	Минимум половины всей памяти или вся память; 160 Мбайт
`Xms`	То же, что Xmx
`NewSize`	3/8 значения, присвоенного Xmx
`UseLargePages`	true
`ResizeTLAB`	false
`TLABSize`	256 Кбайт
`UseParallelGC`	true
`ParallelGCThread`	Совпадает с текущим значением по умолчанию
`YoungPLABSize`	256 Кбайт (по умолчанию 4 Кбайт)
`OldPLABSize`	8 Кбайт (по умолчанию 1 Кбайт)
`CompilationPolicyChoice`	0 (текущее значение по умолчанию)
`ThresholdTolerance`	100 (по умолчанию 10)
`ScavengeBeforeFullDC`	false (по умолчанию true)
`BindGCTaskThreadsToCPUs`	true (по умолчанию false)

Последние шесть флагов достаточно сложны для понимания, поэтому в книге они более не рассматриваются. Вкратце они относятся к следующим областям:

Определение размеров PLAB — буферы PLAB (Promotion-Local Allocation Buffers) — области уровня потоков, используемые в ходе уборки мусора в поколениях GC. Каждый поток может переводить объекты в конкретный буфер PLAB, что снижает необходимость в синхронизации (по аналогии с TLAB).

Политики компиляции — в поставку JVM включены альтернативные алгоритмы JIT-компиляции. Алгоритм, используемый по умолчанию, одно время считался экспериментальным, но он считается рекомендуемым.

Отключение уборки мусора в молодом поколении перед полной уборкой мусора — присваивание флагу `ScavengeBeforeFullGC` значения `false` означает, что при полной уборке мусора JVM не будет выполнять уборку мусора в молодом поколении перед полной уборкой мусора. Обычно это нежелательно, потому что мусорные объекты в молодом поколении (пригодные для уборки) могут помешать уборке объектов в старом поколении. Очевидно, когда-то эта настройка имела смысл (по крайней мере в некоторых ситуациях), но в общем случае изменять этот флаг не рекомендуется.

Привязка потоков уборки мусора к процессорам — установка последнего флага в этом списке означает, что каждый параллельный поток уборки мусора привязывается к конкретному процессору (с использованием специальных функций ОС). В некоторых ситуациях — когда на машине выполняются только потоки уборки мусора, а кучи очень велики — это имеет смысл. В общем случае будет лучше, если потоки уборки мусора могут выполняться на любом доступном процессоре.

Как и с любыми настройками, у вас все может быть иначе. Если вы тщательно протестируете флаг `AggressiveHeap` и обнаружите, что он улучшает производительность — ничто не мешает вам его использовать. Главное — помнить, что происходит за кулисами, и понимать, что при любых обновлениях JVM может оказаться, что относительные преимущества этого флага придется пересматривать.

РЕЗЮМЕ

- Флаг `AggressiveHeap` — устаревшая попытка присвоить параметрам кучи значения, подходящие для одиночной JVM, работающей на очень большой машине.
- Значения, присваиваемые этим флагом, не изменяются с совершенствованием технологии JVM, поэтому его полезность в долгосрочной перспективе выглядит сомнительно (хотя он все еще часто используется на практике).

Полный контроль над размером кучи

В разделе «Определение размера кучи», с. 175, рассматриваются значения по умолчанию для исходного минимального и максимального размера кучи. Эти значения зависят как от объема памяти на машине, так и от используемой JVM, а у представленных данных есть ряд граничных случаев. Если вас интересует, как вычисляется размер кучи по умолчанию, вы найдете нужную информацию в этом разделе. В процессе задействованы некоторые низкоуровневые флаги; в определенных обстоятельствах может быть удобнее настроить сам процесс вычислений (вместо того, чтобы просто задать размер кучи). Например, такая ситуация может возникнуть в том случае, если вы хотите выполнять несколько JVM с общим (но скорректированным) набором эргономических размеров кучи. В основном я привожу эту информацию для того, чтобы объяснение того, как выбираются эти значения по умолчанию, было более полным.

Размеры по умолчанию основаны на объеме памяти на машине, который может задаваться флагом `-XX:MaxRAM=`*N*. Обычно для вычисления этого значения JVM анализирует объем памяти на машине. Тем не менее JVM ограничивает значение `MaxRAM` 4 Гбайт для 32-разрядных серверов Windows и 128 Гбайт для 64-разрядных JVM. Максимальный размер кучи составляет 1/4 `MaxRAM`. Это объясняет, почему размер кучи по умолчанию может изменяться: если объем физической памяти на машине меньше `MaxRAM`, размер кучи по умолчанию составит 1/4 этой величины.

Но даже если на машине доступны сотни гигабайт памяти, по умолчанию JVM будет использовать не более 32 Гбайт (1/4 от 128 Гбайт).

Максимальный размер кучи по умолчанию вычисляется по следующей формуле:

$$\text{Xmx по умолчанию} = \text{MaxRAM} / \text{MaxRAMFraction}$$

Следующий, максимальный размер кучи по умолчанию также можно задать настройкой значения флага `-XX:MaxRAMFraction=`*N*, которое по умолчанию равно 4. Наконец, флагу `-XX:ErgoHeapSizeLimit=`*N* также можно присвоить максимальное значение по умолчанию, которое должно использоваться JVM. По умолчанию это значение равно 0 (это означает, что оно игнорируется); в противном случае используется это ограничение, если оно меньше `MaxRAM/MaxRAMFraction`.

С другой стороны, на машине с очень малым объемом физической памяти JVM нужно позаботиться о том, чтобы операционной системе осталось достаточно памяти. По этой причине на машине, оснащенной всего 192 Мбайт памяти, JVM ограничит максимальную кучу 96 Мбайт и менее. Это вычисление базируется на значении флага `-XX:MinRAMFraction=`*N*, которое по умолчанию равно 2:

```
if ((96 MB * MinRAMFraction) > физическая память) {
Xmx по умолчанию = физическая память / MinRAMFraction;
}
```

Исходный размер кучи выбирается аналогичным образом, хотя и имеет меньше последствий. Значение исходного размера кучи определяется по формуле:

$$\text{Xms по умолчанию} = \text{MaxRAM} / \text{InitialRAMFraction}$$

Как можно заключить по минимальным размерам кучи по умолчанию, значение по умолчанию флага `InitialRAMFraction` равно 64. Единственная проблема возникает тогда, когда значение меньше 5 Мбайт — или, строго говоря, меньше суммы значений флагов `-XX:OldSize=`*N* (по умолчанию 4 Мбайт) и `-XX:NewSize=`*N* (по умолчанию 1 Мбайт). В этом случае сумма старого и нового размера используется как исходный размер кучи.

РЕЗЮМЕ

- Вычисление исходного и максимального размера кучи по умолчанию на большинстве машин происходит достаточно прямолинейно.
- В граничных случаях такие вычисления могут быть достаточно сложными.

Экспериментальные алгоритмы уборки мусора

В JDK 8 и JDK 11 на рабочих виртуальных многопроцессорных машинах используется лишь уборщик G1 либо параллельный уборщик, в зависимости от ваших требований к приложению. На менее мощных машинах используется последовательный уборщик, если он подходит к вашему оборудованию. Все эти уборщики мусора пользуются полноценной поддержкой.

В JDK 12 появились новые уборщики мусора. Хотя они не всегда поддерживаются на уровне реальной эксплуатации, мы кратко познакомимся с ними для экспериментальных целей.

Конкурентное сжатие: ZGC и Shenandoah

Существующие одновременные уборщики мусора на самом деле не являются полностью одновременными. Ни G1, ни CMS не реализуют одновременной уборки молодого поколения: освобождение молодого поколения требует остановки всех потоков приложения. И ни один из этих уборщиков не поддерживает

одновременного сжатия. В уборщике мусора G1 сжатие старого поколения становится эффектом циклов смешанной уборки мусора: в целевой области объекты, которые не были освобождены, сжимаются в пустые области. В CMS старое поколение сжимается тогда, когда оно становится слишком фрагментированным, чтобы стало возможным создание новых объектов. Уборка в молодом поколении также сжимает эту часть кучи посредством перемещения выживших объектов в области выживших или в старое поколение.

Во время сжатия объекты изменяют свое положение в памяти. Это главная причина, по которой JVM останавливает все потоки приложения во время этой операции — алгоритмы обновления ссылок на память значительно упрощаются, если потоки приложения заведомо остановлены. Таким образом, во времени приостановки приложения преобладает время, потраченное за перемещением объектов и обеспечением актуальности ссылок.

Для решения этой проблемы были разработаны два экспериментальных уборщика мусора. Первый — уборщик мусора Z, или *ZGC*; второй — уборщик мусора *Shenandoah*. ZGC впервые появился в JDK 11; уборщик Shenandoah впервые появился в JDK 12, но прошел обратное портирование для JDK 8 и JDK 11. В сборках JVM из AdoptOpenJDK (или откомпилированных вами по исходным кодам) присутствуют оба уборщика мусора; сборки, распространяемые Oracle, содержат только ZGC.

Чтобы использовать этих уборщиков мусора, необходимо установить флаг `-XX:+UnlockExperimentalVMOptions` (по умолчанию `false`). Затем устанавливается флаг `-XX:+UseZGC` или `-XX:+UseShenandoahGC` для использования соответствующего уборщика мусора. Как и другие алгоритмы, они предоставляют определенные средства настройки, но эти средства изменяются, так как алгоритмы находятся в стадии разработки, поэтому пока будут использоваться аргументы по умолчанию. (При этом оба уборщика разрабатывались с расчетом на выполнение с минимальной настройкой.)

Несмотря на различия в подходах, оба уборщика поддерживают конкурентное сжатие кучи — это означает, что объекты в куче могут перемещаться без остановки всех потоков приложения. Отсюда следуют два главных вывода.

Во-первых, куча перестает делиться на поколения (то есть в ней уже нет молодого и старого поколения; есть просто одна куча). В основу молодого поколения заложена идея о том, что уборка в малой части кучи выполняется быстрее, чем уборка во всей куче, и многие (а в идеале — все) такие объекты будут мусорными. Таким образом, молодое поколение во многих случаях сокращает длительность пауз. Если потоки приложения не обязательно приостанавливать в ходе уборки мусора, необходимость в молодом поколении отпадает, и алгоритмам уже не нужно сегментировать кучу на поколения.

Во-вторых, можно ожидать, что задержка операций, выполняемых потоками приложения, сократится (по крайней мере во многих случаях). Возьмем вызов REST, который обычно выполняется за 200 миллисекунд; если такой вызов будет прерван уборкой мусора в молодом поколении в G1, а уборка занимает 500 мс, пользователь увидит, что вызов REST занял 700 мс. Конечно, большинство вызовов с такой ситуацией не столкнется, но в отдельных случаях это произойдет, а эти выбросы повлияют на общую производительность системы. Если потоки приложений останавливать не обязательно, в конкурентных уборщиках со сжатием таких выбросов не будет.

И этот факт несколько упрощает ситуацию. Как говорилось при обсуждении уборщика G1, в работе фоновых потоков, помечающих свободные объекты в областях кучи, иногда возникают короткие паузы. Таким образом, в уборщике G1 возможны паузы трех типов: относительно долгие паузы для полной уборки мусора (в идеале уборщик настроен достаточно хорошо, чтобы таких пауз не было), более короткие паузы для уборки мусора в молодом поколении (включая смешанную уборку, которая освобождает и сжимает часть старого поколения), и очень короткие паузы для пометки потоков.

Как у ZGC, так и у Shenandoah существуют похожие паузы, относящиеся к последней категории; на короткие промежутки времени останавливаются все потоки приложения. Эти уборщики стремятся сделать эти периоды очень короткими, порядка 10 мс.

Такие уборщики также могут создавать задержку в отдельных операциях потоков. Подробности зависят от конкретного алгоритма, но в двух словах обращение к объекту со стороны потока приложения защищается специальным барьером. Если окажется, что объект находится в процессе перемещения, поток приложения ожидает у барьера до завершения перемещения. (И если поток приложения обращается к объекту, поток уборки мусора должен ожидать у барьера, пока он не сможет переместить объект.) По сути это разновидность блокировки по ссылкам на объект, но с этим термином процесс кажется намного более тяжеловесным, чем в действительности. В общем случае он практически не оказывает воздействия на производительность приложения.

Эффект задержки при конкурентном сжатии

Чтобы получить представление об общем воздействии этих алгоритмов, взгляните на данные из табл. 6.5. В таблице приведено время отклика REST-сервера, обрабатывающего фиксированную нагрузку 500 OPS с использованием разных уборщиков. Операция выполняется очень быстро; она просто выделяет память и сохраняет относительно большой массив байтов (с заменой существующего массива для поддержания затрат памяти на постоянном уровне).

Таблица 6.5. Эффекты задержки в работе уборщиков мусора с конкурентным сжатием

Уборщик	Среднее время	90%-ное время	99%-ное время	Максимальное время
Параллельный уборщик мусора	13 мс	60 мс	160 мс	265 мс
Уборщик мусора G1	5 мс	10 мс	35 мс	87 мс
ZGC	1 мс	5 мс	5 мс	20 мс
Уборщик мусора Shenandoah	1 мс	5 мс	5 мс	22 мс

Эти результаты вполне соответствуют тому, чего можно было бы ожидать от разных уборщиков. Время полной уборки мусора параллельного уборщика приводит к максимальному времени отклика 265 мс и большому количеству выбросов с временем отклика, превышающим 50 мс. С уборщиком мусора G1 время полной уборки мусора исчезает, но при уборке мусора в молодом поколении остаются более короткие задержки, что дает максимальное время 87 мс и выбросы за пределами 10 мс. У конкурентных уборщиков эти паузы с уборкой мусора в молодом поколении исчезли, так что максимальное время теперь составляет около 20 мс, а выбросы — только 5 мс.

Предупреждение: паузы уборки мусора традиционно вносили наибольший вклад в аномальные задержки. Но есть и другие причины: временные сетевые заторы между сервером и клиентом, задержки планирования ОС и т. д. Таким образом, хотя многие выбросы в двух предшествующих случаях обусловлены короткими паузами по несколько миллисекунд, возникающими в работе конкурентных уборщиков, мы находимся в области, в которой другие факторы также могут оказывать серьезное влияние на общую задержку.

Влияние конкурентных уборщиков со сжатием на производительность

Воздействие этих уборщиков мусора на производительность трудно разложить на категории. Как и уборщик G1, эти уборщики мусора G1 зависят от сканирования и обработки кучи фоновыми потоками. Если для таких потоков нет достаточных циклов уборки мусора, уборщики сталкиваются с теми же конкурентными сбоями, которые уже встречались нам ранее при выполнении полной уборки мусора. Конкурентные уборщики со сжатием обычно используют еще больший объем фоновой обработки, чем фоновые потоки уборщика G1.

С другой стороны, если для этих фоновых потоков доступны достаточные ресурсы процессора, производительность при использовании таких уборщиков будет выше производительности G1 или параллельного уборщика. И снова это соответствует данным из главы 5. Примеры этой главы показывали, что уборщик G1 обладает более высокой производительностью, чем параллельный уборщик, при передаче уборки мусора фоновым потокам. Конкурентные уборщики мусора со сжатием обладают теми же преимуществами перед параллельным уборщиком и аналогичными (но меньшими) преимуществами перед G1.

Фиктивный эпсилон-уборщик

В JDK 11 также входит уборщик мусора, который не делает ничего: эпсилон-уборщик. При использовании этого уборщика объекты никогда не освобождаются из кучи, а когда куча заполняется, вы получаете ошибку нехватки памяти.

Конечно, традиционные программы не смогут пользоваться этим уборщиком. Он предназначен для внутреннего тестирования JDK, но теоретически может пригодиться в двух ситуациях:

- Программы с очень коротким сроком жизни.
- Программы, которые были написаны специально с расчетом на повторное использование памяти и не должны выделять новой памяти.

Вторая категория встречается в некоторых встроенных средах с ограничениями по памяти. Эта область программирования является узкоспециализированной; здесь она не рассматривается. Однако в первом случае возникают некоторые интересные возможности.

Возьмем программу, которая создает список-массив из 4096 элементов, каждый из которых представляет собой массив байтов с размером 0,5 Мбайт. Время выполнения этой программы с разными уборщиками мусора приведено в табл. 6.6. В примере используется конфигурация уборщиков по умолчанию.

Таблица 6.6. Метрики производительности программы с выделением памяти

Уборщик мусора	Время	Необходимый размер кучи
Параллельный	2,3 секунды	3072 Мбайт
G1	3,24 секунды	4096 Мбайт
Эпсилон	1,6 секунды	2052 Мбайт

Отключение уборки мусора в этом случае обеспечивает значительный выигрыш — улучшение на 30%. Остальные уборщики требовали значительной

дополнительной памяти: как и другие экспериментальные уборщики, которые вы видели, в архитектуре эпсилон-уборщика не используются поколения (так как объекты не могут освобождаться, нет необходимости резервировать отдельное пространство для их быстрого освобождения). Таким образом, для этого теста, создающего объект с размером около 2 Гбайт, общий размер кучи, необходимой для эпсилон-уборщика, будет чуть больше этой величины: этот пример можно запустить с флагом `-Xmx2052m`. Параллельному уборщику нужно примерно на 1/3 больше памяти для хранения нового поколения, тогда как уборщику G1 нужно еще больше памяти для создания всех своих областей. Чтобы использовать этого уборщика, необходимо снова установить флаг `-XX:+UnlockExperimentalVMOptions` с флагом `-XX:+UseEpsilonGC`.

Выполнение программ с этим уборщиком мусора рискованно, если только вы твердо не уверены в том, что программе никогда не понадобится больше памяти, чем вы ей предоставили. Однако в таких случаях он может обеспечить неплохой выигрыш по производительности.

Итоги

В последних двух главах мы уделили достаточно много времени изучению подробностей работы уборки мусора (и разных ее алгоритмов). Если уборка мусора занимает больше времени, чем вам хотелось бы, вся эта информация поможет вам определиться с тем, какие действия помогут улучшить ситуацию.

Теперь, когда вы понимаете все нюансы, давайте отступим на шаг назад и определимся с подходом к выбору и настройке уборщика мусора. Приведенный ниже набор вопросов поможет вам разобраться со всеми нюансами:

Допустимы ли в приложении паузы полной уборки мусора?

> Если нет, выбирайте алгоритм G1. Даже если полные паузы приемлемы, уборщик мусора G1 часто будет эффективнее параллельного уборщика, если только ваше приложение не создает интенсивной вычислительной нагрузки.

Возможно ли добиться нужной вам производительности с настройками по умолчанию?

> Начните с настроек по умолчанию. По мере совершенствования технологии уборки мусора эргономическая (автоматическая) настройка работает все эффективнее. Если вы не добиваетесь нужной производительности, убедитесь в том, что проблема кроется в уборке мусора. Просмотрите журналы уборки мусора; определите, сколько времени проводится за уборкой мусора и с какой частотой возникают долгие паузы. В занятом приложении, если за уборкой мусора проводится 3% и менее времени, настройка вряд ли принесет суще-

ственную пользу (хотя вы всегда можете попробовать сократить выбросы, если это является вашей целью).

Время паузы в какой-то степени близко к вашей цели?

Если да, возможно, настройка максимального времени паузы — все, что вам нужно. Если нет, придется действовать иначе. Если время паузы слишком велико, но эффективность вас устраивает, вы можете сократить размер молодого поколения (а для пауз полной уборки мусора — старого поколения); пауз будет больше, но они станут более короткими.

Эффективность недостаточна даже при коротких паузах в ходе уборки мусора?

Необходимо увеличить размер кучи (или по крайней мере молодого поколения). Больше не всегда значит лучше: увеличение размера кучи приводит к большей длительности пауз. Даже с конкурентным уборщиком увеличение кучи по умолчанию означает увеличение молодого поколения, поэтому вы столкнетесь с более долгими паузами в ходе уборки мусора в молодом поколении. Но если это возможно, увеличьте размер кучи или по крайней мере относительные размеры поколений.

Вы используете конкурентного уборщика мусора и сталкиваетесь с полными уборками мусора из-за сбоев конкурентного режима?

Если в системе имеются доступные ресурсы процессора, попробуйте повысить количество конкурентных потоков уборки мусора или ускорить фоновую очистку, изменяя значение `InitiatingHeapOccupancyPercent`. Для G1 конкурентный цикл не запустится при наличии незавершенных смешанных уборок мусора; попробуйте уменьшить целевое количество смешанных уборок мусора.

Вы используете конкурентного уборщика и сталкиваетесь с полными уборками мусора из-за сбоев повышения?

В уборщике G1 сбой эвакуации указывает на то, что куча фрагментирована, но обычно проблема может быть решена, если уборщик мусора G1 начнет выполнять свою фоновую обработку на более ранней стадии, а смешанные циклы будут выполняться быстрее. Попробуйте увеличить количество конкурентных потоков G1, отрегулировать `InitiatingHeapOccupancyPercent` или сократить целевое количество смешанных уборок мусора.

ГЛАВА 7

Практика работы с памятью кучи

В главах 5 и 6 подробно рассматривается настройка уборщиков мусора, чтобы они оказывали минимальное воздействие на работу программы. Настройка уборщика мусора играет важную роль, но часто лучшего выигрыша по производительности удается добиться за счет использования улучшенных практик программирования. В этой главе обсуждаются некоторые практические приемы использования памяти кучи в Java.

Здесь приходится учитывать две противоречащих друг другу цели. Первое универсальное правило — создавать объекты умеренно и избавляться от них как можно быстрее. Использование меньшего объема памяти — лучший способ повысить эффективность уборки мусора. С другой стороны, частое повторное создание некоторых разновидностей объектов может привести к ухудшению общей производительности (даже если производительность уборки мусора при этом повышается). Если вместо этого будет организовано повторное использование объектов, быстродействие программы значительно повысится. Объекты могут повторно использоваться разными способами, включая потоково-локальные переменные, специальные ссылки на объекты и пулы объектов. Повторное использование объектов означает, что они будут долго существовать и повлияют на работу уборщика мусора, но при их разумном повторном использовании общая производительность улучшится.

В этой главе рассматриваются оба подхода, их относительные достоинства и недостатки. Но сначала мы рассмотрим инструменты, которые помогут понять, что же происходит с кучей.

Анализ кучи

Журналы уборки мусора и инструменты, описанные в главе 5, помогают оценить воздействие уборки мусора на приложение, но для получения дополнительной

информации необходимо заглянуть в саму кучу. Инструменты, описанные в этом разделе, предоставляют информацию об объектах, которые используются приложением в настоящий момент.

В большинстве случаев эти инструменты работают только с живыми объектами в куче — объекты, которые будут освобождены при следующем цикле полной уборки мусора, не включаются в выходные данные. В некоторых случаях программы решают эту задачу, инициируя полную уборку мусора, так что использование программы может повлиять на последующее поведение приложения. В других случаях программы перебирают содержимое кучи и выдают информацию о живых данных без освобождения объектов. Впрочем, в любом случае работа программ требует времени и машинных ресурсов.

Гистограммы кучи

Сокращение потребления памяти — важная цель, но, как и во многих областях производительности, бывает полезно сконцентрироваться на еще более узкой цели для максимизации возможного выигрыша. Позднее в этой главе будет приведен пример отложенной инициализации объекта `Calendar`. Такой подход сэкономит 640 байт в куче, но если приложение всегда инициализирует один такой объект, никаких заметных различий в производительности не будет. Необходимо провести анализ для определения того, какие разновидности объектов потребляют большое количество памяти.

Проще всего это сделать при помощи гистограммы кучи. Гистограммы позволяют быстро оценить количество объектов в приложении без получения полного дампа кучи (так как анализ дампа может потребовать времени, а дамп занимает большой объем дискового пространства). Если несколько конкретных разновидностей объектов приводят к созданию дефицита памяти в приложении, гистограмма кучи позволяет быстро узнать об этом.

Для получения гистограммы кучи можно воспользоваться командой `jcmd` (в данном случае с идентификатором процесса 8898):

```
% jcmd 8998 GC.class_histogram
8898:

 num     #instances         #bytes  class name
----------------------------------------------
   1:        789087       31563480  java.math.BigDecimal
   2:        172361       14548968  [C
   3:         13224       13857704  [B
   4:        184570        5906240  java.util.HashMap$Node
   5:         14848        4188296  [I
   6:        172720        4145280  java.lang.String
```

```
  7:         34217        3127184  [Ljava.util.HashMap$Node;
  8:         38555        2131640  [Ljava.lang.Object;
  9:         41753        2004144  java.util.HashMap
 10:         16213        1816472  java.lang.Class
```

В верхней части гистограммы обычно встречаются символьные массивы (`[C`) и объекты `String` — эти объекты Java создаются чаще других. Также часто встречаются массивы байтов (`[B`) и массивы объектов (`[Ljava.lang.Object;`), потому что загрузчики классов хранят свои данные в этих структурах. Если вы не знакомы с этим синтаксисом, он описан в документации Java Native Interface (JNI).

В этом примере стоит обратить внимание на класс `BigDecimal`: мы знаем, что пример кода создает множество временных объектов `BigDecimal`, но нахождение такого количества этих объектов в куче — не то, чего можно было бы ожидать в обычных условиях. Вывод `GC.class_histogram` включает только живые объекты, так как команда обычно инициирует полную уборку мусора. В команду можно включить флаг `-all`, чтобы пропустить полную уборку мусора, хотя тогда гистограмма будет содержать объекты без ссылок (мусор).

Аналогичный вывод можно получить следующей командой:

```
% jmap -histo process_id
```

Вывод `jmap` включает объекты, пригодные для уборки мусора (мертвые объекты).

Чтобы инициировать полную уборку мусора до вывода гистограммы, выполните следующую команду:

```
% jmap -histo:live process_id
```

Гистограммы невелики, так что построение гистограммы для каждого теста в автоматизированной системе может принести пользу. Но поскольку построение гистограммы и инициирование полной уборки мусора занимает несколько секунд, не следует делать это в стабильном состоянии хронометража производительности.

Дампы кучи

Гистограммы прекрасно подходят для выявления проблем, обусловленных выделением слишком большого количества экземпляров одного или двух конкретных классов, но для более глубокого анализа потребуется дамп кучи. Существует много программ для просмотра дампов кучи; большинство из них может подключаться к работающей программе для генерирования дампа. Часто бывает проще сгенерировать дамп из командной строки, что можно сделать одной из следующих команд:

```
% jcmd process_id GC.heap_dump /path/to/heap_dump.hprof
```

или

```
% jmap -dump:live,file=/path/to/heap_dump.hprof process_id
```

Включение параметра `live` в командную строку `jmap` запускает принудительную уборку мусора перед получением дампа кучи. Этот режим используется по умолчанию для `jcmd`, хотя если вы по какой-то причине захотите включить другие (мертвые) объекты, добавьте `-all` в конец командной строки `jcmd`. Разумеется, использование команды с принудительной полной уборкой мусора создаст длинную паузу в приложении, но даже если не форсировать полной уборки мусора, приложение будет приостановлено на время, необходимое для записи дампа кучи.

Обе команды создают файл с именем heap_dump.hprof в заданном каталоге; затем этот файл может быть открыт в различных программах. Самые популярные из этих программ:

`jvisualvm` — на вкладке Monitor программы `jvisualvm` можно получить дамп кучи работающей программы или открыть ранее созданный дамп. После этого вы сможете просмотреть данные из кучи, проанализировать самые большие хранимые объекты и выполнить произвольные запросы к куче.

`mat` — распространяемая с открытым кодом программа EclipseLink Memory Analyzer tool (`mat`) позволяет загрузить один или несколько дампов кучи и проанализировать их. Она способна строить отчеты, в которых указывается вероятный источник проблем; кроме того, с ее помощью можно просматривать кучу и выполнять с ней SQL-подобные запросы.

Анализ кучи на первом проходе обычно направлен на удерживаемую (retained) память объекта. Удерживаемой памятью объекта называется объем памяти, которая будет освобождена, если сам объект будет пригодным для уничтожения в ходе уборки мусора. На рис. 7.1 удерживаемая память объекта `String Trio` включает память, занимаемую самим объектом, а также память, занимаемую объектами `Sally` и `David`. Она не включает память, используемую объектом `Michael`, потому что на этот объект есть другая ссылка, и при освобождении `String Trio` этот объект не будет пригоден для уборки мусора.

Объекты, удерживающие большой объем памяти кучи, часто называются *доминаторами* кучи. Если программы анализа кучи показывают, что малое количество объектов доминирует в большой части кучи, все просто: все, что нужно сделать — создавать меньшее количество таких объектов, хранить их в течение более короткого периода времени, упростить их граф объектов или уменьшить их размеры. Это проще сказать, чем сделать, но по крайней мере анализ будет несложным.

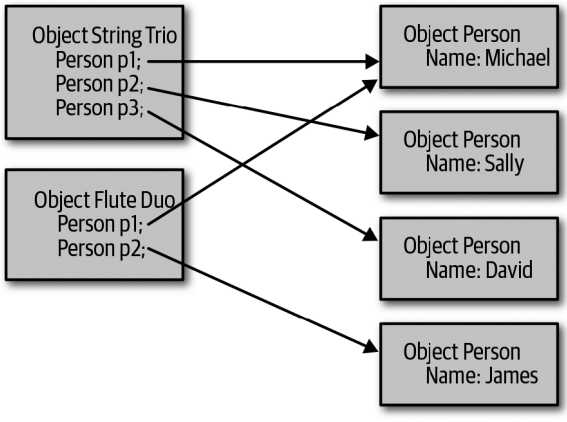


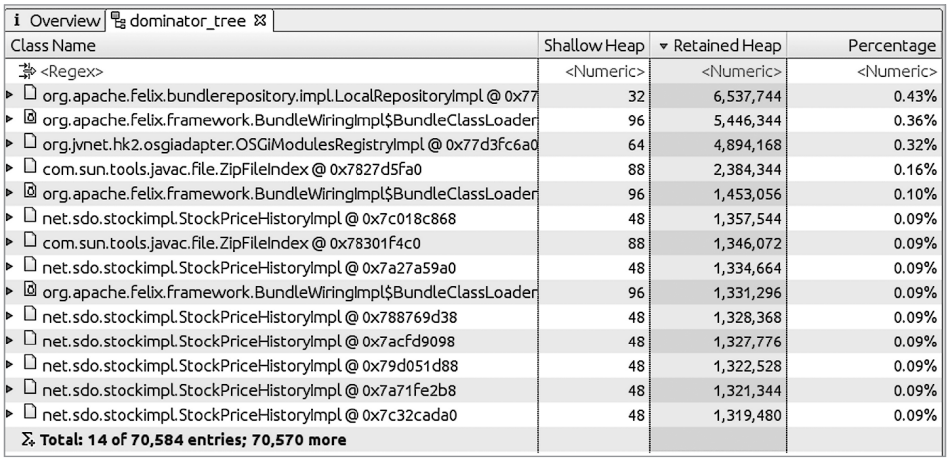
Рис. 7.1. Граф объектов удерживаемой памяти

ПОВЕРХНОСТНЫЕ, УДЕРЖИВАЕМЫЕ И ГЛУБОКИЕ РАЗМЕРЫ ОБЪЕКТОВ

Два других полезных термина для анализа памяти — поверхностный и глубокий размер. Поверхностный размер объекта определяется размером самого объекта. Если объект содержит ссылку на другой объект, то 4 или 8 байт ссылки будут в него включены, но размер целевого объекта не учитывается.

Глубокий размер объекта включает размер объекта, ссылка на который в нем хранится. Различия между глубоким размером объекта и удерживаемой памятью объекта проявляются в совместно используемых объектах. На рис. 7.1 глубокий размер объекта Flute Duo включает память, занимаемую объектом Michael, тогда как в удерживаемый размер объекта Flute Duo она не включается.

Как правило, от вас потребуется аналитическая работа, потому что в программе с большой вероятностью объекты используются совместно. Такие совместные объекты (как объект `Michael` на предыдущей схеме) не включаются в удерживаемую память любого другого объекта, так как освобождение одного отдельного объекта не приводит к освобождению совместно используемого объекта. Кроме того, наибольшими удерживаемыми размерами обычно обладают загрузчики классов, которыми вы управлять не можете. На рис. 7.2 показаны наибольшие удерживаемые объекты в куче из версии сервера биржевых котировок, который использует сильное кэширование данных на уровне клиентского подключения, и слабое — в глобальной хеш-карте (с созданием нескольких ссылок на кэшируемые элементы).



Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.felix.bundlerepository.impl.LocalRepositoryImpl @ 0x77	32	6,537,744	0.43%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	5,446,344	0.36%
org.jvnet.hk2.osgiadapter.OSGiModulesRegistryImpl @ 0x77d3fc6a0	64	4,894,168	0.32%
com.sun.tools.javac.file.ZipFileIndex @ 0x7827d5fa0	88	2,384,344	0.16%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,453,056	0.10%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c018c868	48	1,357,544	0.09%
com.sun.tools.javac.file.ZipFileIndex @ 0x78301f4c0	88	1,346,072	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a27a59a0	48	1,334,664	0.09%
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader	96	1,331,296	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x788769d38	48	1,328,368	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7acfd9098	48	1,327,776	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x79d051d88	48	1,322,528	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a71fe2b8	48	1,321,344	0.09%
net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c32cada0	48	1,319,480	0.09%
Total: 14 of 70,584 entries; 70,570 more			

Рис. 7.2. Представление Retained Memory в Memory Analyzer

Куча содержит 1,4 Гбайт объектов (это значение на вкладке не отображается). Даже при этом наибольший набор объектов, представленных одиночной ссылкой, составляет всего 6 Мбайт (неудивительно, что он является частью библиотеки загрузки классов). Анализ объектов, напрямую удерживающих наибольший объем памяти, не решит проблем с памятью.

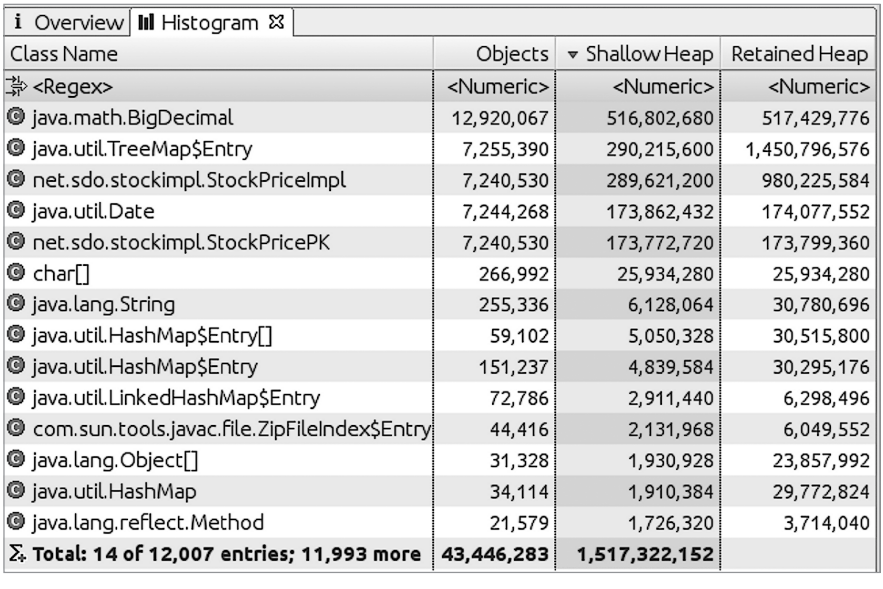
В этом примере в списке присутствуют множественные экземпляры объектов `StockPriceHistoryImpl`, каждый из которых удерживает значительный объем памяти. По объему памяти, занимаемой этими объектами, можно заключить, что именно они являются источником проблем. Однако в общем случае объекты могут совместно использоваться так, что просмотр удерживаемой кучи не даст никаких очевидных результатов.

Вторым полезным шагом станет гистограмма объектов (рис. 7.3).

На гистограмме обобщается информация об объектах одного типа, и в данном примере намного более очевидно, что ключевую роль здесь играют 1,4 Гбайт памяти, удерживаемые семью миллионами `TreeMap$Entry` объектов. Даже не зная, что происходит в программе, достаточно просто использовать средства трассировки объектов в Memory Analyzer, чтобы понять, что удерживает их в памяти.

Средства анализа кучи позволят вам найти корни определенного объекта (или набора объектов в данном случае) в механизме уборки мусора — хотя прямой переход к корням не обязательно вам поможет. Корнями при уборке мусора называются системные объекты, содержащие статическую глобальную ссылку, которая (через длинную цепочку других объектов) ведет к соответствующему

объекту. Как правило, они представлены статическими переменными классов, загруженных в системном или загрузочном каталоге. К ним относятся класс `Thread` и все активные потоки; потоки удерживают объекты либо через потоково-локальные переменные, либо через ссылки на целевой объект `Runnable` (или в случае подкласса класса `Thread` — любые другие ссылки в подклассе).



Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.math.BigDecimal	12,920,067	516,802,680	517,429,776
java.util.TreeMap$Entry	7,255,390	290,215,600	1,450,796,576
net.sdo.stockimpl.StockPriceImpl	7,240,530	289,621,200	980,225,584
java.util.Date	7,244,268	173,862,432	174,077,552
net.sdo.stockimpl.StockPricePK	7,240,530	173,772,720	173,799,360
char[]	266,992	25,934,280	25,934,280
java.lang.String	255,336	6,128,064	30,780,696
java.util.HashMap$Entry[]	59,102	5,050,328	30,515,800
java.util.HashMap$Entry	151,237	4,839,584	30,295,176
java.util.LinkedHashMap$Entry	72,786	2,911,440	6,298,496
com.sun.tools.javac.file.ZipFileIndex$Entry	44,416	2,131,968	6,049,552
java.lang.Object[]	31,328	1,930,928	23,857,992
java.util.HashMap	34,114	1,910,384	29,772,824
java.lang.reflect.Method	21,579	1,726,320	3,714,040
Total: 14 of 12,007 entries; 11,993 more	**43,446,283**	**1,517,322,152**	

Рис. 7.3. Представление Histogram в Memory Analyzer

В некоторых случаях знание корней целевого объекта может пригодиться, но если объект содержит несколько ссылок, у него будет несколько корней GC. Ссылки образуют инвертированную древовидную структуру. Допустим, два объекта ссылаются на конкретный объект `TreeMap$Entry`. На каждый из этих объектов могут ссылаться два других объекта, на каждый из которых могут ссылаться три других объекта, и т. д. Разрастание ссылок при обратном отслеживании корней означает, что у любого заданного объекта с большой вероятностью будет существовать много корней.

Вместо этого будет полезнее поиграть в детектива и найти самую нижнюю точку графа объектов, в которой целевой объект используется совместно. Для этого вы анализируете объекты и их входные ссылки, а затем отслеживаете эти входные ссылки до нахождения дублирующего пути. В данном случае ссылки на объекты `StockPriceHistoryImpl`, хранящиеся в `TreeMap`, исходят из двух источников: объекта `ConcurrentHashMap`, содержащего данные атрибутов сеанса, и объекта `WeakHashMap`, содержащего глобальный кэш.

На рис. 7.4 обратная трассировка раскрывается до такой степени, что о них выводятся минимальные данные. Чтобы сделать вывод, что это сеансовые данные, нужно продолжить раскрывать путь `ConcurrentHashMap`. Аналогичная логика применяется для пути `WeakHashMap`.

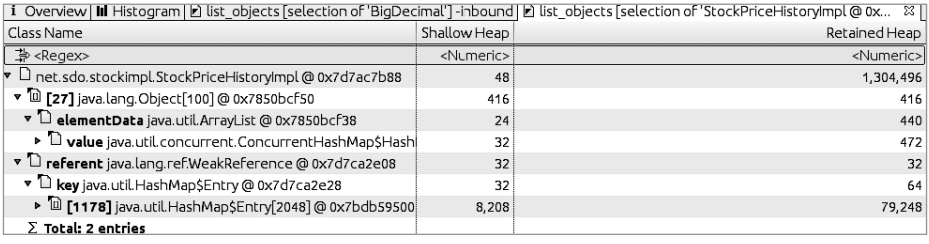


Рис. 7.4. Обратная трассировка ссылок на объекты в Memory Analyzer

Типы объектов, используемые в данном примере, несколько упрощают анализ по сравнению с типичными случаями. Если бы основные данные в этом приложении моделировались объектами `String` вместо объектов `BigDecimal` и хранились в объектах `HashMap` вместо объектов `TreeMap`, ситуация бы усложнилась. В дампе кучи присутствовали бы сотни тысяч других строк и десятки тысяч других объектов `HashMap`. Таким образом, поиск путей к интересным объектам потребует определенного терпения. Обычно следует начинать с объектов коллекций (например, `HashMap`) вместо элементов (например, `HashMap$Entry`) и искать самые большие коллекции.

РЕЗЮМЕ

- Информация о потреблении памяти объектами — первый шаг к определению того, какие объекты следует оптимизировать в вашем коде.
- Гистограммы позволяют легко и быстро выявлять проблемы с памятью, возникающие из-за создания слишком большого количества объектов определенного типа.
- Анализ дампа кучи — самый эффективный способ получения информации об использовании памяти, хотя и требующий некоторого терпения и усилий.

Ошибки нехватки памяти

JVM выдает ошибку нехватки памяти в следующих обстоятельствах:

- JVM недоступна низкоуровневая память.
- В метапространстве не хватает памяти.

- Памяти не хватает в самой куче Java: приложение не может создать дополнительные объекты при заданном размере кучи.
- JVM проводит слишком много времени за выполнением уборки мусора.

Последние два случая (в которых задействована сама куча Java) встречаются чаще, но при возникновении ошибки памяти не стоит автоматически заключать, что проблема связана с кучей. Необходимо сначала разобраться, почему возникла ошибка нехватки памяти (причина указывается в выводе исключения).

Нехватка низкоуровневой памяти

Первый случай в списке — недоступность низкоуровневой памяти для JVM — возникает по причинам, которые вообще никак не связаны с кучей. В 32-разрядной JVM максимальный размер процесса составляет 4 Гбайт (3 Гбайт в некоторых версиях Windows, около 3,5 Гбайт в некоторых старых версиях Linux). При определении очень большой кучи — допустим, 3,8 Гбайт — приложение оказывается в опасной близости от этого лимита. Даже в 64-разрядной JVM у операционной системы может не оказаться достаточной виртуальной памяти для запросов JVM.

Эта тема более полно рассматривается в главе 8. Если в сообщении для ошибки нехватки памяти упоминается выделение низкоуровневой памяти, настройка кучи пользы не принесет: необходимо разобраться в том, какая проблема низкоуровневой памяти упоминается в ошибке. Например, следующее сообщение указывает на исчерпание низкоуровневой памяти стеков потоков:

```
Exception in thread "main" java.lang.OutOfMemoryError:
unable to create new native thread
```

Однако учтите, что JVM иногда выдает эту ошибку из-за причин, не имеющих никакого отношения к памяти. У пользователей обычно ограничивается количество потоков, которые они могут запускать; это ограничение может устанавливаться ОС или контейнером. Например, в Linux пользователям часто разрешается создавать не более 1024 процессов (это значение можно проверить командой `ulimit -u`). При попытке создания 1025-го потока будет выдана та же ошибка `OutOfMemoryError` с заявлением о нехватке памяти для создания потока, тогда как в действительности ошибка была вызвана ограничением на количество процессов в ОС.

Нехватка памяти метапространства

Ошибка нехватки памяти метапространства также не связана с кучей — она возникает из-за заполнения низкоуровневой памяти метапространства. Так как

метапространство не имеет максимального размера по умолчанию, эта ошибка обычно возникает из-за того, что вы решили задать максимальный размер (а причина станет ясна в этом разделе).

Эта ошибка может возникать по двум основным причинам. Первая — что ваше приложение может просто использовать больше классов, чем помещается в выделенном метапространстве (см. «Определение размера метапространства», с. 181). Вторая причина более коварна: она связана с утечкой памяти в загрузчике классов. Чаще всего она проявляется на серверах с динамической загрузкой классов. Одним из примеров служит сервер приложений Java EE. Каждое приложение, развернутое на сервере приложения, выполняется в собственном загрузчике классов (который обеспечивает необходимую изоляцию, чтобы классы из одного приложения не использовались совместно — и не конфликтовали — с классами из другого приложения). В процессе разработки при каждом изменении приложение должно быть развернуто заново: создается новый загрузчик классов для загрузки новых классов, а старый загрузчик получает возможность выйти из области видимости. После того как загрузчик классов выходит из области видимости, метаданные классов могут быть освобождены.

Если старый загрузчик алгоритмов не выходит из области видимости, метаданные классов освобождаться не могут; со временем метапространство заполнится и выдаст ошибку нехватки памяти. В этом случае увеличение размера метапространства принесет пользу, но в конечном итоге оно только отложит возникновение ошибки.

Если эта ситуация возникает в среде сервера приложения, у вас практически нет других вариантов, кроме как связаться с разработчиком сервера и заставить его исправить утечку. Если вы пишете собственное приложение, которое создает и освобождает большое количество загрузчиков классов, убедитесь в том, что сами загрузчики классов правильно освобождаются (в частности, убедитесь в том, что ни один поток не назначает своим контекстным загрузчиком классов один из временных загрузчиков). Для исправления ситуации будет полезно провести только что описанный анализ дампа кучи: найдите на гистограмме все экземпляры класса `ClassLoader`, отследите их до корней GC и определите, что удерживает их в памяти.

Как и прежде, для понимания этой ситуации ключевую роль играет полнотекстовое описание ошибки. При заполнении метапространства текст ошибки будет выглядеть примерно так:

```
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
```

Кстати говоря, загрузчики классов — та причина, по которой следует рассматривать возможность определения максимального размера метапространства. Если

система с утечкой в загрузчиках классов сможет неограниченно расширяться, она захватит всю память на вашей машине.

Нехватка памяти в куче

Если нехватка памяти возникает в самой куче, сообщение об ошибке будет выглядеть так:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

АВТОМАТИЧЕСКОЕ СОХРАНЕНИЕ ДАМПА КУЧИ

Ошибки нехватки памяти могут возникать непредсказуемо, из-за чего может быть сложно определить, когда нужно сохранить дамп кучи. В этом вам помогут некоторые флаги JVM:

`-XX:+HeapDumpOnOutOfMemoryError` — установка этого флага (по умолчанию false) заставляет JVM создавать дамп кучи при каждой выдаче ошибки нехватки памяти;

`-XX:HeapDumpPath=<путь>` — задает место для сохранения дампа; по умолчанию файл `java_pid<pid>.hprof` в текущем рабочем каталоге приложения. Путь может определять либо каталог (в этом случае используется имя файла по умолчанию), либо имя файла для записи дампа;

`-XX:+HeapDumpAfterFullGC` — генерирует дамп кучи после выполнения полной уборки мусора;

`-XX:+HeapDumpBeforeFullGC` — генерирует дамп кучи перед выполнением полной уборки мусора.

При генерировании нескольких дампов кучи (например, из-за выполнения нескольких полных уборок мусора) к имени файла присоединяется порядковый номер дампа.

Попробуйте установить эти флаги, если приложение спонтанно выдает ошибки нехватки памяти из-за памяти кучи и вам нужен дамп памяти для анализа причин сбоя. Однако следует учитывать, что сохранение дампа(-ов) кучи повысит длительность пауз, потому что данные в представлении кучи будут записываться на диск.

Типичные случаи, в которых ошибка возникает из-за нехватки памяти в куче, сходны с рассмотренными выше примерами для метапространства. Возможно,

приложению просто требуется больше памяти в куче: живые объекты, которые в ней удерживаются, не помещаются в области памяти, выделенной для кучи. А может быть, в приложении возникла утечка памяти: оно продолжает выделять новые объекты, не позволяя другим объектам выйти из области видимости. В первом случае увеличение размера кучи решит проблему; во втором случае увеличение размера кучи всего лишь отложит ошибку.

В обоих случаях потребуется анализ дампа кучи для определения того, что же потребляет больше всего памяти; после этого вы можете направить усилия на сокращение количества (или размера) таких объектов. Если в приложении возникает утечка памяти, сохраните несколько последовательных дампов кучи с интервалом в несколько минут и сравните их. В `mat` такая возможность реализована во встроенном виде. Если в `mat` открыты два дампа кучи, присутствует возможность вычисления разности гистограмм двух куч.

На рис. 7.5 изображен классический случай утечки памяти Java, возникающий при использовании класса коллекции — в данном случае `HashMap`. (Классы коллекций являются самой частой причиной утечки памяти: приложение вставляет

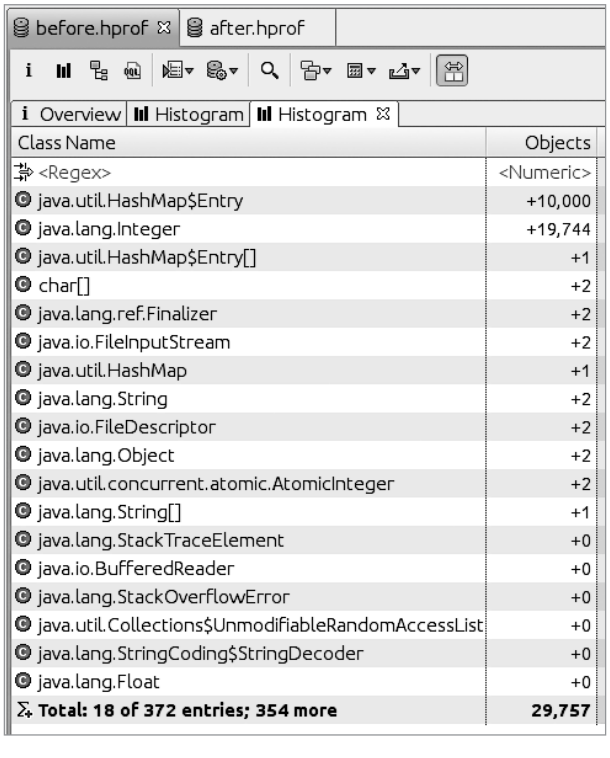


Рис. 7.5. Сравнение гистограмм

элементы в коллекцию и не освобождает их.) На иллюстрации изображена сравнительная гистограмма: на ней выводятся разности количества объектов в двух дампах кучи. Например, целевой дамп кучи содержит на 19 744 объекта `Integer` больше по сравнению с базовым уровнем.

Лучшее решение проблемы — изменить логику приложения так, чтобы элементы удалялись из коллекции, когда надобность в них отпадает. Кроме того, коллекция с использованием слабых (или мягких) ссылок может автоматически освобождать их, когда ничто в приложении не ссылается на них, но за использование этих коллекций приходится расплачиваться (см. далее в этой главе).

Часто при выдаче таких исключений JVM не завершает работу, потому что исключение влияет только на один поток в JVM. Возьмем JVM с двумя потоками, выполняющими вычисления. Один из них может получить ошибку `OutOfMemoryError`. По умолчанию обработчик исключения этого потока выведет трассировку стека, а поток завершит выполнение.

Но у JVM остается еще один активный поток, так что JVM не завершится. А поскольку поток, столкнувшийся с ошибкой, был завершен, значительный объем памяти с большой вероятностью будет освобожден при будущем цикле уборки мусора: все объекты, ссылки на которые содержались в завершенном потоке, но не в других потоках. Таким образом, выживший поток сможет продолжить выполнение, и довольно часто будет располагать достаточной памятью для завершения своей задачи.

Серверные фреймворки с пулами потоков для обработки запросов будут работать практически по той же схеме. Обычно они перехватывают ошибку и предотвращают аварийное завершение потока, но это не существенно; память, связанная с запросом, который выполняется потоком, все равно будет пригодной для уборки.

Таким образом, выдача ошибки будет фатальной для JVM только в том случае, если она приводит к завершению последнего потока JVM, который не является демоном. В серверном фреймворке такое в принципе невозможно, и часто невозможно в автономных программах с несколькими потоками. Обычно в такой схеме все работает хорошо, потому что память, связанная с активным запросом, часто становится пригодной для уборки.

Если вы хотите, чтобы JVM завершалась при исчерпании памяти кучи, установите флаг `-XX:+ExitOnOutOfMemoryError`, который по умолчанию равен `false`.

Превышение порога времени при уборке мусора

Процесс восстановления, описанный для предыдущего случая, предполагает, что при возникновении ошибки нехватки памяти в потоке память, связанная с текущей работой потока, станет пригодной для уборки мусора и JVM сможет

восстановить работоспособное состояние. В действительности это не всегда так, что приводит нас к последнему случаю выдачи ошибки нехватки памяти JVM: когда JVM определяет, что она проводит слишком много времени за выполнением уборки мусора.

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit
exceeded
```

Эта ошибка выдается при выполнении всех следующих условий:

- Время, потраченное на полные уборки мусора, превышает значение, заданное флагом `-XX:GCTimeLimit=`*N*. Значение по умолчанию равно 98 (то есть если 98% времени проводится за уборкой мусора).
- Объем памяти, освобожденной при полной уборке мусора, меньше значения, заданного флагом `-XX:GCHeapFreeLimit=`*N*. Значение по умолчанию равно 2 (то есть если в ходе полной уборки мусора освобождается менее 2% кучи, условие выполнено).
- Два предыдущих условия выполняются для пяти последовательных циклов уборки мусора (это значение не настраивается).
- Значение флага `-XX:+UseGCOverheadLimit` равно `true` (используется по умолчанию).

Обратите внимание: должны быть выполнены *все четыре* условия. Нередко встречаются ситуации, в которых в приложении происходят более пяти последовательных полных уборок мусора без выдачи ошибки памяти. Это объясняется тем, что даже если приложение проводит 98% своего времени за полными уборками мусора, оно может освобождать более 2% кучи в ходе каждой уборки. В таком случае подумайте об увеличении значения `GCHeapFreeLimit`.

Обратите внимание: если первые два условия выполняются для четырех последовательных циклов полной уборки мусора, то JVM делает последнюю отчаянную попытку очистить память и освобождает все мягкие ссылки в JVM перед пятым циклом полной уборки мусора. Часто это позволяет избежать ошибки, так как пятый цикл может освободить более 2% кучи (предполагается, что в приложении используются мягкие ссылки).

РЕЗЮМЕ

- Ошибки нехватки памяти возникают по разным причинам; не следует полагать, что причина всегда связана с кучей.
- Как для метапространства, так и для обычной кучи ошибки нехватки памяти чаще всего возникают из-за утечки памяти; средства анализа кучи помогут найти истинную причину утечки.

Снижение потребления памяти

Первый подход к более эффективному использованию памяти в Java заключается в использовании меньшего объема памяти кучи. Вряд ли это утверждение кого-нибудь удивит: если вы используете меньше памяти, куча будет заполняться с меньшей частотой, а это потребует меньшего количества циклов уборки мусора. Эффект накапливается: меньше уборок в молодом поколении означает, что возраст объекта будет увеличиваться реже — а значит, объект с меньшей вероятностью будет повышен в старое поколение. А значит, количество полных циклов уборки мусора (или конкурентных циклов уборки) будет сокращаться. И если эти полные циклы будут освобождать больше памяти, то они тоже будут происходить с меньшей частотой.

В этом разделе рассматриваются три способа использования меньшего объема памяти: сокращение размера объектов, использование отложенной инициализации объектов и использование канонических объектов.

Уменьшение размеров объектов

Объекты занимают определенный объем памяти в куче, так что простейший способ использования меньшего объема памяти заключается в уменьшении размеров объектов. Из-за ограничений памяти на машине, на которой работает ваша программа, увеличение размера кучи на 10% может оказаться невозможным, но 20%-ное уменьшение половины объектов в куче позволит добиться того же эффекта. Как упоминалось в главе 12, в Java 11 реализована такая оптимизация для объектов `String`; это означает, что пользователи Java 11 часто могут установить максимальный размер кучи на 25% меньше того, который требовался в Java 8 — без каких-либо последствий для уборки мусора или производительности.

Размер объекта можно уменьшить (очевидно) сокращением числа содержащихся в нем переменных экземпляра или (менее очевидно) сокращением размера этих переменных.

В табл. 7.1 приведены размеры переменных экземпляров для всех типов Java.

Тип `reference` представляет ссылку на любую разновидность объектов Java — экземпляров классов или массивов. Эта память предназначена для хранения только самой ссылки. Размер объекта, содержащего ссылки на другие объекты, зависит от того, хотите ли вы рассматривать поверхностный, глубокий или удерживаемый размер объекта, но этот размер также включает некоторые невидимые поля заголовков объектов. Для обычного объекта размер полей заголовка составляет 8 байт на 32-разрядной JVM или 16 байт на 64-разрядной JVM (независимо от размера кучи). Для массива размер полей заголовка составляет 16 байт на

32-разрядной или 64-разрядной JVM, если размер кучи меньше 32 Гбайт, или 24 байта в противном случае.

Таблица 7.1. Размеры переменных экземпляров в Java[1]

Тип	Размер (в байтах)
`byte`	1
`char`	2
`short`	2
`int`	4
`float`	4
`long`	8
`double`	8
`reference`	8 (в JVM для 32-разрядных версий Windows 4)[1]

Например, возьмем следующие определения классов:

```
public class A {
    private int i;
}

public class B {
    private int i;
    private Locale l = Locale.US;
}

public class C {
    private int i;
    private ConcurrentHashMap chm = new ConcurrentHashMap();
}
```

Фактические размеры одного экземпляра этих объектов (на 64-разрядной JVM с размером кучи менее 32 Гбайт) приведены в табл. 7.2.

В классе `B` определение ссылки `Locale` увеличивает размер объекта на 8 байт, но по крайней мере в данном примере объект `Locale` используется совместно с другими классами. Если объект `Locale` не используется классом, включение переменной приведет только к потере дополнительных байтов на хранение

[1] За подробностями обращайтесь к разделу «Сжатые OOP», с. 296.

ссылки. Однако если в приложении создается большое количество экземпляров класса `B`, эти байты быстро накапливаются.

Таблица 7.2. Размеры простых объектов в байтах

	Поверхностный размер	Глубокий размер	Удерживаемый размер
A	16	16	16
B	24	216	24
C	24	200	200

ВЫРАВНИВАНИЕ ОБЪЕКТОВ И РАЗМЕРЫ

Все классы, упомянутые в табл. 7.2, содержат дополнительное целочисленное поле, которое не упоминалось в обсуждении. Для чего оно нужно?

Откровенно говоря, переменная была включена только для того, чтобы вам было проще понять описание этих классов: класс `B` содержит на 8 байт больше, чем класс `A`, что выглядит вполне логично (и позволяет лучше донести суть).

Но при этом упускается одна важная деталь: размеры объектов всегда дополняются, чтобы они были кратны 8 байтам. Без определения `i` в классе `A` экземпляры `A` все равно будут потреблять 16 байт — 4 байта будут использоваться просто для выравнивания размера объекта до значения, кратного 8, а не для хранения ссылки на `i`. Без определения `i` экземпляры класса `B` будут потреблять только 16 байт — столько же, сколько потребляет класс `A`, хотя `B` содержит лишнюю ссылку на объект. Выравнивание также объясняет, почему экземпляр `B` занимает на 8 байт больше экземпляра `A`, хотя он содержит всего одну дополнительную (4-байтовую) ссылку.

JVM также применяет дополнение к объектам с нечетным количеством байт. Это делается для того, чтобы массивы, содержащие такие объекты, хорошо выравнивались по границам адресов, оптимальным для используемой архитектуры.

Таким образом, исключение некоторых полей экземпляров или сокращение размеров полей объекта может принести пользу, а может и не принести — но нет никаких причин не применять их.

В проект OpenJDK входит отдельно загружаемая программа `jol` для вычисления размеров объектов.

С другой стороны, при определении и создании `ConcurrentHashMap` потребляются дополнительные байты для ссылки на объект, а также дополнительные байты для объекта `HashMap`. Если последний не используется, то экземпляры класса `C` неэффективно используют память.

Определение только необходимых переменных экземпляров — единственный способ экономии памяти в объекте. Менее очевидный случай — использование меньших типов данных. Если класс должен отслеживать одно из восьми возможных используемых состояний, вместо `int` можно использовать тип `byte` — с потенциальной экономией 3 байт. Использование `float` вместо `double`, `int` вместо `long` и т. д. позволит сэкономить память, особенно в классах с часто создаваемыми экземплярами. Как было показано в главе 12, использование коллекций с правильно выбранными размерами (или простых переменных экземпляров вместо коллекций) позволяет добиться сходной экономии.

Исключение полей экземпляров в объекте может уменьшить размер объекта, но при этом остается некоторая неопределенность: как насчет полей, в которых хранятся результаты промежуточных вычислений? Здесь возникает классическая проблема выбора баланса между затратами времени и памяти: что лучше — потратить память для хранения значения или время (ресурсы процессора), чтобы вычислять значение по мере надобности? В Java этот выбор также относится к процессорному времени, так как дополнительная память может заставить уборку мусора потреблять больше вычислительных мощностей.

Например, хеш-код `String` вычисляется суммированием, в котором задействованы все символы строки; эти вычисления требуют времени. Из-за этого класс `String` хранит значение в переменной экземпляра, чтобы хеш-код было достаточно вычислить всего один раз: в конечном итоге повторное использование этого значения почти всегда обеспечит прирост производительности, который будет важнее любой экономии памяти при отказе от его хранения. С другой стороны, метод `toString()` многих классов не кэширует строковое представление объекта в переменной экземпляра, что привело бы к затратам памяти на хранение как переменной экземпляра, так и строки, ссылка на которую в нем хранится. Вместо этого время, необходимое для вычисления новой строки, обычно предпочтительнее затрат памяти на хранение ссылки на строку. (Также стоит отметить, что хеш-код `String` используется часто, а строковое представление объекта `toString()` обычно используется гораздо реже.)

Эта ситуация безусловно субъективна. Точка на оси «память/время», в которой происходит переключение между использованием памяти для кэширования значения и повторным вычислением этого значения, зависит от многих факторов. Если целью является сокращение работы по уборке мусора, то баланс смещается в пользу повторного вычисления.

РЕЗЮМЕ

- Сокращение размеров объектов часто повышает эффективность уборки мусора.
- Размер объекта не всегда очевиден: объекты выравниваются по 8-байтовым границам, а размеры ссылок на объекты различаются в 32- и 64-разрядных JVM.
- Даже null-переменные экземпляров расходуют память в классах объектов.

Отложенная инициализация

В большинстве случаев решение о том, действительно ли необходима та или иная переменная экземпляра, не настолько однозначное, как может показаться из предыдущего раздела. Конкретному классу объект `Calendar` может понадобиться всего в 10% случаях, но создание объектов `Calendar` обходится дорого, и такие объекты определенно есть смысл хранить (вместо того чтобы повторно создавать их по мере надобности). В подобных ситуациях отложенная инициализация может пригодиться.

До настоящего момента в нашем обсуждении предполагалось, что переменные экземпляров инициализируются сразу. Класс, которому нужен объект `Calendar` (и которому не обязательно быть потоково-безопасным), может выглядеть примерно так:

```
public class CalDateInitialization {
    private Calendar calendar = Calendar.getInstance();
    private DateFormat df = DateFormat.getDateInstance();

    private void report(Writer w) {
        w.write("On " + df.format(calendar.getTime()) + ": " + this);
    }
}
```

Отложенная инициализация полей сопряжена с небольшим компромиссом в отношении вычислительной эффективности — код должен проверять состояние переменной при каждом выполнении:

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
```

```
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) + ": " + this);
    }
}
```

ПРОИЗВОДИТЕЛЬНОСТЬ ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ НА СТАДИИ ВЫПОЛНЕНИЯ

Обычная потеря производительности при проверке того, была ли инициализирована переменная с отложенной инициализацией, не является неизбежной. Взгляните на пример из класса JDK ArrayList. Класс поддерживает массив хранимых в нем элементов, и в старых версиях Java псевдокод класса выглядел примерно так:

```
public class ArrayList {
    private Object[] elementData = new Object[16];
    int index = 0;
    public void add(Object o) {
        ensureCapacity();
        elementData[index++] = o;
    }
    private void ensureCapacity() {
        if (index == elementData.length) {
            ...Новое выделение памяти для массива и копирование старых
               данных...
        }
    }
}
```

Через несколько лет класс был изменен так, что массив `elementData` инициализируется в отложенном режиме. Но поскольку метод `ensureCapacity()` уже должен проверять размер массива, типичные методы класса не сталкиваются с потерей производительности: код проверки инициализации совпадает с кодом, проверяющим необходимость увеличения размера массива. В новом коде используется статический общий массив нулевой длины, чтобы производительность осталась неизменной:

```
public class ArrayList {
    private static final Object[] EMPTY_ELEMENTDATA = {} ;
    private Object[] elementData = EMPTY_ELEMENTDATA;
}
```

А это означает, что метод `ensureCapacity()` может остаться (практически) неизменным, так как и `index`, и `elementData.length` начинаются с `0`.

Отложенную инициализацию лучше всего применять для достаточно редких операций. Если операция используется часто, экономии памяти не будет (потому что объект всегда будет создаваться), а каждое выполнение частой операции будет сопровождаться незначительной потерей производительности.

Если код должен быть потоково-безопасным, отложенная инициализация усложняется. Для начала проще всего добавить традиционную синхронизацию:

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private synchronized void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) + ": " + this);
    }
}
```

Включение синхронизации в решение создает риск того, что синхронизация станет узким местом в эффективности приложения. Такие случаи должны быть достаточно редкими. Прирост производительности от отложенной инициализации встречается только тогда, когда объект редко инициализирует эти поля — ведь если он почти всегда инициализирует эти поля, никакой экономии памяти не будет. Таким образом, синхронизация становится узким местом для полей с отложенной инициализацией, когда редко используемая ветвь кода внезапно начинает использовать множество потоков конкурентно. Ничего невозможного в такой ситуации нет, но и распространенной ее не назовешь.

Проблему с синхронизацией можно решить только в том случае, если переменные с отложенной инициализацией сами являются потоково-безопасными. Объекты `DateFormat` потоково-безопасными не являются, поэтому в текущем примере не так важно, включает ли блокировка объект `Calendar`: если объекты с отложенной инициализацией внезапно начинают интенсивно использоваться, необходимость синхронизации операций с объектом `DateFormat` создаст проблемы в любом случае. Потоково-безопасный код будет выглядеть примерно так:

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
```

```
        unsychronizedCalendarInit();
        synchronized(df) {
            w.write("On " + df.format(calendar.getTime()) + ": " + this);
        }
    }
}
```

Отложенная инициализация с переменной экземпляра, которая не является потоково-безопасной, всегда может синхронизироваться по этой переменной (например, с использованием `synchronized`-версии метода, приведенной выше).

Возьмем другой пример, в котором выполняется отложенная инициализация большой коллекции `ConcurrentHashMap`:

```
public class CHMInitialization {
    private ConcurrentHashMap chm;

    public void doOperation() {
        synchronized(this) {
            if (chm == null) {
                chm = new ConcurrentHashMap();
                ... Код заполнения коллекции...
            }
        }
        ...Использование chm...
    }
}
```

Так как к `ConcurrentHashMap` могут безопасно обращаться сразу несколько потоков, дополнительная синхронизация в этом примере — один из редких случаев, в которых правильно используемая отложенная инициализация может создать узкое место в производительности приложения. (Впрочем, такие узкие места должны встречаться редко: если обращения к коллекции настолько часты, подумайте, добьетесь ли вы какой-либо реальной экономии с отложенной инициализацией.) Проблема решается идиомой блокировки с двойной проверкой:

```
public class CHMInitialization {
    private volatile ConcurrentHashMap instanceChm;

    public void doOperation() {
        ConcurrentHashMap chm = instanceChm;
        if (chm == null) {
            synchronized(this) {
                chm = instanceChm;
                if (chm == null) {
                    chm = new ConcurrentHashMap();
```

```
                    ... Код заполнения коллекции...
                    instanceChm = chm;
                }
            }
            ...Использование chm...
        }
    }
}
```

При этом есть целый ряд важных нюансов, связанных с потоками: переменная экземпляра должна быть объявлена `volatile`, а присваивание переменной экземпляра локальной переменной обеспечивает небольшое повышение эффективности. Более подробная информация приводится в главе 9; в тех редких случаях, в которых отложенная инициализация многопоточного кода имеет смысл, используйте этот паттерн.

Немедленная деинициализация

Естественным следствием отложенной инициализации переменных становится их немедленная деинициализация присваиванием `null`. Это ускоряет освобождение объекта уборщиком мусора. Хотя теоретически все выглядит вполне логично, этот прием приносит пользу лишь в весьма ограниченных обстоятельствах.

Казалось бы, переменная, которая является кандидатом для отложенной инициализации, тоже становится кандидатом для немедленной деинициализации; в предшествующих примерах объектам `Calendar` и `DateFormat` можно было присвоить `null` при завершении метода `report()`. Но если переменная не будет использоваться при последующих вызовах метода (или в других точках класса), изначально нет причин определять ее как переменную экземпляра. Просто создайте локальную переменную в методе; после завершения метода локальная переменная выйдет из области видимости и будет освобождена уборщиком мусора.

Стандартное исключение из правила об избыточности немедленной деинициализации переменных встречается с классами вроде тех, которые входят в библиотеку коллекций Java — классов, которые хранят ссылки на данные в течение долгого времени, а затем получают оповещения о том, что данные стали ненужными. Рассмотрим реализацию метода `remove()` класса JDK `ArrayList` (часть кода упрощена):

```
public E remove(int index) {
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1,
                         elementData, index, numMoved);
    elementData[--size] = null; // Стереть, чтобы GC выполнил свою работу.
    return oldValue;
}
```

Комментарий об уборке мусора (GC) встречается в исходном коде JDK (который обычно комментируется не слишком щедро): такое присваивание переменной `null` выглядит необычно и заслуживает особых пояснений. В данном случае проследите за тем, что произойдет при удалении последнего элемента массива. Количество элементов, оставшихся в массиве, — переменная экземпляра `size` — уменьшается. Допустим, значение переменной `size` уменьшается с 5 до 4. Теперь к данным, хранящимся в `elementData[4]`, обратиться не удастся: они лежат за пределами действительного размера массива.

`elementData[4]` в данном случае является устаревшей (stale) ссылкой. Вероятно, массив `elementData` будет оставаться активным в течение долгого времени, и всему, к чему ему уже не требуется обращаться, можно активно присвоить `null`.

Концепция устаревшей ссылки играет ключевую роль: если класс с долгим сроком жизни кэширует, а затем освобождает ссылки на объекты, необходимо действовать осторожно и избегать устаревших ссылок. В остальных случаях явное присваивание ссылке `null` не принесет сколько-нибудь заметного выигрыша по производительности.

РЕЗЮМЕ

- Используйте отложенную инициализацию только в том случае, когда в типичных ветвях кода переменные часто остаются неинициализированными.
- Отложенная инициализация потоково-безопасного кода нетипична, но часто может опираться на существующую синхронизацию.
- Используйте блокировку с двойной проверкой для отложенной инициализации кода с потоково-безопасными объектами.

Неизменяемые и канонические объекты

В языке Java многие типы объектов являются неизменяемыми. К этой категории относятся объекты, имеющие соответствующий примитивный тип — `Integer`, `Double`, `Boolean` и т. д., а также некоторые типы, основанные на числах (такие, как `BigDecimal`). Конечно, самым распространенным объектом Java является неизменяемый объект `String`. С точки зрения программной структуры часто бывает полезно, если пользовательские классы тоже будут представлять неизменяемые объекты.

Если такие объекты быстро создаются и уничтожаются, они не оказывают особого влияния на уборку мусора в молодом поколении (см. главу 5). Но, как и для любых других объектов, перемещение многих неизменяемых объектов в старое поколение может привести к потере производительности.

Следовательно, нет особых причин для того, чтобы избегать проектирования и использования неизменяемых объектов, даже если на первый взгляд кажется, что, если такие объекты нельзя изменить, а приходится создавать заново, это в чем-то неэффективно. Однако есть одна оптимизация, часто возможная при работе с такими объектами: следует избегать создания одинаковых копий (дубликатов) одного объекта.

Лучшим примером служит класс `Boolean`. Любому приложению Java необходимы всего два экземпляра класса `Boolean`: для `true` и для `false`. К сожалению, класс `Boolean` спроектирован плохо. Так как он имеет открытый конструктор, приложение может создать столько таких объектов, сколько им потребуется, хотя все они ничем не отличаются от двух канонических объектов `Boolean`. В более правильном решении класс `Boolean` имел бы только приватный конструктор и статические методы, возвращающие `Boolean.TRUE` или `Boolean.FALSE` в зависимости от своего параметра. Если вы будете использовать такую модель в своих неизменяемых классах, вы тем самым предотвратите лишнюю нагрузку на использование кучи в ваших приложениях. (Конечно, в идеале объекты `Boolean` создаваться вообще не должны; вы просто используете `Boolean.TRUE` или `Boolean.FALSE` по мере надобности.)

Эти одиночные представления неизменяемых объектов называются *каноническими* версиями объектов.

Создание канонических объектов

Даже если вселенная объектов конкретного класса практически безгранична, использование канонических значений часто позволяет сэкономить память. JDK предоставляет такую возможность для многих популярных неизменяемых объектов: строки могут вызвать метод `intern()` для получения канонической версии строки. Интернирование строк более подробно рассматривается в главе 12; а пока мы посмотрим, как добиться того же эффекта для пользовательских классов.

Чтобы привести объект к каноническому виду, создайте карту для хранения канонических версий. Чтобы предотвратить утечку памяти, проследите за тем, чтобы ссылки на объекты в карте были слабыми. Заготовка такого класса могла бы выглядеть примерно так:

```
public class ImmutableObject {
    private static WeakHashMap<ImmutableObject, ImmutableObject>
        map = new WeakHashMap();

    public ImmutableObject canonicalVersion(ImmutableObject io) {
        synchronized(map) {
```

```
            ImmutableObject canonicalVersion = map.get(io);
            if (canonicalVersion == null) {
                map.put(io, new WeakReference(io));
                canonicalVersion = io;
            }
            return canonicalVersion;
        }
    }
}
```

В многопоточной среде синхронизация может стать узким местом. Если вы стараетесь ограничиваться классами JDK, простого решения нет, потому что эти классы не предоставляют параллелизованной хеш-карты для слабых ссылок. Впрочем, были предложения включить в JDK класс `CustomConcurrentHashMap` — впервые они появились в запросе JSR 166 (Java Specification Request); кроме того, есть различные сторонние реализации такого класса.

РЕЗЮМЕ

- Неизменяемые объекты предоставляют возможность специального управления жизненным циклом — канонизацию.
- Отказ от создания дубликатов неизменяемых объектов посредством канонизации позволяет значительно сократить размер кучи, используемой приложением.

Управление жизненным циклом объекта

Вторая широкая область управления памятью, рассматриваемая в этой главе, — *управление жизненным циклом объекта*. В большинстве случаев Java пытается свести к минимуму те усилия, которые разработчик должен предпринять для управления жизненным циклом объектов: разработчик создает объекты тогда, когда потребуется, а когда надобность в них отпадет — объекты выходят из области видимости и освобождаются уборщиком мусора.

Иногда такой нормальный жизненный цикл не оптимален. Создание некоторых объектов обходится достаточно дорого, а управление жизненным циклом таких объектов повысит эффективность приложения, даже если это потребует дополнительной работы со стороны уборщика мусора. В этом разделе показано, когда и как можно изменять нормальный жизненный цикл объектов — либо за счет повторного использования объектов, либо за счет поддержания специальных ссылок на них.

Повторное использование объектов

Для повторного использования объектов обычно используются два механизма: *пулы объектов* и *потоково-локальные переменные*. Наверное, специалисты по уборке мусора по всему миру сейчас поморщились, потому что любой из этих механизмов снижает эффективность уборки мусора. В частности, пулы объектов в кругах GC пользуются дурной славой, но если на то пошло, пулы объектов также пользуются дурной славой в кругах разработчиков по многим другим причинам.

На одном уровне причина такого отношения кажется очевидной: повторно используемые объекты задерживаются в куче на долгое время. Если в куче находится много объектов, в ней остается меньше места для создания новых объектов, а следовательно, операции уборки мусора происходят с большей частотой. Однако это всего лишь одна часть истории.

Как было показано в главе 6, когда в приложении создается объект, память для него выделяется в Эдеме. Несколько циклов уборки мусора в молодом поколении пройдут за перемещением между областями выживших, прежде чем объект в итоге будет переведен в старое поколение. Каждый раз, когда обрабатывается только что (или недавно) созданный объект из пула, алгоритм уборки мусора должен выполнить некоторую работу для его копирования и корректировки ссылок, пока объект в конечном итоге не будет переведен в старое поколение.

И хотя может показаться, что перевод объекта в старое поколение завершает историю, он может создать еще больше проблем с производительностью. Время, необходимое для выполнения полной уборки мусора, пропорционально количеству объектов, которые продолжают существовать в старом поколении. Объем живых данных еще важнее размера кучи; 3-гигабайтное старое поколение с несколькими выжившими объектами обрабатывается быстрее, чем 1-гигабайтное старое поколение, в котором выживает 75% объектов.

Использование конкурентного уборщика мусора и предотвращение полной уборки мусора не улучшит ситуацию, потому что время, необходимое для фаз пометки конкурентных уборщиков, также зависит от объема все еще живых данных. И особенно для алгоритма CMS объекты в пуле с большой вероятностью будут повышены в разное время, что повысит риск сбоя конкурентного режима из-за фрагментации. В целом можно сделать вывод: чем дольше объекты хранятся в куче, тем менее эффективной становится уборка мусора.

Итак, повторное использование объектов — плохая идея. А теперь можно поговорить о том, когда и как можно повторно использовать объекты.

JDK предоставляет ряд стандартных пулов объектов: пул потоков, который обсуждается в главе 9, и мягкие ссылки. *Мягкие ссылки*, которые рассматриваются позднее в этом разделе, фактически реализуют большой пул повторно

используемых объектов. Серверы Java используют пулы объектов для обработки подключений к базам данных и других ресурсов. Аналогичная ситуация с потоково-локальными значениями; в JDK полно классов, использующих потоковолокальные переменные для предотвращения повторного создания некоторых разновидностей объектов. Очевидно, даже эксперты Java понимают необходимость повторного использования объектов в некоторых обстоятельствах.

ЭФФЕКТИВНОСТЬ УБОРКИ МУСОРА

В какой степени объем живых данных в куче может повлиять на время уборки мусора? Оказывается, иногда объем работы может изменяться на порядок.

Ниже приведен вывод журнала уборки мусора в тесте, выполненном на моей стандартной четырехъядерной машине с системой Linux с 4-гигабайтной кучей (в которой для молодого поколения была выделена область с фиксированным размером 1 Гбайт):

```
[Full GC [PSYoungGen: 786432K->786431K(917504K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3932159K->3932159K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.5432730 secs]
        [Times: user=1.72 sys=0.01, real=0.54 secs]
...
[Full GC [PSYoungGen: 786432K->0K(917504K)]
        [ParOldGen: 3145727K->210K(3145728K)]
        3932159K->210K(4063232K)
        [PSPermGen: 2349K->2349K(21248K)], 0.0687770 secs]
        [Times: user=0.08 sys=0.00, real=0.07 secs]
...
[Full GC [PSYoungGen: 349567K->349567K(699072K)]
        [ParOldGen: 3145727K->3145727K(3145728K)]
        3495295K->3495295K(3844800K)
        [PSPermGen: 2349K->2349K(21248K)], 0.7228880 secs]
        [Times: user=2.41 sys=0.01, real=0.73 secs]
```

Обратите внимание на среднюю часть вывода: приложение очистило большинство ссылок на объекты в старом поколении, а следовательно, размер данных в старом поколении после уборки мусора составлял всего 210 Кбайт. Операция заняла всего 70 мс. В других случаях большая часть данных в куче все еще жива; операции полной уборки данных, удалившие из кучи очень малый объем данных, заняли от 540 до 730 мс. К счастью, в тесте работали четыре потока уборки мусора. В одноядерной системе короткая уборка мусора в этом примере заняла 80 мс, а на длинную уборку мусора потребовалось 2410 мс (превышение более чем в 30 раз).

Основная причина для повторного использования объектов — высокие затраты на инициализацию многих объектов. Выигрыш из-за повторного использования таких объектов превысит потери из-за возрастания времени уборки мусора. Это утверждение безусловно справедливо для таких конструкций, как пул подключений JDBC: создание сетевого подключения, возможно — с передачей регистрационных данных и созданием сеанса базы данных, обходится недешево. Пулы объектов в данном случае обеспечивают заметный прирост производительности. Потоки объединяются в пул для экономии времени, связанного с созданием потоков; генераторы случайных чисел реализуются в виде потоково-локальных переменных для экономии времени, необходимого для раскрутки генераторов; и т. д.

У всех этих примеров есть одна общая особенность: инициализация объекта занимает много времени. В Java *выделение памяти* для объектов обычно происходит быстро и обходится недорого (аргументы против повторного использования объектов обычно ограничиваются этой частью формулы). Производительность *инициализации* объектов зависит от самого объекта. Возможность повторного использования следует рассматривать только для объектов с очень высокими затратами на инициализацию и только в том случае, если затраты на инициализацию таких объектов становятся одной из доминирующих операций в вашей программе.

Есть и другая общая особенность: количество общих объектов невелико, что сводит к минимуму их влияние на операции уборки мусора. Объектов попросту недостаточно, для того чтобы замедлить циклы уборки мусора. Небольшое количество объектов в пуле не окажет заметного влияния на эффективность уборки мусора; заполнение кучи объектами из пула значительно замедлит уборку мусора.

Несколько примеров того, где (и почему) в JDK и в программах Java стоит повторно использовать объекты:

- *Пулы потоков* — инициализация потоков является затратной операцией.
- *Пулы JDBC* — инициализация подключений к базам данных является затратной операцией.
- *Большие массивы* — Java требует, чтобы при выделении памяти для массива все отдельные элементы по умолчанию инициализировались нулевыми значениями (`null`, 0 или `false`). Для больших массивов эта инициализация может быть затратной операцией.
- *Низкоуровневые буферы NIO* — прямое создание `java.nio.Buffer` (буфер, возвращаемый при вызове метода `allocateDirect()`) является затратной операцией независимо от размера буфера. Лучше создать один большой буфер и управлять буферами, выделяя его части и возвращая их обратно для повторного использования в будущих операциях.

- *Классы безопасности* — создание экземпляров `MessageDigest`, `Signature` и других алгоритмов безопасности является затратной операцией.
- *Объекты кодирования и декодирования String* — различные классы JDK создают и повторно используют эти объекты. Большей частью они также используют мягкие ссылки, как будет показано в следующем разделе.
- *Вспомогательные объекты StringBuilder* — класс `BigDecimal` повторно использует объект `StringBuilder` при вычислении промежуточных результатов.
- *Генераторы случайных чисел* — инициализация экземпляров классов `Random` или (особенно) `SecureRandom` является затратной операцией.
- *Имена, полученные в результате поиска DNS*, — сетевые операции получения имен являются затратными.
- *Кодировщики и декодировщики ZIP* — любопытный момент: инициализация этих объектов обходится достаточно дорого. Однако их освобождение тоже обходится достаточно дорого, потому что они зависят от финализации объектов, гарантирующей, что используемая ими низкоуровневая память также будет освобождена. За подробностями обращайтесь к разделу «Финализаторы и финальные ссылки», с. 289.

Два возможных решения (пулы объектов и потоково-локальные переменные) различаются по производительности. Рассмотрим их более подробно.

Пулы объектов

Пулы объектов пользуются дурной репутацией по многим причинам, только часть из которых связана с их производительностью. Разработчику трудно определиться с их размером. Они также возлагают бремя управления объектами на программиста: вместо того, чтобы просто дать возможность объекту выйти из области видимости, программист должен помнить о необходимости вернуть объект в пул.

Однако нас сейчас интересует производительность пула объектов, на которую влияют следующие факторы:

- *Последствия уборки мусора* — как вы уже видели, хранение множества объектов снижает (иногда кардинально) эффективность уборки мусора.
- *Синхронизация* — пулы объектов неизбежно синхронизируются, и если объекты будут часто удаляться и замещаться, в пуле возникнет значительная конкуренция. В результате обращения к пулу будут происходить медленнее, чем инициализация нового объекта.
- *Регулировка* — пулы также могут благотворно влиять на производительность: они могут предоставлять регулируемый доступ к немногочисленным

ресурсам. Как обсуждалось в главе 2, если вы попытаетесь увеличить нагрузку на систему сверх тех пределов, с которыми она может справиться, производительность уменьшится. Это одна из причин, по которым пулы потоков играют важную роль. Если слишком много потоков работают конкурентно, процессоры будут перегружены и производительность упадет (пример рассматривается в главе 9).

Этот принцип также действителен для доступа к удаленным системам; в частности, он часто проявляется с подключениями JDBC. Если количество созданных подключений к базе данных превышает ее возможности, производительность базы данных упадет. В таких ситуациях лучше регулировать количество ресурсов (например, подключений JDBC), ограничивая размер пула — даже если это означает, что потокам в приложении придется дожидаться освобождения ресурса.

Потоково-локальные переменные

Повторное использование переменных за счет хранения их в потоково-локальных переменных также сопряжено с различными компромиссами производительности:

Управление жизненным циклом — управлять потоково-локальными переменными намного проще и дешевле, чем управлять этими объектами в пуле. Оба механизма требуют получения исходного объекта: его нужно извлечь из пула или вызвать метод `get()` для потоково-локального объекта. Но пулы объектов требуют, чтобы вы вернули объект после завершения работы с ним (в противном случае никто не сможет пользоваться им). Потоково-локальные объекты всегда доступны в границах потока, и явно возвращать их не нужно.

Кратность отношения — потоково-локальные переменные обычно обеспечивают однозначное соответствие между количеством потоков и количеством сохраненных (повторно используемых) объектов. На самом деле это не совсем точно. Копия переменной, принадлежащая потоку, не создается до момента ее первого использования потоком, поэтому сохраненных объектов может быть меньше, чем потоков. Но сохраненных объектов не может быть больше, чем потоков, а в большинстве случаев их количество будет одинаковым.

С другой стороны, пул объектов может иметь произвольные размеры. Если запросу в одних случаях необходимо одно подключение JDBC, а в каких-то отдельных случаях могут понадобиться два, пул JDBC может иметь соответствующий размер (скажем, 12 подключений для 8 потоков). Потоково-локальные переменные не позволяют эффективно реализовать эту возможность; кроме того, они не позволяют регулировать доступ к ресурсу (если только само количество потоков не служит регулятором).

Синхронизация — потоково-локальным переменным синхронизация не нужна, потому что они могут использоваться только в пределах одного потока; потоково-локальный метод `get()` относительно быстр. (Так было не всегда; в ранних версиях Java получение потоково-локальной переменной обходилось дорого. Если вы воздерживаетесь от использования потоково-локальных переменных из-за их плохой производительности в прошлом, пересмотрите свое решение для текущих версий Java.)

С синхронизацией связан один интересный момент, потому что выигрыш по производительности для потоково-локальных объектов часто оценивается по экономии затрат на синхронизацию (вместо экономии от повторного использования объекта). Например, Java предоставляет класс `ThreadLocalRandom`; этот класс (вместо отдельного экземпляра `Random`) используется в наших примерах с акциями. В противном случае во многих примерах в книге появилось бы узкое место синхронизации в методе `next()` одного объекта `Random()`. Использование потоково-локального объекта — хороший способ предотвращения узких мест синхронизации, так как такой объект может использоваться только одним потоком.

Эта проблема синхронизации с таким же успехом была бы решена, если бы в примерах просто создавался новый экземпляр класса `Random` каждый раз, когда потребуется. Однако такое решение проблемы синхронизации не лучшим образом отразилось бы на общей производительности: инициализация объекта `Random` обходится дорого, а постоянное создание экземпляров этого класса привело бы к более худшей производительности по сравнению с узким местом синхронизации из-за совместного использования одного экземпляра класса многими потоками. Использование класса `ThreadLocalRandom` обеспечивает более высокую производительность, как видно из табл. 7.3. Этот пример вычисляет время, необходимое для создания 10 000 случайных чисел в каждом из четырех потоков в трех ситуациях:

- Каждый поток создает новый объект `Random` для вычисления 10 000 чисел.
- Все потоки совместно используют один общий статический объект `Random`.
- Все потоки совместно используют один общий статический объект `ThreadLocalRandom`.

Таблица 7.3. Эффект от использования ThreadLocalRandom при генерировании 10 000 случайных чисел

Операция	Затраченное время
Создание нового объекта Random	134,9 ± 0,01 мкс
ThreadLocalRandom	52,0 ± 0,01 мкс
Совместное использование Random	3,763 ± 200 мкс

Микробенчмаркирование потоков, конкурирующих за блокировку, всегда ненадежно. В последней строке таблицы потоки почти всегда конкурируют за блокировку по объекту `Random`; в реальном приложении объем конкуренции будет намного ниже. Но можно ожидать определенной конкуренции с совместно используемым объектом, тогда как постоянные создания нового объекта обходятся вдвое дороже, чем использование объекта `ThreadLocalRandom`.

Вывод в данном случае (и относительно повторного использования объектов вообще): если инициализация объектов занимает много времени, не бойтесь исследовать решения с пулами объектов или потоково-локальными переменными для повторного использования таких объектов, создание которых обходится достаточно дорого. Однако, как обычно, следует искать баланс: большие пулы объектов обобщенных классов почти наверняка создадут больше проблем с производительностью, чем решат. Оставьте эти решения для классов, инициализация которых требует высоких затрат, а количество повторно используемых объектов невелико.

РЕЗЮМЕ

- Повторное использование объектов нежелательно как операция общего назначения, но может хорошо подойти для небольших групп объектов, инициализация которых обходится дорого.
- Повторное использование объекта через пулы объектов или потоково-локальные переменные сопряжено с определенными компромиссами. В общем случае с потоково-локальными переменными проще работать (если предположить, что между потоками и повторно используемыми объектами есть однозначное соответствие).

Мягкие, слабые и другие ссылки

Мягкие (soft) и *слабые* (weak) ссылки в Java также позволяют повторно использовать объекты, хотя разработчики не всегда рассматривают происходящее под этим углом. Такие ссылки — которые мы будем называть *неопределенными* (indefinite) ссылками — чаще используются для кэширования результатов долгих вычислений или поиска в базе данных, нежели для повторного использования простого объекта. Например, в сервере с акциями косвенная ссылка может использоваться для кэширования результата метода `getHistory()` (который подразумевает либо долгие вычисления, либо долгое обращение к базе данных). Результат представляет собой объект, и когда он кэшируется по неопределенной ссылке, вы просто используете объект повторно, потому что в остальных случаях его инициализация сопряжена с высокими затратами.

ЗАМЕЧАНИЕ ПО ПОВОДУ ТЕРМИНОЛОГИИ

Обсуждение мягких и слабых ссылок может показаться достаточно запутанным, потому что в нем используется много похожих терминов. Краткий список терминов:

Ссылка

Под ссылкой (или ссылкой на объект) поднимается любая ссылка: сильная, слабая, мягкая и т. д. Обычная переменная экземпляра, ссылающаяся на объект, является сильной ссылкой.

Неопределенная ссылка

Я использую этот термин для любых разновидностей ссылок (например, мягких или слабых). Неопределенная ссылка по сути представляет экземпляр объекта (например, экземпляр класса `SoftReference`).

Референт

Работа неопределенных ссылок основана на встраивании другой ссылки (почти всегда сильной) в экземпляр класса неопределенной ссылки. Инкапсулированный объект называется референтом.

Однако многие программисты воспринимают эту ситуацию иначе. Этот факт отражен даже в терминологии: никто не говорит о «кэшировании» потока для повторного использования, но мы рассмотрим повторное использование неопределенных ссылок в контексте кэширования результатов операций баз данных.

Преимущество неопределенной ссылки перед пулом объектов или потоково-локальной переменной заключается в том, что неопределенные ссылки будут (в конечном итоге) освобождены уборщиком мусора. Если пул объектов содержит последние 10 000 выполненных запросов по акциям, а память в куче подходит к концу, приложению не повезло: память в куче, оставшаяся после сохранения этих 10 000 элементов, — все пространство в куче, которое может использоваться приложением. Если результаты поиска сохраняются по неопределенным ссылкам, то JVM может освободить некоторую часть памяти (в зависимости от типа ссылки), улучшая эффективность уборки мусора.

С другой стороны, неопределенные ссылки оказывают чуть более заметное влияние на эффективность уборщика мусора. На рис. 7.6 приведено параллельное сравнение использования памяти без неопределенной ссылки и с ней (в данном случае это мягкая ссылка).

Кэшируемый объект занимает 512 байт. Слева изображена вся потребляемая память (без учета памяти переменной экземпляра, указывающей на объект).

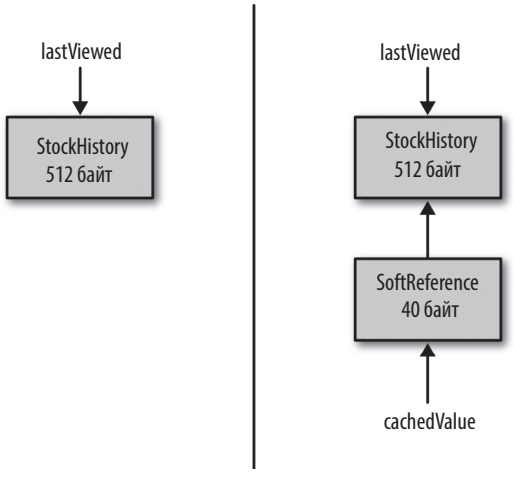


Рис. 7.6. Сравнение гистограмм

В правой части объект кэшируется внутри объекта `SoftReference`, что увеличивает потребление памяти на 40 байт. Неопределенные ссылки ведут себя как любые другие объекты: они потребляют память, а другие сущности (переменная `cachedValue` в правой части диаграммы) содержат сильные ссылки на них.

Таким образом, первое следствие для уборщика мусора заключается в том, что из-за неопределенных ссылок приложение может расходовать больше памяти. Есть и второе, более серьезное следствие: чтобы объект неопределенной ссылки мог быть освобожден уборщиком, потребуется не менее двух циклов уборки мусора.

На рис. 7.7 показано, что происходит, когда на референта нет сильной ссылки (то есть переменной `lastViewed` присвоено значение `null`). Если на объект `StockHistory` не существует ни одной ссылки, он освобождается во время следующей уборки мусора, обрабатывающей поколение, в котором находится объект. Таким образом, левая сторона диаграммы теперь потребляет 0 байт.

В правой части диаграммы память все еще потребляется. Конкретная точка, в которой референт будет освобожден, зависит от типа неопределенной ссылки, но пока рассмотрим случай мягкой ссылки. Референт продолжит существовать, пока JVM не решит, что объект не использовался достаточно давно. Когда это произойдет, первый цикл уборки мусора освободит референта — но не сам объект неопределенной ссылки. Память приложения оказывается в состоянии, изображенном на рис. 7.8.

На сам объект неопределенной ссылки теперь ведут (как минимум) две сильные ссылки: исходная сильная ссылка, созданная приложением, и новая сильная

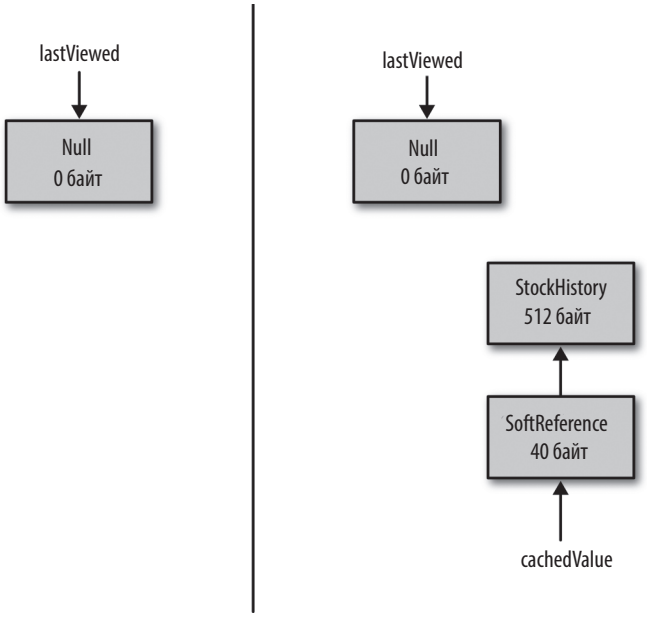


Рис. 7.7. Неопределенные ссылки не очищаются немедленно

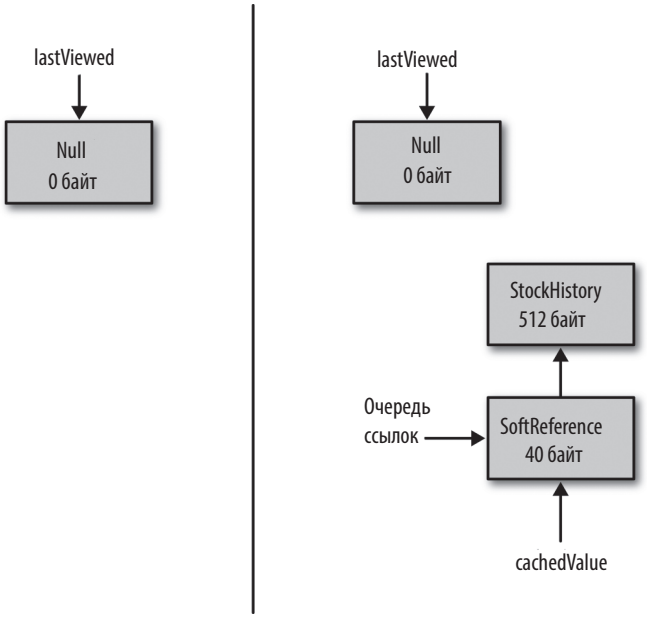


Рис. 7.8. Неопределенные ссылки удерживают память в циклах уборки мусора

ссылка (созданная JVM) в очереди ссылок. Все эти сильные ссылки должны быть очищены, перед тем как сам объект неопределенной ссылки сможет быть освобожден уборщиком мусора.

Как правило, очистка производится кодом, обрабатывающим очередь ссылок. Этот код получает уведомление о появлении нового объекта в очереди и немедленно удаляет все сильные ссылки на этот объект. Затем во время следующего цикла уборки мусора объект неопределенной ссылки (референт) будет освобожден. В худшем случае очередь ссылок не будет обработана немедленно, и перед тем, как все будет очищено, может пройти немало циклов уборки мусора. Впрочем, даже в лучшем случае неопределенной ссылке перед освобождением придется пройти через два цикла уборки мусора.

В зависимости от типа неопределенной ссылки существуют некоторые важные разновидности этого общего алгоритма, но все неопределенные ссылки в той или иной степени страдают от этой нагрузки.

ЖУРНАЛЫ УБОРКИ МУСОРА И ОБРАБОТКА ССЫЛОК

При запуске приложения, использующего большое количество неопределенных ссылок, можно установить флаг `-XX:+PrintReferenceGC` (по умолчанию false). Это позволит вам узнать, сколько времени проводится за обработкой этих ссылок:

```
[GC[SoftReference, 0 refs, 0.0000060 secs]
        [WeakReference, 238425 refs, 0.0236510 secs]
        [FinalReference, 4 refs, 0.0000160 secs]
        [PhantomReference, 0 refs, 0.0000010 secs]
        [JNI Weak Reference, 0.0000020 secs]
        [PSYoungGen: 271630K->17566K(305856K)]
        271630K->17566K(1004928K), 0.0797140 secs]
        [Times: user=0.16 sys=0.01, real=0.08 secs]
```

В данном случае использование 238 425 слабых ссылок добавило 23 мс к уборке в молодом поколении.

Мягкие ссылки

Мягкие ссылки используются тогда, когда у объекта имеется высокая вероятность повторного использования в будущем, но вы хотите разрешить уборщику мусора освободить объект, если он не использовался в последнее время (в вычислениях также учитывается объем доступной памяти в куче). Мягкие ссылки по сути образуют один большой пул наиболее давно используемых (LRU, Least

Recently Used) объектов. Чтобы добиться от них хорошего быстродействия, необходимо позаботиться о том, чтобы эти объекты своевременно очищались.

Рассмотрим пример. Сервер с акциями может создать глобальный кэш истории котировок с доступом по ключу биржевого обозначения (или биржевого обозначения и даты). Когда поступает запрос на выборку акций `TPKS` с 1/9/19 по 31/12/19, можно обратиться к кэшу и проверить, нет ли там результата сходного запроса.

Желательность кэширования этих данных объясняется тем, что запросы на некоторые данные поступают чаще, чем для других. Если данные `TPKS` запрашиваются чаще других, можно ожидать, что они останутся в кэше мягких ссылок. С другой стороны, данные одиночного запроса акций `KENG` некоторое время пробудут в кэше, но в конечном итоге будут освобождены. Такой подход также учитывает распределение запросов по времени: группа запросов акций `DNLD` может повторно использовать результат первого запроса. Когда пользователи поймут, что в `DNLD` деньги вкладывать не стоит, эти кэшированные данные со временем устаревают и покидают кучу.

Когда же именно освобождается мягкая ссылка? Во-первых, на референта не должны существовать сильные ссылки в других местах. Если мягкая ссылка — единственная оставшаяся ссылка на своего референта, то референт будет освобожден при следующем цикле уборки мусора только в том случае, если в последнее время не было обращений по мягкой ссылке. А именно, логика работы функции описывается следующим псевдокодом:

```
long ms = SoftRefLRUPolicyMSPerMB * ОбъемСвободнойПамятиМбайт;
if (now - последнее_обращение_по_ссылке > ms)
   освободить ссылку
```

В этом коде используются два ключевых значения. Первое значение задается флагом `-XX:SoftRefLRUPolicyMSPerMB=`*N*, который по умолчанию равен 1000.

Второе значение — объем свободной памяти в куче (после завершения цикла уборки мусора). Свободная память в куче вычисляется на основании максимального возможного размера кучи за вычетом ее используемой части.

Как это все работает? Возьмем для примера JVM, использующую кучу на 4 Гбайт. После полной уборки мусора (или конкурентного цикла) куча может быть занята на 50%; следовательно, размер свободной кучи составляет 2 Гбайт. Значение `SoftRefLRUPolicyMSPerMB` по умолчанию (1000) означает, что любая мягкая ссылка, которая не использовалась за последние 2048 секунд (2 048 000 мс), будет освобождена; размер свободной кучи 2048 (в мегабайтах) умножается на 1000:

```
long ms = 2048000; // 1000 * 2048
if (System.currentTimeMillis() - последнее_обращение_по_ссылке > ms)
    освободить ссылку
```

Если 4-гигабайтная куча занята на 75%, то объекты, к которым не было обращений за последние 1024 секунды, освобождаются, и т. д.

Чтобы мягкие ссылки освобождались чаще, уменьшите значение флага `SoftRefLRUPolicyMSPerMB`. Присваивание значения 500 означает, что JVM с 4-гигабайтной кучей, заполненной на 75%, будет освобождать объекты, к которым не было обращений за последние 512 секунд.

Настройка этого флага часто становится необходимой, если куча быстро заполняется мягкими ссылками. Допустим, в куче свободно 2 Гбайт, и приложение начинает создавать мягкие ссылки. Если оно создаст мягкие ссылки на 1,7 Гбайт менее чем за 2048 секунд (около 34 минут), ни одна из этих мягких ссылок не будет пригодной для освобождения. В куче останется всего 300 Мбайт для других объектов; в результате уборка мусора будет выполняться достаточно часто (а общая производительность ухудшится).

Если у JVM полностью кончится память или в системе начнется пробуксовка из-за чрезмерной подкачки, JVM очищает все мягкие ссылки, так как единственной альтернативой будет выдача `OutOfMemoryError`. Отказ от выдачи ошибки — хорошее дело, но отбрасывать все кэшированные результаты без разбора, наверное, все же не стоит. Следовательно, если в журналах уборки мусора сообщается о неожиданной очистке слишком большого количества мягких ссылок, стоит подумать о снижении значения `SoftRefLRUPolicyMSPerMB`. Как упоминается в разделе «Превышение порога времени при уборке мусора», с. 260, это происходит только после четырех последовательных циклов полной уборки мусора (и при наличии других факторов).

С другой стороны, долго работающее приложение может рассмотреть возможность повышения этого значения при выполнении двух условий:

- В куче много свободной памяти.
- Обращения по мягким ссылкам встречаются редко.

Это необычная ситуация. Она напоминает ситуацию, которая обсуждалась при описании выбора политик уборки мусора: может показаться, что при увеличении значения политики мягких ссылок (`SoftRefLRUPolicyMSPerMB`) вы сообщаете JVM, что очистка мягких ссылок должна происходить только в крайнем случае. И это действительно так, но вы также приказываете JVM не оставлять запаса для нормальных операций, и в итоге все кончится тем, что за уборкой мусора будет проходить слишком много времени.

Вывод: не стоит использовать слишком много мягких ссылок, потому что они могут легко заполнить всю кучу. И это предупреждение даже еще критичнее, чем предупреждение против создания пула объектов со слишком большим количеством экземпляров: мягкие ссылки хорошо работают, если количество

объектов невелико. В противном случае рассмотрите более традиционный пул объектов с ограниченным размером, реализованный в виде LRU-кэша.

Слабые ссылки

Слабые ссылки следует использовать в том случае, если референт будет использоваться несколькими потоками конкурентно. В противном случае слабая ссылка будет с большой вероятностью освобождена уборщиком мусора: объекты, на которые существуют только слабые ссылки, освобождаются при каждом цикле уборки мусора.

Это означает, что слабые ссылки никогда не попадут в состояние, показанное (для мягких ссылок) на рис. 7.7. При удалении сильных ссылок слабые ссылки немедленно освобождаются. Таким образом, состояние программы переходит непосредственно от рис. 7.6 к рис. 7.8.

Здесь интересно то, где слабые ссылки оказываются в куче. Объекты ссылок ведут себя точно так же, как любые другие объекты Java: они создаются в молодом поколении и со временем переводятся в старое поколение. Если референт слабой ссылки освобождается в то время, когда сама слабая ссылка еще находится в молодом поколении, слабая ссылка будет освобождена быстро (в следующей малой уборке мусора). (Предполагается, что очередь ссылок для объекта, о котором идет речь, обрабатывается быстро.) Если референт продолжает существовать достаточно долго, чтобы слабая ссылка была переведена в старое поколение, слабая ссылка не будет освобождена до следующего конкурентного цикла или цикла полной уборки мусора.

Для примера возьмем сервер с акциями. Допустим, вы знаете, что если конкретный клиент запрашивает данные `TPKS`, он почти всегда обращается к ним повторно. Есть смысл хранить значения для этого вида акций по сильной ссылке, базирующейся на подключении клиента: данные всегда будут наготове, но как только клиент завершит работу в системе, подключение будет сброшено, а память освобождена.

Если другой пользователь запрашивает данные `TPKS`, как он их найдет? Так как объект находится где-то в памяти, проводить его повторную выборку не хотелось бы, но кэш на базе подключения не подойдет второму пользователю. Таким образом, помимо поддержания сильной ссылки на данные `TPKS` на базе подключений, есть смысл поддерживать слабую ссылку на эти данные в глобальном кэше. Теперь второй пользователь сможет найти данные `TPKS` — в предположении, что первый пользователь не закрыл свое подключение. (Этот сценарий используется в разделе «Анализ кучи», с. 248, где на данные существуют две ссылки, и их не удавалось легко найти поиском объектов с наибольшей удерживаемой памятью.)

НЕОПРЕДЕЛЕННЫЕ ССЫЛКИ И КОЛЛЕКЦИИ

Классы коллекций обычно являются источником утечек памяти в Java: приложение помещает объекты (например) в объект `HashMap` и никогда не удаляет их. Со временем `HashMap` увеличивается еще значительнее, расходуя память в куче.

В подобных ситуациях разработчики нередко используют класс коллекции для хранения неопределенных ссылок. JDK предоставляет два таких класса: `WeakHashMap` и `WeakIdentityMap`. Многие сторонние разработчики предлагают классы коллекций, основанные на мягких (и других) ссылках (включая реализации JSR 166 вроде той, которая используется в примерах создания и хранения канонических объектов).

Такие классы удобны, но учтите, что у них есть два недостатка. Во-первых, как обсуждалось в этом разделе, неопределенные ссылки могут отрицательно повлиять на работу уборщика мусора. Во-вторых, сам класс должен периодически выполнять операцию по очистке всех данных в коллекции, на которые нет ссылок (то есть этот класс отвечает за обработку очереди ссылок для неопределенных ссылок, которые в нем хранятся).

Например, класс `WeakHashMap` использует слабые ссылки для своих ключей. Когда ключ со слабой ссылкой становится недоступным, код `WeakHashMap` должен очистить значение в коллекции, ассоциированное с этим ключом. Эта операция выполняется при каждом обращении к коллекции; очередь ссылок для слабого ключа обрабатывается, и значение, ассоциированное с любым ключом в очереди ссылок, удаляется.

Такая схема имеет два последствия для производительности. Во-первых, слабая ссылка и связанное с ней значение не будут реально освобождены до следующего использования коллекции. Следовательно, если коллекция используется нечасто, связанная с ней память будет освобождаться не так быстро, как вам хотелось бы.

Во-вторых, производительность операций с коллекцией непредсказуема. В обычном случае операции с хеш-картой выполняются быстро (собственно, именно поэтому хеш-карты так популярны). Операция с `WeakHashMap`, выполняемая непосредственно после уборки мусора, должна будет обработать очередь ссылок; у операции нет фиксированного короткого времени. Таким образом, даже если ключи освобождаются не так часто, спрогнозировать производительность будет трудно. Что еще хуже, если ключи в карте освобождаются достаточно часто, производительность `WeakHashMap` будет плохой.

Коллекции, основанные на неопределенных ссылках, могут быть полезными, но пользоваться ими следует с осторожностью. Если это возможно, поручите управление коллекцией самому приложению.

Собственно, именно это мы подразумеваем под конкурентным доступом. Все выглядит так, словно мы говорим JVM: «Так, пока кого-то еще интересует этот объект — скажи, где он находится, а когда он станет никому не нужен — уничтожь его, и я сам создам его заново». Сравните с мягкой ссылкой, которая фактически означает: «Постарайся хранить этот объект, пока хватает памяти и пока кто-то к нему время от времени обращается».

Непонимание этого различия — самая частая проблема с производительностью, встречающаяся при использовании слабых ссылок. Не стоит ошибочно полагать, что слабая ссылка — полный аналог мягкой ссылки, не считая того, что она быстрее освобождается: объект с мягкой ссылкой будет доступен (обычно) несколько минут или даже часов, но объект со слабой ссылкой будет доступен только во время существования его референта.

Финализаторы и финальные ссылки

У каждого класса Java есть метод `finalize()`, унаследованный от класса `Object`; этот метод может использоваться для очистки данных, после того как объект станет пригодным для уборки мусора. На первый взгляд это удобно, и данная возможность очень нужна в некоторых обстоятельствах. На практике этого делать не стоит; старайтесь не использовать этот метод, если это только возможно.

Финализаторы настолько плохи, что метод `finalize()` считается устаревшим в JDK 11 (но не в JDK 8). В этом разделе я подробнее расскажу, почему финализаторы нежелательны, но сначала — несколько слов в их пользу. Финализаторы были включены в Java для решения проблем, возникающих при управлении жизненным циклом объектов из JVM. В таких языках, как C++, где разработчик должен явно уничтожать объекты, которые стали ненужными, деконструктор объекта может очистить состояние этого объекта. В языке Java, в котором объект автоматически освобождается при выходе из области видимости, финализатор выполняет функции деконструктора.

Например, JDK использует финализатор в своих классах для работы с ZIP-файлами, потому что при открытии ZIP-файла используется низкоуровневый код, выделяющий низкоуровневую память. Эта память освобождается при закрытии ZIP-файла, но что произойдет, если разработчик забудет вызвать метод `close()`? Финализатор может гарантировать, что метод `close()` будет вызван, даже если разработчик забудет вызвать его.

Многие классы в JDK 8 используют финализаторы подобным образом, но в JDK 11 используется другой механизм: объекты `Cleaner`. Они рассматриваются в следующем разделе. Если вы написали собственный код и у вас появляется искушение воспользоваться финализатором (или вы работаете в JDK 8, где механизм `Cleaner` недоступен), ниже будет рассказано, как с ними справиться.

Финализаторы плохи по соображениям функциональности, и они также плохи для производительности. Финализаторы в действительности являются особым случаем неопределенных ссылок: JVM использует приватный класс ссылки (`java.lang.ref.Finalizer`, который в свою очередь использует `java.lang.ref.FinalReference`) для отслеживания объектов, определивших метод `finalize()`. При создании объекта с методом `finalize()` JVM создает два объекта: сам объект и ссылку `Finalizer`, которая использует объект в качестве референта.

Как и в случае с другими неопределенными ссылками, до освобождения объекта неопределенной ссылки пройдет не менее двух циклов уборки мусора. Однако лишние затраты будут намного больше, чем с другими типами неопределенных ссылок. Когда референт мягкой или слабой ссылки становится пригодным для уборки мусора, он немедленно освобождается; это приводит к распределению памяти, представленному на рис. 7.8. Слабая или мягкая ссылка помещается в очередь ссылок, но объект ссылки более ни на что не ссылается (то есть его метод `get()` возвращает `null` вместо исходного референта). В случае мягких и слабых ссылок двухцикловый штраф уборки мусора применяется только к самому объекту ссылки (но не к референту).

С финальными ссылками дело обстоит иначе. Реализация класса `Finalizer` должна иметь доступ к референту, для того чтобы вызвать метод `finalize()` референта, так что референт не может быть освобожден, когда ссылка на финализатор помещается в очередь ссылок. Когда референт финализатора становится пригодным для уборки мусора, программа находится в состоянии, изображенном на рис. 7.9.

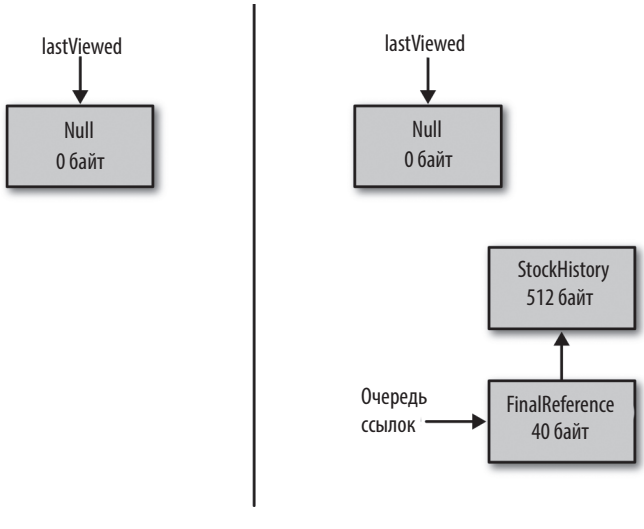


Рис. 7.9. Финальные ссылки удерживают больше памяти

Когда очередь ссылок обрабатывает финализатор, объект `Finalizer` (как обычно) будет удален из очереди и станет пригодным для уборки мусора. Только после этого референт также будет освобожден. По этой причине финализаторы влияют на уборку мусора намного сильнее, чем другие неопределенные ссылки — память, занятая референтом, может быть значительнее памяти, потребляемой объектом неопределенной ссылки.

И здесь мы приходим к функциональной проблеме финализаторов: метод `finalize()` может непреднамеренно создать новую сильную ссылку на референта. И это тоже ухудшает производительность уборки мусора: теперь референт не будет освобожден, до тех пор пока не исчезнет сильная ссылка на него. И с функциональной точки зрения это создает большую проблему, потому что когда референт в следующий раз станет пригодным для уборки мусора, его метод `finalize()` вызван не будет, а ожидаемая очистка референта не произойдет. Подобные ошибки — достаточная причина для того, чтобы финализаторы использовались как можно реже.

Как правило, если вы окажетесь в ситуации, в которой использование финализатора неизбежно, позаботьтесь о том, чтобы память, к которой обращается объект, была сведена к минимуму.

У финализаторов есть альтернатива, которая избегает по крайней мере некоторые из этих проблем — и в частности, позволяет референту освободиться в ходе нормальных операций уборки мусора. Для этого нужно просто использовать другую разновидность неопределенных ссылок вместо неявного использования ссылки `Finalizer`.

Иногда рекомендуется использовать другую разновидность неопределенных ссылок: класс `PhantomReference`. (Именно так действует JDK 11; в этом случае использовать объект `Cleaner` будет намного удобнее, чем приведенный пример, который может быть полезен только в JDK 8.) И это хороший выбор, потому что объект ссылки будет освобожден относительно быстро, после того как на референта не останется сильных ссылок, а в ходе отладки назначение ссылки будет очевидно. Той же цели можно добиться при помощи слабой ссылки (вдобавок слабые ссылки могут использоваться в большем количестве мест). А в некоторых обстоятельствах можно воспользоваться мягкой ссылкой, если семантика кэширования, присущая мягким ссылкам, соответствует потребностям приложения.

Чтобы создать замену для финализатора, необходимо создать подкласс класса неопределенной ссылки для хранения любой информации, которая должна быть освобождена после освобождения референта. Затем необходимая очистка выполняется в методе объекта ссылки (в отличие от определения метода `finalize()` в классе референта).

Ниже приведена заготовка такого класса, в которой используется слабая ссылка. Конструктор выделяет низкоуровневый ресурс. При обычном использовании

должен быть вызван метод `setClosed()`, который освобождает низкоуровневую память.

```
private static class CleanupFinalizer extends WeakReference {
    private static ReferenceQueue<CleanupFinalizer> finRefQueue;
    private static HashSet<CleanupFinalizer> pendingRefs = new HashSet<>();

    private boolean closed = false;

    public CleanupFinalizer(Object o) {
        super(o, finRefQueue);
        allocateNative();
        pendingRefs.add(this);
    }

    public void setClosed() {
        closed = true;
        doNativeCleanup();
    }

    public void cleanup() {
        if (!closed) {
            doNativeCleanup();
        }
    }

    private native void allocateNative();
    private native void doNativeCleanup();
}
```

Слабая ссылка тоже помещается в очередь ссылок. При извлечении ссылки из очереди можно проверить, была ли освобождена низкоуровневая память (и освободить ее, если этого не было сделано).

Обработка очереди ссылок выполняется в потоке-демоне:

```
static {
    finRefQueue = new ReferenceQueue<>();
    Runnable r = new Runnable() {
        public void run() {
            CleanupFinalizer fr;
            while (true) {
                try {
                    fr = (CleanupFinalizer) finRefQueue.remove();
                    fr.cleanup();
                    pendingRefs.remove(fr);
                } catch (Exception ex) {
                    Logger.getLogger(
                           CleanupFinalizer.class.getName()).
                           log(Level.SEVERE, null, ex);
                }
            }
```

```
        }
    };
    Thread t = new Thread(r);
    t.setDaemon(true);
    t.start();
}
```

Все это происходит в приватном статическом внутреннем классе, скрытом от разработчика. Этот класс выглядит так:

```
public class CleanupExample {
    private CleanupFinalizer cf;
    private HashMap data = new HashMap();

    public CleanupExample() {
        cf = new CleanupFinalizer(this);
    }

    ...методы для включения элементов в HashMap...

    public void close() {
        data = null;
        cf.setClosed();
    }
}
```

Разработчик конструирует этот объект точно так же, как любой другой. Ему сообщают, что он должен вызвать метод `close()`, который очистит низкоуровневую память, — но если он этого не сделает, ничего страшного не произойдет. Слабая ссылка незаметно продолжает существовать, так что класс `CleanupFinalizer` имеет возможность очистить эту память при обработке слабой ссылки внутренним классом.

В этом примере есть один нетривиальный момент: необходимость множества слабых ссылок `pendingRefs`. Без этого сами слабые ссылки будут освобождены до того, как появится возможность поместить их в очередь ссылок.

Этот пример справляется с двумя ограничениями традиционного финализатора: он обеспечивает лучшую производительность, потому что память, связанная с объектом референта (хеш-карта `data` в данном случае), будет освобождена сразу же при освобождении референта (а не в методе `finalize()`). Объект референта не удастся «воскресить» в коде очистки, так как он уже был освобожден.

Однако другие возражения, относившиеся к использованию финализаторов, справедливы и для этого кода: вы не можете гарантировать, что уборщик мусора когда-нибудь займется освобождением референта и что поток очереди ссылок когда-нибудь обработает конкретный объект в очереди. Если таких объектов много, обработка очереди ссылок потребует высоких затрат. Это решение, как и все неопределенные ссылки, следует использовать осмотрительно.

ОЧЕРЕДЬ ФИНАЛИЗАТОРОВ

Очередь финализаторов представляет собой очередь ссылок, используемую для обработки ссылок Finalizer, когда референт становится пригодным для уборки мусора.

В ходе анализа дампа кучи часто бывает удобно убедиться в том, что в очереди финализаторов нет объектов: эти объекты все равно скоро будут освобождены, так что исключение их из дампа кучи поможет быстрее понять, что еще происходит в куче. Чтобы запустить обработку очереди финализаторов, введите следующую команду:

```
% jcmd идентификатор_процесса GC.run_finalization
```

Чтобы отслеживать очередь финализаторов и понять, не создает ли она проблем для приложений, проверьте ее размер (который обновляется в реальном времени) на вкладке VM Summary консоли jconsole. В сценариях эту информацию можно получить следующей командой:

```
% jmap -finalizerinfo идентификатор_процесса
```

Объекты Cleaner

В JDK 11 вместо метода `finalize()` гораздо проще использовать новый класс `java.lang.ref.Cleaner`. В этом классе используется класс `PhantomReference` для получения оповещений о том, что на объект не осталось сильных ссылок. Он основан на тех же концепциях, что и класс `CleanupFinalizer`, предложенный мной для использования в JDK 8, но поскольку он входит в число базовых возможностей JDK, разработчику не нужно беспокоиться об обработке потоков и собственных ссылок: они просто регистрируют нужные объекты, которые должны обрабатываться при очистке, а все остальное берут на себя основные библиотеки.

С точки зрения производительности нетривиальным моментом становится регистрация «нужного» объекта в объекте `Cleaner`. Последний хранит сильную ссылку на зарегистрированный объект, так что сам по себе этот объект никогда не станет `PhantomReference`. Вместо этого вы создаете некий «теневой» объект и регистрируете его.

Для примера возьмем класс `java.util.zip.Inflater`. Этому классу необходима завершающая очистка, потому что он должен освободить низкоуровневую память, выделенную в ходе обработки. Код очистки выполняется при вызове метода `end()`, а разработчикам рекомендуется вызвать этот метод при завершении работы с объектом. Но когда объект освобождается, мы должны проследить

за тем, чтобы метод `end()` был вызван; в противном случае возникнет утечка низкоуровневой памяти[1].

В псевдокоде класс `Inflater` выглядит примерно так:

```
public class java.util.zip.Inflater {
    private static class InflaterZStreamRef implements Runnable {
        private long addr;
        private final Cleanable cleanable;
        InflaterZStreamRef(Inflater owner, long addr) {
            this.addr = addr;
            cleanable = CleanerFactory.cleaner().register(owner, this);
        }

        void clean() {
            cleanable.clean();
        }

        private static native void freeNativeMemory(long addr);
        public synchronized void run() {
            freeNativeMemory(addr);
        }
    }

    private InflaterZStreamRef zsRef;

    public Inflater() {
        this.zsRef = new InflaterZStreamRef(this, allocateNativeMemory());
    }

    public void end() {
        synchronized(zsRef) {
            zsRef.clean();
        }
    }
}
```

Этот код проще фактической реализации, которая (по соображениям совместимости) должна отслеживать подклассы, способные переопределить метод `end()`, — и конечно, выделение низкоуровневой памяти тоже происходит сложнее. Здесь следует понимать, что внутренний класс предоставляет объект, на который `Cleaner` может содержать сильную ссылку. Аргумент внешнего класса (`owner`), который также регистрируется с `Cleaner`, предоставляет точку срабатывания: когда он доступен только по ссылке `PhantomReference`, объект `Cleaner`

[1] Если метод end() не будет вызван и вы понадеетесь на то, что механизм уборки мусора очистит низкоуровневую память, все равно возникает видимость утечки низкоуровневой памяти; за дополнительной информацией обращайтесь к главе 8.

активизируется и может использовать сохраненную сильную ссылку для выполнения очистки.

Обратите внимание на то, что внутренний класс объявлен статическим (`static`). В противном случае он бы содержал неявную ссылку на сам класс `Inflater`, и тогда объект `Inflater` никогда не стал бы достижимым только по ссылке `PhantomReference`: всегда существовала бы сильная ссылка из `Cleaner` на объект `InflaterZStreamRef` и сильная ссылка из этого объекта на объект `Inflater`.

Как правило, объект, который будет выполнять очистку, не может содержать ссылку на объект, который должен быть очищен. По этой причине разработчикам не рекомендуется использовать лямбда-выражение вместо класса, так как в лямбда-выражении слишком легко может появиться ссылка на внешний класс.

РЕЗЮМЕ

- Неопределенные (мягкие, слабые, фантомные и финальные) ссылки изменяют обычный жизненный цикл объектов Java, что позволяет повторно использовать их способами, лучше сочетающимися с уборкой мусора, чем пулы или потоково-локальные переменные.
- Слабые ссылки используются в ситуациях, в которых объект представляет интерес для приложения, но только в том случае, если на объект есть сильная ссылка в другой точке приложения.
- Мягкие ссылки удерживают объекты на (возможно) долгие периоды времени, предоставляя простой LRU-кэш, хорошо сочетающийся с уборкой мусора.
- Неопределенные ссылки занимают собственную память и удерживают память других объектов на долгие периоды времени; пользуйтесь ими осмотрительно.
- Финализаторы — особая разновидность ссылок, изначально проектировавшихся для очистки объектов; вместо них следует использовать новый класс `Cleaner`.

Сжатые OOP

При простом программировании 64-разрядные JVM работают медленнее 32-разрядных JVM. Различия в эффективности обусловлены 64-разрядными ссылками на объекты: 64-разрядные ссылки занимают в куче вдвое больше памяти (8 байт), чем 32-разрядные ссылки (4 байта). Это приводит к увеличе-

нию количества циклов уборки мусора, так как в куче остается меньше места для других данных.

JVM может компенсировать дополнительные затраты памяти использованием *сжатых OOP*. Термином «OOP», то есть «указатели на обычные объекты» (Ordinary Object Pointers), обозначаются дескрипторы, используемые JVM в качестве ссылок на объекты. Если длина OOP составляет всего 32 бита, по ним можно обращаться только к 4 Гбайт памяти (2^{32}); именно по этой причине 32-разрядная JVM ограничивается 4-гигабайтным адресным пространством. 64-разрядные OOP способны адресовать эксабайты памяти — намного больше, чем может быть установлено на вашей машине.

Но тут есть промежуточный вариант: что, если бы существовали 35-разрядные OOP? Тогда указатель позволял бы адресовать 32 Гбайт памяти (2^{35}), и при этом занимал бы в куче куда меньше места, чем 64-разрядные ссылки. Проблема в том, что 35-разрядных регистров для хранения таких ссылок нет. С другой стороны, JVM может считать, что последние 3 бита ссылки всегда равны 0. Тогда каждую ссылку можно будет хранить в 32 разрядах кучи. При хранении ссылки в 64-разрядном регистре JVM может сдвинуть ее влево на 3 разряда (добавить три нуля в конце). При сохранении ссылки из регистра JVM сдвигает ее вправо на 3 разряда, отбрасывая завершающие нули.

Так в распоряжении JVM появляются указатели, способные адресовать 32 Гбайт памяти и при этом занимающие только 32 бита в куче. Но это также означает, что JVM не может обратиться к объекту по адресу, который не кратен 8, потому что любой адрес из сжатого OOP кончается тремя нулями. Первый возможный OOP — 0x1 — при сдвиге превращается в 0x8. Второй OOP — 0x2 — при сдвиге превращается в 0x10 (16). Следовательно, объекты в памяти должны начинаться с 8-байтовой границы.

Как выясняется, объекты в JVM уже выровнены по 8-байтовой границе; для большинства процессоров такой способ выравнивания оптимален. Таким образом, при использовании сжатых OOP ничего не теряется. Если первый объект в JVM хранится по адресу 0 и занимает 57 байт, следующий объект будет храниться с 64-го байта — при этом теряются 7 байт, в которых невозможно создавать объекты. Эта потеря памяти приемлема (и неизбежна независимо от того, используются сжатые OOP или нет), потому что 8-байтовое выравнивание ускоряет обращение к объектам.

Но есть причина, по которой JVM не пытается эмулировать 36-разрядные ссылки для адресации 64 Гбайт памяти. В этом случае объекты пришлось бы выравнивать по границе 16 байт, а экономия от хранения сжатого указателя в куче была потеряна из-за неиспользуемых участков между объектами, выровненными в памяти.

Отсюда два важных вывода. Во-первых, для куч, размер которых лежит в диапазоне от 4 Гбайт до 32 Гбайт, используйте сжатые OOP. Сжатие OOP включается флагом `-XX:+UseCompressedOops`; оно включено по умолчанию в том случае, если максимальный размер кучи менее 32 Гбайт. (В разделе «Уменьшение размеров объектов», с. 262, упоминалось о том, что размер ссылки на объект в 64-разрядной JVM с кучей 32 Гбайт составляет 4 байта — это стандартный случай, так как сжатые OOP включаются по умолчанию.)

Во-вторых, программы с кучей 31 Гбайт и сжатыми OOP обычно работают быстрее программ, использующих кучу 33 Гбайт. Хотя во втором случае куча больше, лишняя память, используемая указателями в куче, означает, что большая куча будет выполнять циклы уборки мусора с большей частотой, а ее производительность будет ниже.

Следовательно, лучше использовать кучи, меньшие 32 Гбайт, или кучи, которые по крайней мере на несколько гигабайт больше 32 Гбайт. После добавления в кучу дополнительной памяти, компенсирующей затраты памяти на несжатые ссылки, число циклов уборки мусора уменьшится. Нет четких правил, определяющих объем памяти, необходимой для компенсации влияния несжатых OOP на уборку мусора, — но если учесть, что 20% кучи в среднем может использоваться для ссылок на объекты, планирование стоит начинать как минимум с 38 Гбайт.

РЕЗЮМЕ

- Сжатые OOP включаются по умолчанию тогда, когда они могут принести наибольшую пользу.
- Куча на 31 Гбайт, использующая сжатые OOP, часто превосходит по производительности кучи большего размера, слишком большие для использования сжатых OOP.

Итоги

Скорость программ Java критически зависит от управления памятью. Настройка уборщика мусора важна, но для достижения максимальной производительности память должна эффективно использоваться в приложениях.

Какое-то время оборудование развивалось такими темпами, что разработчики отвыкали думать об оборудовании: если на моем ноутбуке установлено 16 Гбайт памяти, стоит ли беспокоиться об объекте с лишней неиспользуемой 8-байтовой ссылкой? В облачном мире контейнеров, которым выделяется ограниченный объем памяти, эта проблема снова становится очевидной. Однако даже при выполнении приложений с большими кучами на мощном оборудовании, легко

забыть о том, что привычный компромисс между затратами памяти и времени может превратиться в компромисс между затратами времени и затратами памяти/времени: использование слишком большого объема памяти в куче может замедлить работу программы, так как ей потребуется больше уборок мусора. В Java всегда важно управлять состоянием кучи.

Это управление большей частью связано с тем, когда и как следует использовать специальные средства работы с памятью: пулы объектов, потоково-локальные переменные и неопределенные ссылки. Разумное использование этих средств может радикально улучшить производительность приложения, но злоупотребления с таким же успехом приведут к потере производительности. При небольшом количестве объектов эти средства работы с памятью могут быть достаточно эффективными.

ГЛАВА 8

Практика работы с низкоуровневой памятью

Куча обеспечивает наибольшее потребление памяти в приложениях Java, но JVM выделяет и использует большой объем низкоуровневой памяти. И хотя в главе 7 рассматривались возможности эффективного управления кучей с программной точки зрения, конфигурация кучи и ее взаимодействие с низкоуровневой памятью операционной системы становятся еще одной важной составляющей общей производительности приложения. Здесь возникает терминологический конфликт, потому что программисты C обычно называют части низкоуровневой памяти, с которыми они работают, «кучей C». Придерживаясь Java-центрического мировоззрения, мы будем использовать термин «куча» для обозначения кучи Java, а термин «низкоуровневая память» — для обозначения прочей памяти JVM, включая кучу C.

В этой главе рассматриваются эти аспекты низкоуровневой памяти (или памяти операционной системы). Обсуждение начинается с использования всей памяти JVM; особое внимание будет уделено мониторингу использования памяти для выявления проблем производительности. Затем будут рассмотрены различные способы настройки JVM и операционной системы для оптимального использования памяти.

Потребление памяти

Куча (обычно) включает наибольший объем памяти, используемой JVM, но JVM также использует память для своих внутренних операций. Память, не входящая в кучу, относится к *низкоуровневой* памяти. Низкоуровневая память также может выделяться в приложениях (JNI-вызовами `malloc()` и других аналогичных методов, или при использовании NIO (New I/O)). Суммарный объем

низкоуровневой памяти и памяти кучи, используемой JVM, определяет *общее потребление памяти* (footprint) приложения.

С точки зрения операционной системы общее потребление памяти является ключевым фактором производительности. Если в системе недостаточно физической памяти для размещения общего потребления памяти приложения, производительность может пострадать. Здесь стоит обратить внимание на слово «может». Некоторые части низкоуровневой памяти используются только при запуске (например, память, связанная с загрузкой JAR-файлов), и если эта память будет выгружена, это может остаться незамеченным. Часть низкоуровневой памяти, используемая одним процессом Java, используется совместно с другими процессами Java в системе, а некоторые меньшие части могут использоваться совместно с другими видами процессов в системе. Впрочем, для достижения оптимальной производительности обычно необходимо убедиться в том, что общее потребление памяти всех процессов Java не превышает объема физической памяти машины (к тому же часть памяти необходимо оставить для других приложений).

Измерение потребления памяти

Для оценки общего потребления памяти процессом необходимо воспользоваться соответствующими программами операционной системы. В системах семейства Unix такие программы, как `top` и `ps`, выводят эту информацию на простейшем уровне; в Windows можно воспользоваться программой `perfmon` или `VMMap`. Но независимо от программы и платформы необходимо смотреть на фактически выделенную память процесса (в отличие от зарезервированной).

Различия между выделенной и зарезервированной памятью обусловлены тем, как JVM (и все программы) управляют памятью. Возьмем кучу, определяемую параметрами `-Xms512m -Xmx2048m`. Куча в исходном состоянии имеет размер 512 Мбайт и изменяется по мере необходимости для соблюдения целей приложения по уборке мусора.

В этой концепции проявляется принципиальное различие между выделенной (или *закрепленной*) и зарезервированной памятью (иногда называемой *виртуальным размером* процесса). JVM должна сообщить операционной системе, что ей может понадобиться до 2 Гбайт памяти в куче, поэтому эта память *резервируется*: операционная система гарантирует, что если JVM пытается выделить дополнительную память при расширении кучи, эта память будет доступной.

Только 512 Мбайт этой памяти выделены в исходном состоянии, и эти 512 Мбайт — это вся используемая память (для кучи). Эта (фактически выделенная) память называется *закрепленной* (committed). Объем закрепленной памяти колеблется с изменением размера кучи; в частности, с расширением кучи закрепленная память увеличивается соответствующим образом.

СОЗДАЕТ ЛИ ПРОБЛЕМЫ ЧРЕЗМЕРНОЕ РЕЗЕРВИРОВАНИЕ?

Когда речь заходит о производительности, важна только закрепленная память: проблемы с производительностью никогда не возникают из-за того, что программа зарезервировала слишком много памяти.

Но иногда бывает нужно проследить за тем, чтобы JVM не резервировала слишком много памяти. Это особенно важно для 32-разрядных JVM. Поскольку максимальный размер процесса 32-разрядного приложения составляет 4 Гбайт (или меньше в зависимости от операционной системы), чрезмерное резервирование памяти может создать проблемы. У JVM, резервирующей 3,5 Гбайт памяти для кучи, остается всего 0,5 Гбайт низкоуровневой памяти для стеков, кэша команд и т. д. Неважно, если куча в итоге займет всего 1 Гбайт памяти: из-за резервирования 3,5 Гбайт объем памяти для других операций ограничивается 0,5 Гбайт.

64-разрядные JVM не сталкиваются с такими ограничениями из-за размеров процесса, но они ограничиваются общим объемом виртуальной памяти на машине. Допустим, у вас имеется небольшой сервер с 4 Гбайт физической памяти и 10 Гбайт виртуальной памяти и вы запускаете на нем JVM с максимальным размером кучи 6 Гбайт. В такой конфигурации будет зарезервировано 6 Гбайт виртуальной памяти (и еще для разделов памяти, не входящих в кучу). Независимо от того, до какого размера вырастет куча (и закрепленной памяти), вторая JVM сможет зарезервировать только менее 4 Гбайт памяти на этой машине.

При прочих равных условиях удобно завышать размер структур JVM и позволить JVM оптимально использовать эту память. Однако это не всегда возможно.

Это отличие относится практически ко всей значительной памяти, выделяемой JVM. Кэш команд увеличивается от исходного размера до максимального по мере компиляции дополнительного кода. Метапространство выделяется отдельно от кучи и растет от исходного (закрепленного) размера до максимального (зарезервированного).

Стеки потоков составляют исключение из этого правила. Каждый раз, когда JVM создает поток, ОС выделяет немного низкоуровневой памяти для хранения стека этого потока, закрепляя при этом больше памяти (по крайней мере до завершения потока). Впрочем, память стеков потоков полностью выделяется при их создании.

В системах Unix общее потребление памяти приложения можно оценить по размеру резидентного набора (RSS) процесса, который выводится различными

программами ОС. Это значение является хорошей оценкой объема закрепленной памяти, используемой процессом, хотя оно и неточно в двух отношениях. Во-первых, несколько страниц, совместно используемых на уровне ОС между JVM и других процессов (текстовые части общих библиотек), учитываются в RSS каждого процесса. Во-вторых, процесс мог закрепить больше памяти, чем он подгружал в любой момент времени. Но при этом значение RSS процесса станет хорошим предварительным индикатором для оценки общих затрат памяти. В более новых ядрах Linux значение PSS представляет собой уточненную версию RSS, из которой исключаются данные, используемые совместно с другими программами.

В системах Windows эквивалентная концепция называется *рабочим набором* приложения. Именно это значение выводится в Диспетчере задач.

Минимизация потребления памяти

Чтобы минимизировать потребление памяти JVM, ограничьте объем используемой памяти.

Куча — самый большой блок памяти, хотя, как ни странно, он может занимать от 50% до 60% общего потребления памяти. Использование меньшего размера максимальной кучи (или выбор параметров настройки уборки мусора, при которых куча не увеличивается до полного размера) ограничивает потребление памяти программой.

Стеки потоков — стеки потоков довольно велики, особенно для 64-разрядных JVM. В главе 9 описаны возможности ограничения памяти, потребляемой стеками потоков.

Кэш команд — кэш команд использует низкоуровневую память для хранения откомпилированного кода. Как упоминалось в главе 4, его параметры можно настроить (хотя если не весь код удастся откомпилировать из-за ограничений памяти, производительность программы ухудшится).

Выделение памяти низкоуровневыми библиотеками — низкоуровневые библиотеки могут выделять собственную память, которая иногда может быть довольно значительной.

В нескольких следующих разделах речь пойдет о том, как контролировать и сокращать размеры этих областей.

РЕЗЮМЕ

- Общее потребление памяти JVM оказывает серьезное влияние на производительность, особенно если физическая память на машине ограничена. Общее потребление памяти — еще один аспект производительности, который обычно отслеживается в ходе тестов.

Контроль за низкоуровневой памятью

JVM предоставляет неполную информацию о выделении низкоуровневой памяти. Важно понимать, что получаемая информация относится к памяти, выделяемой самой JVM, но не включает память, выделяемую низкоуровневыми библиотеками, которые используются приложением. К этой категории относятся как сторонние низкоуровневые библиотеки, так и низкоуровневые библиотеки (например, `libsocket.so`), входящие в поставку JDK.

Флаг `-XX:NativeMemoryTracking=off|summary|detail` включает вывод этой информации. По умолчанию механизм Native Memory Tracking (NMT) отключен. Если режим сводного или подробного контроля включен, вы можете получить информацию об использовании низкоуровневой памяти командой `jcmd`:

```
% jcmd process_id VM.native_memory summary
```

Если JVM запускается с аргументом `-XX:+PrintNMTStatistics` (по умолчанию `false`), то JVM выведет информацию о выделенной памяти при завершении программы.

Ниже приведен сводный вывод JVM, работающей с исходным размером кучи 512 Мбайт и максимальным размером кучи 4 Гбайт:

```
Native Memory Tracking:

 Total: reserved=5947420KB, committed=620432KB
```

И хотя JVM зарезервировала память на 5,9 Гбайт, реально было использовано гораздо меньше памяти: всего 620 Мбайт. Такая картина довольно типична (это одна из причин, по которым не стоит обращать особого внимания на виртуальный размер процесса, выводимый средствами ОС, так как он отражает только зарезервированную память).

Разберемся со структурой этого использования памяти. Куча (как и следовало ожидать) занимает самую большую часть зарезервированной памяти (4 Гбайт). Однако динамическое изменение размера кучи означает, что она выросла только до 268 Мбайт (в данном случае размер кучи определялся параметрами `-Xms256m -Xmx4g`, так что фактическое использование кучи увеличилось на небольшую величину):

```
-                 Java Heap (reserved=4194304KB, committed=268288KB)
                            (mmap: reserved=4194304KB, committed=268288KB)
```

Затем идет низкоуровневая память, используемая для хранения метаданных класса. И снова обратите внимание на то, что JVM резервирует больше памяти, чем использовалось для хранения 24 316 классов в программе. Закрепленный

размер начинается со значения флага `MetaspaceSize` и увеличивается по мере необходимости, пока не достигнет значения флага `MaxMetaspaceSize`:

```
-                     Class (reserved=1182305KB, committed=150497KB)
                            (classes #24316)
                            (malloc=2657KB #35368)
                            (mmap: reserved=1179648KB, committed=147840KB)
```

Была выделена память для 77 стеков потоков приблизительно по 1 Мбайт каждый:

```
-                    Thread (reserved=84455KB, committed=84455KB)
                            (thread #77)
                            (stack: reserved=79156KB, committed=79156KB)
                            (malloc=243KB, #314)
                            (arena=5056KB, #154)
```

Затем идет кэш команд JIT: 24 316 классов — не так много, поэтому закреплена лишь небольшая часть кэша команд:

```
-                      Code (reserved=102581KB, committed=15221KB)
                            (malloc=2741KB, #4520)
                            (mmap: reserved=99840KB, committed=12480KB)
```

Далее идет область за пределами кучи, которая используется алгоритмом уборки мусора для обработки. Размер этой области зависит от используемого алгоритма уборки мусора: (простой) последовательный уборщик резервирует куда меньше памяти, чем более сложный алгоритм G1 (хотя в общем случае эта величина никогда не будет слишком большой):

```
-                         GC (reserved=199509KB, committed=53817KB)
                            (malloc=11093KB #18170)
                            (mmap: reserved=188416KB, committed=42724KB)
```

Аналогичным образом следующая область используется компилятором для своих операций, не считая кода, помещенного в кэш команд:

```
-                   Compiler (reserved=162KB, committed=162KB)
                             (malloc=63KB, #229)
                             (arena=99KB, #3)
```

В этой области представлены внутренние операции JVM. Большинство из них имеет малый размер, но одно важное исключение составляют прямые байтовые буферы, которые выделяются здесь:

```
-                   Internal (reserved=10584KB, committed=10584KB)
                             (malloc=10552KB #32851)
                             (mmap: reserved=32KB, committed=32KB)
```

Ссылки из таблицы символических имен (константы из файлов классов) хранятся здесь:

```
-                     Symbol (reserved=12093KB, committed=12093KB)
                             (malloc=10039KB, #110773)
                             (arena=2054KB, #1)
```

Механизму NMT также необходима память для работы (это одна из причин, по которым он не включается по умолчанию):

```
-    Native Memory Tracking (reserved=7195KB, committed=7195KB)
                            (malloc=16KB #199)
                            (tracking overhead=7179KB)
```

Наконец, остаются некоторые разделы JVM для хранения второстепенной учетной информации:

```
-               Arena Chunk (reserved=188KB, committed=188KB)
                            (malloc=188KB)
-                   Unknown (reserved=8192KB, committed=0KB)
                            (mmap: reserved=8192KB, committed=0KB)
```

ПОДРОБНАЯ ИНФОРМАЦИЯ ОБ ИСПОЛЬЗОВАНИИ ПАМЯТИ

Если JVM запускается с флагом `-XX:NativeMemoryTracking=detail`, программа jcmd (с завершающим аргументом `detail`) предоставит подробную информацию о выделении низкоуровневой памяти. Она включает в себя карту всего пространства памяти, которая содержит строки следующего вида:

```
0x00000006c0000000 - 0x00000007c0000000] reserved 4194304KB for Java Heap
        from [ReservedSpace::initialize(unsigned long, unsigned long,
                             bool, char*, unsigned long, bool)+0xc2]
        [0x00000006c0000000 - 0x00000006fb100000] committed 967680KB
            from [PSVirtualSpace::expand_by(unsigned long)+0x53]
        [0x000000076ab00000 - 0x00000007c0000000] committed 1397760KB
            from [PSVirtualSpace::expand_by(unsigned long)+0x53]
```

4 Гбайт памяти кучи были зарезервированы в функции `initialize()`, при этом два блока памяти из нее были выделены в функции `expand_by()`.

Такая информация повторяется для всего пространства процесса. Технические специалисты по JVM найдут в ней немало интересного, но для нас хватит и сводной информации.

NMT предоставляет два ключевых значения:

Общий размер закрепленной памяти

Общий размер закрепленной памяти JVM (в идеале) близок к объему физической памяти, которую может потреблять процесс. В свою очередь, последний должен быть близок к RSS (или рабочему набору) приложения, но эти метрики, предоставляемые ОС, не включают память, которая была закреплена, но выгружена из процесса. Собственно, если RSS процесса меньше объема закрепленной памяти, это часто говорит о том, что ОС сталкивается с трудностями при размещении JVM в физической памяти.

Отдельные размеры закрепленной памяти

Когда наступает момент настройки максимальных значений — кучи, кэша команд и метапространства, — будет полезно знать, какую часть этой памяти использует JVM. Чрезмерное выделение памяти для этих областей обычно приводит к безвредному резервированию памяти, хотя, если зарезервированная память важна, NMT поможет отследить, где можно сократить эти максимальные размеры.

С другой стороны, как я упоминал в начале этого раздела, NMT не выдает информации об использовании низкоуровневой памяти совместными библиотеками, так что в некоторых случаях общий размер процесса будет больше закрепленного размера структур данных JVM.

NMT во времени

NMT также позволяет следить за выделением памяти с течением времени. После запуска JVM с включенным механизмом NMT можно установить базовую линию использования памяти следующей командой:

```
% jcmd идентификатор_процесса VM.native_memory baseline
```

Команда заставляет JVM пометить текущие выделенные блоки памяти. Позднее вы можете сравнить текущее использование памяти с этой пометкой:

```
% jcmd идентификатор_процесса VM.native_memory summary.diff
Native Memory Tracking:

Total:  reserved=5896078KB  -3655KB, committed=2358357KB -448047KB
-             Java Heap (reserved=4194304KB, committed=1920512KB -444927KB)
                        (mmap: reserved=4194304KB, committed=1920512KB
-444927KB)
....
```

В данном случае JVM зарезервировала 5,8 Гбайт памяти, и в настоящее время использует 2,3 Гбайт. Закрепленный размер на 448 Мбайт меньше установлен-

ной базовой линии. Аналогичным образом, закрепленная память, используемая кучей, уменьшилась на 444 Мбайт (также можно проанализировать остальной вывод и найти, где еще используется сокращенная память, чтобы разыскать недостающие 4 Мбайт).

Этот прием может пригодиться для анализа потребления памяти JVM со временем.

АВТОМАТИЧЕСКОЕ ОТКЛЮЧЕНИЕ NMT

В выводе NMT было показано, что сам механизм NMT требует низкоуровневой памяти. Кроме того, при включении NMT будут созданы фоновые потоки, которые помогают в процессе отслеживания состояния памяти.

Если у JVM возникнет серьезная нехватка памяти или ресурсов процессора, NMT автоматически отключится для экономии ресурсов. Обычно это правильное поведение — если только вы не пытаетесь провести диагностику именно в такой ситуации. Чтобы приказать NMT продолжить работу в такой ситуации, сбросьте флаг `-XX:-AutoShutdownNMT` (по умолчанию true).

РЕЗЮМЕ

- Механизм NMT (Native Memory Tracking) предоставляет подробную информацию об использовании низкоуровневой памяти JVM. С точки зрения операционной системы эта память включает кучу JVM (которая для ОС является простым разделом низкоуровневой памяти).
- Сводный режим NMT достаточен для большей части аналитических задач. Он позволяет определить, сколько памяти закрепила JVM (и для чего используется эта память).

Низкоуровневая память общих библиотек

С точки зрения архитектуры NMT является частью HotSpot: ядра C++, выполняющего байт-код Java вашего приложения. HotSpot располагается ниже уровня самого JDK и поэтому не отслеживает выделения памяти чем-либо, находящимся на уровне JDK. Эта память выделяется общими библиотеками (загружаемыми вызовом `System.loadLibrary()`).

Общие библиотеки часто рассматриваются как сторонние расширения Java: например, Oracle WebLogic Server использует ряд низкоуровневых библиотек

для выполнения ввода/вывода с большей эффективностью, чем у JDK[1]. Однако и сам JDK использует несколько низкоуровневых библиотек; как и все общие библиотеки, они лежат за пределами видимости NMT.

Таким образом, низкоуровневые утечки памяти — при которых RSS или рабочий набор приложения непрерывно возрастают со временем — обычно не обнаруживаются NMT. Все пулы памяти, которые отслеживает NMT, обычно имеют верхнюю границу (например, максимальный размер кучи). NMT может сообщить нам, какой из пулов использует много памяти (а следовательно, какой следует настроить для использования меньшего объема памяти), но неограниченная утечка памяти в приложении обычно происходит из-за проблем с низкоуровневой библиотекой.

Никакие средства уровня Java не помогут определить, где приложение использует низкоуровневую память из общих библиотек. Средства уровня ОС позволяют определить, что рабочий набор процесса постоянно растет, и если процесс продолжает расти до размера рабочего набора 10 Гбайт, а NMT сообщает, что JVM закрепила только 6 Гбайт памяти, мы знаем, что оставшиеся 4 Гбайт памяти должны происходить от выделения памяти низкоуровневыми библиотеками.

Для определения того, какая из низкоуровневых библиотек отвечает за утечку памяти, потребуются средства уровня ОС, а не средства JDK. Различные отладочные версии `malloc` могут использоваться для этой цели. Они полезны до определенной степени, хотя часто низкоуровневая память выделяется вызовом `mmap`, а эти вызовы игнорируются многими библиотеками, отслеживающими вызовы `malloc`.

Хорошей альтернативой служит профилировщик, который может осуществлять профилирование как низкоуровневого кода, так и кода Java. Например, в главе 3 упоминался Oracle Studio Profiler — многоязыковой профилировщик. Он также предоставляет возможность отслеживания операций выделения памяти — правда, он способен отслеживать выделение памяти низкоуровневым кодом, но не кодом Java, но нам именно это и нужно.

На рис. 8.1 изображено представление низкоуровневого выделения памяти в Studio Profiler.

Из графа вызовов видно, что низкоуровневая функция WebLogic `mapFile` использовала `mmap` для выделения около 150 Гбайт низкоуровневой памяти в нашем процессе. Эта информация не совсем точна: есть несколько отображений для этого файла, а профилировщик недостаточно сообразителен, чтобы понять, чтобы все они совместно используют одну память: если бы, например, существовали 100 отображений этого 15-гигабайтного файла, то использование

[1] В основном это исторический артефакт: эти библиотеки разрабатывались до появления NIO и в основном повторяют его функциональность.

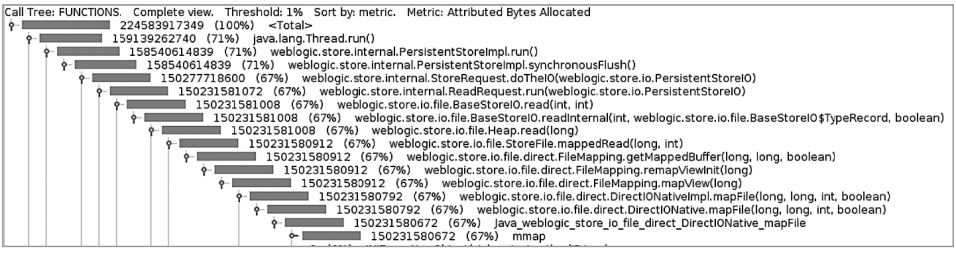


Рис. 8.1. Профилирование низкоуровневой памяти

памяти увеличилось бы только на 15 Гбайт. (Откровенно говоря, я намеренно подправил этот файл до такого размера; пример ни в коем случае не типичен для реального использования.) Однако низкоуровневый профилировщик указал на источник проблемы.

Внутри самого JDK две распространенные операции могут привести к использованию больших объемов низкоуровневой памяти: использование объектов `Inflater`/`Deflater` и использование буферов NIO. Даже без профилирования можно обнаружить, приводят ли эти операции к росту низкоуровневой памяти.

Низкоуровневая память и Inflater/Deflater

Классы `Inflater` и `Deflater` выполняют различные виды сжатия: zip, gzip и т. д. Они могут использоваться напрямую или опосредованно через разные входные потоки. Эти разные алгоритмы используют платформенно-зависимые низкоуровневые библиотеки для выполнения своих операций. Такие библиотеки могут выделять значительный объем низкоуровневой памяти.

При использовании одного из этих классов вы должны — как сказано в документации — вызвать метод `end()` после завершения операции. Среди прочего, вызов освобождает низкоуровневую память, используемую объектом. Если вы используете поток, вы должны закрыть его (а класс потока вызовет метод `end()` для своего внутреннего объекта).

Даже если вы забудете вызвать метод `end()`, еще не все потеряно. Вспомните, о чем говорилось в главе 7: у всех объектов имеется специальный механизм завершения именно для таких ситуаций: метод `finalize()` (в JDK 8) или метод `Cleaner`, связанный с объектом (в JDK 11), может вызвать метод `end()` при освобождении объекта `Inflater`. Следовательно, утечки низкоуровневой памяти здесь не возникнет; со временем объекты будут освобождены и пройдут финальную очистку, а низкоуровневая память будет освобождена.

Но это может занять много времени. Размер объекта `Inflater` относительно мал, и в приложении с большой кучей, которое редко выполняет полную уборку

мусора, такие объекты легко могут перейти в старое поколение и оставаться в нем часами. Таким образом, хотя формально здесь утечки нет (низкоуровневая память будет в конечном итоге освобождена при выполнении полной уборки мусора), пропущенный вызов `end()` может обладать всеми признаками утечки низкоуровневой памяти.

Если на то пошло и сам объект `Inflater` создает утечку в коде Java, то низкоуровневая память будет действительно пропадать.

Если в системе имеется значительная утечка низкоуровневой памяти, может быть полезно сохранить дамп кучи приложения и поискать в нем объекты `Inflater` и `Deflater`. Скорее всего, эти объекты не создадут проблем в самой куче (они для этого слишком малы), но большое их количество укажет на значительные затраты низкоуровневой памяти.

Низкоуровневые буферы NIO

Байтовые буферы NIO выделяют низкоуровневую (не входящую в кучу) память, если они создаются методом `allocateDirect()` класса `ByteBuffer` или методом `map()` класса `FileChannel`.

Низкоуровневые байтовые буферы важны с точки зрения производительности, так как они позволяют низкоуровневому коду и коду Java обмениваться данными без их копирования. Наиболее типичным примером служат буферы, используемые для операций файловой системы и сокетов. Запись данных в низкоуровневый буфер NIO и последующая отправка данных в канал (файл или сокет) не требуют копирования данных между JVM и библиотекой C, используемой для передачи данных. Если вместо этого будет использоваться байтовый буфер в куче, то содержимое буфера должно быть скопировано JVM.

Вызов метода `allocateDirect()` обходится достаточно дорого; прямые байтовые буферы следует использовать повторно по мере возможности. Идеальная ситуация встречается тогда, когда потоки независимы и каждый может поддерживать прямой байтовый буфер в виде потоково-локальной переменной. Иногда это может привести к перерасходу низкоуровневой памяти, если многим потокам требуются буферы разных размеров, так как в итоге у каждого потока появится буфер максимально возможного размера. В подобных ситуациях (или если потоково-локальные буферы не соответствуют архитектуре приложения) пул объектов или прямые байтовые буферы могут оказаться более полезными.

Управление байтовыми буферами также может осуществляться посредством сегментации. Приложение выделяет один очень большой прямой байтовый буфер, а отдельные запросы могут выделять части этого буфера методом `slice()` класса `ByteBuffer`. Такое решение становится крайне неудобным, если сегменты не всегда имеют одинаковый размер: исходный байтовый буфер может оказаться

фрагментированным точно так же, как куча фрагментируется при выделении и освобождении объектов разных размеров. Однако в отличие от кучи, отдельные сегменты байтового буфера не могут сжиматься, так что это решение хорошо работает только в том случае, если все сегменты имеют постоянный размер.

УТЕЧКА ПАМЯТИ В СИСТЕМАХ LINUX

В больших системах Linux иногда встречается утечка низкоуровневой памяти, возникающая из-за структуры их библиотек работы с памятью. Эти библиотеки разбивают низкоуровневую память на сегменты, что упрощает выделение памяти из нескольких потоков (вследствие ограниченной конкуренции со стороны потоков).

Однако управление низкоуровневой памятью отличается от управления кучей Java: в частности, низкоуровневая память никогда не сжимается. Однако схема выделения памяти может привести к такой же фрагментации, какая была описана в главе 5.

В Java низкоуровневая память может быть исчерпана из-за фрагментации низкоуровневой памяти; чаще всего это происходит в больших системах (например, с более чем 8 ядрами), так как количество разделов памяти в Linux зависит от количества ядер в системе.

Два фактора помогут в диагностике этой проблемы: во-первых, приложение выдает ошибку `OutOfMemoryError`, указывающую на нехватку низкоуровневой памяти. Кроме того, обратившись к smaps-файлу процесса, вы найдете в нем много мелких выделений памяти (обычно 64 Кбайт). В этом случае проблема решается присваиванием переменной среды `MALLOC_ARENA_MAX` низкого значения (например, 2 или 4). Значение по умолчанию этой переменной равно количеству ядер в системе, умноженному на 8 (вот почему проблема чаще встречается в больших системах). В этом случае низкоуровневая память все равно будет фрагментирована, но фрагментация должна быть менее серьезной.

С точки зрения настройки при использовании любой из этих программных моделей важно понимать, что размер прямых байтовых буферов, которые могут выделяться приложением, может ограничиваться JVM. Общий объем памяти, который может быть выделен для прямых байтовых буферов, задается флагом `-XX:MaxDirectMemorySize=`*N*. Значение этого флага по умолчанию часто изменялось, но в последних версиях Java 8 (и всех версиях Java 11) максимальное значение равно максимальному размеру кучи; если максимальный размер кучи составляет 4 Гбайт, вы также сможете выделить 4 Гбайт памяти, не входящей в кучу, в прямых и/или отображаемых байтовых буферах. При необходимости значение можно поднять выше максимального размера кучи.

Память, выделенная для прямых байтовых буферов, включается в секцию `Internal` отчета NMT; если этот объем велик, это почти всегда происходит из-за буферов. Если вы хотите точно знать, сколько памяти потребляют сами буферы, эту информацию можно получить из объектов mbean. Проверка объекта mbean `java.nio.BufferPool.direct.Attributes` или `java.nio.BufferPool.mapped.Attributes` сообщит вам объем памяти, выделенной каждым типом. На рис. 8.2 изображен пример с отображением 10 буферов, занимающих в сумме 10 Гбайт памяти.

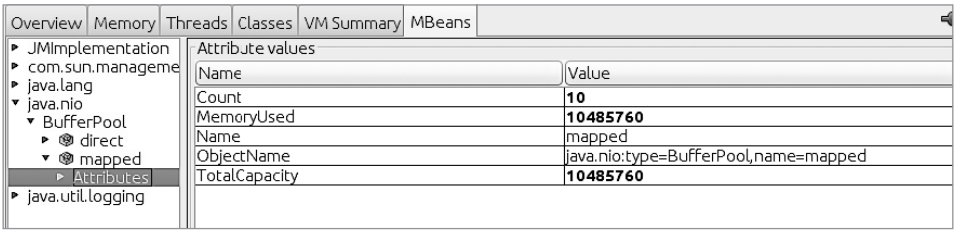


Рис. 8.2. Анализ низкоуровневой памяти в байтовом буфере

РЕЗЮМЕ

- Если вам кажется, что приложение использует слишком много низкоуровневой памяти, то, скорее всего, причиной являются низкоуровневые библиотеки, а не сама JVM.
- Низкоуровневые профили могут эффективно использоваться для определения источника выделения памяти.
- Некоторые часто используемые классы JDK часто вносят свой вклад в использование низкоуровневой памяти; следите за тем, чтобы эти классы использовались правильно.

Настройки JVM для операционной системы

JVM может использовать ряд настроек, повышающих эффективность использования памяти ОС.

Большие страницы

Операции выделения памяти и подкачки осуществляются в страницах. *Страница* (page) — единица памяти, используемая операционными системами для

управления физической памятью. Размер страницы определяет размер минимального блока, выделяемого операционной системой: при выделении 1 байта операционная система должна выделить целую страницу. Дальнейшие блоки памяти, выделяемые этой программой, будут выделяться в той же странице вплоть до ее заполнения, после чего выделяется новая страница.

Операционная система выделяет намного больше страниц, чем может поместиться в физической памяти. Для решения этой проблемы есть механизм подкачки: страницы адресного пространства перемещаются в пространство подкачки и из него (или на другой носитель в зависимости от данных, содержащихся в странице). Это означает, что должно существовать отображение между этими страницами и их местоположением в оперативной памяти компьютера. Такие отображения создаются двумя способами. Все отображения страниц хранятся в глобальной таблице страниц (которую ОС может просматривать для поиска определенного отображения), а наиболее часто используемые отображения хранятся в буферах быстрого преобразования адресов (TLB, Translation Lookaside Buffer).

TLB хранятся в быстром кэше, так что обращение к страницам через элементы TLB выполняется намного быстрее, чем через таблицу страниц.

Количество элементов TLB на машине ограничено, поэтому важно максимизировать частоту попаданий в элементы TLB (таблица действует как LRU-кэш). Так как каждый элемент представляет страницу памяти, повышение размера страницы, используемой приложением, часто приносит пользу. Если каждая страница представляет больше памяти, то для всей программы понадобится меньше элементов TLB и повысится вероятность того, что страница окажется в TLB тогда, когда потребуется. Обычно это правило истинно для любых программ, поэтому оно также истинно для серверов приложений Java и других программ Java даже с кучей умеренного размера.

Большие страницы должны быть включены как на уровне Java, так и на уровне ОС. На уровне Java флаг `-XX:+UseLargePages` включает использование больших страниц; по умолчанию этот флаг равен `false`. Не все операционные системы поддерживают большие страницы; разумеется, способы их активизации зависят от конкретной системы.

Если флаг `UseLargePages` установлен в системе, в которой большие страницы не поддерживаются, предупреждение не выводится, а JVM использует обычные страницы. Если флаг `UseLargePages` устанавливается в системе с поддержкой больших страниц, для которой большие страницы недоступны (либо потому, что все они используются, либо потому, что операционная система настроена неправильно), JVM выводит предупреждение.

Большие страницы в Linux

Конфигурация больших страниц (huge pages) в Linux изменяется между выпусками; за более точными инструкциями обращайтесь к документации вашего выпуска. Общая процедура выглядит примерно так:

1. Определите размер больших страниц, поддерживаемых ядром. Размер зависит от процессора компьютера и параметров загрузки, заданных при запуске ядра, но самый распространенный размер равен 2 Мбайт:

```
# grep Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

2. Определите, сколько больших страниц понадобится. Если JVM выделяет кучу 4 Гбайт, а в системе используются большие страницы по 2 Мбайт, для кучи понадобятся 2048 больших страниц. Количество больших страниц определяется глобально в ядре Linux, поэтому повторите этот процесс для всех запускаемых JVM (а также всех остальных программ, использующих большие страницы). Увеличьте это значение на 10%, чтобы учесть другие использования больших страниц, не связанных с кучей (поэтому в приведенном примере используется 2200 больших страниц).
3. Сохраните значение в операционной системе (чтобы оно немедленно вступило в силу):

```
# echo 2200 > /proc/sys/vm/nr_hugepages
```

4. Сохраните это значение в файле /etc/sysctl.conf, чтобы оно сохранилось после перезагрузки:

```
sys.nr_hugepages=2200
```

5. Во многих версиях Linux объем памяти больших страниц, которую может выделить пользователь, ограничен. Отредактируйте файл /etc/security/limits.conf и добавьте записи `memlock` для пользователя, запустившего JVM (пользователь `appuser` в следующем примере):

```
appuser soft    memlock        4613734400
appuser hard    memlock        4613734400
```

Если файл limits.conf был изменен, то пользователь должен снова войти в систему, чтобы изменение вступило в силу. После этого JVM сможет выделить необходимые большие страницы. Чтобы убедиться в том, что выделение работает, выполните следующую команду:

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

Успешное завершение этой команды показывает, что большие страницы настроены правильно. Если конфигурация памяти большой страницы неправильна, будет выведено предупреждение:

```
Java HotSpot(TM) 64-Bit Server VM warning:
Failed to reserve shared memory (errno = 22).
```

Даже в этом случае программа будет работать; вместо больших страниц она использует обычные.

Прозрачные большие страницы Linux

В ядрах Linux начиная с версии 2.6.32 поддерживаются *прозрачные* большие страницы. Они обеспечивают (теоретически) такой же выигрыш по производительности, как и традиционные большие страницы, но в некоторых отношениях отличаются от них.

Во-первых, традиционные большие страницы фиксируются в памяти, то есть не могут выгружаться из нее. Для Java это является преимуществом, потому что, как упоминалось ранее, выгрузка частей кучи плохо отражается на производительности уборки мусора. Прозрачные большие страницы могут выгружаться на диск, а это ухудшает производительность.

Во-вторых, процесс выделения прозрачной большой страницы значительно отличается от выделения традиционной большой страницы. Традиционные большие страницы резервируются во время загрузки ядра; они доступны всегда. Прозрачные большие страницы выделяются по требованию; когда приложение запрашивает страницу размером 2 Мбайт, ядро пытается найти 2 Мбайт непрерывного пространства в физической памяти для страницы. Если физическая память фрагментирована, то ядро может принять решение о переупорядочении страниц в процессе по аналогии с уже известным вам процессом сжатия памяти в куче Java. Это означает, что время выделения страницы может существенно возрасти из-за ожидания того, пока ядро освободит место в памяти.

Эти различия влияют на все программы, но для Java они могут породить очень долгие паузы на уборку мусора. В процессе уборки мусора JVM может решить, что кучу нужно расширить, и запросить новые страницы. Если выделение страницы занимает несколько сотен миллисекунд или даже секунду, время уборки мусора значительно возрастет.

В-третьих, прозрачные большие страницы по-разному настраиваются на уровнях ОС и Java. Ниже настройка рассматривается более подробно.

На уровне операционной системы прозрачные большие страницы настраиваются изменением содержимого файла /sys/kernel/mm/transparent_hugepage/enabled:

```
# cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
# cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

Поддерживаются три варианта:

`always`

Большие страницы предоставляются всем программам, если это возможно.

`madvise`

Большие страницы предоставляются тем программам, которые их запрашивают; остальным программам предоставляются обычные страницы (4 Кбайт).

`never`

Большие страницы не предоставляются даже тем программам, которые их запрашивают.

В разных версиях Linux используются разные значения этого параметра по умолчанию (и они могут изменяться в будущих версиях). Например, в Ubuntu 18.04 LTS по умолчанию используется значение `madvise`, а в CentOS 7 (и в фирменных выпусках — таких, как Red Hat и в Oracle Enterprise Linux) выбрано значение `always`. Также учтите, что на облачных машинах значение может быть изменено поставщиком образа ОС; я видел образы Ubuntu, в которых параметру присваивалось значение `always`.

Если выбрано значение `always`, никакая конфигурация на уровне Java не нужна: JVM будут предоставляться большие страницы. Собственно, все программы, выполняемые в системе, будут выполняться с большими страницами.

Как нетрудно предположить, при выборе значения `never` никакой аргумент на уровне Java не позволит JVM получить большие страницы. Однако в отличие от традиционных больших страниц, если установлен флаг `UseTransparentHugePages`, а система не может представить прозрачные большие страницы, никакое предупреждение не выводится.

Из-за различий в подкачке и выделении прозрачных больших страниц часто не рекомендуется использовать их с Java; несомненно, их использование может привести к непредсказуемым перепадам длительности пауз. С другой стороны, особенно в тех системах, в которых они включены по умолчанию, вы (в прозрачном режиме, как и обещается) ощутите прирост производительности

в большинстве случаев. Но если вы хотите быть уверены в том, что вы получаете оптимальную производительность с большими страницами, лучше настроить систему для использования прозрачных больших страниц только по запросу, а JVM — для использования традиционных больших страниц.

Большие страницы Windows

Большие страницы Windows могут включаться только в серверных версиях Windows. Ниже приводятся инструкции для Windows 10; в других версиях процедура настройки может отличаться:

1. Запустите Microsoft Management Center. Щелкните на кнопке Пуск и введите в поле поиска строку `mmc`.
2. Если на левой панели не отображается значок Local Computer Policy, выберите команду Add/Remove Snap-in в меню File и добавьте пункт Group Policy Object Editor. Если этот пункт недоступен, значит, используемая версия Windows не поддерживает большие страницы.
3. На левой панели раскройте узел Local Computer Policy ▸ Computer Configuration ▸ Windows Settings ▸ Security Settings ▸ Local Policies и щелкните на папке User Rights Assignment.
4. На правой панели сделайте двойной щелчок на команде Lock Pages in memory.
5. В открывшемся окне добавьте пользователя или группу.
6. Щелкните на кнопке OK.
7. Закройте MMC.
8. Перезагрузите машину.

На этой стадии JVM должна выделить необходимые большие страницы. Чтобы убедиться в том, что все работает, выполните следующую команду:

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

Если команда завершается успешно, значит, большие страницы настроены успешно. Если же конфигурация больших страниц неверна, выводится предупреждение:

```
Java HotSpot(TM) Server VM warning: JVM cannot use large page memory
because it does not have enough privilege to lock pages in memory.
```

Помните, что команда не выведет сообщения об ошибке в системе Windows, не поддерживающей большие страницы (например, в «домашних» версиях): когда

JVM обнаружит, что ОС не поддерживает большие страницы, она присваивает флагу `UseLargePages` значение `false` независимо от конфигурации командной строки.

РЕЗЮМЕ

- Использование больших страниц обычно заметно ускоряет работу приложений.
- Поддержка больших страниц должна быть явно включена в большинстве операционных систем.

Итоги

Хотя из всех областей памяти больше всего внимания обычно достается куче Java, общее потребление памяти JVM критично для ее производительности, особенно в связи с операционной системой. Программы, описанные в этой главе, позволяют отслеживать потребление памяти со временем (и особенно сосредоточиться на закрепленной памяти JVM вместо зарезервированной памяти).

Некоторые возможности использования памяти ОС в JVM — особенно большие страницы — также могут настраиваться для улучшения производительности. JVM, рассчитанные на длительное выполнение, почти всегда выигрывают от использования больших страниц, особенно при большом размере кучи.

ГЛАВА 9

Производительность многопоточных программ и синхронизации

С первых дней существования платформы Java одной из ее сильных сторон была многопоточность. Даже до того, как многоядерные и многопроцессорные системы стали обычным явлением, возможность простого написания многопоточных программ считалась одной из фирменных черт Java.

В контексте производительности выигрыш очевиден: если в системе доступны два процессора, то приложение сможет выполнить вдвое больше работы — или ту же работу вдвое быстрее. Предполагается, что задачу можно разбить на сегменты, потому что Java не принадлежит к числу автопараллелизуемых языков, определяющих алгоритмические части. К счастью, в наши дни вычисления часто направлены на решение раздельных задач: сервер, обрабатывающий конкурентные запросы от разных клиентов, пакетное задание, выполняющее одну операцию с разными наборами данных, математические алгоритмы, состоящие из отдельных фаз, и т. д.

В этой главе вы узнаете, как добиться максимальной производительности от многопоточного выполнения и средств синхронизации в Java.

Многопоточное выполнение и оборудование

Вспомните, что говорилось в главе 1 о многоядерных и гиперпоточных системах. Многопоточность на программном уровне позволяет извлечь пользу из наличия нескольких ядер и гиперпоточной поддержки на машине.

Удвоение числа ядер на машине позволяет удвоить производительность правильно написанного приложения, хотя, как упоминалось в главе 1, поддержка гиперпоточности процессором не удваивает его производительности.

Почти все примеры этой главы запускались на машине с четырьмя однопоточными процессорами — исключением стал первый пример, демонстрирующий различия между гиперпоточными и негиперпоточными процессорами. После этого масштабирование будет рассматриваться только в контексте однопоточных ядер, чтобы вы лучше поняли, как влияет на производительность добавление новых потоков. Это не означает, что гиперпоточные процессоры не нужны; 20–40%-ный прирост производительности от дополнительного потока на аппаратном уровне безусловно улучшит общую производительность вашего приложения. С точки зрения Java мы все равно должны рассматривать гиперпотоки как реальные процессоры и настраивать приложение на четырехъядерной машине с восемью гиперпотоками так, как если бы она была оснащена восемью процессорами. Но с точки зрения измерения производительности следует ожидать только пятишестикратного улучшения по сравнению с одним ядром.

Пулы потоков и объекты ThreadPoolExecutor

В Java разработчик может написать собственный код управления потоками, или же приложения могут воспользоваться пулом потоков. Серверы Java обычно строятся на основе концепции одного или нескольких пулов потоков для обработки запросов: каждое обращение к серверу обрабатывается потоком из пула. Аналогичным образом другие приложения используют объект Java `ThreadPoolExecutor` для параллельного выполнения задач.

Собственно, некоторые серверные фреймворки используют экземпляры класса `ThreadPoolExecutor` для управления своими задачами, хотя многие из них написали собственные реализации пулов потоков (хотя бы из-за того, что они появились до включения `ThreadPoolExecutor` в API). И хотя реализации пулов в таких случаях могут различаться, основные концепции остались неизменными.

Ключевой фактор использования пула потоков заключается в том, что настройка размера пула критична для достижения оптимальной производительности. Производительность пула потоков зависит от основных решений разработчика относительно размера пула потоков, а при некоторых обстоятельствах слишком большой пул потоков отрицательно отразится на производительности.

Все пулы потоков работают практически по одному принципу. Задачи ставятся в очередь (таких очередей может быть несколько, но концепция остается неизменной). Затем несколько потоков извлекают задачи из очереди и выполняют их. Результат задачи может быть отправлен обратно клиенту (например, в случае сервера), сохранен в базе данных, сохранен во внутренней структуре данных и т. д. Но после завершения задачи поток возвращается к очереди задач, чтобы извлечь другое задание (и если задач не осталось, поток ожидает их поступления).

Для пула потоков определяется минимальное и максимальное количество потоков. Минимальное количество присутствует в пуле постоянно, ожидая назначения им задач. Так как создание потока является относительно затратной операцией, это ускоряет работу пула при постановке задачи в очередь: предполагается, что задачу заберет уже существующий поток. С другой стороны, потоки требуют системных ресурсов (включая низкоуровневую память для их стеков), и наличие слишком многих простаивающих потоков приведет к потере ресурсов, которые могли бы использоваться другими процессами. Максимальное количество потоков обеспечивает необходимую регулировку и предотвращает конкурентное выполнение слишком большого числа потоков.

`ThreadPoolExecutor` и сопутствующие классы используют несколько отличающуюся терминологию. В ней используются термины «базовый размер пула» и «максимальный размер пула», а смысл этих терминов зависит от способа создания пула. Иногда базовый размер пула соответствует минимальному размеру пула, иногда — максимальному, а иногда он вообще игнорируется. При этом максимальный размер пула иногда соответствует максимальному, а иногда игнорируется.

Подробности приводятся в конце этого раздела, но для простоты мы в своих тестах установим одинаковые значения базового и максимального размера и будем говорить только о максимальном размере. Следовательно, пулы потоков в примерах всегда будут использовать заданное количество потоков.

Назначение максимального количества потоков

Начнем с максимального количества потоков: как определить оптимальное максимальное количество потоков для заданной рабочей нагрузки на заданном оборудовании? Простого ответа на этот вопрос нет; он зависит от характеристик рабочей нагрузки и оборудования, на котором эта нагрузка выполняется. В частности, оптимальное количество потоков зависит от того, с какой частотой блокируется каждая отдельная задача.

В следующем обсуждении будет использоваться машина с четырьмя одноядерными процессорами. Обратите внимание: неважно, имеет ли система всего четыре ядра, или же 128 ядер, из которых вы используете только четыре, или используется контейнер Docker, ограничивающий использование процессоров только четырьмя. Ваша цель — использовать эти четыре ядра с максимальной эффективностью.

Очевидно, максимальное количество потоков должно быть не менее 4. Правда, некоторые потоки в JVM будут выполнять другую работу помимо обработки этих задач, но этим потокам почти никогда не потребуется целое ядро. Одним из исключений является уборщик мусора конкурентного режима (см. главу 5) — фо-

новые потоки должны располагать достаточными ресурсами процессора (ядрами) для своей работы, иначе они не смогут обрабатывать кучу в достаточном темпе.

Исправит ли ситуацию существование более четырех потоков? Здесь в игру вступают характеристики рабочей нагрузки. Возьмем простой случай, когда все задачи создают интенсивную вычислительную нагрузку: они не генерируют внешние сетевые вызовы (например, обращения к базам данных) и не создают значительной конкуренции по внутренней блокировке. Одним из таких приложений служит пакетная программа анализа истории цен акций (с использованием фиктивного менеджера сущностей): обработка данных сущностей может происходить полностью параллельно.

В табл. 9.1 приведены данные производительности вычисления истории 10 000 фиктивных сущностей с использованием пула потоков, настроенного для заданного количества потоков на машине с четырьмя ядрами. С одним потоком в пуле на вычисление набора данных потребовалось 55,2 секунды; с четырьмя потоками — только 13,9 секунды. После этого добавление новых потоков приводило к некоторому увеличению времени обработки.

Таблица 9.1. Время, необходимое для вычисления 10 000 фиктивных историй цен акций

Количество потоков	Необходимое время (в секундах)	Проценты от эталонного времени
1	55,2 ± 0,6	100%
2	28,3 ± 0,3	51,2%
4	13,9 ± 0,6	25,1%
8	14,3 ± 0,2	25,9%
16	14,5 ± 0,3	26,2%

Если бы задачи в приложении были полностью параллельными, то в столбце «Проценты от эталонного времени» для двух потоков содержалось бы значение 50%, а для четырех — 25%. Точное линейное масштабирование невозможно по нескольким причинам: как минимум потоки должны координироваться для выбора задачи из очереди (и в общем случае потоки должны синхронизироваться). К тому моменту, когда будут использоваться четыре потока, система потребляет 100% доступных ресурсов процессора, и хотя на машине не будут выполняться другие приложения пользовательского уровня, различные процессы системного уровня будут вмешиваться в работу и создавать некоторую нагрузку на процессор, не позволяя JVM задействовать 100% процессорного времени. При этом наше приложение неплохо масштабируется, и даже если количество потоков в пуле будет завышено, потери относительно невелики.

ЭФФЕКТ ГИПЕРПОТОЧНОСТИ

А если четыре процессора будут гиперпоточными, так что будут использоваться два ядра и четыре аппаратных потока? В табл. 9.2 приведены результаты того же эксперимента на такой машине. Как и в предыдущем примере, с точки зрения JVM используется четырехъядерная машина, но масштабирование результатов получается совсем иным.

Таблица 9.2. Время, необходимое для обработки 10 000 фиктивных историй цен акций с гиперпоточностью

Количество потоков	Необходимое время (в секундах)	Проценты от эталонного времени
1	55,7 ± 0,1	100%
2	28,1 ± 0,4	50,4%
4	25,5 ± 0,4	45,7%
8	25,7 ± 0,2	46,1%
16	26,0 ± 0,2	46,6%

Программа хорошо масштабируется с одного процессора до двух, потому что оба процессора являются полноценными ядрами. С другой стороны, добавление гиперпотоков дает минимальный выигрыш. Преимущества гиперпоточности более наглядно проявляются, если потоки приостанавливаются для выполнения ввода/вывода или ожидают освобождения блокировки, удерживаемой другим потоком. Обычно при добавлении гиперпотока прирост производительности составляет около 20%.

Впрочем, в других обстоятельствах потери от слишком большого количества потоков могут возрасти. В REST-версии программы вычисления истории цен акций завышение числа потоков приводит к более серьезным последствиям, как показано в табл. 9.3. Сервер приложения настроен для использования заданного количества потоков, а генератор нагрузки посылает серверу 16 конкурентных запросов.

Учитывая, что REST-серверу доступны четыре процессора, максимальная производительность достигается при четырех потоках в пуле.

В главе 1 рассматривался вопрос выявления узких мест при анализе проблем производительности. В данном примере узким местом очевидным образом яв-

ляется процессор: при четырех процессорах обеспечивается 100%-ная загрузка. Однако потери при добавлении лишних потоков в данном случае близки к минимуму (по крайней мере до четырехкратного превышения количества потоков).

Таблица 9.3. Производительность REST-сервера (операции в секунду)

Количество потоков	Операции в секунду	Проценты от эталонного значения
1	46,4	27%
4	169,5	100%
8	165,2	97%
16	162,2	95%

Но что, если узкое место находится где-то еще? Данный пример также немного нетипичен в том, что задачи связаны с интенсивными вычислениями: они не выполняют ввода/вывода. Как правило, потоки обращаются с вызовами к базам данных, куда-то записывают свой вывод или даже взаимодействуют с другими ресурсами. В этом случае узким местом не обязательно является процессор: им вполне может оказаться внешний ресурс.

В таких случаях добавление новых потоков в пул только ухудшит ситуацию. Хотя в главе 1 я говорил (лишь отчасти в шутку), что база данных всегда является узким местом, на самом деле узким местом может быть любой внешний ресурс.

Для примера возьмем REST-сервер с акциями, на котором роли меняются местами: что, если целью является оптимальное использование машины с генератором нагрузки (которая, в конце концов, всего лишь выполняет многопоточную программу Java)?

В типичном случае, если REST-приложение выполняет сервер с четырьмя процессорами и только одним клиентом, запрашивающим данные, REST-сервер будет загружен всего на 25% и клиентская машина будет практически простаивать. Если увеличить нагрузку до четырех конкурентно подключенных клиентов, сервер будет загружен на 100%, а клиентская машина может быть загружена только на 20%.

Если рассматривать только клиента, напрашивается вывод, что при наличии избыточных ресурсов процессора у клиента, должна быть возможность добавить к клиенту новые потоки для повышения его производительности. В табл. 9.4 показано, насколько ошибочно это заключение: добавление новых потоков к клиенту очень серьезно отражается на его производительности.

Таблица 9.4. Среднее время отклика для обработки фиктивных историй цен акций

Количество клиентских потоков	Среднее время отклика	Проценты от эталонного значения
1	0,022 секунды	100%
2	0,022 секунды	100%
4	0,024 секунды	109%
8	0,046 секунды	209%
16	0,093 секунды	422%
32	0,187 секунды	885%

После того как REST-сервер станет узким местом в этом примере (то есть с четырьмя клиентскими потоками), увеличение нагрузки на сервер только вредит.

Этот пример может показаться неестественным. Кто станет добавлять новые клиентские потоки, когда сервер и так загружен? Но я использовал этот пример просто потому, что он понятен и в нем используются только программы Java. Вы можете запустить его самостоятельно, чтобы понять, как он работает, без настройки подключений к базам данных, создания схем и т. д.

Дело в том, что этот принцип справедлив для REST-сервера, который отправляет запросы к базе данных, которая выполняет интенсивные вычисления. Вы можете ограничиться загрузкой процессора на сервере, увидеть, что она заметно ниже 100%, а в очереди имеются запросы для обработки, и решить, что на сервере стоит увеличить количество потоков. Это приведет к большим сюрпризам, потому что увеличение количества потоков в такой ситуации на самом деле уменьшит общую производительность (возможно, значительно) — как это было при увеличении количества клиентских потоков в примере с Java.

Есть и другая причина, по которой важно понять, где в системе находится узкое место: увеличение загрузки на критическом участке приведет к серьезному снижению производительности. И наоборот, если нагрузка на текущем критическом участке снижается, производительность может повыситься.

Это еще одна причина, затрудняющая самонастройку потоков пулов. Пулы потоков обычно обладают некоторой информацией о текущем объеме незавершенной работы и, возможно, даже о ресурсах процессоров, доступных на машине, — но обычно они ничего не знают о других аспектах окружения, в котором они выполняются. Следовательно, добавление потоков при наличии незавершенной работы — ключевая особенность многих самонастраивающихся пулов потоков (а также некоторых конфигураций `ThreadPoolExecutor`) — часто оказывается тем, чего делать не следует.

В табл. 9.4 в конфигурации по умолчанию сервера REST на машине с четырьмя процессорами создавались 16 потоков. Это имеет смысл в стандартной обобщенной ситуации, потому что можно предполагать, что потоки будут генерировать внешние вызовы. Если такие вызовы блокируются в ожидании ответа, в это время могут выполняться другие задачи, и серверу может потребоваться более четырех потоков для выполнения этих задач. Таким образом, конфигурация по умолчанию, которая создает незначительное количество избыточных потоков, может быть разумным компромиссом: она будет создавать небольшие затраты для задач, требующих интенсивных вычислений, и позволит повысить производительность для выполнения задач, выполняющих блокирующие операции ввода/вывода. Другой сервер мог бы создавать по умолчанию 32 потока, что приводило бы к большим потерям для нашего теста с интенсивными вычислениями, но также обеспечивало бы большие преимущества для обработки нагрузки, связанной в основном с интенсивными вычислениями.

К сожалению, это также объясняет, почему выбор максимального размера пула потоков часто становится в большей степени искусством, нежели наукой. В реальном мире самонастраивающийся пул потоков может обеспечить 80–90% возможной производительности тестируемой системы, и переоценка количества потоков, необходимых для пула, может привести к незначительным потерям. Но если при таком определении размера будет допущена ошибка, последствия могут быть очень серьезными. К сожалению, адекватное тестирование в этом отношении остается ключевой необходимостью.

Настройка минимального количества потоков

После того как максимальное количество потоков в пуле будет определено, можно переходить к определению минимального количества потоков. Перейдем к сути дела: это редко имеет значение, и для простоты практически во всех случаях можно присвоить минимальному количеству потоков то же значение, что и максимальному.

Один из доводов в пользу выбора другого значения для минимального количества потоков (например, 1) — он не позволяет системе создавать слишком много потоков, обеспечивая экономию системных ресурсов. Правда, каждый поток требует определенного объема памяти, особенно для стека (эта тема рассматривается далее в этой главе). Впрочем, в соответствии с одним из правил главы 2, размер системы должен выбираться таким образом, чтобы справиться с максимально возможной нагрузкой; в этот момент ей придется создать все эти потоки. Если система не способна справиться с максимальным количеством потоков, выбор меньшего минимального количества потоков особо не поможет: если система столкнется с ситуацией, требующей макси-

мального количества потоков (с которым она справиться не может), то такая система обречена на неудачу. Лучше создать все потоки, которые могут когда-либо понадобиться, и убедиться в том, что система справится с максимальной ожидаемой нагрузкой.

НУЖНО ЛИ СОЗДАВАТЬ ПОТОКИ ЗАРАНЕЕ?

По умолчанию при создании объекта `ThreadPoolExecutor` он начинает существование только с одним потоком. Представьте конфигурацию, в которой пул запрашивает 8 базовых потоков и 16 максимальных потоков; в этом случае базовая настройка может рассматриваться как минимальная, потому что восемь потоков будут выполняться даже в том случае, если они простаивают. Но эти восемь потоков не будут создаваться при создании пула; они создаются по мере надобности и продолжают выполняться после этого.

В серверной конфигурации это означает, что обработка первых восьми запросов будет слегка замедлена из-за создания потоков. Как было показано в этом разделе, этот эффект минимален, но при необходимости потоки можно создать заранее (методом `prestartAllCoreThreads()`).

С другой стороны, потери от определения минимального количества потоков весьма незначительны. Они встречаются только при первом конкурентном выполнении нескольких задач: тогда пулу потребуется создать новый поток. Создание потоков отрицательно сказывается на производительности (собственно, это является главной причиной для использования пулов потоков), но эти одноразовые затраты на создание потоков, скорее всего, останутся незамеченными, если поток после этого останется в пуле.

В пакетном приложении не так важно, создается ли поток при создании пула (что происходит, если задать минимальному и максимальному количеству потоков одинаковое значение) или же поток создается по требованию: время выполнения приложения останется неизменным. В других приложениях новые потоки с большей вероятностью будут создаваться в период разогрева (и снова общее время создания потоков останется неизменным); влияние на производительность приложения будет минимальным. Даже если потоки создаются в цикле хронометража, при ограниченном количестве создаваемых потоков это, скорее всего, остается незаметным.

Еще одна настройка, которая может применяться в данном случае, — время бездействия потока. Допустим, при определении размеров пула указывается минимальный размер 1 и максимальный размер 4. Теперь предположим, что обычно один поток выполняет задачу, а затем приложение запускает цикл, в котором за

каждые 15 секунд средняя рабочая нагрузка составляет 2 задачи. При первом проходе этого цикла пул создаст второй поток — и теперь желательно, чтобы этот второй поток оставался в пуле в течение хотя бы некоторого периода времени. Нужно избежать ситуации, в которой второй поток создается, завершает свою задачу за 5 секунд, 5 секунд простаивает, а затем завершается — потому что через 5 секунд второй поток потребуется для следующей задачи. В общем случае, после того как поток создается в пуле для минимального размера, он должен просуществовать хотя бы несколько минут, чтобы обработать возможный резкий рост нагрузки. Время бездействия может оцениваться на основании этого фактора — в той степени, в которой вы можете моделировать интенсивность изменения нагрузки. В остальных случаях планируйте время бездействия в минутах, по крайней мере от 10 до 30.

Поддержание бездействующих потоков обычно не оказывает особого влияния на приложение. Как правило, сам поток объекта занимает не так много пространства в куче. Исключение из этого правила встречается тогда, когда поток удерживает большой объем потоково-локальной памяти или когда исполняемый объект потока поддерживает ссылку на большой объем памяти. В любом из этих случаев освобождение потока может предоставить значительную экономию в отношении живых данных, остающихся в куче (а это, в свою очередь, влияет на эффективность уборки мусора).

Впрочем, для пулов потоков такие ситуации встречаться не должны. Когда поток в пуле бездействует, он не должен быть связан ни с каким исполняемым объектом (а если связан — значит, где-то ошибка). В зависимости от реализации пула потоково-локальные переменные могут оставаться на своих местах — но хотя в некоторых обстоятельствах потоково-локальные переменные могут быть эффективным механизмом, способствующим повторному использованию объектов (см. главу 7), общий объем памяти, занимаемой этими потоково-локальными объектами, должен быть ограничен.

Важное исключение из этого правила составляют пулы потоков, которые могут вырасти до очень больших размеров (а следовательно, работать на очень больших размерах). Допустим, ожидается, что средний размер очереди задач в пуле потоков равен 20; в этом случае 20 становится хорошим минимальным размером пула. Теперь предположим, что пул работает на очень большой машине и он разрабатывался в расчете на пиковую нагрузку 2000 задач. Поддержание 2000 бездействующих потоков в пуле повлияет на производительность в то время, когда выполняются только 20 задач — эффективность этого пула при 1980 бездействующих потоках может составлять всего 50% (по сравнению с ситуацией, когда он содержит всего 20 базовых занятых потоков). Пулы потоков обычно не сталкиваются с подобными проблемами выбора размера, но когда это происходит, стоит проследить за тем, чтобы им было назначено хорошее минимальное значение.

Размеры задач для пула потоков

Незавершенные задачи для пула потоков хранятся в очереди или в списке; когда поток в пуле может выполнить задачу, он извлекает ее из очереди. Это может привести к нарушению баланса, так как количество задач в очереди может стать очень большим. При очень большой очереди задачам в очереди придется подолгу ожидать, пока предшествующие задачи будут завершены. Представьте перегруженный веб-сервер: если задача добавляется в очередь и не будет выполнена в течение 3 секунд, скорее всего, пользователь перейдет к другой странице.

По этой причине пулы потоков обычно ограничивают размер очереди незавершенных задач. Объект `ThreadPoolExecutor` делает это разными способами в зависимости от структуры данных, включенной в настройку (подробнее об этом в следующем разделе); серверы обычно предоставляют параметры для настройки этого значения.

Как и в случае с максимальным размером пула потоков, универсальных правил по настройке этого значения нет. Сервер с 30 000 элементами в очереди и четырьмя доступными процессорами сможет расчистить очередь за 6 минут, если на выполнение задачи уходит всего 50 мс (предполагается, что за это время новые задачи не появляются). Это время может быть приемлемым, но если на выполнение каждой задачи уходит 1 секунда, на расчистку очереди уйдет 2 часа. И снова только тестирование реального приложения позволит с уверенностью сказать, какое значение обеспечит необходимую производительность.

В любом случае при достижении лимита очереди дальнейшие попытки добавления задач должны завершаться неудачей. Класс `ThreadPoolExecutor` содержит метод `rejectedExecution()` для обработки такой ситуации (по умолчанию он выдает исключение `RejectedExecutionException`, но это поведение можно переопределить). Серверы приложений должны возвращать разумный ответ пользователю (с сообщением, описывающим произошедшее), а серверы REST должны возвращать код статуса 429 (слишком много запросов) или 503 (сервис недоступен).

Определение размера ThreadPoolExecutor

В общем случае пул потоков начинается с минимального количества потоков, и если при поступлении новой задачи все имеющиеся потоки заняты, запускается новый поток (до максимального количества потоков) и задача выполняется немедленно. Если запущено максимальное количество потоков, но все они заняты, задача ставится в очередь — если только очередь уже не содержит слишком много задач; в этом случае задача отвергается. Хотя это описание представляет каноническое поведение пула потоков, `ThreadPoolExecutor` также может вести себя иначе.

`ThreadPoolExecutor` принимает решение о запуске нового потока в зависимости от типа очереди, используемой для хранения задач. Возможны три варианта:

- `SynchronousQueue` — при использовании `SynchronousQueue` пул потоков использует ожидаемое поведение в отношении количества потоков: новые задачи запускают новый поток, если все существующие потоки заняты, а количество потоков в пуле меньше максимального. Но у такой очереди нет возможности хранения незавершенных задач: если поступает задача, а максимальное количество потоков уже занято, задача всегда отвергается. Таким образом, этот вариант хорошо подходит для управления небольшим числом задач, но в остальных случаях может оказаться неприемлемым. В документации класса рекомендуется задать очень большое число для максимального количества потоков — это может быть нормально для чисто вычислительных задач, но, как вы уже видели, может оказаться нерациональным в других ситуациях. С другой стороны, если вам понадобится пул с легко настраиваемым количеством потоков, этот вариант подходит лучше других.

  В этом случае базовый размер соответствует минимальному размеру пула: количество потоков, которые будут поддерживаться даже в том случае, если они бездействуют. Максимальный размер определяет максимальное количество потоков в пуле.

  Этот тип пула (с неограниченным максимальным количеством потоков) возвращается методом `newCachedThreadPool()` класса `Executor`.

- *Неограниченные очереди* — если `ThreadPoolExecutor` использует неограниченную очередь (например, `LinkedBlockedingQueue`), никакие задачи отвергаться не будут (так как размер очереди не ограничен). В этом случае будет использоваться количество потоков, не превышающее базового размера пула: максимальный размер пула игнорируется. Фактически это поведение соответствует традиционному пулу потоков, у которого базовый размер интерпретируется как максимальный размер пула. Если очередь не ограничена, возникает риск поглощения слишком большого объема памяти, если скорость поступления задач превышает скорость их обработки.

  Этот тип пула потоков возвращается методами `newFixedThreadPool()` и `newSingleThreadScheduledExecutor()` класса `Executors`. Базовый (или максимальный) размер пула из первого случая определяется параметром, переданным при конструировании пула; во втором случае базовый размер пула равен 1.

- *Ограниченные очереди* — при использовании ограниченной очереди (например, `ArrayBlockingQueue`) момент запуска нового потока определяется сложным алгоритмом. Например, допустим, что базовый размер пула равен 4, максимальный размер равен 8, а максимальный размер `ArrayBlockingQueue` равен 10. По мере того как задачи поступают и помещаются в очередь, в пуле работают максимум 4 потока (базовый размер пула). Даже если оче-

редь заполнится — то есть будет содержать 10 незавершенных задач, — объект `ThreadPoolExecutor` будет использовать 4 потока.

Дополнительный поток будет использоваться только в случае добавления новой задачи в заполненную очередь. Вместо того чтобы отвергать задачу (так как в очереди не осталось места), объект запускает новый поток. Этот новый поток выполняет первую задачу в очереди, освобождая место для добавления в очередь незавершенной задачи.

В этом примере пул может содержать 8 потоков (заданный максимум) только в одном случае: если 7 задач выполняются, 10 задач находятся в очереди, а в очередь добавляется новая задача.

В основе этого алгоритма лежит идея о том, что большую часть времени пул будет работать только с базовыми потоками (4), даже если в очереди находится умеренное количество задач, ожидающих выполнения. Таким образом регулируется нагрузка на пул. Если же незавершенных запросов становится слишком много, пул пытается запустить новые потоки для разгрузки очереди (в соответствии со вторым регулятором — максимальным количеством потоков).

Если в системе нет узких мест, а ресурсы процессора доступны, все работает нормально: с добавлением новых потоков очередь обрабатывается быстрее и, скорее всего, вернется к желаемому размеру. Ситуации, в которых этот алгоритм уместен, представить нетрудно.

С другой стороны, алгоритму неизвестно, из-за чего увеличился размер очереди. Если это произошло из-за поступления внешней незавершенной работы, то добавление новых потоков — неподходящая мера. То же самое можно сказать, если пул занят чисто вычислительной работой. Добавление потоков имеет смысл только в том случае, если незавершенная работа образовалась из-за поступления нагрузки в систему (например, запросы HTTP стали поступать от большего количества клиентов). (Впрочем, даже в этом случае зачем медлить с добавлением потоков до того, как очередь достигнет определенной границы? Если доступны дополнительные ресурсы, позволяющие использовать большее число потоков, быстрое их добавление улучшит общую производительность системы.)

У каждого варианта есть много аргументов «за» и «против», но в попытках добиться максимальной производительности следует руководствоваться принципом KISS («Keep it Simple, Stupid», то есть «будь проще, тупица»). Как обычно, потребности приложения могут потребовать других решений, но в общем случае не используйте классы `Executor` для получения стандартных неограниченных пулов потоков, которые не позволяют управлять использованием памяти в приложении. Вместо этого создайте собственную реализацию `ThreadPoolExecutor` с таким же количеством базовых и максимальных потоков, которая использует `ArrayBlockingQueue` для ограничения количества запросов, находящихся в памяти в ожидании выполнения.

РЕЗЮМЕ

- Пулы потоков — один из случаев, в которых пулы объектов стоит использовать в приложениях: инициализация потоков обходится дорого, а пул потоков позволяет легко отрегулировать количество потоков в системе.
- Пулы потоков требуют тщательной настройки. В некоторых обстоятельствах бездумное добавление новых потоков в пул может привести к потере производительности.
- Использование более простых реализаций `ThreadPoolExecutor` обычно обеспечивает наилучшую (и самую предсказуемую) производительность.

ForkJoinPool

Кроме реализаций `ThreadPoolExecutor` общего назначения, Java предоставляет специализированный пул: класс `ForkJoinPool`. Внешне этот класс похож на любой другой пул потоков: как и класс `ThreadPoolExecutor`, он реализует интерфейсы `Executor` и `ExecutorService`. С этими интерфейсами `ForkJoinPool` использует внутренний неограниченный список задач, которые будут выполняться количеством потоков, заданным в конструкторе. Если конструктору не передается аргумент, то пул определяет свой размер на основании количества процессоров, доступных на машине (или на основании количества процессоров, доступных для контейнера Docker).

Класс `ForkJoinPool` спроектирован для работы с алгоритмами, действующими по принципу «разделяй и властвуй», — то есть алгоритмами, позволяющими рекурсивно разбивать задачу на подзадачи. Эти подзадачи обрабатываются параллельно, после чего результаты всех подзадач объединяются в один результат. Классическим примером служит алгоритм быстрой сортировки.

У алгоритмов «разделяй и властвуй» есть одна важная особенность: они создают множество подзадач, для управления которыми используется относительно небольшой набор потоков. Допустим, вы хотите отсортировать массив из 10 миллионов элементов. Сортировка начинается с создания отдельных задач для выполнения трех операций: сортировки подмассива, содержащего первые 5 миллионов элементов; сортировки подмассива, содержащего вторые 5 миллионов элементов; и последующего слияния двух подмассивов.

Аналогичным образом сортировка массивов с 5 миллионами элементов осуществляется сортировкой подмассивов с 2,5 миллиона элементов и последующим слиянием этих массивов. Рекурсия продолжается до определенного момента (например, когда в подмассиве остается 47 элементов), в котором будет более

эффективно воспользоваться сортировкой методом вставки и отсортировать массив напрямую. На рис. 9.1 показано, как работает этот алгоритм.

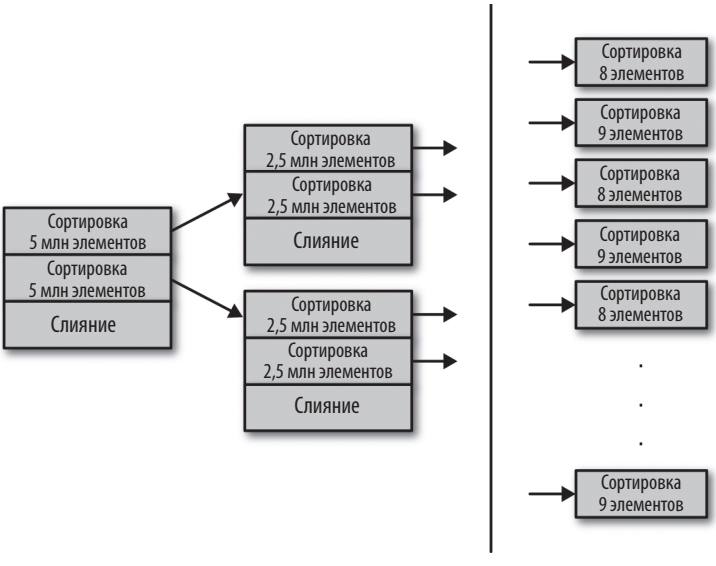


Рис. 9.1. Рекурсивная быстрая сортировка с разбиением на подзадачи

В итоге мы получаем 262 144 задачи для сортировки листовых массивов, каждый из которых содержит 47 (и менее) элементов. (Число 47 зависит от алгоритма и определяется на основании серьезного анализа, но для быстрой сортировки в Java используется именно это значение.)

Для слияния этих отсортированных массивов требуются еще 131 072 задачи, для слияния следующего набора отсортированных массивов — еще 65 536 задач и т. д. В сумме получается 524 287 задач.

Здесь важно то, что ни одна из этих задач не может быть завершена до того, как будут также завершены все порожденные ею задачи. Сначала должны завершиться задачи сортировки массивов, содержащих менее 47 элементов; затем объединяются два маленьких массива, которые были ими созданы, и т. д.; слияние выполняется вверх по цепочке, пока весь массив не будет объединен в окончательное отсортированное состояние.

Эффективно реализовать этот алгоритм на базе `ThreadPoolExecutor` невозможно, потому что родительская задача должна дождаться завершения своих дочерних задач. Поток в `ThreadPoolExecutor` не сможет добавить другую задачу в очередь, а затем дождаться ее завершения; если поток находится в ожидании, он не сможет использоваться для выполнения одной из своих подзадач. С другой стороны,

`ForkJoinPool` позволяет своим потокам создавать новые задачи, а затем приостанавливать свою текущую задачу. Пока задача приостановлена, поток может выполнять другие незавершенные задачи.

Возьмем простой пример: допустим, имеется массив чисел двойной точности. Требуется подсчитать количество значений в массиве, меньших 0,5. Можно последовательно просканировать массив; это простое решение (и возможно, рациональное, как будет показано позднее в этом разделе), но пока попробуем разделить массив на подмассивы и просканировать их параллельно (эмуляция более сложных алгоритмов быстрой сортировки и других алгоритмов, работающих по принципу «разделяй и властвуй»). Ниже приведена заготовка кода для решения этой задачи с использованием `ForkJoinPool`.

```
private class ForkJoinTask extends RecursiveTask<Integer> {
    private int first;
    private int last;

    public ForkJoinTask(int first, int last) {
        this.first = first;
        this.last = last;
    }

    protected Integer compute() {
        int subCount;
        if (last - first < 10) {
            subCount = 0;
            for (int i = first; i <= last; i++) {
                if (d[i] < 0.5)
                    subCount++;
            }
        }
        else {
            int mid = (first + last) >>> 1;
            ForkJoinTask left = new ForkJoinTask(first, mid);
            left.fork();
            ForkJoinTask right = new ForkJoinTask(mid + 1, last);
            right.fork();
            subCount = left.join();
            subCount += right.join();
        }
        return subCount;
    }
}
```

Методы `fork()` и `join()` играют ключевую роль: реализовать подобную рекурсию без этих методов будет нелегко (и эти методы недоступны для задач, выполняемых `ThreadPoolExecutor`). Эти методы используют серию внутренних очередей уровня потока для манипуляций с задачами и переключения потоков от выполнения одной задачи на выполнение другой. Подробности реализации

прозрачны для разработчика, хотя если вас заинтересуют алгоритмы, чтение их кода становится в высшей степени увлекательным занятием. Но нас в данном случае интересует производительность: к каким последствиям приводит выбор между классами `ForkJoinPool` и `ThreadPoolExecutor`?

Первое и самое важное: парадигма приостановки `fork/join` позволяет выполнять все задачи в небольшом количестве потоков. Подсчет значений в массиве из 2 миллионов элементов с использованием этого кода создает более 4 миллионов задач, но эти задачи легко выполняются всего несколькими потоками (и даже одним, если это имеет смысл на машине, на которой запускается тест). Выполнение аналогичного алгоритма с использованием `ThreadPoolExecutor` потребует более 4 миллионов потоков, так как каждому потоку придется ожидать завершения своих подзадач, а эти подзадачи смогут завершиться только при наличии дополнительных потоков в пуле. Таким образом, приостановка `fork/join` позволяет использовать алгоритмы, которые не могли бы использоваться в других обстоятельствах, что обеспечивает выигрыш по производительности.

С другой стороны, подобный простой алгоритм плохо подходит для реального применения пула `fork/join`. Этот пул идеально подходит для следующих случаев:

- Фаза слияния алгоритма выполняет некоторую нетривиальную работу (вместо простого суммирования двух чисел, как в нашем примере).
- Листовые вычисления алгоритма выполняют достаточный объем работы, компенсирующий затраты на создание задачи.

При отсутствии этих двух критериев массив достаточно легко разбить на блоки и воспользоваться `ThreadPoolExecutor` для многопоточного сканирования массива:

```
public class ThreadPoolTest {
    private double[] d;

    private class ThreadPoolExecutorTask implements Callable<Integer> {
        private int first;
        private int last;

        public ThreadPoolExecutorTask(int first, int last) {
            this.first = first;
            this.last = last;
        }

        public Integer call() {
            int subCount = 0;
            for (int i = first; i <= last; i++) {
                if (d[i] < 0.5) {
```

```
                    subCount++;
                }
            }
            return subCount;
        }
    }

    public static void main(String[] args) {
        d = createArrayOfRandomDoubles();
        ThreadPoolExecutor tpe = new ThreadPoolExecutor(4, 4,
                                        Long.MAX_VALUE,
                                        TimeUnit.SECONDS,
                                new LinkedBlockingQueue());
        Future[] f = new Future[4];
        int size = d.length / 4;
        for (int i = 0; i < 3; i++) {
            f[i] = tpe.submit(
                        new ThreadPoolExecutorTask(i * size, (i + 1) * size - 1);
        }
        f[3] = tpe.submit(new ThreadPoolExecutorTask(3 * size, d.length - 1);
        int n = 0;
        for (int i = 0; i < 4; i++) {
            n += f.get();
        }
        System.out.println("Found " + n + " values");
    }
}
```

На 4-процессорной машине этот код полностью использует все доступные процессоры. Массив обрабатывается параллельно, избегая создания и постановки в очередь 4 миллионов задач, используемых в примере `fork/join`. Как и следовало ожидать, производительность увеличивается, что показано в табл. 9.5.

Таблица 9.5. Время подсчета в массиве из 2 миллионов элементов

Количество потоков	ForkJoinPool	ThreadPoolExecutor
1	125 ± 1 мс	1,731 ± 0,001 мс
4	37,7 ± 1 мс	0,55 ± 0,002 мс

Два теста различаются по времени уборки мусора, но настоящие различия происходят от алгоритма «разделяй и властвуй», особенно с листовым значением 10. Затраты на создание и управление 4 миллионами объектов задач ухудшают производительность `ForkJoinPool`. Если доступна сходная альтернатива, вполне возможно, что она будет работать быстрее — по крайней мере в этом простом случае.

Кроме того, более раннее завершение рекурсии потребовало бы гораздо меньшего количества задач. Возьмем один из крайних случаев: рекурсию можно было бы остановить, когда подмассив содержит 500 000 элементов; при этом общий объем работы аккуратно делится на четыре задачи, как в примере с пулом потоков. На этой стадии производительность теста была бы той же (хотя если работа разделяется настолько легко, нет никаких причин применять алгоритм «разделяй и властвуй»).

Для демонстрации можно легко справиться со вторым пунктом нашего критерия, добавив работу в фазу листовых вычислений нашей задачи:

```
for (int i = first; i <= last; i++) {
    if (d[i] < 0.5) {
        subCount++;
    }
    for (int j = 0; j < 500; j++) {
        d[i] *= d[i];
    }
}
```

Теперь в тесте доминирует вычисление `d[i]`. Но поскольку в фазе слияния алгоритма не выполняется никакой значительной работы, создание всех задач все еще сопряжено с потерями, как видно из табл. 9.6.

Таблица 9.6. Время подсчета в массиве из 2 миллионов элементов с дополнительной работой

Количество потоков	ForkJoinPool	ThreadPoolExecutor
4	271 ± 3 мс	258 ± 1 мс

Теперь, когда во времени доминируют непосредственные вычисления в тесте, пул `fork/join` выглядит не так плохо по сравнению с разбиением. Однако время создания задач остается значительным, и когда задачи могут разбиваться на подзадачи (то есть когда в фазе слияния не выполняется никакой значительной работы), простой пул потоков будет работать быстрее.

Перехват работы

Одно из правил использования этой разновидности пулов требует убедиться в том, что разбиение задач имеет смысл. Однако вторая особенность пула `ForkJoinPool` делает его еще более мощным: он реализует *перехват работы*. По сути речь идет о подробности реализации: термин означает, что каждый поток в пуле имеет собственную порожденную им очередь задач. Потоки предпочита-

ют работать с задачами из собственной очереди, но если эта очередь пуста, они перехватывают задачи из очередей других потоков. Как следствие, даже если выполнение одной из 4 миллионов задач занимает много времени, другие потоки в `ForkJoinPool` могут завершать любые из остальных задач. У `ThreadPoolExecutor` такая возможность отсутствует: если одна из задач занимает много времени, другие потоки не смогут взять дополнительную работу.

При добавлении работы в исходный пример объем работы на одно значение был постоянным. Что, если эта работа будет изменяться в зависимости от позиции элемента в массиве?

```
for (int i = first; i <= last; i++) {
    if (d[i] < 0.5) {
        subCount++;
    }
    for (int j = 0; j < i; j++) {
        d[i] += j;
    }
}
```

Так как внешний цикл (с индексированием по `j`) основан на позиции элемента в массиве, вычисления требуют времени, пропорционального позиции элемента: вычисление значения `d[0]` будет очень быстрым, тогда как вычисление значения `d[d.length - 1]` потребует больше времени.

Теперь простое разбиение теста `ThreadPoolExecutor` будет невыгодным. Поток, вычисляющий первое разбиение массива, будет работать очень долго — намного больше времени, проведенного четвертым потоком, работающим с последним разбиением. После завершения четвертого потока он остается бездействующим: всем приходится ожидать, пока первый поток выполняет свою долгую задачу.

Степень детализации 4 миллионов задач в `ForkJoinPool` означает, что хотя один поток погрязнет за выполнением очень долгих вычислений с первыми 10 элементами массива, у остальных потоков будет работа, которую они должны выполнять, и процессор будет оставаться занятым в ходе большей части теста. Различия представлены в табл. 9.7.

Таблица 9.7. Время обработки массива из 2 миллионов элементов с несбалансированной рабочей нагрузкой

Количество потоков	ForkJoinPool	ThreadPoolExecutor
1	22,0 ± 0,01 с	21,7 ± 0,1 с
4	5,6 ± 0,01 с	9,7 ± 0,1 с

Если пул содержит единственный поток, вычисления занимают практически то же время. Это логично: объем вычислений остается неизменным независимо от реализации пула, и поскольку эти вычисления никогда не выполняются параллельно, можно ожидать, что они займут то же время (с некоторыми дополнительными затратами на создание 4 миллионов задач). Но когда пул содержит четыре потока, степень детализации задач в пуле `ForkJoinPool` дает ему решительное преимущество: он может обеспечить занятость процессора почти на все время теста.

Такая ситуация называется *несбалансированной*, потому что некоторые задачи занимают больше времени, чем другие (поэтому задачи из предыдущего примера называются *сбалансированными*). В общем случае использование `ThreadPoolExecutor` с разбиением обеспечит повышение производительности, если задачи легко разбиваются в сбалансированное множество, а `ForkJoinPool` — если задачи не сбалансированы.

Также есть другая, более тонкая рекомендация из области производительности: тщательно выберите точку, в которой рекурсия парадигмы `fork/join` должна завершиться. В нашем примере было произвольно решено, что завершение должно происходить тогда, когда размер массива становится меньше 10. Как было показано ранее, в сбалансированном случае завершение рекурсии будет оптимальным при 500 000.

С другой стороны, рекурсия в несбалансированном случае обеспечивает еще лучшую производительность для меньших листовых значений. Характерные точки данных приведены в табл. 9.8.

Таблица 9.8. Время обработки массива из 2 миллионов элементов с переменными листовыми значениями

Целевой размер листового массива	ForkJoinPool
500 000	9842 ± 5 мс
50 000	6029 ± 100 мс
10 000	5764 ± 55 мс
1000	5657 ± 56 мс
100	5598 ± 20 мс
10	5601 ± 15 мс

С листовым размером 500 000 был продублирован случай `ThreadPoolExecutor`. С уменьшением листового размера мы выигрываем от несбалансированной

природы теста вплоть до диапазона 1000–10 000, где производительность выравнивается.

Подобная настройка листового значения характерна для подобных алгоритмов. Как упоминалось ранее в этом разделе, в Java используется значение 47 в качестве листового значения в своей реализации алгоритма быстрой сортировки: в этой точке (для данного алгоритма) затраты на создание задач перевешивают выигрыш от подхода «разделяй и властвуй».

Автоматическая параллелизация

В Java предусмотрена возможность автоматической параллелизации некоторых видов кода. Такая параллелизация основана на использовании класса `ForkJoinPool`. JVM создает для этой цели обычный пул `fork/join`; это статический элемент класса `ForkJoinPool`, размер по умолчанию для которого равен количеству процессоров на целевой машине.

Такая параллелизация применяется во многих методах класса `Array`: методах сортировки массива с использованием параллельной быстрой сортировки, методах обработки каждого отдельного элемента массива и т. д. Также она используется с функциональностью потоков данных, позволяющей выполнять операции (последовательные или параллельные) с каждым элементом коллекции. Основные последствия от использования потоков данных для производительности обсуждаются в главе 12; в этом разделе рассматривается автоматизация параллельной обработки потоков данных.

Допустим, имеется коллекция, содержащая последовательность целых чисел. Следующий код вычисляет историю цен акций для биржевого обозначения, соответствующего заданному целому числу:

```
List<String> symbolList = ...;
Stream<String> stream = symbolList.parallelStream();
stream.forEach(s -> {
    StockPriceHistory sph = new StockPriceHistoryImpl(s, startDate,
                                    endDate, entityManager);
    blackhole.consume(sph);
});
```

Этот код вычисляет истории фиктивных цен параллельно: метод `forEach()` создает задачу для каждого элемента в массиве, а каждая задача будет обрабатываться пулом для объектов `ForkJoinTask`. По сути это эквивалентно тесту, приведенному в начале главы, когда пул потоков использовался для параллельного вычисления историй — хотя написать такой код намного проще, чем напрямую работать с пулом потоков.

Определение размера общего пула `ForkJoinTask` не менее важно, чем определение размера любого другого пула потоков. По умолчанию количество потоков в общем пуле равно количеству процессоров на целевых машинах. Если на одной машине запущено несколько JVM, то ограничение этого числа имеет смысл, чтобы JVM не конкурировали за процессор друг с другом. Аналогичным образом, если сервер выполняет другие запросы параллельно и вы хотите, чтобы процессор был доступен для других задач, рассмотрите возможность понижения размера общего пула. С другой стороны, если задачи общего пула блокируются в ожидании результатов ввода/вывода или других данных, то, возможно, размер пула стоит увеличить.

Размер пула задается системным свойством `-Djava.util.concurrent.ForkJoinPool.common.parallelism=`*N*. Как обычно для контейнеров Docker, в версиях Java до обновления 192 он должен задаваться вручную.

Ранее в этой главе в табл. 9.1 была показана зависимость производительности параллельных вычислений истории цен акций от размера пула. В табл. 9.9 эти данные сравниваются с конструкцией `forEach()`, использующей общий пул `ForkJoinPool` (с присваиванием системному свойству `parallelism` заданного значения).

Таблица 9.9. Время, необходимое для вычисления 10 000 фиктивных историй цен акций

Количество потоков	ThreadPoolExecutor	ForkJoinPool
1	40 ± 0,1 с	20,2 ± 0,2 с
2	20,1 ± 0,07 с	15,1 ± 0,05 с
4	10,1 ± 0,02 с	11,7 ± 0,1 с
8	10,2 ± 0,3 с	10,5 ± 0,1 с
16	10,3 ± 0,03 с	10,3 ± 0,7 с

По умолчанию общий пул содержит четыре потока (на обычной 4-процессорной машине), поэтому третья строка в таблице представляет типичный случай. Результаты для размеров 1 и 2 в точности соответствуют результатам, способным привести в бешенство специалиста по производительности: они кажутся абсолютно нелогичными, потому что `ForkJoinPool` работает намного эффективнее, чем можно было бы ожидать.

При подобных аномальных результатах самой вероятной причиной является ошибка тестирования. Однако в данном случае выясняется, что метод `forEach()`

делает кое-что неочевидное: он использует как поток, выполняющий команду, так и потоки в общем пуле для обработки данных, поступающих из потока данных. И хотя общий пул в первом тесте настроен для использования одного потока, получается, что для вычисления результата используются два потока. Соответственно время для `ThreadPoolExecutor` с двумя потоками и `ForkJoinPool` с одним потоком практически одинаково.

Если вам нужно настроить размер общего пула при использовании параллельных конструкций потоков данных и других средств автопараллелизма, подумайте о том, чтобы уменьшить желаемое значение на 1.

РЕЗЮМЕ

- Класс `ForkJoinPool` следует использовать для рекурсивных алгоритмов типа «разделяй и властвуй». Этот класс не подходит для ситуаций, в которых можно действовать простым разбиением.
- Постарайтесь определить оптимальную точку, в которой рекурсия задач в алгоритме должна остановиться. Создание слишком большого числа задач повредит производительности, но недостаточное количество задач также повредит производительности, если выполнение задач занимает разное время.
- Функции с автоматической параллелизацией совместно используют общий экземпляр класса `ForkJoinPool`. Возможно, вам придется отрегулировать размер этого экземпляра по умолчанию.

Синхронизация потоков

В идеальном мире — или в примерах этой книги — потоки без особых усилий обходятся без обязательной синхронизации. Впрочем, в реальности все обычно сложнее.

СИНХРОНИЗАЦИЯ И СРЕДСТВА ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ В JAVA

В этом разделе термин «синхронизация» относится к коду, находящемуся в блоке, в котором доступ к набору переменных выглядит строго последовательным: с этой памятью в любой момент времени может работать только один поток. К этой категории относятся блоки, защищенные ключевым словом `synchronized`. Также в нее входит код, защищенный экземпляром класса `java.util.concurrent.lock.Lock`, а также код пакетов `java.util.concurrent` и `java.util.concurrent.atomic`.

Строго говоря, атомарные классы (atomic) не используют синхронизацию — по крайней мере в контексте традиционного программирования. Атомарные классы используют команду процессора CAS (Compare and Swap), тогда как синхронизация требует монопольного доступа к ресурсу. Потоки, использующие команды CAS, не блокируются при конкурентном обращении к одному ресурсу, тогда как поток, требующий синхронизационной блокировки, будет заблокирован, если ресурс удерживается другим потоком.

Эти два механизма обладают разной производительностью (см. далее в этом разделе). Несмотря на то что команды CAS не требуют блокировки и не приостанавливают выполнение, в них проявляется большая часть поведения блокирующих конструкций: результат их выполнения создает у разработчика иллюзию того, что потоки могут обращаться к защищенной памяти только последовательно.

Затраты на синхронизацию

Синхронизируемые блоки кода влияют на производительность в двух отношениях. Во-первых, время, проводимое приложением в синхронизируемом блоке, влияет на масштабируемость приложения. Во-вторых, на захват блокировки расходуются ресурсы процессора.

Синхронизация и масштабируемость

Начнем по порядку: когда приложение разбивается для выполнения в нескольких потоках, наблюдаемый прирост скорости его выполнения определяется по формуле, называемой *законом Амдала*:

$$Ускорение = \frac{1}{(1-P)+\frac{P}{N}}\ .$$

P — доля вычислений, которая может быть выполнена параллельно, а N — количество используемых потоков (предполагается, что каждый поток всегда располагает доступными вычислительными мощностями). Таким образом, если 20% кода выполняется в последовательных блоках (то есть значение P равно 80%), можно рассчитывать, что с 8 доступными процессорами код будет выполняться (только) в 3,33 раза быстрее.

Один из ключевых аспектов этой формулы заключается в том, что с уменьшением P (то есть с выполнением большей части кода в последовательных блоках)

прирост производительности от многопоточного выполнения также уменьшается. Вот почему так важно ограничивать объем кода в последовательных блоках. В этом примере с 8 доступными процессорами вы, возможно, надеялись на восьмикратное увеличение скорости. Если только 20% кода размещается в последовательном блоке, выгода от наличия нескольких потоков уменьшилась более чем на 50% (то есть скорость возросла только в 3,3 раза).

Затраты на блокирование объектов

Кроме влияния на масштабируемость, с операцией синхронизации связаны два вида затрат.

Прежде всего это затраты на получение синхронизационной блокировки. Если конкуренция за блокировку отсутствует — то есть два потока не пытаются обратиться к блокировке конкурентно, — эти затраты будут минимальными. И здесь кроется небольшое различие между ключевым словом `synchronized` и конструкциями на основе команды CAS. Блокировки, за которые отсутствует конкуренция, называются *бесконфликтными*; затраты на получение бесконфликтной блокировки составляют несколько сотен наносекунд. Бесконфликтный код CAS сталкивается с еще меньшими затратами для быстродействия (пример приведен в главе 12).

Конфликтные конструкции обходятся дороже. Когда второй поток пытается обратиться к синхронизируемой блокировке, блокировка становится (как и следовало ожидать) конфликтной. Время захвата блокировки слегка возрастает, но настоящие последствия заключаются в том, что второму потоку придется ожидать, пока первый поток освободит блокировку. Конечно, время ожидания зависит от приложения.

Затраты на конфликтную операцию в коде, использующем команды CAS, непредсказуемы. Классы, использующие примитивы CAS, основаны на оптимистичной стратегии: поток задает значение, выполняет код, а затем проверяет, что исходное значение не изменилось. Если оно изменилось, то код на основе CAS должен выполнить код снова. В худшем случае два потока попадают в бесконечный цикл: каждый изменяет значение, защищенное CAS, только для того, чтобы увидеть, что другой поток конкурентно изменил его. На практике с двумя потоками такого бесконечного цикла не будет, но с увеличением числа потоков, конкурирующих за значение на базе CAS, количество повторных попыток возрастет.

Второй вид синхронизационных затрат относится к специфике Java и зависит от модели памяти Java. В отличие от таких языков, как C++ и C, Java предоставляет жесткие гарантии относительно семантики памяти, связанной с синхронизацией; эта гарантия распространяется на защиту на основе CAS, на традиционную синхронизацию и на ключевое слово `volatile`.

ИСПОЛЬЗОВАНИЕ VOLATILE В ПРИМЕРАХ

В примере блокировки с двойной проверкой из главы 7 было необходимо использовать переменную с ключевым словом volatile:

```
private volatile ConcurrentHashMap instanceChm;
...
public void doOperation() {
    ConcurrentHashMap chm = instanceChm;
    if (chm == null) {
        synchronized(this) {
            chm = instanceChm;
            if (chm == null) {
                chm = new ConcurrentHashMap();
                ... код заполнения коллекции
                instanceChm = chm;
            }
        }
    }
    ...Использование chm...
}
```

Ключевое слово `volatile` в этом примере решает две задачи. Во-первых, обратите внимание на то, что коллекция сначала инициализируется с использованием локальной переменной, и только итоговое (полностью инициализированное) значение присваивается переменной `instanceChm`. Если бы код, заполняющий коллекцию, использовал переменную экземпляра напрямую, то второй поток мог бы «увидеть» частично заполненную коллекцию. Кроме того, такое решение гарантирует, что при полной инициализации карты другие потоки немедленно «увидят» это значение, сохраненное в переменной экземпляра `instanceChm`.

Также в главе 2 было указано, что в микробенчмарках одним из способов гарантировать использование результата в программе было сохранение его в volatile-переменной. Такое сохранение не может оптимизироваться компилятором, поэтому оно гарантирует, что код, который сгенерировал результат, сохраненный в переменной, тоже не будет исключен в результате оптимизации. Класс `Blackhole` в jmh использует `volatile` для той же цели (хотя этот класс также предотвращает ряд других эффектов микробенчмарков, и это еще одна причина, по которой использование jmh предпочтительнее написания собственных микробенчмарков с нуля).

Цель синхронизации заключается в защите обращений к значениям (или переменным) в памяти. Как упоминалось в главе 4, переменные могут временно храниться в регистрах. Обращения к регистрам намного эффективнее прямых

обращений к ним в основной памяти. Значения регистров не видны другим потокам; поток, изменивший значение в регистре, должен в какой-то момент записать содержимое регистра в основную память, чтобы другие потоки смогли получить доступ к значению. Момент времени, в который значения регистров должны записываться в память, определяется синхронизацией потоков.

Семантика становится достаточно сложной, но проще всего считать, что при выходе из синхронизируемого блока поток должен записать все измененные переменные в основную память. Это означает, что другие потоки, входящие в синхронизируемый блок, увидят самые последние обновленные значения. Аналогичным образом конструкции на основе CAS гарантируют, что переменные, измененные во время их выполнения, будут записаны в основную память, а переменная с пометкой `volatile` всегда правильно обновляется в основной памяти при изменении.

В главе 1 я упоминал о том, что вы должны научиться избегать непроизводительных конструкций в Java, даже если на первый взгляд это кажется «преждевременной оптимизацией» кода (такое впечатление ошибочно). Интересный и вполне реалистичный пример встречается в следующем цикле:

```
Vector v;
for (int i = 0; i < v.size(); i++) {
    process(v.get(i));
}
```

В рабочем коде этот цикл занимает неожиданно много времени, и вроде бы логично предположить, что виновником является метод `process()`. Однако это не так; проблема даже не связана с тем, что методы `size()` и `get()` вызывают сами себя (компилятор встраивает эти вызовы). Методы `get()` и `size()` класса `Vector` синхронизированы, и, как выясняется, запись данных из регистров в память, необходимая для этих вызовов, создает большие проблемы с производительностью[1].

Этот код не идеален и по другим причинам. В частности, состояние вектора может измениться между тем моментом, когда поток вызывает метод `size()`, и моментом вызова метода `get()`. Если второй поток удалит последний элемент из вектора между двумя вызовами из первого потока, метод `get()` выдаст исключение `ArrayIndexOutOfBoundsException`. Даже если не считать семантических проблем с кодом, излишне детализированная синхронизация в данном случае была не лучшим выбором.

[1] Хотя в современном коде будет использоваться другой класс коллекции, суть примера останется неизменной с любой коллекцией-оберткой для метода `synchronizedCollection()` или любого цикла с избыточной очисткой регистров.

Для устранения этой проблемы можно написать несколько последовательных детализированных вызовов синхронизации в блоке `synchronized`:

```
synchronized(v) {
    for (int i = 0; i < v.size(); i++) {
        process(v.get(i));
    }
}
```

Такое решение плохо работает, если метод `process()` занимает много времени, потому что вектор уже не будет обрабатываться параллельно. Возможно, также понадобится скопировать и разделить вектор, чтобы его элементы могли обрабатываться параллельно с копиями, тогда как другие потоки в это время смогут изменять исходный вектор.

Эффект выгрузки регистров в память также зависит от того, на каком процессоре выполняется программа; процессоры с большим количеством регистров на поток потребуют больше выгрузки, чем простые процессоры. Этот код долгое время выполнялся без каких-либо проблем в тысячах разных сред. Проблемы возникли только тогда, когда он запускался на большой машине на базе SPARC с множеством регистров на поток.

Означает ли это, что вы не столкнетесь с проблемами из-за выгрузки регистров в меньших средах? Возможно. Но в наши дни, когда многоядерные процессоры стали нормой на простых ноутбуках, более сложные процессоры с большими объемами кэшей и количеством регистров также получат более широкое распространение, и при этом могут проявиться скрытые проблемы с производительностью.

РЕЗЮМЕ

- Синхронизация потоков связана с двумя видами затрат производительности: она ограничивает масштабируемость приложения и требует захвата блокировок.
- Семантика работы с памятью при синхронизации, средства на базе CAS и ключевое слово `volatile` могут отрицательно повлиять на производительность, особенно на больших машинах с множеством регистров.

Предотвращение синхронизации

Если вам удастся полностью отказаться от синхронизации, то потери, связанные с блокировками, не повлияют на производительность приложения. Для достижения этой цели можно воспользоваться двумя обобщенными решениями.

Во-первых, можно использовать в разных потоках разные объекты, чтобы при обращении к объектам не было конфликтов. Многие объекты Java синхронизируются для обеспечения потоковой безопасности, но они не обязаны использоваться совместно. Класс `Random` относится к этой категории; в главе 12 был приведен пример из JDK, в котором потоково-локальные средства использовались при разработке нового класса для предотвращения синхронизации в этом классе.

С другой стороны, создание многих объектов Java требует высоких затрат ресурсов памяти или значительных объемов памяти. Возьмем хотя бы класс `NumberFormat`: экземпляры этого класса не являются потоково-безопасными, а интернационализация, необходимая для создания экземпляра, повышает затраты на создание новых объектов. Программа может обойтись простым совместно используемым глобальным экземпляром `NumberFormat`, но обращение к этому объекту должно быть синхронизировано.

Вместо этого лучше воспользоваться объектом `ThreadLocal`:

```
public class Thermometer {
    private static ThreadLocal<NumberFormat> nfLocal = new ThreadLocal<>() {
        public NumberFormat initialValue() {
            NumberFormat nf = NumberFormat.getInstance();
            nf.setMinumumIntegerDigits(2);
            return nf;
        }
    }
    public String toString() {
        NumberFormat nf = nfLocal.get();
        nf.format(...);
    }
}
```

Использование потоково-локальной переменной ограничивает общее количество объектов (что приводит к минимизации последствий для уборки мусора), и объекты никогда не будут подвержены конкуренции со стороны потоков.

Второй способ предотвращения синхронизации основан на использовании альтернативных решений на основе CAS. В каком-то смысле речь идет не о предотвращении синхронизации, а скорее о другом решении задачи. Однако в данном контексте за счет сокращения потерь на синхронизацию эффект по сути оказывается тем же.

Казалось бы, различия в производительности между защитой на основе CAS и традиционной синхронизацией предоставляют идеальный случай для применения микробенчмарков: написание кода, сравнивающего операцию на основе CAS с традиционным методом `synchronized`, должно быть тривиальным. Например, JDK предоставляет простой механизм счетчика с CAS-защитой: `AtomicLong` и аналогичные классы. Тогда микробенчмарк может сравнить код, использую-

щий защиту на основе CAS, с традиционной синхронизацией. Допустим, поток должен получить глобальный индекс и выполнить его атомарное увеличение (чтобы следующий поток получил следующий индекс). При использовании операций на основе CAS это делается так:

```
AtomicLong al = new AtomicLong(0);
public long doOperation() {
    return al.getAndIncrement();
}
```

Версия этой операции с традиционной синхронизацией выглядит так:

```
private long al = 0;
public synchronized doOperation() {
    return al++;
}
```

Как выясняется, различия между этими двумя реализациями невозможно измерить при помощи микробенчмарков. Если поток только один (что исключает саму возможность конфликта), микробенчмарк, использующий этот код, может выдать разумную оценку затрат при использовании двух разных решений в бесконфликтной среде (результат этого теста описан в главе 12). Но этот результат не дает никакой информации относительно того, что происходит в среде с конкуренцией (а если в коде нет конкуренции, его изначально незачем было делать потоково-безопасным).

В микробенчмарке, построенном на основе этих фрагментов кода, который запускается всего с двумя потоками, будет существовать огромная конкуренция за общий ресурс. Кроме того, такая ситуация выглядит нереалистично: маловероятно, чтобы в реальном приложении два потока обращались к общему ресурсу конкурентно. Добавление новых потоков только повышает уровень нереалистичной конкуренции.

Как обсуждалось в главе 2, результаты микробенчмарков обычно сильно завышают влияние узких мест синхронизации в тестируемой ситуации. Данное обсуждение идеально поясняет это замечание. При использовании этого кода в реальном приложении можно было бы получить намного более реалистичную картину.

В общем случае при сравнении производительности средств на основе CAS с традиционной синхронизацией стоит руководствоваться следующими рекомендациями:

- Если обращения к ресурсу обходятся без конкуренции, защита на основе CAS работает чуть быстрее традиционной синхронизации. Впрочем, при полном отсутствии конкуренции отсутствие какой-либо защиты работает

еще быстрее и позволяет обойти граничные случаи вроде того, который был продемонстрирован при выгрузке регистров для класса `Vector`.

- Если обращения к ресурсу сопряжены с незначительной или умеренной конкуренцией, защита на основе CAS будет быстрее (часто намного быстрее) традиционной синхронизации.
- При наличии серьезной конкуренции за обращения к ресурсу традиционная синхронизация в какой-то момент становится более эффективным вариантом. На практике это происходит только на очень больших машинах с множеством потоков.
- Защита на основе CAS не подвержена конкуренции, если значения только читаются, но не записываются.

КОНФЛИКТНЫЕ АТОМАРНЫЕ КЛАССЫ

Классы в пакете `java.util.concurrent.atomic` используют примитивы на основе CAS вместо традиционной синхронизации. В результате такие классы (например, `AtomicLong`) обычно работают быстрее специально написанных синхронизированных методов для увеличения переменной long — по крайней мере до того, как конкуренция за примитив CAS станет слишком высокой.

В Java имеются классы для такой ситуации, когда слишком много потоков конкурирует за обращение к примитивным атомарным значениям: атомарные сумматоры и аккумуляторы (например, класс `LongAdder`). Эти классы масштабируются лучше традиционных атомарных классов. Когда `LongAdder` обновляется несколькими потоками, класс может хранить обновления отдельно для каждого потока. Потокам не нужно ждать, пока другие потоки завершат свои операции; вместо этого значения сохраняются (фактически) в массиве, и каждый поток может быстро вернуть управление. Позднее значения будут суммироваться или аккумулироваться, когда поток попытается получить текущее значение.

При минимальной конкуренции значение аккумулируется в процессе работы программы, а поведение сумматора будет таким же, как у традиционного атомарного класса. При жесткой конкуренции обновления будут происходить намного быстрее, хотя экземпляр начнет использовать больше памяти для хранения массива значений. Выборка значения в этом случае также немного замедлится, потому что она должна обработать все незавершенные обновления в массиве. В условиях очень высокой конкуренции эти новые классы будут работать еще лучше, чем их атомарные аналоги.

В конечном итоге ничто не заменит тщательного тестирования в реальной среде, в которой будет выполняться код: только после него можно будет точно утверждать, какая реализация определенного метода лучше. Впрочем, даже в этом случае конкретное утверждение будет относиться только к этим условиям.

РЕЗЮМЕ

- Предотвращение конкуренции за синхронизируемые объекты — хороший способ подавления их влияния на производительность.
- Потоково-локальные переменные никогда не подвержены конкуренции; они идеально подходят для хранения синхронизированных объектов, которые не должны совместно использоваться разными потоками.
- Средства на основе CAS позволяют избежать традиционной синхронизации объектов, не предназначенных для совместного использования.

Ложное совместное использование данных

У синхронизации есть одно следствие для производительности, о котором редко говорят: речь идет о *ложном совместном использовании данных* (также называемом *совместным использованием строк кэша*). Когда-то оно было довольно экзотическим артефактом многопоточного программирования, но по мере того, как многоядерные машины стали нормой — и были решены другие, более очевидные проблемы производительности синхронизации, — ложное совместное использование данных стало играть все более важную роль.

Эффект ложного совместного использования данных возникает из-за особенностей работы процессора с кэшем. Возьмем следующий простой класс:

```
public class DataHolder {
    public volatile long l1;
    public volatile long l2;
    public volatile long l3;
    public volatile long l4;
}
```

Все значения `long` хранятся в памяти по соседству друг с другом; например, `l1` может храниться по адресу 0xF20. Тогда `l2` будет храниться в памяти по адресу 0xF28, `l3` — по адресу 0xF2C и т. д. Когда программе потребуется работать с `l2`, она загрузит относительно большой объем памяти — например, 128 байт от адреса 0xF00 до 0xF80 — в строку кэша одного из ядер процессора. Второй по-

ток, который захочет работать с l3, загрузит тот же блок памяти в строку кэша на другом ядре.

В большинстве случаев подобная загрузка соседних значений выглядит разумно: если приложение обращается к одной конкретной переменной экземпляра в объекте, она также с большой вероятностью обратится к ближайшим переменным экземпляра. Если эти переменные уже загружены в кэш ядра, это обращение к памяти будет очень, очень быстрым — с заметным выигрышем по производительности.

Однако у этой схемы есть один недостаток: каждый раз, когда программа обновляет значение в своем локальном кэше, это ядро должно уведомить все остальные ядра о том, что содержимое памяти изменилось. Другие ядра должны сделать свои строки кэша недействительными и заново загрузить данные из памяти.

Посмотрим, что произойдет при интенсивном использовании класса DataHolder несколькими потоками:

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
public class ContendedTest {
    private static class DataHolder {
        private volatile long l1 = 0;
        private volatile long l2 = 0;
        private volatile long l3 = 0;
        private volatile long l4 = 0;
    }
    private static DataHolder dh = new DataHolder();

    Thread[] threads;

    @Setup(Level.Invocation)
    public void setup() {
        threads = new Thread[4];
        threads[0] = new Thread(() -> {
                for (long i = 0; i < nLoops; i++) {
                        dh.l1 += i;
                }
        });
        threads[1] = new Thread(() -> {
                for (long i = 0; i < nLoops; i++) {
                        dh.l2 += i;
                }
        });
        //...аналогично для 2 и 3...
    }

    @Benchmark
    public void test(Blackhole bh) throws InterruptedException {
        for (int i = 0; i < 4; i++) {
            threads[i].start();
```

```
        }
        for (int i = 0; i < 4; i++) {
            threads[i].join();
        }
    }
}
```

Имеются четыре разных потока, которые не используют переменные совместно: каждый обращается только к одному полю класса `DataHolder`. С точки зрения синхронизации никакой конкуренции нет, и можно с разумными основаниями ожидать, что код будет выполняться (на четырехъядерной машине) за то же время независимо от того, сколько потоков используется — один или четыре.

На практике все обстоит не так. Когда один конкретный поток записывает `volatile`-значение в цикле, строка кэша каждого из остальных потоков становится недействительной и значения приходится перезагружать из памяти. Результат показан в табл. 9.10: с добавлением новых потоков производительность ухудшается.

Таблица 9.10. Время суммирования 100 000 значений с ложным совместным использованием данных

Количество потоков	Затраченное время
1	0,8 ± 0,001 мс
2	5,7 ± 0,3 мс
3	10,4 ± 0,6 мс
4	15,5 ± 0,8 мс

Этот тест был спроектирован специально для демонстрации самых жестких потерь из-за ложного совместного использования данных: фактически с каждой записью все остальные строки кэша становятся недействительными, а производительность изменяется последовательно.

Строго говоря, ложное совместное использование не обязательно включает синхронизированные (или `volatile`) переменные: каждый раз при записи любого значения данных в кэше процессора другие кэши, содержащие тот же диапазон данных, должны быть объявлены недействительными. Однако стоит напомнить, что модель памяти Java требует, чтобы данные записывались в основную память только в конце примитива синхронизации (включая конструкции CAS и `volatile`). Именно в этой ситуации данная проблема будет возникать чаще всего. Если бы в приведенном примере переменные `long` не были объявлены `volatile`, то компилятор хранил бы значения в регистрах, а тест выполнялся бы приблизительно за 0,7 мс независимо от количества задействованных потоков.

Очевидно, в примере представлен крайний случай, но он поднимает вопрос о том, как обнаружить и исправить ложное совместное использование данных. К сожалению, ответ получается неясным и неполным. Никакие средства из стандартного набора, описанного в главе 3, не решают проблемы ложного совместного использования, потому что она требует знания архитектуры процессора.

Если повезет, производитель целевого процессора для вашего приложения предоставит программу для диагностики ложного совместного использования. У Intel, например, есть программа VTune Amplifier, позволяющая обнаружить ложное совместное использование анализом промахов кэша. Некоторые низкоуровневые профилировщики предоставляют информацию о количестве тактовых сигналов на команду (CPI) для заданной строки кода; высокое значение CPI для простой команды в цикле может означать, что код ожидает повторной загрузки целевой памяти в кэш процессора.

В противном случае обнаружение ложного совместного использования требует интуиции и экспериментов. Если обычный профиль указывает, что конкретный цикл занимает неожиданно большое время, проверьте, не обращаются ли несколько потоков к переменным, не рассчитанным на совместное использование, в цикле. (В области производительности настройка скорее является искусством, нежели наукой, хотя в руководстве Intel VTune Amplifier сказано, что «основным средством предотвращения ложного совместного использования является анализ кода».)

Предотвращение ложного совместного использования требует изменений в коде. В идеальной ситуации можно снизить частоту записи соответствующих переменных. В нашем примере в вычислениях могут быть задействованы локальные переменные, и в переменную `DataHolder` записывается только конечный результат. При очень малом количестве операций записи вряд ли возникнет конкуренция за строки кэша, и они не будут иметь последствий для производительности, даже если все четыре потока конкурентно обновляют свои результаты в конце цикла.

Во втором варианте переменные выравниваются в памяти так, чтобы они не могли быть загружены в одну строку кэша. Если целевой процессор имеет 128-байтовые строки кэша, может сработать следующее выравнивание (хотя может и не сработать):

```
public class DataHolder {
    public volatile long l1;
    pubilc long[] dummy1 = new long[128 / 8];
    public volatile long l2;
    pubilc long[] dummy2 = new long[128 / 8];
    public volatile long l3;
    pubilc long[] dummy3 = new long[128 / 8];
    public volatile long l4;
}
```

Такое использование массивов вряд ли сработает, потому что JVM может изменить структуру переменных экземпляров, чтобы все массивы располагались рядом друг с другом в одном блоке, а все переменные `long` — в другом блоке. Использование примитивных значений для дополнения полей структуры скорее сработает, хотя оно может быть непрактичным из-за количества требуемых переменных.

При использовании дополнений для предотвращения ложного совместного использования также необходимо учитывать другие аспекты. Размер дополнения трудно спрогнозировать, потому что разные процессоры имеют разные размеры кэша. Кроме того, дополнения значительно увеличивают размер экземпляров, что неизбежно повлияет на уборку мусора (конечно, это зависит от количества требуемых экземпляров). Но при отсутствии алгоритмического решения выравнивание данных посредством дополнения их в памяти часто приносит значительную пользу.

АННОТАЦИЯ @CONTENDED

У приватных классов JDK предусмотрена возможность для сокращения конкуренции за кэш для отдельных полей (JEP 142). Для этого переменные, которые должны автоматически дополняться JVM в памяти, помечаются аннотацией `@sun.misc.Contended`.

Предполагается, что эта аннотация должна быть приватной. В Java 8 она входит в пакет `sun.misc`, хотя ничто не мешает вам использовать этот пакет в своем коде. В Java 11 он принадлежит пакету `jdk.internal.vm.annotation`, а с системой модулей в Java 11, классы, использующие этот пакет, не могут компилироваться без флага `-add-exports`; этот флаг добавляет пакет в набор классов, экспортируемых модулем `java.base`.

По умолчанию JVM игнорирует эту аннотацию везде, кроме классов JDK. Чтобы эта аннотация могла использоваться в коде приложения, добавьте флаг `-XX:-RestrictContended`, который по умолчанию равен `true` (это означает, что аннотация ограничивается классами JDK).

С другой стороны, чтобы отключить автоматическое дополнение, применяемое в JDK, установите флаг `-XX:-EnableContended`, который по умолчанию равен `true`. Это приведет к сокращению размера классов `Thread` и `ConcurrentHashMap`, использующих эту аннотацию для дополнения в памяти своих реализаций с целью защиты от ложного совместного использования данных.

РЕЗЮМЕ

- Ложное совместное использование данных может значительно замедлить выполнение кода, который часто изменяет `volatile`-переменные или выходит из синхронизируемых блоков.
- Обнаружить ложное совместное использование сложно. Если вам кажется, что цикл занимает слишком много времени, проанализируйте код и посмотрите, не построен ли он по схеме, в которой может происходить ложное совместное использование.
- Для предотвращения ложного совместного использования лучше всего переместить данные в локальные переменные и сохранить их позднее. Также иногда можно воспользоваться дополнением данных в памяти для перемещения конфликтующих переменных в разные строки кэша.

Настройки потоков JVM

В JVM предусмотрен ряд настроек, влияющих на производительность потоков и синхронизации. Эти настройки оказывают второстепенное влияние на производительность приложений.

Настройка размеров стеков потоков

В средах с крайне ограниченной памятью можно отрегулировать объем памяти, используемой потоками. У каждого потока имеется низкоуровневый стек, в котором ОС хранит информацию стека вызовов потока (например, тот факт, что метод `main()` вызвал метод `calculate()`, который вызвал метод `add()`).

Размер низкоуровневого стека составляет 1 Мбайт (за исключением JVM для 32-разрядных версий JVM, в которых он составляет 320 Кбайт). В 64-разрядной JVM обычно нет причин задавать это значение, разве что на машине сильно не хватает физической памяти, а уменьшение размера стека не позволит приложению исчерпать всю низкоуровневую память. Это относится прежде всего к контейнерам Docker с их ограниченной памятью.

Простое практическое правило: многие программы могут выполняться с размером стека 256 Кбайт, некоторым требуется полный 1 Мбайт. Потенциальный недостаток слишком низкого значения состоит в том, что поток с очень большим стеком вызовов выдаст исключение `StackOverflowError`.

НЕХВАТКА НИЗКОУРОВНЕВОЙ ПАМЯТИ

Ошибка `OutOfMemoryError` может возникнуть в том случае, если для создания потока не хватает низкоуровневой памяти. Это может свидетельствовать об одной из трех ситуаций:

- В 32-разрядной JVM процесс достиг своего максимального размера 4 Гбайт (или менее в зависимости от ОС).
- В системе не хватает виртуальной памяти.
- В системах семейства Unix пользователь уже создал (во всех программах, которые он запустил) максимальное количество процессов, настроенное для его учетной записи. Отдельные потоки в этом отношении рассматриваются как процессы.

Сокращение размера стека может решить первые две проблемы, но не поможет с третьей. К сожалению, по ошибке JVM невозможно понять, с каким из трех случаев вы столкнулись; при возникновении такой ошибки рассмотрите все три возможности.

Чтобы изменить размер стека для потока, используйте флаг `-Xss=`*N* (например, `-Xss=256k`).

РЕЗЮМЕ

- Размеры стека потоков можно уменьшить на машине с недостаточным объемом памяти.

Смещенная блокировка

Когда за блокировки возникает конкуренция, JVM (и операционная система) могут принимать решения относительно того, как должна выделяться блокировка. Блокировка может предоставляться справедливо, это означает, что блокировка будет предоставляться всем потокам по круговому принципу. Также возможен другой вариант: предпочтение может отдаваться потоку, который последним обращался к блокировке.

В основе привязанной блокировки лежит следующая теория: если поток недавно обращался к блокировке, то кэш процессора с большей вероятностью будет содержать данные, которые понадобятся потоку при следующем выполнении кода, защищенного той же блокировкой. Если поток получает приоритет для

повторного получения блокировки, вероятность попаданий в кэш возрастает. Если это условие выполняется, производительность улучшается. Но поскольку смещенная блокировка требует хранения дополнительной информации, иногда она ухудшает производительность.

В частности, приложения, использующие пул потоков — включая некоторые приложения и серверы REST, — часто работают хуже при смещенной блокировке. В этой модели программирования предполагается, что разные потоки с равной вероятностью обращаются к блокировкам. В таких приложениях можно добиться небольшого прироста производительности за счет отключения смещенной блокировки при помощи флага `-XX:-UseBiasedLocking`. По умолчанию смещенная блокировка включена.

Приоритеты потоков

Каждый поток Java имеет *приоритет*, назначенный разработчиком. Он указывает операционной системе, насколько (с точки зрения программы) важен данный поток. Если разные потоки выполняют разные операции, можно подумать, что приоритет потока можно использовать для повышения производительности некоторых задач за счет других задач, выполняемых потоками с меньшим приоритетом. К сожалению, система приоритетов работает немного не так.

Операционная система вычисляет *текущий приоритет* для каждого потока, выполняемого на машине. Текущий приоритет учитывает приоритет, назначенный Java, но также включает ряд других факторов, самым главным из которых является время с момента последнего выполнения потока. Это гарантирует, что все потоки получат возможность выполниться в какой-то момент времени. Никакой поток не будет подолгу простаивать в ожидании доступа к процессору независимо от приоритета.

Баланс между этими двумя факторами зависит от операционной системы. В системах семейства Unix при вычислении общего приоритета определяющим фактором является время с момента последнего выполнения потока — приоритет потока уровня Java особой роли не играет.

В системе Windows потоки с более высоким приоритетом Java обычно получают больше времени, чем потоки с низким приоритетом, но даже низкоприоритетным потокам достается достаточное процессорное время.

В любом случае вы не можете рассчитывать на приоритет потока, чтобы повлиять на частоту его выполнения. Если некоторые задачи важнее других задач, их приоритетное выполнение должно обеспечиваться логикой приложения.

Мониторинг потоков и блокировок

При анализе производительности приложения в контексте эффективности многопоточности и синхронизации следует учитывать два фактора: общее количество потоков (оно должно быть не слишком большим и не слишком малым) и время, проводимое потоками в ожидании блокировки или другого ресурса.

Видимость потоков

Практически каждое средство мониторинга JVM предоставляет информацию о количестве потоков (и о том, чем они занимаются). Интерактивные программы (такие, как `jconsole`) отображают состояние потоков в JVM. На панели Threads программы `jconsole` можно в реальном времени следить за ростом и уменьшением количества потоков в ходе выполнения программы. Пример представлен на рис. 9.2.

В какой-то момент приложение (NetBeans) использовало максимум 45 потоков. В начале графика виден всплеск, при котором приложение использовало до 38 потоков, но потом оно стабилизировалось на уровне 30–31. `jconsole` также выводит отдельный стек потока; как видно из иллюстрации, поток Java2D Disposer в настоящее время ожидает блокировки для очереди ссылок.

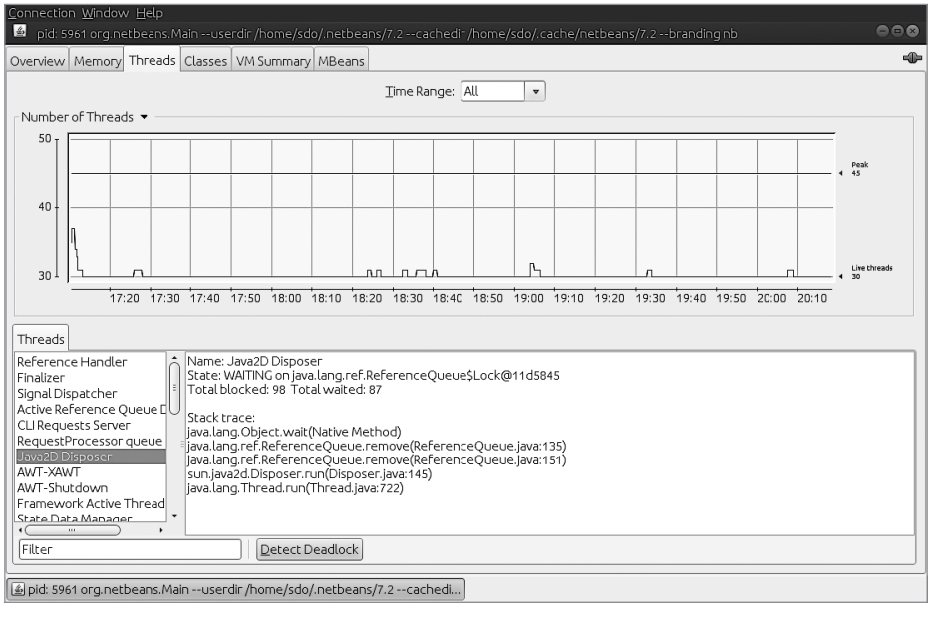


Рис. 9.2. Вывод информации о потоках в jconsole

Вывод информации о заблокированных потоках

Мониторинг потоков в реальном времени позволяет получить высокоуровневое представление о потоках, выполняемых в приложении, но не дает никакой информации о том, чем этим потоки занимаются. Чтобы определить, где потоки проводят процессорное время, необходимо воспользоваться профилировщиком (см. главу 3). Профилировщики дают полную информацию о том, какие потоки выполняются. Обычно они достаточно совершенны, для того чтобы вы могли найти области кода, в которых более разумный выбор алгоритмов и кода позволит ускорить выполнение приложения в целом.

Сложнее проводить диагностику заблокированных потоков, хотя эта информация часто играет важную роль в общем выполнении приложения — особенно если этот код работает в многопроцессорной системе и не использует все доступные процессоры.

Для выполнения этой диагностики можно воспользоваться тремя способами. Во-первых, вы можете снова воспользоваться профилировщиком, так как в большинстве средств профилирования выводится временная шкала выполнения потоков, по которой можно определить, в каких точках поток был заблокирован. Пример приведен в главе 3.

Заблокированные потоки и JFR

Несомненно, лучший способ выявления заблокированных потоков — использование программ, которые могут обратиться к JVM и низкоуровневыми средствами определить, когда поток будет заблокирован. Одним из таких инструментов является механизм Java Flight Recorder, описанный в главе 3. Разработчик может раскрывать информацию событий, которую сохраняет JFR, и искать события, вызывающие блокировку потока. Как правило, следует искать потоки, ожидающие получения монитора, но если понаблюдать за потоками с долгими операциями чтения (и изредка долгими операциями записи) в сокет, они с большой вероятностью также будут заблокированы. Эти события можно просматривать на панели Histogram консоли Java Mission Control (рис. 9.3).

В этом примере блокировка, связанная с `HashMap` в методе `sun.awt.AppContext.get()`, оспаривалась 163 раза (за 66 секунд), что привело к среднему росту времени отклика для запроса на 31 мс. Из трассировки стека следует, что конкуренция возникает из-за особенностей записи объекта `java.util.Date` в JSP. Для улучшения масштабируемости кода можно воспользоваться потоково-локальным форматировщиком даты (вместо простого вызова метода `toString()` объекта даты).

Этот процесс — выбор блокирующего события на гистограмме и анализ вызывающего кода — работает с любыми разновидностями блокирующих событий; он становится возможным благодаря тесной интеграции JFR с JVM.

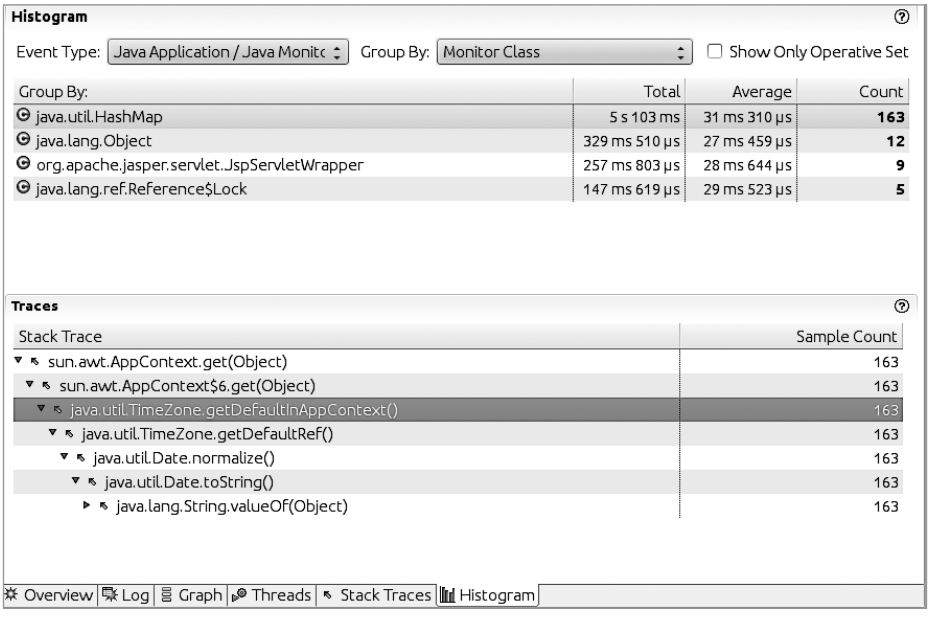


Рис. 9.3. Потоки, заблокированные по монитору в JFR

Заблокированные потоки и JStack

Если запись данных JFR программы недоступна, то альтернативой становится сохранение стеков потоков из программы с их последующим анализом. `jstack`, `jcmd` и другие программы могут предоставлять информацию о состоянии каждого потока в VM: выполнение, ожидание захвата блокировки, ожидание ввода/вывода и т. д. Эта информация может весьма пригодиться для определения того, что происходит в приложении (если только не ожидать от нее слишком многого).

Первое, что необходимо учитывать при анализе стеков потоков — JVM может сохранять стеки потоков только в точках безопасного состояния. Во-вторых, стеки для каждого потока сохраняются последовательно, поэтому полученная из них информация может быть противоречивой: два потока могут удерживать одну блокировку или же поток ожидает блокировку, которая не удерживается никаким другим потоком. Стеки потоков также могут показать, насколько серьезно блокируются потоки (так как заблокированный поток уже находится в безопасном состоянии). Если последовательно сохраненные дампы потоков показывают, что многие потоки находятся в ожидании блокировки, можно сделать вывод, что за эту блокировку есть значительная конкуренция. Если последовательно сохраненные дампы потоков показывают, что многие потоки блокируются в ожидании ввода/вывода, можно сделать вывод, что ввод/вы-

вод нуждается в настройке (например, если они обращаются с вызовом к базе данных, то выполняемый код SQL необходимо оптимизировать или сама база данных нуждается в настройке).

ПРОФИЛИРОВЩИК JSTACK

Появляется соблазнительная мысль: а нельзя ли сохранить несколько дампов стеков в быстрой последовательности и использовать эти данные как кустарный профилировщик? В конце концов, профилировщики с выборкой работают фактически по тому же принципу: они периодически проверяют стек выполняемого потока и на основании полученных данных экстраполируют время, проводимое в методах. Но между безопасными состояниями и ненадежными снимками состояния такой подход работает не лучшим образом; иногда можно получить общее представление о затратных методах вашего приложения, просматривая стеки потоков, но нормальный профилировщик предоставит гораздо более точную информацию.

В примерах этой книги содержится рудиментарный парсер для вывода `jstack`, который может выдать сводку состояния всех потоков из одного или нескольких дампов. К сожалению, формат вывода `jstack` может изменяться между версиями, так что создать универсальный парсер может быть сложно. Нет гарантий того, что парсер из примеров не придется изменять для вашей конкретной JVM.

Базовый вывод парсера `jstack` выглядит примерно так:

```
% jstack pid > jstack.out
% java ParseJStack jstack.out
[Неполный вывод...]
Threads in start Running
    8 threads in java.lang.Throwable.getStackTraceElement(Native
Total Running Threads: 8

Threads in state Blocked by Locks
    41 threads running in
     com.sun.enterprise.loader.EJBClassLoader.getResourceAsStream
        (EJBClassLoader.java:801)
Total Blocked by Locks Threads: 41

Threads in state Waiting for notify
    39 threads running in
     com.sun.enterprise.web.connector.grizzly.LinkedListPipeline.getTask
        (LinkedListPipeline.java:294)
    18 threads running in System Thread
```

```
Total Waiting for notify Threads: 74

Threads in state Waiting for I/O read
    14 threads running in com.acme.MyServlet.doGet(MyServlet.java:603)
Total Waiting for I/O read Threads: 14
```

Парсер обобщает информацию всех потоков и сообщает, сколько из них пребывает в том или ином состоянии. В настоящее время работают восемь потоков (они выполняют трассировку стека — затратную операцию, которой лучше избегать).

41 поток находится в ожидании по блокировке. В выходных данных указывается первый метод, не относящийся к JDK, — в данном примере это метод `GlassFish EJBClassLoader.getResourceAsStream()`. Следующим шагом должно стать обращение к трассировке стека, поиск метода и определение ресурса, по которому блокируется поток.

В этом примере все потоки блокируются в ожидании чтения одного JAR-файла, а трассировка стека этих потоков показывает, что все вызовы происходили от создания нового экземпляра парсера SAX (Simple API for XML). Как выясняется, парсер SAX может определяться динамически — включением ресурса в файл манифеста JAR-файлов приложения; это означает, что JDK должен провести поиск этих записей по всем каталогам `classpath`, пока не найдет нужные приложению (или не найдет ничего и вернется к использованию системного парсера). Так как чтение JAR-файла требует синхронизационной блокировки, все эти потоки, пытающиеся создать парсер, в итоге будут конкурировать за одну блокировку, что очень сильно отразится на производительности приложения. (Чтобы справиться с этой проблемой, установите свойство `-Djavax.xml.parsers.SAXParserFactory` для запрета такого поиска.)

Итак, большое количество заблокированных потоков ухудшает производительность. Какой бы ни была причина блокирования потоков, необходимо внести изменения в конфигурацию или код приложения для ее предотвращения.

Как насчет потоков, ожидающих уведомления? Такие потоки часто объединяются в пул, в котором они ожидают уведомления о готовности задачи (например, метод `getTask()` в предыдущем выводе ожидает запроса). Системные потоки выполняют такие операции, как распределенная уборка мусора RMI или мониторинг JMX, — они отображаются в выводе `jstack` как потоки, в стеке которых присутствуют только классы JDK. Эти условия не обязательно указывают на проблему с производительностью; ожидание уведомлений для них нормально.

Другая проблема встречается в потоках, ожидающих результатов чтения: они выполняют блокирующий вызов ввода/вывода (обычно метод `socketRead0()`). В такой ситуации производительность также страдает: поток ожидает, пока внешний ресурс ответит на его запрос. Здесь нужно переходить к анализу производительности базы данных или другого внешнего ресурса.

РЕЗЮМЕ

- Основные средства получения информации о потоках в системе предоставляют информацию о количестве работающих потоков.
- Информация о потоке позволяет определить причину блокировки потока: например, ожидание ресурса или завершения ввода/вывода.
- Java Flight Recorder предоставляет простые средства для анализа событий, которые стали причиной блокировки потока.
- jstack предоставляет информацию о ресурсах, по которым блокируются потоки.

Итоги

Понимая, как работают потоки, вы сможете добиться заметных улучшений производительности. Впрочем, производительность потоков не сводится к настройке — количество флагов у JVM относительно невелико, и эти флаги оказывают ограниченный эффект.

Скорее, хорошая производительность потоков определяется соблюдением рекомендаций по управлению количеством потоков и ограничению эффектов синхронизации. Соответствующие средства профилирования и анализа блокировок помогут вам проанализировать приложения и внести в них необходимые изменения, чтобы проблемы многопоточности и управления блокировками не снижали производительности.

ГЛАВА 10

Серверы Java

В этой главе рассматриваются некоторые аспекты, связанные с серверными технологиями Java. По сути эти технологии обеспечивают передачу данных (обычно по протоколу HTTP) между клиентами и серверами. Следовательно, основное место в этой главе будут занимать вопросы, общие для серверных технологий: масштабирование серверов с разными потоковыми моделями, асинхронные ответы, асинхронные запросы и эффективная обработка данных JSON.

Масштабирование серверов в основном сводится к эффективному использованию потоков, а это требует неблокирующего ввода/вывода, управляемого событиями. Традиционные серверы EE Java/Jakarta — такие, как Apache Tomcat, IBM WebSphere Application Server и Oracle WebLogic Server, — в течение довольно долгого времени использовали для этой цели Java NIO API. Современные серверные фреймворки — такие, как Netty и Eclipse Vert.x, — инкапсулируют сложность Java NIO API, чтобы предоставить простые и удобные структурные элементы для построения серверов с уменьшенными затратами памяти. Кроме того, этими фреймворками пользуются такие серверы, как Spring WebFlux и Helidon (оба используют Netty).

Более новые фреймворки предлагают программные модели, основанные на концепции реактивного программирования. Реактивное программирование базируется на обработке асинхронных потоков данных с использованием событийной парадигмы. Хотя реактивное программирование рассматривает события под несколько иным углом, для наших целей реактивное и асинхронное программирование обеспечивает одно преимущество из области производительности: возможность масштабирования программ (и особенно ввода/вывода) для многих подключений или источников данных.

Java NIO

Если вам известно, как работает неблокирующий ввод/вывод, можете переходить к следующему разделу. Если нет — в этом разделе приведено краткое

объяснение того, как работает неблокирующий ввод/вывод и почему он играет такую важную роль в этой главе.

В ранних версиях Java весь ввод/вывод был блокирующим. Поток, пытающийся прочитать данные из сокета, должен ждать (блокироваться) до появления данных или тайм-аута операции чтения. Что еще важнее, определить, имеются ли в сокете данные, можно только одним способом — попытавшись выполнить чтение из сокета. Таким образом, поток, который хотел обработать данные по клиентскому подключению, должен был выдать запрос на чтение данных, блокироваться до появления данных, обработать запрос и отправить ответ, а затем вернуться к блокирующему чтению из сокета. Ситуация изображена на рис. 10.1.

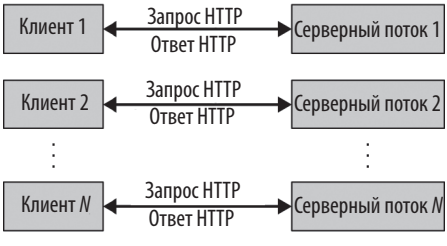


Рис. 10.1. Блокирование потоков при вводе/выводе

Блокирующий ввод/вывод требует однозначного соответствия между клиентскими подключениями и серверными потоками; каждый поток обрабатывает только одно подключение. Это особенно важно для клиентов, которые используют постоянные HTTP-подключения для предотвращения потерь производительности при создании нового сокета с каждым запросом. Допустим, 100 клиентов отправляют запросы со средними 30-секундными интервалами и серверу требуется 500 мс для обработки запроса. В этом случае в любой момент времени в среднем будут обрабатываться два запроса, но для обработки всех клиентов серверу понадобится 100 потоков. Такое соотношение крайне неэффективно.

Поэтому когда в Java появился неблокирующий NIO API, серверные фреймворки перешли на эту модель для работы с клиентами. Новая схема обработки изображена на рис. 10.2.

Теперь сокет, ассоциированный с каждым клиентом, регистрируется у селектора на сервере (селектор представляет собой экземпляр класса `Selector`; он обеспечивает интерфейс к операционной системе, предоставляющий уведомления при появлении данных в сокете). Когда клиент отправляет запрос, селектор получает событие от операционной системы и оповещает поток из серверного пула потоков о том, что у конкретного клиента появились данные

ввода/вывода, которые можно прочитать. Поток читает данные от клиента, обрабатывает запрос, возвращает ответ, а затем возвращается к ожиданию следующего запроса[1].

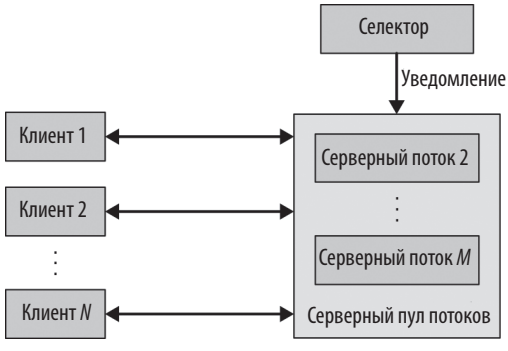


Рис. 10.2. Потоки с уведомлениями для чтения

И хотя на диаграмме представлены *N* клиентов, они обрабатываются *M* потоками. Теперь, когда клиенты уже не привязаны к конкретному серверному потоку, пул серверных потоков может быть настроен для обработки того количества конкурентных запросов, которое должно обрабатываться сервером. В примере, приведенном выше, пула потоков с размером 2 будет достаточно для обработки нагрузки от всех 100 клиентов. Если запросы могут поступать неравномерно, но без нарушения общих параметров (30-секундное время обработки), для обработки нужного количества конкурентных запросов могут понадобиться 5 или 6 потоков. Неблокирующий ввод/вывод позволяет использовать гораздо меньше потоков, чем клиентов, что обеспечивает значительный выигрыш по эффективности.

Серверные контейнеры

Масштабирование серверных подключений по многим клиентам становится первым препятствием на пути производительности сервера, которая зависит от использования неблокирующего ввода/вывода для работы с подключениями. Также важно, использует ли сервер неблокирующие функции API для других операций; эта тема будет рассмотрена позднее в этой главе, а пока мы займемся настройкой подключений.

[1] У этой схемы есть много разновидностей. Некоторые из них будут представлены в следующем разделе.

Настройка серверных пулов потоков

На современных серверах запросы, поступающие от клиентов, обрабатываются произвольным потоком из серверного пула. Следовательно, настройка пула потоков весьма важна для производительности сервера.

Как упоминалось в предыдущем разделе, серверные фреймворки различаются по схемам управления подключениями и пулами потоков. В базовой модели, описанной в тексте, один или несколько потоков действуют как селекторы: они уведомляют систему о доступности ввода/вывода и называются *селекторными потоками*. Затем отдельный пул *рабочих потоков* обрабатывает фактический запрос/ответ для клиента, после того как селектор уведомит их о наличии незавершенного ввода/вывода для клиента.

Селектор и рабочие потоки могут создаваться по разным схемам:

- Пулы рабочих потоков и селекторов могут создаваться по отдельности. Селекторы ожидают уведомлений по всем сокетам и передают запросы пулу рабочих потоков.
- Получив уведомление о вводе/выводе, селектор читает ввод/вывод (возможно, только частично) для определения информации о запросе. Затем селектор передает запрос другому пулу серверных потоков в зависимости от типа запроса.
- Пул селекторов принимает новые подключения через `ServerSocket`, но после создания подключений вся работа выполняется пулом рабочих потоков. Поток из пула рабочих потоков иногда использует класс `Selector` для ожидания необработанного ввода/вывода от существующего подключения, а иногда обрабатывает уведомления от рабочего потока о наличии необработанного ввода/вывода для клиента (например, он будет выполнять запрос/ответ для клиента).
- Между потоками-селекторами и потоками, обрабатывающими запросы, нет никаких различий. Поток, получающий уведомления о вводе/выводе, доступном в сокете, может обработать весь запрос. При этом все потоки в пуле получают уведомления о вводе/выводе в других сокетах и обрабатывают запросы для этих сокетов.

Несмотря на эти различия, есть два основных соображения, которые должны учитываться при настройке серверных пулов потоков. Первое (и самое важное): рабочих потоков должно хватать для обработки количества конкурентных запросов (не конкурентных подключений!), на которое рассчитан сервер. Как упоминалось в главе 9, это отчасти зависит от того, выполняют ли эти запросы интенсивные вычисления или же выдают другие блокирующие вызовы. Также в этом случае необходимо учитывать, что произойдет при выполнении сервером других неблокирующих вызовов.

Возьмем сервер REST, выполняющий интенсивные вычисления. Как и во всех ситуациях с высокой вычислительной нагрузкой, нет смысла создавать потоки в количестве, превышающем количество виртуальных процессоров на машине или контейнере, в котором работает сервер. Лишние потоки все равно выполняться не смогут.

А если сервер REST, в свою очередь, будет обращаться с внешними вызовами к другому ресурсу — скажем, к другому серверу REST или к базе данных? Тогда все зависит от того, являются ли эти вызовы блокирующими или неблокирующими. Предположим, вызовы являются блокирующими. Тогда для каждого конкурентного исходящего блокирующего вызова понадобится отдельный поток, и по сути сервер возвращается к неэффективной модели «один поток на клиента».

Представьте, что для выполнения конкретного запроса от клиента рабочий поток должен потратить 900 мс на получение данных из базы и еще 100 мс на подготовку обращения к базе данных и обработку данных в ответе для клиента в системе с двумя негиперпоточными процессорами. Сервер располагает достаточным количеством процессоров для обработки 20 запросов в секунду. Если запросы поступают от клиента через каждые 30 секунд, сервер сможет обслуживать 600 клиентов. Так как клиентские подключения осуществляются без блокирования, вам не понадобится пул из 600 рабочих потоков, но и обойтись всего 2 потоками (по одному на процессор) не удастся. В среднем в любой момент времени будут заблокированы 20 потоков, поэтому пул рабочих потоков должен содержать по крайней мере столько потоков.

Теперь допустим, что исходящий запрос также является неблокирующим, и за 900 мс, необходимые базе данных для возвращения ответа, поток, обратившийся к ней с вызовом, может обрабатывать другие запросы. Мы снова возвращаемся к ситуации, в которой достаточно всего двух рабочих потоков: они могут проводить все свое время за обработкой 100-миллисекундных периодов, необходимых для обработки информации из базы данных; тем самым будет обеспечена полная загрузка процессоров и поддержание эффективности сервера на максимальном уровне.

Как обычно, обсуждение несколько упрощено: также понадобится время на чтение и создание запросов и т. д. Однако основное правило остается неизменным: рабочий пул должен содержать по крайней мере столько потоков, сколько необходимо для конкурентного выполнения кода и конкурентного блокирования по другим ресурсам.

Другой аспект настройки — количество потоков, которые должны выполнять функции селекторов в любой отдельный момент времени. Одного потока для этого недостаточно. Поток-селектор выполняет вызов `select()` для определения того, в каких сокетах из множества доступных доступен ввод/вывод. Затем он

должен провести время за обработкой этих данных, как минимум уведомить другие рабочие потоки о том, у каких клиентов имеются запросы для обработки. Затем он может вернуть управление и снова вызвать метод `select()`. Но во время обработки результатов вызова `select()` другой поток должен выполнять вызовы `select()` для других сокетов, чтобы узнать, когда в них появятся доступные данные.

Таким образом, во фреймворках с отдельным пулом потоков для селекторов необходимо позаботиться о том, чтобы пул содержал хотя бы несколько потоков (как правило, по умолчанию используются три потока). В фреймворках, в которых селекторы и рабочие потоки объединены в один пул, следует добавить еще несколько потоков сверх количества рабочих потоков, которое обсуждалось выше.

Асинхронные серверы REST

Вместо того чтобы настраивать пул потоков запросов сервера, также можно поручить работу другому пулу потоков. Этот подход используется реализацией асинхронного сервера JAX-RS, задачами исполнителей событий Netty (предназначенными для долго выполняемых задач), а также в других фреймворках.

Рассмотрим эту возможность с точки зрения JAX-RS. В простом сервере REST запросы и ответы обрабатываются в одном потоке. Этот факт ограничивает степень параллелизма таких серверов. Например, размер пула потоков по умолчанию для сервера Helidon на 8-процессорной машине равен 32. Рассмотрим следующую конечную точку:

```
@GET
@Path("/sleep")
@Produces(MediaType.APPLICATION_JSON)
public String sleepEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay
    ) throws ParseException {
    try { Thread.sleep(delay); } catch (InterruptedException ie) {}
    return "{\"sleepTime\": \"" + delay + "\"}";
}
```

Вызов `sleep` включен в этот пример только ради тестирования: предполагается, что `sleep` обращается с вызовом к удаленной базе данных или другому серверу REST, и эти удаленные вызовы занимают 100 мс. Если выполнить этот тест на сервере Helidon в конфигурации по умолчанию, он будет обрабатывать 32 параллельных запроса. Генератор нагрузки с параллелизмом 32 будет сообщать, что каждый запрос занимает 100 мс (плюс 1–2 мс для обработки). Генератор нагрузки с параллелизмом 64 сообщит, что каждый запрос занимает 200 мс, так

как каждому запросу придется ожидать завершения другого запроса, прежде чем он сможет приступить к обработке.

Другие серверы могут иметь другую конфигурацию, но эффект остается тем же: размер пула запросов становится своего рода регулятором. Часто это хорошо: если бы в данном примере расходовалось 100 мс активного процессорного времени (вместо ожидания), то сервер не смог бы обрабатывать 32 запроса конкурентно, если только он не работает на очень большой машине.

Однако в данном случае о плотном распределении вычислительной нагрузки нет и речи: для обработки нагрузки может быть достаточно 20–30% одного ядра (или если эти 100-миллисекундные интервалы всего лишь представляют удаленный вызов другого сервиса). Таким образом, параллелизм на машине можно повысить, изменив конфигурацию пула потоков по умолчанию для обработки большего количества вызовов. Ограничение будет определяться в зависимости от степени параллелизма удаленных систем; обращения к этим системам все равно нужно регулировать, чтобы избежать их перегрузки.

JAX-RS предоставляет второй способ повышения параллелизма, основанный на использовании асинхронных ответов. Асинхронный ответ позволяет передать обработку бизнес-логики в другой пул потоков:

```
ThreadPoolExecutor tpe = Executors.newFixedThreadPool(64);
@GET
@Path("/asyncsleep")
@Produces(MediaType.APPLICATION_JSON)
public void sleepAsyncEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay,
    @Suspended final AsyncResponse ar
    ) throws ParseException {
    tpe.execute(() -> {
        try { Thread.sleep(delay); } catch (InterruptedException ie) {}
        ar.resume("{\"sleepTime\": \"" + delay + "\"}");
    });
}
```

В этом примере исходный запрос поступает в пул потоков сервера по умолчанию. Этот запрос организует вызов выполнения бизнес-логики в отдельном пуле потоков (асинхронном пуле потоков), после чего метод `sleepAsyncEndpoint()` немедленно возвращает управление. Таким образом поток из пула по умолчанию освобождается, чтобы он мог немедленно обработать другой запрос. Тем временем асинхронный ответ (помеченный аннотацией `@Suspended`) ожидает завершения логики; когда это происходит, он возобновляет работу с отправкой ответа пользователю.

Это позволяет нам запустить 64 (или другое число по значению параметра, переданного пулу потоков) параллельных запроса до того, как запросы начнут

накапливаться. Но откровенно говоря, мы не добились ничего, что бы принципиально отличалось от определения пула потоков по умолчанию с размером 64. Собственно, в нашем случае время отклика будет немного хуже, потому что запрос будет передан другому потоку для обработки, что займет несколько миллисекунд.

Асинхронные ответы используются по трем причинам:

- Для расширения параллелизма в бизнес-логике. Представьте, что вместо ожидания в течение 100 мс наш код выдает три (не связанных друг с другом) вызова JDBC для получения данных, необходимых для ответа. Использование асинхронного ответа позволяет коду обрабатывать вызовы параллельно, при этом каждый вызов JDBC будет использовать отдельный поток в асинхронном пуле потоков.
- Для ограничения количества активных потоков.
- Для регулировки нагрузки на сервер.

У большинства серверов REST в случае простой регулировки пула потоков запросов новые запросы будут ожидать обработки, а очередь пула потоков начнет расти. Часто эта очередь не ограничена (или по крайней мере имеет очень большую границу), так что общее количество запросов выходит из-под контроля. Запросы, проводящие много времени в очереди пула потоков, часто становятся бесполезными к моменту своей обработки, но даже если они еще нужны — долгое время ожидания подрывает общую производительность системы.

Правильнее будет проверить статус асинхронного пула потоков, перед тем как ставить запрос в очередь, и отвергнуть запрос, если система слишком занята.

```
@GET
@Path("/asyncreject")
@Produces(MediaType.APPLICATION_JSON)
public void sleepAsyncRejectEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay,
    @Suspended final AsyncResponse ar
    ) throws ParseException {
    if (tpe.getActiveCount() == 64) {
        ar.cancel();
        return;
    }
    tpe.execute(() -> {
        // Моделирование задержки обработки вызовом sleep
        try { Thread.sleep(delay); } catch (InterruptedException ie) {}
        ar.resume("{\"sleepTime\": \"" + delay + "\"}");
    });
}
```

Это можно сделать многими разными способами, но в этом простом примере мы проверяем количество активных потоков, работающих в пуле. Если оно равно размеру пула, ответ немедленно отменяется. (В более сложном решении можно было бы создать ограниченную очередь для пула и отменить запрос в обработчике отклоненных исключений пула потоков.)

В результате сторона вызова немедленно получит статус HTTP 503 («Сервис недоступен»), который указывает, что запрос не может быть обработан в данный момент. Так рекомендуется поступать в случае перегрузки серверов в мире REST, и немедленное возвращение этого статуса понизит нагрузку на перегруженный сервер, что в итоге приведет к значительному улучшению общей производительности.

РЕЗЮМЕ

- Неблокирующий ввод/вывод с использованием Java NIO API предоставляет возможность масштабирования серверов за счет уменьшения количества потоков, требуемого для обработки нескольких клиентов.
- Этот метод означает, что серверу понадобится один или несколько пулов потоков для обработки клиентских запросов. Настройка этого пула базируется на максимальном количестве конкурентных запросов, которые должны обрабатываться сервером.
- Еще несколько дополнительных потоков понадобятся для селекторов (в составе пула рабочих потоков или в отдельном пуле, в зависимости от серверного фреймворка).
- У серверных фреймворков часто имеется механизм передачи длительных запросов в другой пул потоков, что обеспечивает более надежную обработку запросов в основном пуле потоков.

Асинхронные исходящие вызовы

В предыдущем разделе был приведен пример сервера с двумя процессорами, которому требовался пул из 20 потоков для достижения максимальной пропускной способности. Это объяснялось тем, что потоки проводили 90% своего времени за блокированием по вводу/выводу при выдаче внешнего вызова к другому ресурсу.

Неблокирующий ввод/вывод может помочь и в этом случае: если исходящие вызовы HTTP или JDBC являются неблокирующими, можно отказаться от выделения отдельного потока для вызова; это позволит сократить размер пула потоков.

Асинхронный HTTP

Клиенты HTTP представляют собой классы, которые (как нетрудно догадаться) выдают запросы HTTP к серверу. В нормальной ситуации клиентов много и все они обладают разными характеристиками как по функциональности, так и по производительности. В этом разделе рассматриваются характеристики производительности для типичных сценариев использования.

В Java 8 базовым клиентом HTTP является класс `java.net.HttpURLConnection` (а для безопасных подключений — подкласс `java.net.HttpsURLConnection`). В Java 11 был добавлен новый клиент: класс `java.net.http.HttpClient` (который также поддерживает HTTPS). Среди других классов клиентов HTTP из других пакетов можно упомянуть `org.apache.http.client.HttpClient` от Apache Foundation, `org.asynchttpclient.AsyncHttpClient` на базе Netty Project и `org.eclipse.jetty.client.HttpClient` от Eclipse Foundation.

И хотя, возможно, класс `HttpURLConnection` позволяет выполнять основные операции, большинство вызовов REST осуществляется из таких фреймворков, как JAX-RS. Таким образом, большинство клиентов HTTP напрямую реализует эти API (или их разновидности), но реализация по умолчанию JAX-RS также предоставляет коннекторы для большинства популярных клиентов HTTP. Следовательно, JAX-RS можно использовать с клиентом HTTP более низкого уровня, что обеспечит лучшую производительность. С JAX-RS и нижележащими клиентами HTTP связаны два основных аспекта производительности.

Во-первых, коннекторы JAX-RS предоставляют объект `Client`, используемый для создания вызовов REST; при непосредственном использовании клиентов они предоставляют клиентский объект с именем вида `HttpClient` (класс `HttpURLConnection` — исключение; он не может использоваться повторно). Типичный пример создания и использования клиента:

```
private static Client client;
static {
    ClientConfig cc = new ClientConfig();
    cc.connectorProvider(new JettyConnectorProvider());
    client = ClientBuilder.newClient(cc);
}

public Message getMessage() {
    Message m = client.target(URI.create(url)
                  .request(MediaType.APPLICATION_JSON)
                  .get(Message.class);
    return m;
}
```

В этом примере важно то, что клиентский объект представляет собой статический, совместно используемый объект. Все клиентские объекты являются

потоково-безопасными, а создание их экземпляров требует высоких затрат, поэтому в их приложении они должны использоваться в ограниченном количестве (например, 1).

Второй фактор производительности — вы должны позаботиться о том, чтобы клиент HTTP правильно организовал пул подключений и использовал постоянное (keepalive) подключение, для того чтобы поддерживать открытые подключения. Открытие сокета для коммуникаций HTTP является затратной операцией, особенно если используется протокол HTTPS, а клиент и сервер должны выполнять согласование (handshake) SSL. Как и подключения JDBC, подключения HTTP(S) должны использоваться повторно.

Все клиенты HTTP предоставляют механизм организации пула, хотя принцип работы `HttpURLConnection` часто понимается неправильно. По умолчанию класс создает пул из пяти подключений (на сервер). Однако в отличие от традиционных пулов, пул этого класса не регулирует подключения: если запросить шестое подключение, то новое подключение будет создано, а затем уничтожено после завершения. Подобное кратковременное поведение нетипично для традиционных пулов подключений. По этой причине в конфигурации класса `HttpURLConnection` по умолчанию часто есть множество временных подключений, и предполагается, что подключения не объединены в пулы (причем документация Java здесь тоже не помогает; функциональность пулов в ней вообще не упоминается, хотя это поведение документировано в других местах).

Размер пула можно изменить при помощи системного свойства `-Dhttp.maxConnections=`*N*, по умолчанию равного 5. Несмотря на имя, это свойство также применяется к подключениям HTTPS, но этот класс не ограничивает количество подключений.

В новом классе `HttpClient` из JDK 11 этот пул работает по похожему принципу, но с двумя важными различиями. Во-первых, размер пула по умолчанию не ограничен, хотя его можно установить при помощи системного свойства `-Djdk.httpclient.connectionPoolSize=`*N*. Это свойство все равно не является ограничителем; если вы запросите больше подключений, чем было настроено, они будут созданы по мере надобности, а затем уничтожены после завершения. Во-вторых, этот пул существует на уровне объекта `HttpClient`; если вы не используете этот объект, функциональность пула подключений отсутствует.

Непосредственно в JAX-RS часто рекомендуется использовать другой коннектор вместо коннектора по умолчанию, если вам нужна функциональность пула подключений. Так как коннектор по умолчанию использует класс `HttpURLConnection`, это не так: если только вы не хотите ограничивать количество подключений, вы можете настроить количество подключений для класса только что описанным способом. Другие популярные коннекторы также используют пулы подключений.

Таблица 10.1. Настройка пулов подключений HTTP для популярных клиентов

Коннектор	Класс клиента HTTP	Механизм настройки пула
По умолчанию	`java.net.HttpURLConnection`	Изменение системного свойства `maxConnections`
Apache	`org.apache.http.client.HttpClient`	Создание `PoolingHttpClientConnectionManager`
Grizzly	`com.ning.http.client.AsyncHttpClient`	Пул используется по умолчанию; возможно изменение конфигурации
Jetty	`org.eclipse.jetty.client.HttpClient`	Пул используется по умолчанию; возможно изменение конфигурации

В JAX-RS менеджер подключений Grizzly использует клиента `com.ning.http.client.AsyncHttpClient`. Этот клиент был переименован в `org.asynchttpclient.AsyncHttpClient`; это асинхронный клиент, построенный на базе Netty.

Асинхронные клиенты HTTP

Асинхронные клиенты HTTP, как и асинхронные серверы HTTP, позволяют более эффективно управлять потоками в приложении. Поток, который выдает асинхронный вызов, отправляет запрос удаленному серверу и принимает меры к обработке ответа в другом (фоновом) потоке, когда он будет доступен.

Столь неопределенная формулировка («принимает меры») выбрана намеренно, потому что механизм, который для этого используется, сильно отличается для разных клиентов HTTP. Но с точки зрения производительности суть в том, что использование асинхронного клиента повышает производительность, так как обработка ответа передается в другой поток, что позволяет выполнять больше операций параллельно.

Асинхронные клиенты HTTP относятся к функциональности JSX-RS 2.0, хотя многие автономные клиенты HTTP также поддерживают асинхронную функциональность напрямую. Возможно, вы заметили, что в именах некоторых из рассмотренных клиентов присутствует часть `async`; эти клиенты асинхронны по умолчанию. Хотя они поддерживают синхронный режим, эта поддержка есть в реализации синхронных методов: эти методы выдают асинхронный вызов, ожидают завершения ответа, а затем возвращают ответ (синхронно) вызывающей стороне.

Асинхронный режим поддерживается реализациями JAX-RS 2.0, включая эталонные реализации Jersey. Эта реализация включает ряд коннекторов, которые могут использоваться асинхронно, хотя не все эти коннекторы действительно

асинхронны. Во всех случаях обработка ответа передается другому потоку, но эта обработка может осуществляться двумя основными способами. В одном случае другой поток просто использует стандартный блокирующий ввод/вывод Java. В этом случае фоновый пул потоков должен содержать один поток для каждого запроса, который должен обрабатываться конкурентно с другими. Ситуация такая же, как с асинхронным сервером: параллелизм обеспечивается добавлением множества других потоков.

Во втором случае клиент HTTP использует неблокирующий ввод/вывод. Для подобной обработки фоновому потоку понадобятся несколько (по крайней мере один, но обычно больше) потоков для обработки ключей выбора NIO, а также другие потоки для обработки поступающих ответов. Во многих случаях эти клиенты HTTP используют меньшее общее количество потоков. NIO является классическим примером событийного программирования: когда на подключении к сокету появляются данные, доступные для чтения, поток (обычно из пула) уведомляется об этом событии. Поток читает данные, обрабатывает их (или передает данные другому потоку для обработки), а затем возвращается в пул.

Асинхронное программирование обычно рассматривается как событийное, поэтому формально асинхронные клиенты HTTP, использующие блокирующий ввод/вывод (и закрепление потока на весь запрос), асинхронными не являются. API создает иллюзию асинхронного поведения, даже если потоковая масштабируемость будет не той, которую вы ожидаете.

С точки зрения производительности асинхронный клиент обладает примерно такими же преимуществами, как и асинхронный сервер: вы можете повысить параллелизм во время запроса, чтобы он выполнялся быстрее, а также повысить эффективность управления запросами (и ограничивать их количество) за счет использования разных пулов потоков.

А теперь возьмем общий сценарий для асинхронных примеров: службу REST, которая агрегирует информацию от трех других служб REST. Схема такой службы на псевдокоде выглядит так:

```
public class TestResource {
    public static class MultiCallback extends InvocationCallback<Message> {
        private AsyncResponse ar;
        private AtomicDouble total = new AtomicDouble(0);
        private AtomicInteger pendingResponses;
        public MultiCallback(AsyncResponse ar, int targetCount) {
            this.ar = ar;
            pendingResponse = new AtomicInteger(targetCount);
        }
        public void completed(Message m) {
            double d = total.getAndIncrement(Message.getValue());
            if (targetCount.decrementAndGet() == 0) {
```

```
                ar.resume("{\"total\": \"" + d + "\"}");
            }
        }
    }

    @GET
    @Path("/aggregate")
    @Produces(MediaType.APPLICATION_JSON)
    public void aggregate(@Suspended final AsyncResponse ar)
                    throws ParseException {
        MultiCallback callback = new MultiCallback(ar, 3);
        target1.request().async().get(callback);
        target2.request().async().get(callback);
        target3.request().async().get(callback);
    }
}
```

Обратите внимание: в этом примере также используется асинхронный ответ, но отдельный пул, как в предыдущих примерах, здесь не нужен: запрос будет возобновлен в одном из потоков, обрабатывающих запросы.

Такой подход вводит в операцию требуемый параллелизм, но давайте внимательней присмотримся к использованию потоков. На рис. 10.3 представлено значительное использование потоков сервера Helidon при выполнении этого примера.

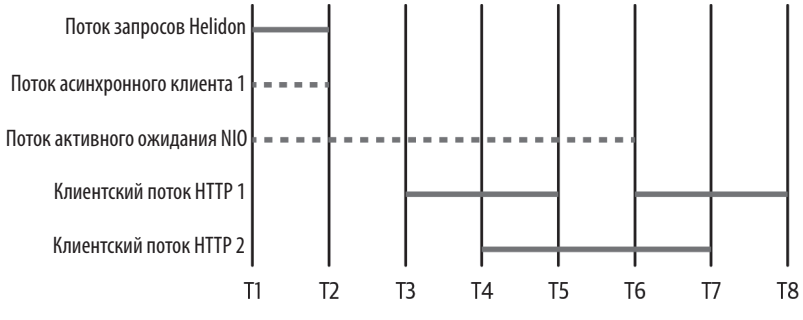


Рис. 10.3. Простое использование потоков для асинхронных клиентов HTTP

В момент T1 поступает запрос, который начинает выполняться в потоке запросов Helidon. Поток создает три удаленных вызова; каждый вызов в действительности отправляется потоком из пула асинхронных клиентов. (На диаграмме все три отправляются одним потоком, но это зависит от временной последовательности: они могут выполняться в трех разных потоках в зависимости от того, с какой скоростью генерируются запросы и сколько времени уходит на отправку ими данных.) Три сокета, связанных с этими вызовами, также регистрируются

в очереди событий, обрабатываемой потоком активного ожидания NIO. Поток запросов завершает обработку в момент T2.

В момент T3 поток активного ожидания NIO получает событие, сообщающее о наличии данных в одном из сокетов. Он создает клиентский поток HTTP 1 для чтения и обработки этих данных. Эта обработка продолжается до момента T5. Тем временем в момент T4 поток активного ожидания NIO получает событие, сообщающее о наличии данных для чтения в другом сокете; эти данные читаются и обрабатываются клиентским потоком HTTP 2 (это продолжается до момента времени T7). Затем в момент времени T5 третий сокет становится готовым к обработке. Так как клиентский поток HTTP 1 бездействует, он может прочитать и обработать этот запрос, который завершается в момент времени T8 (в этот момент для объекта ответа вызывается метод `resume()`, и ответ передается клиенту).

В этой схеме ключевую роль играет продолжительность обработки в клиентских потоках. Если обработка выполняется очень быстро, а ответы распределяются во времени достаточно хорошо, один поток может обработать все ответы. Если обработка занимает много времени или ответы группируются, понадобится один поток на один запрос. Приведенный пример занимает промежуточное положение: мы использовали меньше потоков, чем в модели «один поток на один запрос», но больше одного потока. В этом заключается ключевое отличие сервера REST от чего-то вроде сервера статического контента nginx: в конечном итоге даже в полностью асинхронной реализации потребности бизнес-логики в вычислительных ресурсах потребуют значительного количества потоков для достижения хорошего параллелизма.

В этом примере предполагается, что клиент HTTP использует NIO. Если клиент использует традиционный механизм NIO, диаграмма будет выглядеть немного иначе. При первом вызове к потоку асинхронного клиента этот вызов продолжается до момента времени T7. Второму вызову к асинхронному клиенту потребуется новый поток: этот запрос будет продолжаться до момента времени T8. Третий поток асинхронного клиента будет работать до момента T5 (мы не ожидаем, что клиенты будут завершаться в том порядке, в котором они были запущены). Отличия изображены на рис. 10.4.

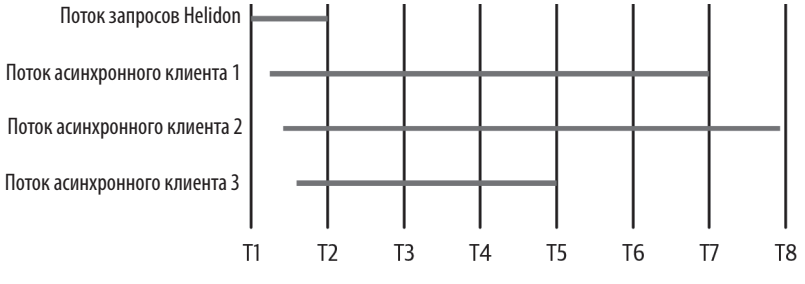


Рис. 10.4. Простое использование потоков для блокирующих клиентов HTTP

В любом случае для конечного пользователя результат будет одним и тем же: все три запроса обрабатываются параллельно с ожидаемым выигрышем по быстродействию. Однако уровень использования потоков (а следовательно, общая эффективность системы) будет совершенно другим.

Асинхронные клиенты HTTP и использование потоков

Конечно, пулы фоновых потоков будут обеспечивать ограничение, и их необходимо настраивать: их размер был достаточным для обеспечения степени параллелизма, необходимой вашему приложению, но не слишком большим, чтобы не перегружать внешнего ресурса. Нередко настроек по умолчанию оказывается достаточно, но если вам захочется больше узнать о различных коннекторах эталонной реализации JAX-RS и их фоновых пулах, я приведу дополнительную информацию о них.

Коннектор по умолчанию использует блокирующий ввод/вывод. Один пул потоков асинхронных клиентов в Jersey (эталонная реализация JAX-RS) обрабатывает все запросы; имена потоков в этом пуле начинаются с `jersey-client-async-executor`. Пул должен содержать один поток для каждого параллельного запроса, как показано на рис. 10.4. По умолчанию размер пула не ограничен; вы можете установить границу при настройке клиента, задав следующее свойство:

```
ClientConfig cc = new ClientConfig();
cc.property(ClientProperties.ASYNC_THREADPOOL_SIZE, 128);
client = ClientBuilder.newClient(cc);
```

Коннектор Apache — хотя в библиотеках Apache имеется полноценный асинхронный клиент (использующий NIO для чтения ответа, вместо того чтобы требовать выделенный поток), коннектор Apache в Jersey использует традиционного клиента Apache с блокирующим вводом/выводом. В отношении пулов потоков он ведет себя и настраивается так же, как клиент по умолчанию.

Коннектор Grizzly — клиент HTTP, используемый коннектором Grizzly, является асинхронным и строится в соответствии с моделью на рис. 10.3. В нем используются разные пулы: один (`grizzly-ahc-kernel`) записывает запросы, другой (`nioEventLoopGroup`) ожидает события NIO, а третий (`pool-N`) читает и обрабатывает запросы. Последний пул особенно важно настроить по соображениям производительности/регулировки нагрузки, а его размер не ограничен; его можно регулировать при помощи свойства `ASYNC_THREADPOOL_SIZE`.

Коннектор Jetty — Jetty использует асинхронный класс. Запросы отправляются и читаются из одного пула потоков (и активное ожидание событий также происходит в этом пуле). В Jersey этот пул также настраивается при помощи свойства `ASYNC_THREADPOOL_SIZE`, хотя сервер, использующий Jetty, также

использует два служебных пула потоков: стандартный пул потоков `jersey-client-async-executor` (который выполняет различные учетные операции) и пул потоков, обслуживающих клиентов Jetty (имена этих потоков начинаются с `HttpClient`). Если свойство не задано, то размер пула `HttpClient` будет равен 200.

РЕЗЮМЕ

- Всегда следите за тем, чтобы пул подключений клиентов HTTP был правильно настроен.
- Производительность асинхронных клиентов в HTTP можно улучшить за счет распределения потоков между несколькими потоками и повышения параллелизма.
- Асинхронные клиенты HTTP, построенные с использованием NIO, требуют меньшего количества потоков, чем клиенты, использующие традиционный ввод/вывод, но сервер REST все равно требует относительно большого количества потоков для обработки асинхронных запросов.

Асинхронные обращения к базам данных

Если исходящий вызов является обращением к реляционной базе данных, сделать его полностью асинхронным будет сложно. Стандартный JDBC API не подходит для неблокирующего ввода/вывода, поэтому общее решение требует нового API или новых технологий. Различные предложения по такому API создавались и отклонялись; в настоящее время надежды возлагаются на новый облегченный модуль задач — *волокна* (fibers), которые сделают возможным масштабирование существующих асинхронных API без необходимости асинхронного программирования. Волокна являются частью проекта OpenJDK Loom, но дата выпуска еще не определена (на момент написания книги).

Предложения (и реализации) асинхронных оберток JDBC часто передают работу JDBC отдельному пулу потоков. В этом они похожи на асинхронного клиента HTTP по умолчанию из реализации Jersey; с точки зрения программирования API кажется асинхронным. Но в реализации фоновые потоки блокируются по каналам ввода/вывода, поэтому если мы пойдем в этом направлении, никакой масштабируемости добиться не удастся.

Различные проекты за пределами JDK позволят заполнить пробел. Наибольшей популярностью пользуется технология Spring Data R2DBC из проекта Spring. Она требует другого API, а драйверы доступны не для всех баз данных. Но если вам нужен неблокирующий доступ к реляционным базам данных, на сегодняшний день это лучшее из имеющихся решений.

Для баз данных NoSQL ситуация выглядит аналогично. С другой стороны, никаких стандартов Java для работы с базами данных NoSQL нет, поэтому программирование зависит от API, специализированного для конкретной базы данных. Проекты Spring для реактивных баз данных NoSQL могут использоваться для полноценного асинхронного доступа.

Обработка JSON

От механики передачи данных серверам Java перейдем к самим данным. В этом разделе в основном будет рассматриваться обработка данных в формате JSON. Другие программы Java часто используют XML (плюсы и минусы JSON и XML во многом идентичны); также существуют новые форматы (например, Apache Avro и буферы протоколов Google).

ДАННЫЕ JSON ДЛЯ ПРИМЕРОВ

В примерах этого раздела используются данные JSON, полученные от REST-службы eBay, возвращающей 100 предложений для заданного ключевого слова. Приведу часть этих данных:

```
{"findItemsByKeywordsResponse": [
    {"ack":["Success"],
     "version":["1.13.0"],
     "timestamp":["2019-08-26T19:48:28.830Z"],
     "searchResult":[
        {"@count":"100",
         "item":[
             {"itemId":["153452538114"],
              "title":["Apple iPhone 6"],
              ... другие поля ...
             }
         ],
         ... другие элементы ...
        }
     ]
    }
]}
```

Разбор данных и маршалинг

Программа, получившая серию строк JSON, должна преобразовать эти строки в данные, пригодные для обработки в Java. Этот процесс называется *маршалингом* (marshaling) или *разбором* (parsing) в зависимости от контекста и получае-

мого результата. Если на выходе создается объект Java, то процесс называется маршалингом; если данные обрабатываются в процессе чтения, то процесс называется разбором. Обратный процесс — формирование строк JSON по другим данным — называется *демаршалингом*.

При обработке данных JSON используются три основных метода:

- *Активные парсеры* — входные данные связываются с парсером, и программа извлекает серию лексем от парсера.
- *Модели документов* — входные данные преобразуются в объект, напоминающий документ, так что приложение может перебирать его содержимое в поиске фрагментов данных. Интерфейс строится на концепциях обобщенных документо-ориентированных объектов.
- *Объектные модели* — входные данные преобразуются в один или несколько объектов Java с использованием набора заранее определенных классов, отражающих структуру данных (например, заранее определенный класс `Person` предназначен для данных, представляющих человека). Обычно такие объекты обозначаются сокращением POJO (от Plain Old Java Object).

Методы перечислены приблизительно по убыванию скорости (от быстрых к медленным), но я еще раз подчеркну, что функциональные различия между ними важнее различий по производительности. Парсер ограничивается простым последовательным сканированием, поэтому он не так хорошо подходит для данных, к которым нужно обращаться в произвольном порядке или анализировать многократно. В таких ситуациях программе, использующей только простой парсер, придется строить внутреннюю структуру данных, что в общем-то несложно. Однако модели документов и объектов Java уже предоставляют структурированные данные, и работать с ними обычно бывает проще, чем определять новые структуры самостоятельно.

В сущности, в этом и состоит отличие парсера от маршалера данных. Первый пункт списка — парсер; логика приложения должна обрабатывать данные, по мере того как парсер предоставляет эти данные. Следующие два пункта — маршалеры; они должны использовать парсер для обработки данных, но при этом обеспечивают представление данных, которые более сложные программы могут использовать в своей логике.

Таким образом, основной выбор относительно используемого метода определяется тем, как должно быть написано приложение. Если программа должна совершить один простой проход по данным, достаточно использовать самый быстрый парсер. Прямое использование парсера также уместно в том случае, если приложение должно быть сохранено в простой структуре, определяемой

приложением, например, цены товаров в примере данных можно сохранить в `ArrayList`, чтобы упростить их обработку из логики приложения.

Использование модели документа более уместно в тех ситуациях, когда важен формат данных. Если формат данных необходимо сохранить, то данные можно прочитать в формат документа, каким-то образом изменить, после чего формат документа можно просто записать в новый поток данных.

Для достижения наивысшей гибкости модель документа обеспечивает представление данных на уровне языка Java. С данными можно работать, пользуясь знакомыми понятиями объектов и их атрибутов. Дополнительная сложность маршалинга (в основном) остается прозрачной для разработчика и может немного замедлить эту часть приложения, но выигрыш по производительности при работе с кодом может компенсировать эту проблему.

Объекты JSON

Данные JSON имеют два объектных представления. Первое, обобщенное — простые объекты JSON. Для работы с объектами используются обобщенные интерфейсы: `JsonObject`, `JsonArray` и т. д. Они предоставляют механизм построения или анализа документов JSON без создания конкретных представлений данных в виде классов.

Второе объектное представление JSON связывает данные JSON с полноценными классами Java с использованием привязок JSON (JSON-B), в результате чего создаются POJO-объекты. Например, элементы данных JSON в нашем примере будут представляться классом `Item`, который содержит атрибуты для своих полей.

Два объектных представления отличаются тем, что первое является обобщенным и не требует классов. Если у вас имеется объект `JsonObject`, представляющий элемент наших данных, то название товара может быть определено следующим образом:

```
JsonObject jo;
String title = jo.getString("title");
```

В JSON-B для получения названия будут использоваться интуитивно более понятные `get`- и `set`-методы:

```
Item i;
String title = i.getTitle();
```

В любом случае сам объект создается используемым парсером, поэтому очень важно настроить парсер для оптимальной производительности. Однако в до-

полнение к разбору данных реализации объектов позволяют построить строку JSON по объекту (то есть выполнить демаршалинг). В табл. 10.2 приведены данные о производительности этих операций.

Таблица 10.2. Производительность объектных моделей JSON

Объектная модель	Производительность маршалинга
Объект JSON	2318 ± 51 мкс
Классы JSON-B	7071 ± 71 мкс
Преобразователь Jackson	1549 ± 40 мкс

Простой объект JSON строится значительно быстрее, чем объекты специализированных классов Java, хотя с этими классами Java проще работать с точки зрения программирования.

Преобразователь Jackson в этой таблице — альтернативное решение, которое на настоящее время в значительной степени вытеснило другие примеры. Хотя Jackson предоставляет реализацию API стандартного разбора JSON (JSON-P), также имеется альтернативная реализация, которая осуществляет маршалинг и демаршалинг данных JSON в объекты Java, но не следует стандарту JSON-B. Эта реализация построена на базе класса `ObjectMapper`, также предоставляемого Jackson. Код JSON-B для маршалинга данных в объект выглядит примерно так:

```
Jsonb jsonb = JsonbBuilder.create();
FindItemsByKeywordsResponse f =
    jsonb.fromJson(inputStream, FindItemsByKeywordsResponse.class);
```

Код с `ObjectMapper` немного отличается от него:

```
ObjectMapper mapper = new ObjectMapper();
FindItemsByKeywordsResponse f =
    mapper.readValue(inputStream, FindItemsByKeywordsResponse.class);
```

С точки зрения производительности у использования `ObjectMapper` имеется ряд проблем. При маршалинге данных преобразователь создает множество классов-посредников, которые используются для создания полученных POJO-объектов. Все это занимает некоторое время при первом использовании класса. Для решения этой проблемы на практике часто допускается одна ошибка (которая становится второй проблемой) — в программе создается множество объектов-преобразователей (например, один статический объект на класс, выполняющий маршалинг). Это часто приводит к чрезмерному расходованию

памяти, потерям процессорного времени и даже ошибкам `OutOfMemory`. В приложении должен быть только один объект `ObjectMapper`, что положительно отражается на затратах как процессорного времени, так и памяти. Даже в этом случае представление данных в объектной модели потребует некоторых затрат памяти для этих объектов.

Разбор JSON

Прямой разбор данных JSON обладает двумя преимуществами. Во-первых, если объектная модель JSON расходует слишком много памяти, прямой разбор JSON с последующей обработкой сэкономит эту память. Во-вторых, если разметка JSON, с которой вы работаете, содержит много данных, прямой разбор будет более эффективным.

Все парсеры JSON являются активными (pull); их работа основана на получении данных из потока по требованию. У простого активного парсера для тестов из этого раздела основная логика представлена следующим циклом:

```
parser = factory.createParser(inputStream);
int idCount = 0;
while (parser.hasNext()) {
    Event event = parser.next();
    switch (event) {
        case KEY_NAME:
            String s = parser.getString();
            if (ID.equals(s)) {
                isID = true;
            }
            break;
        case VALUE_STRING:
            if (isID) {
                if (addId(parser.getString())) {
                    idCount++;
                    return;
                }
                isID = false;
            }
            continue;
        default:
            continue;
    }
}
```

Этот код получает токены от парсера. В коде большинство токенов просто отбрасывается. При обнаружении стартового токена код сначала проверяет, представляет ли он идентификатор элемента. Если проверка дает положительный

результат, то следующим символьным токеном в приложении будет идентификатор, который приложение хочет сохранить.

Тест также позволяет фильтровать данные: в данном случае фильтрация ограничивает выборку первыми 10 элементами в данных JSON. Это делается при обработке идентификатора: идентификатор сохраняется методом `addItemId()`, который возвращает `true`, если было сохранено нужное количество идентификаторов. Когда это происходит, цикл может просто вернуть управление без обработки остальных данных во входном потоке.

ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ ФАБРИК И ПАРСЕРОВ

Создание фабрик и парсеров JSON сопряжено с высокими затратами. К счастью, фабрики являются потоково-безопасными, поэтому фабрику можно легко сохранить в глобальной статической переменной и повторно использовать ее по мере надобности.

В общем случае сами парсеры не могут использоваться повторно и не обладают потоковой безопасностью. Таким образом, парсеры обычно создаются тогда, когда в них возникнет необходимость.

Как же на самом деле работают эти парсеры? В табл. 10.3 приведено время разбора документа-примера в микросекундах с остановкой после 10 элементов и с обработкой всего документа. Как и следовало ожидать, сокращение количества элементов на 90% приводит к 90%-ному выигрышу по производительности.

Таблица 10.3. Производительность активных парсеров

Обрабатываемые элементы	Парсер по умолчанию	Парсер Jackson
10	159 ± 2 мкс	86 ± 5 мкс
100	1662 ± 46 мкс	770 ± 4 мкс

Как и раньше, парсер Jackson обеспечивает лучшую производительность, но оба решения заметно быстрее чтения объектов.

РЕЗЮМЕ

- Есть два варианта обработки JSON: создание POJO-объектов и прямой разбор.

- Выбор зависит от потребностей приложения, но прямой разбор обладает преимуществами в отношении фильтрации и производительности. Создание объектов JSON часто приводит к проблемам из области уборки мусора при больших размерах объектов.
- Парсер Jackson обычно работает быстрее всех остальных; выбирайте его вместо реализаций по умолчанию.

Итоги

Неблокирующий ввод/вывод образует основу эффективного масштабирования серверов, потому что он позволяет обрабатывать относительно большое количество подключений при относительно небольшом количестве потоков. Традиционные серверы используют эту возможность для базовой обработки клиентских подключений, а более новые серверные фреймворки продвигают неблокирующую природу операций для других применений.

ГЛАВА 11

Практика работы с базами данных

Эта глава посвящена быстродействию приложений для работы с базами данных на Java. В приложениях, обращающихся к базам данных, возникают некоторые проблемы производительности, не связанные с Java: если база данных выполняет интенсивный ввод/вывод или выполняет запросы SQL, требующие полного сканирования таблиц из-за отсутствия индекса, никакая настройка Java или приемы программирования не решат проблем производительности. Имея дело с технологиями баз данных, приготовьтесь учиться (по другому источнику) тому, как настраивать и программировать базу данных.

SQL И NOSQL

Эта глава посвящена реляционным базам данных и технологиям Java для работы с ними. В Java существуют средства для работы с нереляционными базами данных, но они не являются стандартными. Реляционные базы данных соответствуют различным стандартам SQL, включая стандарт ANSI/ISO SQL (SQL:2003). Соответствие этому стандарту означает, что сама платформа Java может предоставить стандартный интерфейс к этим базам данных; таким интерфейсом является JDBC (и спецификация JPA).

Соответствующего стандарта для баз данных NoSQL нет, а следовательно, нет и стандартной платформенной поддержки для работы с ними. Ожидается, что в предстоящую версию Jakarta Enterprise Edition 9 будет включена спецификация для стандартной работы к базам данных NoSQL, хотя подробности еще не приняли окончательной формы.

Хотя в примерах этой главы базы данных NoSQL не рассматриваются, изложенные концепции, безусловно, применимы к ним: такие концепции, как пакетная обработка и границы транзакций, играют для баз данных NoSQL не менее важную роль, чем для реляционных баз данных.

Это не означает, что производительность приложения, использующего базу данных, не зависит от факторов, находящихся под контролем JVM и используемых технологий Java. Скорее, для достижения хорошей производительности необходимо позаботиться о том, чтобы как база данных, так и приложение были правильно настроены и выполняли лучший возможный код.

Эта глава начинается с рассмотрения драйверов JDBC, потому что они влияют на фреймворки данных, взаимодействующие с реляционными базами данных. Многие фреймворки, включая JPA и модули данных Spring, абстрагируют технические подробности JDBC.

База данных для примера

Примеры этой главы используют базу данных для хранения данных 256 акций за один год. В году 261 рабочий день.

Цены акций хранятся в таблице `STOCKPRICE`, имеющей первичный ключ для биржевого обозначения акций и даты. Таблица содержит 66 816 строк данных (256 × 261).

С каждым видом акций также связан набор из пяти опционов, цена которых также определяется ежедневно. Эти данные хранятся в таблице `STOCKOPTIONPRICE` с первичным ключом из биржевого обозначения, даты и целочисленного номера опциона. Таблица содержит 334 080 строк данных (256 × 261 × 5).

JDBC

В этой главе рассматривается производительность работы с базами данных в JPA версии 2.x. Однако JPA использует JDBC во внутренней реализации, и многие разработчики все еще пишут приложения, напрямую использующие JDBC API, — поэтому очень важно представлять себе важнейшие факторы производительности JDBC. Даже в приложениях, использующих JPA (или другой фреймворк баз данных, хотя бы Spring Data), понимание основ производительности JDBC поможет добиться максимальной производительности от фреймворка.

Драйверы JDBC

Драйвер JDBC — самый важный фактор производительности приложений, работающих с базами данных. Базы данных поставляются с собственными драйверами JDBC, а для большинства популярных баз данных доступны альтернативные драйверы JDBC. Часто существование этих альтернативных

драйверов объясняется тем, что они обеспечивают более высокую производительность.

Невозможно вынести заключение о производительности всех драйверов баз данных. Ниже перечислены некоторые обстоятельства, которые следует учитывать при оценке драйверов.

Где выполняется работа

Драйверы JDBC могут быть написаны так, чтобы большая часть работы выполнялась в приложениях Java (клиентах баз данных), или так, чтобы большая часть работы выполнялась на серверах баз данных. Лучший пример — тонкие и толстые драйверы для баз данных Oracle. Тонкий драйвер пишется так, чтобы он создавал минимальную нагрузку в приложениях Java: он полагается на сервер базы данных, выполняющий большую часть обработки. Толстый драйвер идет по обратному пути: он снимает часть нагрузки с базы данных за счет выполнения большего объема обработки и бóльших затрат памяти в клиенте Java. Подобные компромиссы встречаются при работе с большинством баз данных.

По поводу того, какая модель обеспечивает лучшую производительность, разные стороны расходятся во мнениях. Но по правде говоря, ни одна модель не обеспечивает однозначных преимуществ — выбор драйвера, который обеспечивает наилучшую производительность, зависит от особенностей среды, в которой он работает. Допустим, хостом приложения является небольшая двухъядерная машина, подключающаяся к огромной, хорошо настроенной базе данных. Процессор хоста приложения с большой вероятностью будет перенасыщен нагрузкой задолго до того, как на базе данных появится сколько-нибудь значительная нагрузка. Тонкий драйвер обеспечит более высокую производительность в этом случае. И наоборот, в организации с 100 филиалов, которые работают с единой базой данных, наилучшая производительность будет достигнута при экономии ресурсов базы данных и использовании толстых драйверов на стороне клиентов[1].

По этой причине не стоит безоговорочно доверять любым заявлениям о производительности, относящимся к драйверам JDBC: слишком легко выбрать драйвер, хорошо подходящий для конкретной среды, и показать, что он превосходит драйверы других производителей, которые плохо работают в этой конкретной конфигурации. Как обычно, проводите тестирование в собственной

[1] Возможно, вы предпочтете масштабировать саму базу данных, но часто это бывает непросто в реальных средах установки.

среде и следите за тем, чтобы эта среда отражала те условия, в которых будет эксплуатироваться продукт.

Типы драйверов JDBC

Драйверы JDBC делятся на четыре типа (1–4). В наши дни широко используются драйверы типа 2 (использующие низкоуровневый код) и типа 4 (написанные исключительно на Java).

Драйверы типа 1 образуют связь между ODBC (Open Database Connectivity) и JBDC. Если приложение желает взаимодействовать с базой данных через ODBC, то оно должно использовать этот драйвер. Как правило, драйверы типа 1 обладают достаточно низкой производительностью; выбирайте их только в том случае, если вы вынуждены работать со старой базой данных по протоколу ODBC.

Драйверы типа 3, как и драйверы типа 4, написаны исключительно на Java, но они предназначены для конкретной архитектуры, в которой некий связующий компонент (иногда, хотя и не так часто, это сервер приложения) предоставляет промежуточное преобразование. В этой архитектуре клиент JDBC (обычно автономная программа, хотя это может быть и сервер приложения) отправляет запрос по протоколу JDBC связующему компоненту, который преобразует запросы в протокол, специфический для базы данных, и передает запрос базе данных (и выполняет обратное преобразование ответа).

В некоторых ситуациях такая архитектура обязательна: связующий компонент может находиться в обособленном сетевом сегменте (DMZ) и обеспечивать дополнительную безопасность для подключений к базе данных. С точки зрения производительности у такой архитектуры есть как потенциальные достоинства, так и недостатки. Связующий компонент может кэшировать информацию базы данных, что позволяет снять часть нагрузки с базы данных (ускоряя ее работу) и быстрее возвращать данные клиенту (снижая задержку запроса). Однако без этого кэширования производительность снизится, так как для выполнения операции с базой данных теперь требуются два круговых сетевых запроса.

В идеальном случае плюсы и минусы скомпенсируют друг друга (или кэширование будет быстрее). Впрочем, на практике такая архитектура не получила широкого распространения. Как правило, проще разместить сам сервер на промежуточном уровне (или в DMZ, если потребуется). Затем сервер может выполнять операции с базой данных, но он не обязан предоставлять клиентам интерфейс JDBC: лучше он будет предоставлять интерфейсы сервлетов, интерфейсы веб-служб и т. д., чтобы изолировать клиента от любой информации о базе данных.

Остаются драйверы типов 2 и 4. Оба типа достаточно популярны, и ни один из них не обладает изначальными преимуществами по производительности перед другим.

Драйверы типа 2 используют низкоуровневую библиотеку для обращения к базе данных. Эти драйверы популярны у некоторых разработчиков баз данных, потому что они позволяют драйверу Java воспользоваться многолетним опытом, используемым при написании библиотеки C, которая позволяет другим программам работать с базой данных. Зависимость драйверов от низкоуровневой библиотеки усложняет их развертывание: разработчик базы данных должен предоставить платформеннозависимую низкоуровневую библиотеку для драйвера, а приложение Java должно настроить переменные среды для использования этой библиотеки. Однако с учетом усилий, уже вложенных разработчиком в библиотеку C, драйверы типа 2 обычно показывают очень хорошие результаты.

Драйверы типа 4 — драйверы, написанные исключительно на Java; они реализуют протокол передачи данных, определенный разработчиком базы данных для работы с его базой данных. Так как они пишутся исключительно на Java, это упрощает их развертывание: приложению достаточно добавить файл JAR в свои настройки `classpath`. Драйверы типа 4 обычно не уступают по производительности драйверам типа 2, потому что оба типа используют один протокол передачи данных по каналу связи. С точки зрения производительности драйвер типа 4 может потребовать написания дополнительного кода, но с точки зрения пользователя они обычно оказываются самыми простыми в использовании.

Не путайте тип драйвера (2 или 4) с тонкими/толстыми драйверами, описанными в предыдущем разделе. На практике драйверы типа 2 часто бывают толстыми, а драйверы типа 4 — тонкими, но это не обязательно. В конечном итоге вопрос о том, какой драйвер лучше — типа 2 или 4, больше зависит от среды и специфики самих драйверов. Невозможно заранее определить, какой драйвер будет работать лучше.

РЕЗЮМЕ

- Выделите время на выбор наилучшего драйвера JDBC для приложения.
- Лучший драйвер часто зависит от конкретной среды развертывания. Одно приложение может лучше работать с одним драйвером JDBC в одной среде и с другим драйвером JDBC в другой среде.
- Если у вас есть выбор, избегайте драйверов ODBC и драйверов JDBC типа 1.

Пулы подключений JDBC

Создание подключений к базе данных занимает относительно много времени, поэтому подключение JDBC — еще один прототипический объект, повторно используемый в Java.

В большинстве серверных сред все подключения JDBC выделяются из пула подключений сервера. В среде Java SE с IPA многие провайдеры JPA используют пул подключений прозрачно, и пул подключений можно настроить в файле persistence.xml. В автономной среде Java SE подключениями должно управлять приложение. В этом случае можно использовать одну из библиотек пулов подключений, доступных из разных источников. Впрочем, часто бывает проще создать подключение и сохранить его в потоково-локальной переменной в автономном приложении.

Как обычно, важно выдержать правильный баланс между памятью, занимаемой объектами пула, и объемом дополнительной уборки мусора, порождаемой использованием пула. Это особенно справедливо из-за кэшей подготовленных команд, которые будут рассматриваться в следующем разделе. Возможно, сами объекты подключений не столь велики, но кэши команд (которые существуют на уровне подключений) могут вырасти до значительных размеров.

В данном случае соблюдение правильного баланса также относится к базе данных. Каждое подключение к базе данных требует ресурсов базы данных (в дополнение к памяти, удерживаемой приложением). При добавлении подключений базе данных требуется больше ресурсов: она выделяет дополнительную память для каждой подготовленной команды, используемой драйвером JDBC. Избыток открытых подключений может отрицательно отразиться на производительности базы данных.

Общее практическое правило для пулов подключений рекомендует создать одно подключение для каждого потока в приложении. На сервере для начала определите одинаковые размеры для пула потоков и пула подключений. В автономном приложении определите размер пула подключений на основании количества потоков, создаваемых приложением. В типичном случае это обеспечит наилучшую производительность: ни одному потоку программы не придется ожидать доступного подключения базы данных, и обычно у базы данных будет достаточно ресурсов для обработки нагрузки, созданной приложением.

Но если база данных станет узким местом, то это правило только принесет вред. Наличие слишком большого числа подключений к базе данных недостаточного размера — еще одно проявление принципа, гласящего, что внедрение нагрузки в занятую систему приведет к снижению ее производительности. Использование пула подключений для регулирования объема работы, передаваемого базе данных недостаточного размера, позволит улучшить производительность в такой ситуации. Возможно, потокам приложений придется ожидать свободного подключения, но общая пропускная способность системы достигает максимума, если база данных не перегружена работой.

РЕЗЮМЕ

- Инициализация объектов подключений сопряжена с высокими затратами; в Java они обычно организуются в пулы — либо в самом драйвере JDBC, либо в JPA и других фреймворках.
- Как и с другими пулами объектов, важно настроить пул подключений, чтобы он не оказывал отрицательного влияния на уборку мусора. В данном случае также необходимо настроить пул подключений, чтобы он не оказывал отрицательного влияния на саму базу данных.

Подготовленные команды и пулы команд

В большинстве случаев для вызовов JDBC в коде следует использовать класс `PreparedStatement` вместо `Statement`. Это улучшает быстродействие: подготовленные команды позволяют базе данных повторно использовать информацию о выполняемых командах SQL. Тем самым снижается нагрузка на базу данных при последующих выполнениях подготовленной команды. Подготовленные команды также предоставляют преимущества в области безопасности и удобства программирования, особенно при определении параметров вызова.

Ключевую роль здесь играют слова «повторное использование»: при первом использовании выполнение подготовленной команды базой данных занимает больше времени, потому что она должна сгенерировать и сохранить информацию. Если команда используется только один раз, эта работа будет проделана напрасно; в таких ситуациях лучше использовать обычные команды.

Если количество обращений к базе данных невелико, то интерфейс `Statement` позволит приложению завершиться быстрее. Но даже пакетно-ориентированные программы могут содержать сотни или тысячи вызовов JDBC для одних и тех же команд SQL; в более поздних примерах этой главы будет использоваться пакетная программа для заполнения базы данных 400 896 записей. Пакетным программам с множеством вызовов JDBC — и серверам, обслуживающим многочисленные запросы на протяжении своего жизненного цикла, — лучше воспользоваться интерфейсом `PreparedStatement` (фреймворки делают это автоматически).

Подготовленные команды реализуют свой выигрыш по производительности при объединении их в пул — то есть при повторном использовании объектов `PreparedStatement`. Для правильной организации пула необходимо учесть два фактора: пул подключений JDBC и конфигурацию драйвера JDBC[1]. Параметры

[1] Разработчики баз данных часто называют организацию пулов команд кэшированием команд.

конфигурации действуют в любой программе, использующей JDBC — напрямую или через фреймворк.

Создание пула команд

Пулы подготовленных команд работают на уровне подключений. Если один поток в программе забирает подключение JDBC из пула и использует подготовленную команду с этим подключением, то информация, связанная с командой, останется действительной только для этого подключения. Второй поток, использующий второе подключение, создаст в пуле второй экземпляр подготовленной команды. В итоге каждый объект подключения будет содержать свой пул всех подготовленных команд, используемых приложением (предполагается, что все они используются на протяжении всего срока жизни приложения).

Это одна из причин, по которым автономное приложение JDBC должно использовать пул подключений. Также из нее следует, что размер пула подключений играет важную роль (для программ JPA и JDBC). Все это особенно справедливо на ранней стадии выполнения программы: при использовании подключения, которое еще не использовало конкретную подготовленную команду, первый запрос будет выполняться немного медленнее.

Размер пула подключений также важен, потому что пул кэширует подготовленные команды, которые расходуют память в куче (и нередко довольно значительную). Безусловно, повторное использование объектов в данном случае желательно, но вы должны учесть, сколько памяти занимают эти повторно используемые объекты, и убедиться в том, что они не окажут отрицательного влияния на время уборки мусора.

Управление пулами подключений

Второе, что необходимо учитывать относительно пулов подготовленных команд — какой код занимается фактическим созданием и управлением пулом. Включение и отключение пулов команд осуществляется методом `setMaxStatements()` класса `ConnectionPoolDataSource`. Пулы команд отключаются, если значение, переданное методу `setMaxStatements()`, равно 0. При этом интерфейс не определяет, где должен находиться пул команд — в драйвере JDBC или на другом уровне (например, в сервере приложения). И этого интерфейса недостаточно для некоторых драйверов JDBC, требующих дополнительной настройки.

Итак, при написании приложения Java SE, которое использует вызовы JDBC напрямую, возможны два варианта: либо драйвер JDBC должен быть настроен для создания и управления пула команд, либо создание и управление пулом должно осуществляться в коде приложения. При использовании фреймворка пулом команд часто управляет фреймворк.

Проблема в том, что в этой области нет никаких стандартов. Некоторые драйверы JDBC вообще не предоставляют механизма создания пулов команд; предполагается, что они будут использоваться только в рамках сервера приложения, который организует пул команд и хочет предоставить более простой драйвер. Некоторые серверы приложений не предоставляют возможности управления пулами; они ожидают, что драйвер JDBC займется решением этой задачи, и не хотят усложнять свой код. У обоих вариантов есть свои достоинства (хотя драйвер JDBC, не поддерживающий пула команд, перекладывает бремя на разработчика автономного приложения). В конечном итоге вам придется разобраться в происходящем и позаботиться о том, чтобы пул команд где-то создавался.

Так как стандартов нет, вы можете столкнуться с ситуацией, в которой и драйвер JDBC, и фреймворк уровня данных способны управлять пулом подготовленных команд. В этом случае важно, чтобы только одна из этих сторон была настроена для решения этой задачи. С точки зрения производительности лучший выбор снова зависит от конкретной комбинации драйвера и сервера. Как правило, можно ожидать, что драйвер JDBC лучше справится с управлением пулом команд. Так как драйвер (обычно) пишется под конкретную базу данных, можно ожидать, что для этой базы данных он будет использовать более эффективные оптимизации, чем более общий код сервера приложения.

Чтобы сделать возможной организацию пула (то есть кэширование) команд для конкретного драйвера JDBC, обратитесь к документации этого драйвера. Во многих случаях достаточно настроить драйвер так, чтобы свойству `maxStatements` было присвоено нужное значение (то есть размер пула команд). Другие драйверы могут требовать дополнительных настроек: например, драйверы Oracle JDBC требуют задания свойств, которые указывают, должно ли кэширование команд быть явным или неявным, а драйверы MySQL требуют задания специального свойства для включения кэширования команд.

РЕЗЮМЕ

- Приложения Java часто многократно выполняют одну и ту же команду SQL. В таких случаях повторное использование команд обеспечит значительный прирост производительности.
- Подготовленные команды должны объединяться в пулы на уровне подключений. Многие драйверы JDBC и фреймворки данных делают это автоматически.
- Подготовленные команды могут потреблять значительный объем памяти в куче. Размер пула команд должен тщательно настраиваться для предотвращения проблем с уборкой мусора при размещении в пуле слишком большого количества объектов.

Транзакции

У приложений имеются требования, которые в конечном итоге определяют, как в системе обрабатываются транзакции. Транзакция, требующая семантики повторяемого чтения, будет работать медленнее транзакции, требующей только семантики чтения зафиксированных данных, но эта информация вряд ли принесет пользу в приложении, которое не допускает неповторяемого чтения. Хотя в этом разделе я рассказываю о том, как использовать в приложении семантику с наименьшим внешним воздействием, скорость не должна доминировать над правильностью приложения.

С транзакциями баз данных связаны два фактора производительности. Во-первых, создание и последующее закрепление транзакции базой данных требует времени. Для этого необходимо позаботиться о том, чтобы описания изменений в базе данных полностью сохранялись на диске, чтобы журналы транзакций базы данных были согласованными, и т. д. Во-вторых, во время транзакции базы данных часто захватывается блокировка для конкретного набора данных (не всегда строки, но я буду использовать этот вариант в своем примере). Если две транзакции конкурируют за блокировку одной строки базы данных, масштабируемость приложения пострадает. С точки зрения Java ситуация полностью аналогична обсуждению конфликтных и неконфликтных блокировок из главы 9.

Для достижения оптимальной производительности необходимо учитывать оба аспекта: как запрограммировать транзакцию, чтобы сама транзакция была эффективной, и как удерживать блокировки во время транзакции, чтобы само приложение в целом могло масштабироваться.

Управление транзакциями JDBC

Транзакции присутствуют как в JDBC, так и в приложениях JPA, но в JPA управление транзакциями организовано иначе (подробности рассматриваются позднее в этой главе). Для JDBC транзакции начинаются и завершаются в зависимости от того, как используется объект `Connection`.

При простом варианте использования JDBC у подключений имеется режим автозакрепления (включаемый методом `setAutoCommit()`). Если режим автозакрепления включен (как это делается по умолчанию для большинства драйверов JDBC), каждая команда в программе JDBC составляет отдельную транзакцию. В таком случае программе не нужно предпринимать специальных действий для закрепления транзакций (более того, если метод `commit()` будет вызван, производительность приложения от этого пострадает).

При отключении режима автозакрепления транзакция неявно начинается при первом вызове к объекту подключения (например, при вызове метода `executeQuery()`). Транзакция продолжается до вызова метода `commit()` (или метода `rollback()`). Новая транзакция начнется при использовании подключения для следующего обращения к базе данных.

Закрепление транзакций обходится дорого, поэтому одна из целей заключается в том, чтобы в транзакции выполнялось как можно больше работы. К сожалению, этот принцип полностью противоречит другой цели: так как транзакции должны удерживать блокировки, они должны быть настолько короткими, насколько возможно. В этой ситуации определенно приходится искать баланс, зависящий от приложения и его требований к блокировке. В следующем разделе, посвященном изоляции транзакций и блокировкам, эта тема рассматривается более подробно; а пока рассмотрим возможности оптимизации самого процесса обработки транзакций.

Рассмотрим пример кода вставки данных в базу данных, используемую биржевым приложением. По данным за один день одна строка должна быть вставлена в таблицу `STOCKPRICE` и пять строк — в таблицу `STOCKOPTIONPRICE`. Базовый цикл для решения этой задачи выглядит так:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
         PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
            ps.clearParameters();
            ps.setBigDecimal(1, sp.getClosingPrice());
      ... настройка других параметров ...
            ps.executeUpdate();
            for (int j = 0; j < 5; j++) {
                ps2.clearParameters();
                ps2.setBigDecimal(1,
                    sp.getClosingPrice().multiply(
                        new BigDecimal(1 + j / 100.)));
                ... настройка других параметров ...
                ps2.executeUpdate();
            }
        }
    }
}
```

В полном коде цены предварительно вычисляются в массиве `stockPrices`. Если этот массив представляет данные за 2019 год, то цикл вставит 261 строку в таблицу `STOCKPRICE` (первый вызов метода `executeUpdate()`) и 1305 строк в таблицу `STOCKOPTIONPRICE` (первый цикл). В действующем по умолчанию

режиме автозакрепления это соответствует 1566 транзакциям, что обойдется довольно дорого.

Лучшей производительности можно добиться, отключив режим автозакрепления и выполнив явное закрепление в конце цикла:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    c.setAutoCommit(false);
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
         PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        ... тот же код, что и выше ....
    }
    c.commit();
}
```

С точки зрения логики это, пожалуй, имеет смысл: база данных либо будет заполнена данными за год, либо не будет содержать данных.

Если этот цикл повторяется для нескольких видов акций, приходится выбрать между закреплением всех данных сразу или закреплением всех данных для одного вида акций:

```
    try (Connection c = DriverManager.getConnection(URL, p)) {
        c.setAutoCommit(false);
        String lastSymbol = null;
        try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
             PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
            for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
            if (lastSymbol != null && !symbol.equals(lastSymbol)) {
                // Переход к обработке нового вида акций;
                // операции для предыдущего вида акций закрепляются.
          c.commit();
            }
        }
    }
    c.commit();
}
```

Конкурентное закрепление всех данных обеспечивает наивысшее быстродействие. Впрочем, в этом примере семантика приложения может потребовать, чтобы данные за каждый год закреплялись по отдельности. Иногда в попытки достижения наивысшего быстродействия вмешиваются другие требования.

Каждый раз, когда в предшествующем коде вызывается метод `executeUpdate()`, совершается удаленный вызов к базе данных, по которому должна быть выпол-

нена некоторая работа. Кроме того, при обновлении данных будет происходить захват блокировки (который гарантирует, среди прочего, что другая транзакция не сможет вставить запись с тем же видом акций и датой). В данном случае обработку транзакций можно дополнительно оптимизировать *пакетированием* (batching) операций вставки. При объединении вставок драйвер JDBC хранит их до того, как пакет будет сформирован; затем все команды передаются одним удаленным вызовом JDBC.

Пример реализации пакетирования:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
         PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
            ps.clearParameters();
            ps.setBigDecimal(1, sp.getClosingPrice());
            ... настройка других параметров ...
            ps.addBatch();
            for (int j = 0; j < 5; j++) {
                ps2.clearParameters();
                ps2.setBigDecimal(1,
                    sp.getClosingPrice().multiply(
                        new BigDecimal(1 + j / 100.)));
                ... настройка других параметров ...
                ps2.addBatch();
            }
        }
  ps.executeBatch();
  ps2.executeBatch();
    }
}
```

В этом коде также можно было бы выбрать обработку пакетов на уровне видов акций (аналогии с тем, как мы выполняли закрепление после смены вида акций). У некоторых драйверов JDBC устанавливаются ограничения на количество команд, которые могут быть объединены в пакет (а пакетирование потребляет память в приложении), так что даже если данные закрепляются в конце всей операции, возможно, обработка пакетов должна будет производиться с большей частотой.

Такие оптимизации могут обеспечивать очень большой прирост производительности. В табл. 11.1 приведено время, необходимое для вставки данных 256 акций за один год (всего 400 896 вставок).

Обратите внимание на один интересный факт, который не очевиден на первый взгляд: строки 1 и 2 различаются тем, что автозакрепление было отключено, а код явно вызывает метод `commit()` в конце каждого цикла. Строки 1 и 4 различаются

тем, что команды объединяются в пакет — но автозакрепление остается включенным. Пакет рассматривается как одна транзакция, что и объясняет однозначное соответствие между вызовами к базе данных и закреплениями (а исключение более чем 400 000 вызовов обеспечивает впечатляющее ускорение).

Таблица 11.1. Время вставки данных 256 акций в секундах

Режим программирования	Необходимое время	Вызовы к БД	Закрепления БД
Автозакрепление включено, пакетирование не используется	537 ± 2 секунды	400 896	400 896
1 закрепление для каждого вида акций	57 ± 4 секунды	400 896	256
1 закрепление для всех данных	56 ± 14 секунд	400 448	1
1 пакет на закрепление для каждого вида акций	4,6 ± 2 секунды	256	256
1 пакет на каждый вид акций; 1 закрепление	3,9 ± 0,7 секунды	256	1
1 пакет/закрепление для всех данных	3,8 ± 1 секунда	1	1

Также интересно заметить, что время для 400 896 и 256 вызовов для закрепления данных (строки 1 и 2) отличается на порядок, тогда как различия для 1 и 256 закреплений не столь значительны (строки 2 и 3 или строки 5 и 6). При 256 вызовах закрепление происходит достаточно быстро, и лишние затраты — всего лишь шум; при 400 896 вызовах они накапливаются[1].

Изоляция транзакций и блокировка

Второй фактор производительности транзакций связан с масштабируемостью базы данных, так как данные в транзакциях защищаются блокировками. Блокировка обеспечивает целостность данных; в контексте баз данных она позволяет изолировать одну транзакцию от других. Как JDBC, так и JPA поддерживают четыре основных режима изоляции транзакций баз данных, хотя и делают это по-разному.

Режимы изоляции кратко описаны ниже. Впрочем, поскольку программирование для конкретного режима транзакций в действительности не относится

[1] В первом издании этой книги при проведении тестов с Oracle 11g это было не так; строки 2 и 3, как и строки 5 и 6, заметно отличались друг от друга. Приведенные результаты были получены в версии Oracle 18c, в которой были реализованы новые улучшения.

к специфике Java, вам стоит обратиться к книге по программированию баз данных за дополнительной информацией.

Ниже перечислены основные режимы изоляции транзакций (по убыванию затрат):

- `TRANSACTION_SERIALIZABLE` — самый затратный режим транзакций; требует, чтобы все данные, используемые из транзакций, были заблокированы на все время транзакции. Это относится как к обращениям к данным по первичному ключу, так и к обращениям через условие `WHERE` — а с условием `WHERE` таблица блокируется так, чтобы никакие новые записи, удовлетворяющие условию, не могли быть добавлены за время транзакции. Транзакция всегда будет видеть одни и те же данные при многократном выполнении запроса.
- `TRANSACTION_REPEATABLE_READ` — требует, чтобы все данные блокировались на время транзакции. Однако другие транзакции могут вставлять новые записи в таблицу в любой момент времени. В этом режиме возможно *фантомное чтение*: транзакция, повторно выдающая запрос с условием `WHERE`, может получить другие данные при повторном выполнении запроса.
- `TRANSACTION_READ_COMMITTED` — в этом режиме блокируются только строки, записанные в ходе транзакции. Это приводит к неповторяемым чтениям: данные, прочитанные в одной точке транзакции, могут отличаться от данных, прочитанных в другой точке транзакции.
- `TRANSACTION_READ_UNCOMMITTED` — наименее затратный режим транзакций. Блокировки в нем не используются, так что одна транзакция может прочитать данные, записанные (но не закрепленные) другой транзакцией. Эта ситуация называется *чтением грязных данных*; проблема возникает из-за того, что первая транзакция может быть отменена (так что запись не будет выполнена), а следовательно, вторая транзакция будет работать с некорректными данными.

Базы данных работают в режиме изоляции транзакций по умолчанию: MySQL по умолчанию начинает с режима `TRANSACTION_REPEATABLE_READ`; Oracle и IBM Db2 по умолчанию начинают с режима `TRANSACTION_READ_COMMITTED`; и т. д. Здесь возможно множество комбинаций для разных баз данных. Db2 называет свой режим транзакций по умолчанию CS (сокращение от Cursor Stability) и использует другие названия для трех других режимов JDBC. Oracle не поддерживает режимы `TRANSACTION_READ_UNCOMMITTED` и `TRANSACTION_REPEATABLE_READ`. Также возможно вызвать метод `setTransactionIsolation()` для объекта подключения JDBC, чтобы база данных установила требуемый уровень изоляции (а если база данных этот уровень не поддерживает, то драйвер JDBC либо выдаст исключение, либо незаметно повысит уровень изоляции до ближайшего поддерживаемого уровня).

TRANSACTION_NONE И АВТОЗАКРЕПЛЕНИЕ

В спецификации JDBC определен пятый режим изоляции `TRANSACTION_NONE`. Теоретически этот режим невозможно задать при вызове `setTransactionIsolation()`, потому что этот уровень не может быть назначен уже существующей транзакции. Некоторые драйверы JDBC (особенно для Db2) позволяют совершать такие вызовы (и даже использовать этот режим по умолчанию). Другие драйверы позволяют задать режим `TRANSACTION_NONE` в свойствах, используемых для инициализации драйвера.

Строго говоря, команда, выполняемая из подключения с семантикой `TRANSACTION_NONE`, не может закреплять данные в базе данных; она должна быть запросом только для чтения. Если данные записываются, должна присутствовать некоторая форма блокировки; в противном случае, если один пользователь запишет длинную строку в таблицу с семантикой `TRANSACTION_NONE`, второй пользователь может увидеть неполную строку, записанную в таблице. Базы данных могут работать в этом режиме, хотя это было бы нетипично; как минимум ожидается, что запись данных в одну таблицу происходит атомарно. Следовательно, операция, которая записывает данные, на практике будет обладать (как минимум) семантикой `TRANSACTION_READ_UNCOMMITTED`.

Запрос `TRANSACTION_NONE` не может быть закреплен, но драйверы JDBC, использующие `TRANSACTION_NONE`, могут допускать написание запросов при включенном автозакреплении. Это означает, что база данных рассматривает каждый запрос как отдельную транзакцию. Даже в этом случае, поскольку база данных (скорее всего) не позволит другим транзакциям увидеть результаты неполной записи, в действительности используется семантика `TRANSACTION_READ_UNCOMMITTED`.

Для простых программ JDBC этого достаточно. Чаще — и особенно при использовании с JPA — в программах может потребоваться использование разных уровней изоляции с данными в пределах одной транзакции. В приложении, которое запрашивает информацию о работнике, доступ к записи работника должен быть защищен: эти данные должны рассматриваться как находящиеся на уровне `TRANSACTION_REPEATABLE_READ`. Но транзакция также с большой вероятностью будет обращаться к данным в других таблицах — например, таблице, в которой хранится табельный номер работника. Блокировать эти данные во время транзакции не нужно, так что обращение к этой записи вполне может выполняться на уровне `TRANSACTION_READ_COMMITTED` (а возможно, и ниже).

JPA позволяет задавать уровни блокировки на уровне сущностей (и конечно, сущность — по крайней мере обычно — соответствует строке в базе данных).

Так как правильная настройка этих уровней изоляции может быть непростой задачей, проще воспользоваться JPA, чем управлять блокировками в командах JDBC. Тем не менее в приложениях JDBC можно использовать разные уровни блокировки, применяя те же семантики пессимистичных и оптимистичных блокировок, которые использует JPA (а если вы незнакомы с этими семантиками, этот пример станет хорошим введением в тему).

На уровне JDBC обычно устанавливается уровень изоляции подключения `TRANSACTION_READ_UNCOMMITTED`, а затем блокировка включается явно только для тех данных, которые должны блокироваться в ходе транзакции.

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    c.setAutoCommit(false);
    c.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
    try (PreparedStatement ps1 = c.prepareStatement(
        "SELECT * FROM employee WHERE e_id = ? FOR UPDATE")) {
        ... обработка информации из ps1 ...
    }
    try (PreparedStatement ps2 = c.prepareStatement(
           "SELECT * FROM office WHERE office_id = ?")) {
        ... обработка информации из ps2 ...
    }
    c.commit();
}
```

Команда `ps1` получает явную блокировку для таблицы данных `employee`: ни одна другая транзакция не сможет обратиться к этой строке на протяжении всей транзакции. Синтаксис SQL для решения этой задачи не является стандартным. Чтобы узнать, как добиться желаемого уровня блокировки, следует обратиться к документации разработчика вашей базы данных; чаще всего для этого включается условие `FOR UPDATE`. Такая разновидность блокировки называется *пессимистической*. Она активно предотвращает обращение к данным со стороны других транзакций.

Производительность блокировок часто удается улучшить за счет использования оптимистичных блокировок. Если обращение к данным является неконфликтным, оно приведет к значительному приросту производительности. Если же обращения к данным сопровождаются хотя бы незначительной конкуренцией, программирование усложняется.

В базе данных для реализации оптимистичного параллелизма используется столбец версии. При чтении данных из строки команда должна включать как нужные данные, так и столбец версии. Например, для получения информации о работнике можно воспользоваться следующей командой SQL:

```
SELECT first_name, last_name, version FROM employee WnHERE e_id = 5058;
```

Этот запрос вернет имя и фамилию (например, `Scott` и `Oaks`), а также текущий номер версии (скажем, 1012). Когда придет время завершения транзакции, транзакция обновит столбец версии:

```
UPDATE employee SET version = 1013 WHERE e_id = 5058 AND version = 1012;
```

Если строка данных, к которой обращается приложение, требует семантики повторяемого чтения или сериализации, это обновление должно быть выполнено даже в том случае, если данные только читались в ходе транзакции, — эти уровни изоляции требуют блокировки данных только для чтения, используемых в транзакции. Для семантики чтения закрепленных данных столбец версии необходимо обновлять только в том случае, если другие данные в строке тоже обновляются.

Если в этой схеме две транзакции конкурентно используют запись работника, каждая из них прочитает номер версии 1012. Первая завершенная транзакция успешно обновит номер версии до 1013 и продолжит работу. Вторая транзакция не сможет обновить запись работника — записи с номером версии 1012 уже не существует, так что попытка выполнения команды обновления SQL завершится неудачей. Транзакция столкнется с исключением и будет отменена.

В этом проявляется крупнейшее различие между оптимистичной блокировкой базы данных и атомарных примитивов Java: при программировании баз данных транзакция, получившая такое исключение, не будет (и не может быть) прозрачно повторена. Если вы программируете непосредственно для JDBC, метод `commit()` получит исключение `SQLException`; в JPA ваше приложение получит исключение `OptimisticLockException` при закреплении транзакции.

В зависимости от точки зрения это может быть как хорошо, так и плохо. В главе 9 мы рассмотрели производительность атомарных средств, использующих средства на базе CAS для предотвращения явной синхронизации. Фактически эти средства используют оптимистичный параллелизм с бесконечными автоматическими повторными попытками. В ситуациях с высокой конкуренцией производительность пострадает, если многочисленные повторные попытки поглощают значительные ресурсы процессора, хотя на практике эта проблема обычно не возникает. С базами данных дело обстоит намного хуже, так как код, выполняемый в транзакции, обычно намного сложнее простого увеличения значения, хранящегося в ячейке памяти. Повторная попытка неудачной оптимистичной транзакции в базе данных сопряжена с гораздо большим риском возникновения бесконечной спирали повторных попыток. Кроме того, часто бывает достаточно сложно автоматически определить, какую именно операцию следует повторить.

Таким образом, отказ от прозрачных повторных попыток — это хорошее (а часто и единственно возможное) решение, но, с другой стороны, это означает, что за обработку исключения теперь отвечает приложение. Приложение может вы-

брать между повторной попыткой выполнения транзакции (возможно, один или два раза), оно может запросить у пользователя другие данные или же просто проинформировать пользователя о том, что операция завершилась неудачей. Единственно правильного ответа на все случаи жизни нет.

Итак, оптимистичная блокировка лучше всего работает при малом риске коллизий между двумя источниками. Представьте совместный расчетный счет: есть небольшая вероятность того, что супруги, находящиеся в разных частях города, попытаются в точности одновременно снять деньги со счета. Это приведет к исключению оптимистичной блокировки для одного из них. Но даже если это произойдет, просьба к одному из супругов повторить операцию не слишком обременительна, и на этот раз риск исключения оптимистичной блокировки близок к нулю (не будем учитывать, с какой частотой мы снимаем средства с банковской карты). Сравните эту ситуацию с использованием нашего приложения с биржевыми котировками. В реальном мире данные обновляются настолько часто, что применять к ним оптимистичную блокировку было бы нерационально. На самом деле биржевые приложения часто вообще не используют блокировку там, где это возможно, просто из-за объема изменений (хотя реальные обновления данных на торгах потребуют некоторой степени блокировки).

РЕЗЮМЕ

- Транзакции влияют на быстродействие приложений в двух отношениях: их закрепление обходится дорого, а блокировка, связанная с транзакциями, может помешать масштабированию базы данных.
- Эти два эффекта конфликтуют друг с другом: слишком долгое ожидание закрепления транзакции увеличивает время удержания блокировок, связанных с транзакцией. Особенно для транзакций, использующих более строгую семантику, баланс смещается в сторону более частых закреплений, а не к удержанию блокировок в течение большего времени.
- Для точного управления транзакциями в JDBC используйте уровень по умолчанию `TRANSACTION_READ_UNCOMMITTED` и явно устанавливайте блокировки данных, когда потребуется.

Обработка итоговых наборов

Типичные приложения баз данных работают с диапазонами данных. Например, биржевое приложение может работать с диапазоном цен для отдельного вида акций. История цен загружается одной командой `SELECT`:

```
SELECT * FROM stockprice WHERE symbol = 'TPKS' AND
  pricedate >= '2019-01-01' AND pricedate <= '2019-12-31';
```

Команда возвращает 261 строку данных. Если бы вас также интересовали опционные цены, то был бы выполнен аналогичный запрос, который возвращает впятеро больше данных. Код SQL для загрузки всех записей в базе данных (256 видов акций за 1 год) вернет 400 896 строк данных:

```
SELECT * FROM stockprice s, stockoptionprice o WHERE
  o.symbol = s.symbol AND s.pricedate >= '2019-01-01'
  AND s.pricedate <= '2019-12-31';
```

Чтобы использовать эти данные, код должен перебрать весь итоговый набор:

```
try (PreparedStatement ps = c.prepareStatement(...)) {
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            ... чтение текущей строки ...
        }
    }
}
```

Вопрос в том, где должны храниться данные этих 400 896 строк. Если при вызове `executeQuery()` возвращается весь набор, то у приложения в куче окажется огромный блок данных, что, скорее всего, приведет к проблемам с уборкой мусора (и не только). Если вместо этого при вызове `next()` будет возвращаться только одна строка данных, процесс обработки итогового набора породит значительный круговой трафик между приложением и базой данных.

Как обычно, единственно правильного ответа нет: в одних случаях более эффективно хранить основной массив данных в базе данных и извлекать их по мере надобности, в других — загрузить все данные непосредственно при выполнении запроса. Чтобы управлять этим процессом, используйте метод `setFetchSize()` объекта `PreparedStatement` для передачи драйверу JDBC количества передаваемых записей.

Значение по умолчанию зависит от драйвера JDBC; например, в драйверах JDBC Oracle значение по умолчанию равно 10. При вызове метода `executeQuery()` в цикле, приведенном выше, база данных будет возвращать 10 строк данных, которые буферизуются во внутренней реализации драйвером JDBC. Каждый из первых 10 вызовов метода `next()` будет обрабатывать одну из буферизованных строк. 11-й вызов обратится к базе данных для загрузки следующих 10 записей, и т. д.

Хотя конкретное значение изменяется, драйверы JDBC обычно задают относительно небольшой размер выборки по умолчанию. Это разумно во многих обстоятельствах: в частности, он снижает вероятность проблем с памятью в при-

ложении. Если быстродействие метода `next()` (или быстродействие первого get-метода итогового набора) оказывается особенно низким, рассмотрите возможность увеличения размера выборки.

ДРУГИЕ СПОСОБЫ ОПРЕДЕЛЕНИЯ РАЗМЕРА ВЫБОРКИ

Я порекомендовал использовать метод `setFetchSize()` с объектом (подготовленной) команды, но этот метод также существует в интерфейсе `ResultSet`. В любом случае размер всего лишь является рекомендацией. Драйвер JDBC может проигнорировать это значение, округлить его до другого значения или вообще сделать все, что сочтет нужным. Никаких гарантий нет, но определение этого значения перед выполнением запроса (например, с объектом команды) значительно повысит вероятность соблюдения рекомендации.

Некоторые драйверы JDBC также позволяют задать размер выборки по умолчанию при создании подключения; это делается передачей свойства методу `getConnection()` объекта `DriverManager`. Если этот путь вам покажется более удобным, обращайтесь к документации разработчика.

РЕЗЮМЕ

- В приложениях, обрабатывающих большие объемы данных из запроса, можно рассмотреть возможность изменения размера выборки данных.
- Следует выдержать баланс между загрузкой слишком большого объема данных в приложении (что повышает нагрузку на уборщика мусора) и частыми обращениями к базе данных для получения набора данных.

JPA

Производительность JPA напрямую зависит от производительности используемого драйвера JDBC, причем большая часть факторов производительности, относящихся к драйверу JDBC, также относится к JPA. Также у JPA имеются дополнительные факторы производительности.

Многие улучшения производительности JPA достигаются за счет изменения байт-кода классов сущностей. В большинстве серверных фреймворков это происходит прозрачно. В среде Java SE важно проследить за тем, чтобы обработка байт-кода была правильно настроена. В противном случае производительность

приложений JPA становится непредсказуемой: поля, которые должны загружаться в отложенном режиме, могут загружаться немедленно; сохраняемая в базе данных информация может оказаться избыточной; данные, которые должны находиться в кэше JPA, могут повторно загружаться из базы данных и т. д.

Нет стандарта обработки байт-кода, определенного на уровне JPA. Как правило, такая обработка становится частью компиляции — после того, как классы сущностей будут откомпилированы (но до их загрузки в файлы JAR или выполнения JVM), они обрабатываются специальным постпроцессором, зависящим от реализации. Постпроцессор «улучшает» байт-код, создавая измененный файл класса с нужными оптимизациями. Например, Hibernate делает это при помощи плагина Maven или Gradle во время компиляции.

Некоторые реализации JPA также предоставляют возможность динамической модификации байт-кода при загрузке классов в JVM. Для этого в JVM должен существовать агент, который получает уведомление о загрузке классов; агент перехватывает загрузку классов и изменяет байт-код перед его использованием для определения класса. Агент задается в командной строке приложения; например, для EclipseLink включается аргумент `-javaagent:path_to/eclipselink.jar`.

Оптимизация записи JPA

В JDBC были рассмотрены два важнейших приема повышения производительности: повторное использование подготовленных команд и пакетное выполнение обновлений. Обе оптимизации могут быть достигнуты в JPA, но конкретный способ их выполнения зависит от используемой реализации JPA; в JPA API нет вызовов для решения этой задачи. Для Java SE такие оптимизации обычно требуют настройки конкретного свойства в файле persistence.xml приложения.

ЗАПИСЬ МЕНЬШЕГО КОЛИЧЕСТВА ПОЛЕЙ

Один из распространенных приемов оптимизации записи в базу данных — запись только тех полей, которые были изменены. Код, используемый системой отдела кадров для удвоения моей зарплаты, может прочитать 20 полей из моей записи работника, но обратно в базу данных должно быть записано только одно (очень важное) поле.

Следует ожидать, что реализации JPA будут выполнять эту оптимизацию прозрачно. Это одна из причин для модификации байт-кода JPA, потому что именно этот процесс используется провайдером JPA для отслеживания изменяемых значений в коде. Если код JPA будет правильно модифицирован, код SQL для записи удвоенной зарплаты в базу данных обновит только этот один столбец.

Например, при использовании эталонной реализации JPA EclipseLink повторное использование команд активизируется добавлением следующего свойства в файл persistence.xml:

```
<property name="eclipselink.jdbc.cache-statements" value="true" />
```

Это свойство включает повторное использование команд в реализации EclipseLink. Если драйвер JDBC способен реализовать управление пулом команд, обычно лучше включить кэширование команд в драйвере и исключить свойство из конфигурации JPA.

Пакетирование команд в эталонной реализации JPA включается добавлением следующих свойств:

```
<property name="eclipselink.jdbc.batch-writing" value="JDBC" />
<property name="eclipselink.jdbc.batch-writing.size" value="10000" />
```

Драйверы JDBC не могут автоматически реализовать пакетирование команд, поэтому это свойство полезно задавать практически во всех случаях. Размером пакета можно управлять двумя способами: во-первых, вы можете задать свойство `size`, как сделано в этом примере. Во-вторых, приложение может периодически вызывать метод `flush()` менеджера свойств, что приведет к немедленному выполнению всех пакетированных команд.

В табл. 11.2 приведены данные о влиянии повторного использования команд и пакетирования на операции создания и записи сущностей в базу данных.

Таблица 11.2. Продолжительность вставки данных 256 видов акций через JPA

Режим программирования	Необходимое время
Пакетирование не используется, без пула команд	83 ± 3 секунды
Пакетирование не используется, с пулом команд	64 ± 5 секунд
Пакетирование используется, без пула команд	10 ± 0,4 секунды
Пакетирование используется, с пулом команд	10 ± 0,3 секунды

РЕЗЮМЕ

- Приложения JPA, как и приложения JDBC, выиграют от ограничения количества операций записи в базу данных (с потенциальными потерями на удержание блокировок транзакций).

- Кэширование команд может быть достигнуто на уровне JPA уровня JDBC. Но сначала нужно исследовать возможности кэширования на уровне JDBC.
- Пакетирование обновлений JPA может быть включено на декларативном уровне (в файле persistence.xml) или на программном уровне (вызовом метода `flush()`).

Оптимизация операций чтения JPA

Оптимизация того, когда и как JPA читает данные из базы данных, — задача более сложная, чем кажется на первый взгляд, потому что JPA кэширует данные в надежде, что они могут использоваться для выполнения будущих запросов. Обычно это хорошо для производительности, но это означает, что сгенерированный средствами JPA код SQL, используемый для чтения, по крайней мере на первый взгляд кажется субоптимальным. Выборка данных оптимизирована под потребности кэша JPA, а не под конкретный выполняемый запрос.

Технические подробности кэширования будут рассмотрены в следующем разделе. А пока рассмотрим базовые возможности применения оптимизаций чтения из базы данных в JPA. JPA читает данные из базы данных в трех случаях: при вызове метода `find()` объекта `EntityManager`, при выполнении запроса JPA и при переходе кода к новой сущности с использованием отношения другой сущности. В классе, представляющем акции, последнее означает вызов метода `getOptions()` для сущности `Stock`.

Вызов метода `find()` — самый прямолинейный случай: в нем задействована только одна строка, и (как минимум) эта одна строка читается из базы данных. Единственное, чем можно управлять, — это объем загружаемых данных. JPA может прочитать только часть полей в строке данных, может прочитать всю строку или же осуществить предварительную выборку других сущностей, связанных с читаемой строкой. Эти оптимизации также применимы к запросам.

Возможны два пути: чтение меньшего объема данных (потому что какие-то данные не понадобятся) или чтение большего объема за раз (потому что эти данные определенно понадобятся в будущем).

Чтение меньшего объема данных

Чтобы читать меньше данных, укажите, что некоторые поля должны загружаться в отложенном режиме. При чтении сущности поля с аннотацией `lazy` будут исключены из кода SQL, используемого для загрузки данных. Если get-метод

для этого поля когда-либо будет выполнен, это будет означать дополнительное обращение к базе данных для получения этих данных.

Эта аннотация редко применяется к простым столбцам базовых типов, однако вам стоит рассмотреть ее применение для столбцов, содержащих большие `BLOB`- и `CLOB`-объекты:

```
@Lob
@Column(name = "IMAGEDATA")
@Basic(fetch = FetchType.LAZY)
private byte[] imageData;
```

В данном случае сущность представляет таблицу, в которой хранятся двоичные графические данные. Двоичные данные велики, и пример предполагает, что они не должны загружаться без необходимости. Отказ от загрузки ненужных данных в этом случае служит двум целям: он ускоряет выполнение SQL при чтении сущности и экономит много памяти, что приводит к снижению нагрузки на уборку мусора.

ГРУППЫ ВЫБОРКИ

Если сущность содержит поля с отложенной загрузкой, то обычно эти поля загружаются по одному при обращениях к ним. А что, если три поля сущности используют отложенную загрузку и если потребуется одно из этих полей, то потребуются и все остальные? Тогда будет разумнее загрузить все отложенные поля сразу.

Сделать это со стандартным JPA невозможно, но большинство реализаций JPA позволяет определить группу выборки для решения этой задачи. Группы выборки позволяют указать, что некоторые поля с отложенной загрузкой должны загружаться как группа при обращении к одному из них. Как правило, можно определить несколько независимых групп полей; каждая группа будет загружаться по требованию.

Так как группы выборки не входят в стандарт JPA, код, использующий группы выборки, будет связан с одной конкретной реализацией JPA. Но если эта возможность вам покажется полезной, обращайтесь к документации реализации JPA за подробностями.

Также следует помнить, что аннотация `lazy` в конечном итоге всего лишь является рекомендацией для реализации JPA. Реализация JPA все равно может потребовать, чтобы база данных немедленно предоставляла эти данные.

С другой стороны, возможно, другие данные должны загружаться заранее — например, при выборке одной сущности также должны возвращаться данные других (связанных) сущностей. Этот механизм называется *упреждающей выборкой*, и для него используется похожая аннотация:

```
@OneToMany(mappedBy="stock", fetch=FetchType.EAGER)
private Collection<StockOptionPriceImpl> optionsPrices;
```

По умолчанию связанные сущности загружаются посредством упреждающей выборки для типов отношений `@OneToOne` и `@ManyToOne` (поэтому к ним также можно применять противоположную оптимизацию: снабдить их пометкой `FetchType.LAZY`, если они почти никогда не используются).

Эта аннотация тоже является всего лишь рекомендацией для реализации JPA, но по сути она означает, что каждый раз, когда загружается цена акций, также должны загружаться все связанные с ней цены опционов. Будьте внимательны: нередко считается, что для немедленной выборки в сгенерированный код SQL включается секция JOIN. Для типичных провайдеров JPA это не так; они выдают единственный запрос SQL для выборки первичного объекта, после чего одну или несколько команд SQL для выборки других, связанных объектов. Из простого метода `find()` управлять этим невозможно: если вам необходима команда `JOIN`, придется использовать запрос и запрограммировать в нем условие `JOIN`.

Использование JOIN в запросах

JPQL (JPA Query Language) не позволяет указать, какие поля объекта должны быть прочитаны из базы данных. Взгляните на следующий запрос JPQL:

```
Query q = em.createQuery("SELECT s FROM StockPriceImpl s");
```

Этот запрос всегда генерирует следующую команду SQL:

```
SELECT <список не-LAZY полей> FROM StockPriceTable
```

Если вы хотите читать меньше полей в сгенерированном SQL, у вас нет другого варианта, кроме как пометить их как отложенные. Аналогичным образом для полей, помеченных как отложенные, нет реальной возможности выбрать их по запросу.

Если между сущностями существуют отношения, сущности можно явно объединить в запросе в JPQL, в результате чего исходные сущности и связанные с ними сущности будут загружены за один раз.

Например, для сущностей в биржевом приложении можно выдать следующий запрос:

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s " +
                         "JOIN FETCH s.optionsPrices");
```

В результате будет сгенерирована команда SQL следующего вида:

```
SELECT t1.<fields>, t0.<fields> FROM StockOptionPrice t0, StockPrice t1
WHERE ((t0.SYMBOL = t1.SYMBOL) AND (t0.PRICEDATE = t1.PRICEDATE))
```

ДРУГИЕ МЕХАНИЗМЫ ВЫБОРКИ С СОЕДИНЕНИЕМ

Многие провайдеры JPA позволяют создать выборку с соединением, задав соответствующие рекомендации для запроса. Например, в EclipseLink этот код сгенерирует запрос JOIN:

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s");
q.setQueryHint("eclipselink.join-fetch", "s.optionsPrices");
```

Некоторые провайдеры JPA также поддерживают специальную аннотацию `@JoinFetch`, которая может использоваться с отношением.

Конкретный код SQL зависит от провайдера JPA (пример взят из EclipseLink), но общий процесс выглядит так.

Выборка с соединением действительна для отношений между сущностями независимо от того, помечено ли отношение как отложенное или немедленное. Если соединение выполняется с отложенным отношением, сущности с отложенной аннотацией, удовлетворяющие запросу, все равно извлекаются из базы данных, и если эти сущности будут использоваться позднее, дополнительного обращения к базе данных не понадобится.

Если все данные, возвращаемые запросом с использованием выборки с соединением, будут использоваться, то выборка с соединением часто обеспечивает большой прирост производительности. Однако выборка с соединением также взаимодействует с кэшем JPA неочевидным образом. Пример приведен в разделе «Кэширование JPA» на с. 418: вы должны хорошо понимать последствия, прежде чем писать собственные запросы с использованием выборки с соединением.

Пакетирование и запросы

Запросы JPA обрабатываются по той же схеме, что и запросы JDBC, порождающие итоговый набор: реализация JPA может получить все результаты сразу,

может получать результаты по одному при переборе результатов приложением, или же получать по несколько результатов за раз (по аналогии с тем, как размер выборки работает с JDBC).

Стандартных средств управления этим процессом нет, но разработчики JPA используют специальные механизмы для назначения размера выборки. В EclipseLink размер выборки задается рекомендацией для запроса:

```
q.setHint("eclipselink.JDBC_FETCH_SIZE", "100000");
```

Hibernate вместо этого предоставляет специальную аннотацию `@BatchSize`.

При обработке очень большого набора данных код может перебирать список, возвращенный запросом, по страницам. Такой перебор естественным образом связан с тем, как данные могут выводиться для пользователя на веб-странице: выводится подмножество данных (допустим, 100 строк) со ссылками на следующую и предыдущую страницу для (страничного) перебора данных.

Эта задача решается определением диапазона для запроса:

```
Query q = em.createNamedQuery("selectAll");
query.setFirstResult(101);
query.setMaxResults(100);
List<? implements StockPrice>  = q.getResultList();
```

Код возвращает список, подходящий для вывода на второй странице веб-приложения: элементы 101–200. Получение данных из необходимого диапазона будет более эффективным, чем загрузка 200 строк, первые 100 из которых будут отброшены.

Обратите внимание на то, что в этом примере используется именованный запрос (метод `createNamedQuery()`) вместо произвольного (метод `createQuery()`). Во многих реализациях JPA именованные запросы работают быстрее: реализация JPA почти всегда использует подготовленную команду с параметрами, используя кэш команд. Ничто не мешает реализациям JPA использовать аналогичную логику для неименованных произвольных запросов, хотя это несколько усложнит реализацию, и реализация JPA может по умолчанию каждый раз создавать новую команду (то есть объект `Statement`).

РЕЗЮМЕ

- JPA может выполнить ряд оптимизаций для ограничения (или увеличения) объема данных, читаемых за одну операцию.
- Большие поля (например, BLOB-объекты), которые используются относительно редко, должны загружаться в отложенном режиме в сущностях JPA.

- Если есть отношение, связывающее сущности JPA, данные связанных элементов могут загружаться немедленно или в отложенном режиме. Выбор зависит от потребностей приложения.
- При немедленной загрузке зависимостей именованные запросы могут использоваться для выдачи отдельной команды SQL с условием JOIN. Учтите, что это влияет на кэш JPA и это не всегда стоит делать (как описано в следующем разделе).
- Чтение данных с использованием именованных запросов всегда выполняется быстрее чтения обычными запросами, потому что реализации JPA проще использовать `PreparedStatement` для именованных запросов.

Кэширование JPA

Одна из канонических структур из области производительности в Java — определение промежуточного уровня, который кэширует данные от ресурсов баз данных внутренней подсистемы. Уровень Java выполняет функции, полезные с точки зрения архитектуры (например, запрет прямых обращений к базе данных со стороны клиентов). С точки зрения производительности кэширование часто используемых данных на уровне Java может значительно ускорить время отклика для клиентов.

Технология JPA проектировалась с учетом этой архитектуры. В JPA существуют две разновидности кэшей. Каждый экземпляр менеджера сущностей сам по себе является кэшем: он обеспечивает локальное кэширование данных, прочитанных в ходе транзакции. Он также выполняет локальное кэширование данных, записанных в ходе транзакции; данные передаются в базу данных только при закреплении транзакции. Программа может содержать несколько экземпляров менеджера сущностей, каждый из которых выполняет свою транзакцию и каждый имеет собственный локальный кэш. (В частности, менеджеры сущностей, внедряемые в серверы Java, являются отдельными экземплярами.)

Когда менеджер сущностей закрепляет транзакцию, все данные в локальном кэше могут быть объединены в глобальный кэш. Глобальный кэш совместно используется всеми менеджерами сущностей в приложении. Глобальный кэш также называется *кэшем 2-го уровня* или *кэшем L2*; кэш в менеджере сущностей называется *кэшем уровня 1* или *кэшем L1*.

В кэше транзакций менеджера сущностей (кэш L1) настраивать практически нечего, и кэш L1 включается во всех реализациях JPA. С кэшем L2 дело обстоит иначе: он поддерживается многими реализациями JPA, но не всегда включается по умолчанию (например, это делается в EclipseLink, но не в Hibernate). После

включения настройка и использование кэша L2 могут значительно повлиять на производительность.

Кэш JPA работает только с сущностями, к которым производятся обращения по первичным ключам (то есть элементам, которые читаются вызовом метода `find()`) или элементам, прочитанным в результате обращения (или немедленной загрузки) к связанным сущностям. Когда менеджер сущностей пытается найти объект по первичному ключу или по отношению, он может обратиться к кэшу L2 и вернуть объект(-ы), если он будет найден; таким образом экономится обращение к базе данных.

Данные, прочитанные по запросу, не сохраняются в кэше L2. В некоторых реализациях JPA имеется механизм кэширования результатов запросов, специфический для разработчика, но эти результаты будут использованы повторно только при повторном выполнении точно такого же запроса. Даже если реализация JPA поддерживает кэширование запроса, сами сущности не сохраняются в кэше L2 и не могут быть возвращены при последующем вызове метода `find()`.

Связи между кэшем L2, запросами и загрузкой объектов оказывают разностороннее влияние на производительность. Для анализа этого влияния будет использоваться код, основанный на следующем цикле:

```
EntityManager em = emf.createEntityManager();
Query q = em.createNamedQuery(queryName);
List<StockPrice> l = q.getResultList(); (1)
for (StockPrice sp : l) {
    ... обработка sp ...
    if (processOptions) {
        Collection<? extends StockOptionPrice> options = sp.getOptions();(2)
        for (StockOptionPrice sop : options) {
      ... обработка sop ...
        }
    }
}
em.close();
```

(1) — точка вызова 1

(2) — точка вызова 2

Из-за кэша L2 этот цикл покажет один результат при первом выполнении и другой (обычно более быстрый) результат при последующих выполнениях. Конкретные различия в производительности зависят от специфики запроса и отношений сущностей. В нескольких ближайших подразделах результаты будут описаны более подробно.

В некоторых случаях различия в примере происходят от различий в конфигурациях JPA, но также и из-за того, что некоторые тесты выполняются без перебора

отношений между классами `Stock` и `StockOptions`. В этих тестах без перебора отношения значение `processOptions` в цикле равно `false`; реально используются только объекты `StockPrice`.

Кэширование по умолчанию (отложенная загрузка)

В примере кода цены акций загружаются посредством именованного запроса. По умолчанию для загрузки данных акций выполняется простой запрос:

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceImpl s ORDER BY s.id.symbol")
```

Класс `StockPrice` связан с классом `StockOptionPrice` отношением `@OneToMany` с использованием переменной экземпляра `optionsPrices`:

```
@OneToMany(mappedBy="stock")
private Collection<StockOptionPrice> optionsPrices;
```

Отношения `@OneToMany` по умолчанию используют отложенную загрузку. В табл. 11.3 приведены данные о времени выполнения этого цикла.

Таблица 11.3. Продолжительность чтения данных 256 видов акций (в конфигурации по умолчанию)

Тестовый сценарий	Первое выполнение	Последующие выполнения
Отложенное отношение	22,7 ± 2 секунды (66 817 вызовов SQL)	1,1 ± 0,7 секунды (1 вызов SQL)
Отложенное отношение, без перебора	2,0 ± 0,3 секунды (1 вызов SQL)	1,0 ± 0,02 секунды (1 вызов SQL)

При первом выполнении цикла в этом сценарии (для 256 видов акций за один год) код JPA выполняет одну команду SQL в вызове метода `executeQuery()`. Эта команда выполняется в точке вызова 1 в приведенном листинге.

По мере того как код в цикле перебирает виды акций и посещает каждую коллекцию опционов, JPA выдает команды SQL для получения всех опционов, связанных с конкретной сущностью (то есть читает сразу всю коллекцию для одной комбинации вид акций/дата). Это происходит в точке вызова 2 и приводит к выполнению 66 816 отдельных команд `SELECT` (261 день × 256 видов акций); итого 66 817 вызовов.

В этом примере первое выполнение цикла заняло почти 23 секунды. На второе выполнение кода потребовалось немногим более 1 секунды. Дело в том, что при втором выполнении цикла выполняется только именованный запрос. Сущности, прочитанные через отношение, все еще находятся в кэше L2, поэтому в данном

случае обращения к базе данных не нужны. (Вспомните, что кэш L2 работает только для сущностей, загруженных из отношения или операции поиска. Таким образом, сущности опционов могут быть найдены в кэше L2, но цены акций — потому что они были загружены из запроса — в кэше L2 отсутствуют и должны быть загружены заново.)

Вторая строка в табл. 11.3 представляет код, который не обходит все опционы в отношении (то есть переменная `processOptions` равна `false`). В этом случае код работает намного быстрее: первая итерация цикла занимает 2 секунды, а последующие итерации — всего 1 секунду. (Различия в производительности между этими двумя случаями обусловлены периодом разогрева компилятора. Разогрев также происходил в первом примере, хотя он не был столь заметен.)

Кэширование и немедленная загрузка

В следующих двух экспериментах отношения между ценами акций и ценами опционов переопределяются так, чтобы цены опционов загружались немедленно.

НЕМЕДЛЕННАЯ ЗАГРУЗКА ОТНОШЕНИЙ

Независимо от того, является ли выборка отношения отложенной или немедленной, этот цикл выполняет 66 816 команд SELECT для получения цен опционов (как упоминалось в предыдущем разделе, JOIN по умолчанию не используется).

Различия между немедленной и отложенной загрузкой отношения в этой ситуации проявляются во времени выполнения этих команд SQL. Если отношение помечено аннотацией немедленной загрузки, итоговый набор обрабатывается непосредственно при выполнении запроса (в вызове `getResultList()`). Фреймворк JPA просматривает каждую сущность, возвращаемую этим вызовом, и выполняет команду SQL для получения связанных с ними сущностей. Все эти команды SQL выполняются в точке вызова 1 — тогда как в случае немедленного отношения в точке вызова 2 никакие команды SQL не выполняются.

Если отношение помечено аннотацией отложенной загрузки, в точке вызова 1 загружаются только цены акций (с использованием именованного запроса). Цены опционов для отдельных видов акций загружаются при переборе отношения в точке вызова 2. Цикл выполняется 66 816 раз, что приводит к выполнению 66 816 вызовов SQL.

Впрочем, независимо от того, когда используется SQL, количество команд SQL остается прежним — предполагается, что в примере с отложенной загрузкой все данные реально используются.

Когда используются все данные (то есть первые строки в табл. 11.3 и 11.4), производительность случаев немедленной и отложенной загрузки практически одинакова. Но если данные отношений реально не используются (вторые строки в обеих таблицах), отложенное отношение экономит время — особенно при первом выполнении цикла. Последующие выполнения цикла не экономят время, так как код немедленной загрузки не перезагружает данные в этих последующих итерациях; данные загружаются из кэша L2.

Таблица 11.4. Продолжительность чтения данных 256 видов акций (немедленная загрузка)

Тестовый сценарий	Первое выполнение	Последующие выполнения
Немедленное отношение	23 ± 1,0 секунды (66 817 вызовов SQL)	1,0 ± 0,8 секунды (1 вызов SQL)
Немедленное отношение, без перебора	23 ± 1,3 секунды (66 817 вызовов SQL)	1,0 ± 0,5 секунды (1 вызов SQL)

Выборка с соединением и кэширование

Как обсуждалось в предыдущем разделе, запрос можно записать с явным использованием синтаксиса `JOIN`:

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceEagerLazyImpl s " +
    "JOIN FETCH s.optionsPrices ORDER BY s.id.symbol")
```

При использовании именованного запроса (с полным перебором) будут получены данные, представленные в табл. 11.5.

Таблица 11.5. Продолжительность чтения данных 256 видов акций (запрос JOIN)

Тестовый сценарий	Первое выполнение	Последующие выполнения
Конфигурация по умолчанию	22,7 ± 2 секунды (66 817 вызовов SQL)	1,1 ± 0,7 секунды (1 вызов SQL)
Выборка с соединением	9,0 ± 0,3 секунды (1 вызов SQL)	5,6 ± 0,4 секунды (1 вызов SQL)
Выборка с соединением, кэш запросов	5,8 ± 0,2 секунды (1 вызов SQL)	0,001 ± 0,0001 секунды (0 вызовов SQL)

При первом выполнении цикла с запросом `JOIN` наблюдается большой выигрыш по производительности: выполнение занимает всего 9 секунд. Это происходит из-за того, что выполняется только один запрос SQL вместо 66 817.

К сожалению, при следующем выполнении кода эта одна команда SQL попрежнему необходима, потому что результаты запроса не сохраняются в кэше L2. Последующие выполнения примера занимают 5,6 секунды — потому что выполняемая команда SQL содержит условие `JOIN` и загружает более 400 000 строк данных.

Если провайдер JPA реализует кэширование запросов, несомненно, это идеальное время для его применения. Если команды SQL не нужны во время второго выполнения кода, при последующих выполнениях потребуется всего 1 мс. Учтите, что кэширование запросов работает только в том случае, если параметры, используемые в запросе, в точности совпадают при каждом выполнении запроса.

Предотвращение запросов

Если сущности никогда не читаются с использованием запроса, то все сущности будут доступны в кэше L2 после исходного периода разогрева. Кэш L2 можно разогреть загрузкой всех сущностей, и незначительная модификация предыдущего примера дает следующий код:

```
EntityManager em = emf.createEntityManager();
ArrayList<String> allSymbols = ... все действительные акции ...;
ArrayList<Date> allDates = ... все действительные даты ...;
for (String symbol : allSymbols) {
    for (Date date = allDates) {
        StockPrice sp =
            em.find(StockPriceImpl.class, new StockPricePK(symbol, date);
          ... обработка sp ...
        if (processOptions) {
         Collection<? extends StockOptionPrice> options = sp.getOptions();
          ... обработка options ...
        }
    }
}
```

Результаты выполнения этого кода приведены в табл. 11.6.

Таблица 11.6. Продолжительность чтения данных 256 видов акций (с использованием кэша L2)

Тестовый сценарий	Первое выполнение	Последующие выполнения
Конфигурация по умолчанию	22,7 ± 2 секунды (66 817 вызовов SQL)	1,1 ± 0,7 секунды (1 вызов SQL)
Без запроса	35 ± 3 секунды (133 632 вызова SQL)	0,28 ± 0,3 секунды (0 вызовов SQL)

Первое выполнение этого цикла требует 133 632 команды SQL: 66 816 для вызова метода `find()` и еще 66 816 для вызова метода `getOptions()`. Последующие выполнения этого кода действительно оказываются очень быстрыми, потому что все сущности находятся в кэше L2 и никакие команды SQL выдавать не придется.

РАЗОГРЕВ ДЛЯ ТЕСТА

У тестов производительности Java — и особенно хронометражных тестов — обычно имеется период разогрева. Как упоминалось в главе 4, разогрев позволяет компилятору оптимально компилировать код.

Рассмотрим еще один пример ситуации, в которой период разогрева приносит пользу. Во время разогрева приложения JPA наиболее часто загружаемые сущности будут загружаться в кэш L2. Во время измерений вы получите совершенно иные показатели производительности, так как нужные сущности будут загружены заранее. В частности, это особенно справедливо, если, как и в предыдущем примере, запросы не используются для загрузки сущностей; обычно именно этим отличаются все таблицы, приведенные в этом разделе, между первым и вторым выполнением.

Напомню, что база данных в нашем примере включает пять цен опционов для каждой даты и вида акций, то есть 334 080 цен опционов для 256 видов акций за один год. Когда для каждой комбинации вида акций и даты через отношение приложение обращается к пяти опционам, все цены могут быть загружены одновременно. Именно по этой причине для загрузки всех данных опционов потребовалось всего 66 816 команд SQL. И хотя эти команды SQL возвращают множество строк данных, JPA все равно может кэшировать сущности — это не то же самое, что выполнение запроса. Если кэш L2 разогревается перебором сущностей, не перебирайте связанные сущности по отдельности — это следует делать простым посещением отношения.

При оптимизации кода следует учитывать эффекты кэширования (и особенно кэша L2). Даже если вы полагаете, что написанный вами код SQL эффективнее кода, сгенерированного JPA (причем в этом коде должны использоваться сложные именованные запросы), убедитесь в том, что использование этого кода будет оправданно с учетом кэширования. Даже если кажется, что использование простого именованного запроса позволит ускорить загрузку данных, подумайте, что произойдет в долгосрочной перспективе, если эти сущности будут загружены в кэш L2 вызовом метода `find()`.

Определение размера кэша JPA

Как это всегда бывает при повторном использовании объектов, кэширование JPA имеет потенциальные недостатки по производительности: если кэш потребляет слишком много памяти, это создаст лишнюю нагрузку на систему уборки мусора. Возможно, для этого потребуется настроить кэш и изменить его размер или же управлять режимом кэширования сущностей. К сожалению, все эти возможности не являются стандартными, поэтому вы должны выполнять эти настройки на основании используемого провайдера JPA.

Реализации JPA обычно предоставляют возможность настройки размера кэша — либо глобальной, либо на уровне сущностей. Второй вариант, очевидно, обладает большей гибкостью, хотя он также требует дополнительной работы для определения оптимального размера для каждой сущности. Альтернативное решение основано на использовании мягких и/или слабых ссылок реализацией JPA для кэша L2. Например, EclipseLink предоставляет пять типов кэшей (а также еще несколько типов, которые считаются устаревшими) на основании различных комбинаций мягких и слабых ссылок. Хотя этот подход потенциально проще нахождения оптимальных размеров для каждой сущности, он требует некоторого планирования: в частности, как говорилось в главе 7, слабые ссылки не выживают при операциях уборки мусора, из-за чего их применение для кэширования выглядит сомнительно.

При использовании кэша, основанного на мягких или слабых ссылках, производительность приложения также будет зависеть от того, что еще происходит в куче. Во всех примерах этого раздела используется куча большого размера, так что кэширование 400 896 объектов сущностей в приложении не создаст проблем с уборщиком мусора. Настройка кучи при больших кэшах JPA L2 играет важную роль для достижения хорошей производительности.

РЕЗЮМЕ

- Кэш JPA L2 автоматически кэширует сущности для приложения.
- Кэш L2 не кэширует сущности, полученные в результате запросов. Это означает, что в долгосрочной перспективе может быть лучше вообще обходиться без запросов.
- Если только кэширование запросов не поддерживается используемой реализацией JPA, выясняется, что запросы JOIN часто отрицательно влияют на производительность, поскольку они работают в обход кэша L2.

Spring Data

Хотя JDBC и JPA являются стандартными частями платформы Java, также существуют другие сторонние Java API и фреймворки, управляющие работой с базами данных. Все разработчики решений NoSQL определяют собственные API для обращения к базам данных, а различные фреймворки предоставляют доступ к базам данных через другие абстракции, кроме JPA.

Наибольшей популярностью пользуется Spring Data — набор модулей для работы с базами данных (как реляционными, так и NoSQL). Фреймворк содержит ряд модулей, включая следующие:

Spring Data JDBC — проектировался как простая альтернатива для JPA; предоставляет функциональность отображения сущностей, сходную с JPA, но без кэширования, отложенной загрузки или отслеживания грязных сущностей. Работает на базе стандартных драйверов JDBC, поэтому обладает широкой поддержкой. Это означает, что вы можете отслеживать аспекты производительности, описанные в этой главе, в коде Spring: убедитесь в том, что вы используете подготовленные команды для повторных вызовов, реализуйте необходимые интерфейсы в коде для поддержки моделей пакетных команд Spring и/или работайте напрямую с объектами подключений для изменения семантики автозакрепления.

Spring Data JPA — проектировался как обертка для стандартного JPA. Одно из больших преимуществ — сокращение объема шаблонного кода, который должны писать разработчики (что хорошо для производительности разработки, но не влияет напрямую на производительность, о которой мы здесь говорим). Так как Spring Data JPA является оберткой для стандартного JPA, к нему применяются все аспекты производительности JPA, упоминаемые в этой главе: настройка немедленной или отложенной загрузки, пакетирование обновлений и вставок, кэши L2 и т. д.

Spring Data for NoSQL — Spring включает различные коннекторы для технологий NoSQL (и сходных с NoSQL), включая MongoDB, Cassandra, Couchbase и Redis. Они отчасти упрощают работу с NoSQL, так как средства для обращения к хранилищу данных остаются неизменными, несмотря на некоторые различия в настройке и инициализации.

Spring Data R2DBC — модуль Spring Data R2DBC, упоминавшийся в главе 10, обеспечивает асинхронный доступ JDBC к базам данных Postgres, H2 и Microsoft SQL Server. Он следует типичной модели программирования Spring Data (вместо непосредственного использования JDBC), поэтому в некоторых отношениях напоминает Spring Data JDBC: для работы используются простые сущности в репозиториях, хотя и без кэширования, отложенной загрузки и других возможностей JPA.

Итоги

Правильная настройка доступа к базам данных через JDBC и JPA — один из самых эффективных способов управления производительностью приложений среднего уровня. Помните следующие рекомендации:

- Старайтесь группировать операции чтения и записи, насколько это возможно, за счет соответствующей настройки конфигурации JDBC или JPA.
- Оптимизируйте код SQL, выдаваемый приложением. Для приложений JDBC это вопрос базовых, стандартных команд SQL. Для приложений JPA обязательно рассмотрите эффект кэша L2.
- По возможности минимизируйте блокировки. Используйте оптимистичную блокировку, если конфликты за данные маловероятны, и пессимистичную блокировку, если доступ к данным становится предметом конкуренции.
- Старайтесь использовать пул подготовленных команд.
- Проследите за тем, чтобы пул подключений имел правильные размеры.
- Правильно выберите масштаб транзакции: она должна иметь максимально возможный размер, при котором она не будет отрицательно влиять на масштабируемость приложения из-за блокировок, удерживаемых в ходе транзакции.

ГЛАВА 12

Рекомендации по использованию Java SE API

В этой главе рассматриваются области Java SE API, имеющие особенности реализации, влияющие на их производительность. В JDK есть множество таких подробностей реализации; в этих областях я постоянно обнаруживаю проблемы с производительностью (даже в моем собственном коде). В этой главе подробно рассматривается работа со строками (и особенно дублирование строк); способы буферизации ввода/вывода; загрузка классов и способы ускорения запуска приложений, использующих большое количество классов; правильное использование коллекций; и такие возможности JDK 8, как лямбда-выражения и потоки данных.

Строки

Как и следовало ожидать, строка является самым популярным объектом Java. В этом разделе мы рассмотрим разнообразные возможности работы с памятью, потребляемой объектами строк; эти приемы часто позволяют сократить объем памяти, необходимой вашей программе. Также будет рассмотрена новая возможность работы со строками в JDK 11, использующая конкатенацию.

Компактные строки

В Java 8 все строки представляются массивами 16-разрядных символов независимо от кодировки строки. Такое представление неэффективно: многие западные локальные контексты могут кодировать строки массивами 8-разрядных символов, и даже в локальном контексте, требующем 16 бит для всех символов, некоторые строки (например, константы в программе) часто могут кодироваться 8-разрядными байтами.

В Java 11 строки представляются массивами 8-разрядных символов, если только им не требуются именно 16-разрядные символы; такие строки называются *компактными*. В Java 6 существовала похожая (экспериментальная) возможность — *сжатые строки*; компактные строки концептуально близки к ним, но реализованы совершенно иначе.

Таким образом, размер средней строки в Java 11 составляет приблизительно половину от размера той же строки в Java 8. Обычно это обеспечивает значительную экономию; в среднем около 50% типичной кучи Java может быть занято объектами строк. Конечно, программы бывают разными, но в среднем затраты памяти кучи такой программы в Java 11 составляют всего 75% от затрат той же программы, работающей в Java 8.

Довольно легко представить пример, в котором это изменение будет иметь колоссальные преимущества. В Java 8 можно запустить программу, которая будет проводить непропорционально большое время за уборкой мусора. При запуске той же программы в Java 11 с тем же размером кучи она практически не потребует уборки мусора, что по сообщениям приведет к возрастанию производительности от 3 до 10 раз. К таким утверждениям следует относиться с долей скепсиса; при столь ограниченной куче запускать программы Java вообще не рекомендуется. Впрочем, при прочих равных условиях вы увидите сокращение времени, проведенного за уборкой мусора.

В хорошо настроенном приложении настоящий выигрыш проявляется в затратах памяти: вы можете немедленно сократить максимальный размер кучи средней программы на 25% без потери для производительности. И наоборот, если оставить размер кучи неизменным, вы сможете повысить нагрузку на приложение, не столкнувшись с какими-либо узкими местами в области уборки мусора (хотя остальные части приложения должны быть способны справиться с возросшей нагрузкой). Этой функциональностью управляет флаг `-XX:+CompactStrings`, по умолчанию равный `true`.

Но в отличие от сжатых строк Java 6, компактные строки надежны и эффективны; почти всегда для этого флага рекомендуется оставлять значение по умолчанию. Одним из возможных исключений может стать программа, в которой все строки требуют 16-разрядной кодировки; в такой программе операции со сжатыми строками могут занимать чуть больше времени, чем с несжатыми.

Дублирование и интернирование строк

В программах довольно часто создаются множественные строковые объекты, содержащие одну последовательность символов. Такие объекты лишь напрасно занимают память в куче; так как строки неизменяемы, часто будет более эффективно повторно использовать существующие строки. Общий случай такого

рода был рассмотрен в главе 7 для произвольных объектов в каноническом представлении; в этом разделе идея расширяется для строк.

Чтобы определить, содержит ли ваша программа большое количество строк-дубликатов, необходимо воспользоваться анализом кучи. Один из способов решения этой задачи в Eclipse Memory Analyzer:

1. Загрузите дамп кучи.
2. В Query Browser выберите команду Java Basics ▸ Group By Value.
3. В поле аргумента `objects` введите строку `java.lang.String`.
4. Щелкните на кнопке Finish.

Результат показан на рис. 12.1. Каждая из строк, `Name`, `Memnor` и `Parent Name`, встречается более чем в 300 000 экземпляров. Ряд других строк тоже существует во многих экземплярах; всего куча содержит более 2,3 миллиона дубликатов строк.

Overview | group_by_value java.lang.String

String Value	Objects	Shallow Heap	Avg. Retained Size	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
Name	361,959	8,687,016	48	>= 17,374,032
Memnor	361,809	8,683,416	56	>= 20,261,304
Parent Name	361,808	8,683,392	64	>= 23,155,712
Memnor|Name|Parent Name	241,202	5,788,848	88	>= 21,225,776
English	120,615	2,894,760	56	>= 6,754,440
English|Memnor|Name|Parent Name	120,601	2,894,424	104	>= 12,542,504
FY16	47,060	1,129,440	48	>= 2,258,880
[Year].[FY16]	47,060	1,129,440	72	>= 3,388,320
FY17	46,163	1,107,912	48	>= 2,215,824
[Year].[FY17]	46,163	1,107,912	72	>= 3,323,736
Rolling	27,183	652,392	56	>= 1,522,248
[Period].[Rolling]	27,183	652,392	80	>= 2,174,640
YearTotal	5,200	124,800	64	>= 332,800
Q1	5,187	124,488	48	>= 248,976
Q2	5,187	124,488	48	>= 248,976
Q3	5,187	124,488	48	>= 248,976
Q4	5,187	124,488	48	>= 248,976
Apr	5,149	123,576	48	>= 247,152
Dec	5,149	123,576	48	>= 247,152
Feb	5,149	123,576	48	>= 247,152
Jul	5,149	123,576	48	>= 247,152
Jun	5,149	123,576	48	>= 247,152
Mar	5,149	123,576	48	>= 247,152
May	5,149	123,576	48	>= 247,152
Nov	5,149	123,576	48	>= 247,152
Oct	5,149	123,576	48	>= 247,152
[Period].[YearTotal]	5,148	123,552	80	>= 411,840
[Period].[YearTotal].[Q1]	5,148	123,552	96	>= 494,208
[Period].[YearTotal].[Q1].[Feb]	5,148	123,552	104	>= 535,392
Total: 29 of 20,480 entries; 20,451 more	2,335,390	56,049,360		

Рис. 12.1. Затраты памяти на хранение дубликатов строк

Дубликаты строк можно исключить тремя способами:

- Провести автоматическое удаление дубликатов средствами уборщика мусора G1.
- Воспользоваться методом `intern()` класса `String` для создания канонической версии строки.
- Воспользоваться нестандартным методом для создания канонической версии строки.

Исключение дубликатов строк

В простейшем варианты вы поручаете JVM найти дубликаты строк и избавиться от них: перевести все ссылки на одну копию и освободить остальные копии. Это можно сделать только при использовании уборщика мусора G1 и только при установленном флаге `-XX:+UseStringDeduplication` (который по умолчанию равен `false`). Данная возможность есть в Java 8 только после версии 20 и во всех версиях Java 11.

Она не включается по умолчанию по трем причинам. Во-первых, она требует дополнительной обработки в фазах уборки в молодом и смешанном поколении G1, что несколько повышает их длительность. Во-вторых, она требует дополнительного потока, который выполняется параллельно с приложением, и возможно, отнимает вычислительные ресурсы у потоков приложения. В-третьих, если строк с исключаемыми дубликатами немного, затраты памяти в приложении увеличатся (вместо того, чтобы понизиться); дополнительные затраты памяти вызваны служебными операциями, связанными с отслеживанием всех строк в поисках дубликатов.

Эта одна из тех настроек, которые включаются в рабочей версии только после тщательного тестирования: они могут помочь вашему приложению, но в некоторых случаях только ухудшают ситуацию. Впрочем, шансы в вашу пользу: по оценкам технических специалистов Java, ожидаемый выигрыш от включенного исключения дубликатов строк составляет 10%.

Если вы захотите узнать, как исключение дубликатов строк ведет себя в вашем приложении, запустите его с флагом `-XX:+PrintStringDeduplicationStatistics` в Java 8 или с флагом `-Xlog:gc+stringdedup*=debug` в Java 11. Полученный журнал выглядит примерно так:

```
[0.896s][debug][gc,stringdedup]   Last Exec: 110.434ms, Idle: 729.700ms,
                                  Blocked: 0/0.000ms
[0.896s][debug][gc,stringdedup]     Inspected:           62420
[0.896s][debug][gc,stringdedup]       Skipped:               0(  0.0%)
[0.896s][debug][gc,stringdedup]       Hashed:            62420(100.0%)
```

```
[0.896s][debug][gc,stringdedup]       Known:                 0(  0.0%)
[0.896s][debug][gc,stringdedup]       New:               62420(100.0%)
                                                         3291.7K
[0.896s][debug][gc,stringdedup]     Deduplicated:        15604( 25.0%)
                                                         731.4K( 22.2%)
[0.896s][debug][gc,stringdedup]       Young:                 0(  0.0%)
                                                         0.0B(  0.0%)
[0.896s][debug][gc,stringdedup]       Old:               15604(100.0%)
                                                         731.4K(100.0%)
```

Выполнение потока исключения дубликатов строк заняло 110 мс. За это время было обнаружено 15 604 дубликата строк (из 62 420 строк, которые были идентифицированы как кандидаты для исключения дубликатов). Общая экономия памяти составила 731,4 Кбайт — около 10% того, на что мы надеялись для этой оптимизации.

Код, сгенерировавший этот журнал, был настроен так, что дубликатами были 25% строк — по утверждениям технических специалистов JVM, такая картина типична для приложений Java. (По моему опыту — как я упоминал ранее — доля дубликатов в куче ближе к 50%; у вас может быть по-другому.)

Почему же мы не сэкономили 25% памяти строк? Дело в том, что оптимизация обеспечивает совместное использование только символьного или байтового массива с данными, но не остальной части объекта строки. Объект содержит от 24 до 32 байт служебной информации в других полях (в зависимости от платформенных реализаций). Таким образом, две идентичные строки из 16 символов будут занимать 44 (или 52) байта каждая, до того как будут дедуплицированы до 80 байт; после дедупликации они займут 64 байта. Если бы строки были интернированы (эта тема рассматривается в следующем разделе), то они бы занимали только 40 байт.

Как упоминалось ранее, обработка строк выполнялась конкурентно с потоками приложения. На самом деле это последняя стадия процесса. В ходе уборки мусора в молодом поколении проверяются все строки молодого поколения. Те из них, которые повышаются до старого поколения, становятся кандидатами, которые проверяются фоновым потоком (после завершения уборки в молодом поколении). Кроме того, вспомните, что говорилось в главе 6 о хранении объектов в областях выживших в молодом поколении: объекты могут какое-то время переходить между областями выживших, прежде чем они будут перемещены в старое поколение. Строки с возрастом хранения 3 (по умолчанию) — это означает, что они копируются в область выживших три раза, — также становятся кандидатами для исключения дубликатов и будут обработаны фоновым потоком.

Это приводит к тому, что строки с коротким сроком жизни не подвергаются исключению дубликатов, что, скорее всего, хорошо: не стоит тратить ресурсы процессора и память на исключение дубликатов данных, которые все равно

скоро будут потеряны. Как и при настройке цикла хранения вообще, изменение точки, в которой это происходит, требует серьезного тестирования и выполняется только в нетипичных обстоятельствах. Но на всякий случай замечу, что для управления точкой, в которой хранимая строка становится пригодной для уборки, используется флаг `-XX:StringDeduplicationAgeThreshold=`*N*, значение которого по умолчанию равно 3.

Интернирование строк

Типичное решение проблемы дубликатов строк на программном уровне основано на использовании метода `intern()` класса `String`.

Как и большинство оптимизаций, интернирование строк следует применять осмотрительно. Тем не менее оно может быть эффективным, если множество дубликатов строк занимает значительную часть кучи. При этом оно часто требует специальной настройки (а в следующем разделе будет рассмотрено нестандартное решение, которое может принести пользу в некоторых обстоятельствах).

Интернированные строки хранятся в специальной хеш-таблице, находящейся в низкоуровневой памяти (хотя сами строки находятся в куче). Эта хеш-таблица отличается от хеш-таблиц и хеш-карт, с которыми вы знакомы по Java, потому что эта низкоуровневая хеш-таблица имеет фиксированный размер: 60 013 в Java 8 и 65 536 в Java 11. (Если вы работаете на 32-разрядной Windows JVM, ее размер будет равен 1009.) Таким образом, вы сможете сохранить только около 32 000 интернированных строк, прежде чем в хеш-таблице начнут возникать коллизии.

Размер таблицы можно задать при запуске JVM при помощи флага `-XX:StringTableSize=`*N* (который по умолчанию равен 1009, 60 013 или 65 536, как упоминалось ранее). Если приложение интернирует множество строк, это число следует увеличить. Таблица интернирования работает наиболее эффективно, если это значение является простым числом.

В производительности метода `intern()` определяющим фактором является качество настройки размера таблицы. Например, в табл. 12.1 приведено общее время создания и интернирования миллиона случайных строк с этой настройкой и без нее.

Таблица 12.1. Время интернирования 1 миллиона строк

Настройка	100% попаданий в кэш	0% попаданий в кэш
Размер таблицы строк 60 013	4,992 ± 2,9 секунды	2,759 ± 0,13 секунды
Размер таблицы строк 1 000 000	2,446 ± 0,6 секунды	2,737 ± 0,36 секунды

Обратите внимание: неправильно выбранный размер таблицы интернирования строк приводит к значительным потерям при 100% попаданий в кэш. После того как размер таблицы был приведен в соответствие с ожидаемыми данными, производительность кардинально улучшается.

ХЕШ-ТАБЛИЦЫ ФИКСИРОВАННОГО РАЗМЕРА

Если вы не знакомы с базовой структурой хеш-таблиц и хеш-карт, возможно, вас интересует, что же собой представляет хеш-таблица фиксированного размера (особенно потому, что размер Java-реализаций этих классов не фиксирован).

На концептуальном уровне хеш-таблица содержит массив, в котором может храниться определенное количество элементов. Каждый элемент такого массива называется гнездом (bucket). Когда в хеш-таблице сохраняется какое-либо значение, индекс для его хранения вычисляется по формуле `хешКод%количествоГнезд`. В этой схеме два объекта с разными хешами вполне могут отображаться на одно гнездо, поэтому каждое гнездо в действительности представляет собой связанный список всех хранимых элементов, соответствующих этому гнезду. Если два объекта отображаются на одно гнездо, такая ситуация называется *коллизией*.

По мере того как в таблицу вставляется все больше объектов, коллизии происходят все чаще; в каждом связанном списке сохраняется все больше элементов. В этом случае поиск элемента сводится к поиску по связанному списку. Эта процедура может быть очень медленной, особенно при увеличении длины списка.

Проблему можно обойти за счет определения размера хеш-таблицы, чтобы в ней было больше гнезд (и, как следствие, меньше коллизий). Многие реализации делают это динамически; по этому принципу работают классы Java `Hashtable` и `HashMap`.

Но другие реализации — включая внутреннюю реализацию JVM, которая здесь рассматривается, — изменять свои размеры не могут; размер массива фиксируется при создании карты.

Данные для 0% попаданий в кэш могут показаться немного неожиданными, потому что производительность с настройкой и без почти не отличается. В нашем тестовом примере строки отбрасываются сразу же после интернирования. Внутренняя таблица строк функционирует так, словно ключи являются слабыми ссылками, так что после того, как строка будет освобождена, она может

быть удалена из таблицы. Таким образом, в данном тестовом примере таблица строк реально никогда не заполняется; в конечном итоге она содержит лишь несколько элементов (потому что в любой момент времени только несколько строк удерживаются по сильным ссылкам).

Чтобы понаблюдать за работой таблицы строк, запустите свое приложение с аргументом `-XX:+PrintStringTableStatistics` (по умолчанию `false`). При завершении JVM выводится таблица следующего вида:

```
StringTable statistics:
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :   2002784 =  48066816 bytes, avg  24.000
Number of literals      :   2002784 = 606291264 bytes, avg 302.724
Total footprint         :           = 654838184 bytes
Average bucket size     :    33.373
Variance of bucket size :    33.459
Std. dev. of bucket size:     5.784
Maximum bucket size     :        60
```

Этот результат относится к примеру стопроцентных попаданий в кэш. После выполнения в таблице содержатся 2 002 784 интернированные строки (2 миллиона из теста с одним разогревом и одним циклом измерений; остальные принадлежат jmh и классам JDK). Из всех характеристик нас больше всего интересуют средний и максимальный размер гнезда (`Average bucket size` и `Maximum bucket size`): чтобы найти элемент в хеш-таблице, необходимо перебрать в среднем 33 и максимум 60 элементов связанного списка. В идеале средняя длина должна быть меньше 1, а максимальная близка к 1. Мы это наблюдаем в примере с 0% попаданий в кэш:

```
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :      2753 =     66072 bytes, avg  24.000
Number of literals      :      2753 =    197408 bytes, avg  71.707
Total footprint         :           =    743584 bytes
Average bucket size     :     0.046
Variance of bucket size :     0.046
Std. dev. of bucket size:     0.214
Maximum bucket size     :         3
```

Так как строки быстро освобождаются из таблицы, в итоге таблица содержит всего 2753 элемента, что неплохо для размера по умолчанию 60 013.

Количество интернированных строк, созданных приложением (и их общий размер), также можно определить командой `jmap`:

```
% jmap -heap process_id
... другие данные ...
36361 interned Strings occupying 3247040 bytes.
```

Потери от завышения размера таблицы строк минимальны; каждое гнездо занимает только 8 байт, поэтому превышение оптимального размера на несколько тысяч элементов становится одноразовой потерей нескольких килобайт низкоуровневой памяти (не кучи).

ИНТЕРНИРОВАНИЕ СТРОК И ПРОВЕРКА РАВЕНСТВА

Раз уж речь зашла об интернировании строк, почему бы не использовать метод `intern()` для ускорения работы программы — ведь интернированные строки можно сравнивать оператором `==`? Это довольно популярная идея, однако в большинстве случаев она оказывается мифом. Метод `String.equals()` работает довольно быстро. Для начала он знает, что строки разной длины никогда не бывают равными, а если строки имеют одинаковую длину, он должен просканировать строку и сравнить все символы (по крайней мере пока не обнаружит, что строки не совпадают). Сравнение строк оператором `==` безусловно выполняется быстрее, но не стоит забывать и о затратах на интернирование строк. Для этого (среди прочего) необходимо вычислить хеш-код строки, для чего сканируется вся строка и выполняется операция с каждым из ее символов (по аналогии с тем, как это должен делать метод `equals()`).

Ожидать выигрыша от использования метода `intern()` при сравнении строк можно только в одном случае: если приложение выполняет большое количество повторяющихся сравнений с набором строк равной длины. Если обе строки ранее были интернированы, сравнение `==` осуществляется быстрее; затраты на вызов метода `intern()` будут одноразовыми. Однако в общем случае затраты в целом равны.

Специальное интернирование строк

Настраивать таблицу строк неудобно; нельзя ли добиться лучших результатов, если просто воспользоваться специальной схемой интернирования, которая хранит важные строки в хеш-карте? Схема такого решения также была представлена в главе 2.

Информация в табл. 12.2 поможет найти ответ на этот вопрос. Кроме использования обычной коллекции `ConcurrentHashMap` для хранения интернированных строк, в таблице также продемонстрировано использование коллекции `CustomConcurrentHashMap` из дополнительных классов, разработанных в составе JSR166. Эта специализированная карта позволяет использовать слабые ссылки для ключей, поэтому ее поведение точнее воспроизводит таблицу интернированных строк.

Таблица 12.2. Время интернирования 1 миллиона строк со специальным кодом

Реализация	100% попаданий в кэш	0% попаданий в кэш
`ConcurrentHashMap`	7,665 ± 6,9 секунды	5,490 ± 2,462 секунды
`CustomConcurrentHashMap`	2,743 ± 0,4 секунды	3,684 ± 0,5 секунды

В тесте со 100% попаданий `ConcurrentHashMap` страдает от тех же проблем, которые были представлены ранее для внутренней таблицы строк: при каждой итерации создается значительная нагрузка на процесс уборки мусора. Данные получены для размера кучи 30 Гбайт; с меньшим размером кучи результаты будут еще хуже.

Как и с любыми микробенчмарками, здесь необходимо тщательно продумать сценарий использования. Коллекцией `ConcurrentHashMap` можно управлять явно, а не использовать схему из нашего примера, которая просто продолжает вставлять только что созданные строки. В зависимости от приложения это может быть не так просто; впрочем, если это просто, тест `ConcurrentHashMap` продемонстрирует те же преимущества, что и обычное интернирование или тест `CustomConcurrentHashMap`. В реальном приложении самым важным фактором становится давление на систему уборки мусора; мы будем использовать этот метод только для удаления дубликатов строк, пытаясь сэкономить на циклах уборки мусора.

Ни один из примеров не показывает результатов лучших, чем в тесте с правильно настроенной таблицей строк. Преимущество нестандартной карты заключается в том, что ее размер не нужно задавать заранее: она может сама изменять свои размеры по мере необходимости. А следовательно, она гораздо лучше адаптируется к различным применениям, чем использование метода `intern()` и настройка размера таблицы строк в зависимости от приложения.

Конкатенация строк

Конкатенация строк — еще одна область потенциальных потерь для производительности. Возьмем простую конкатенацию строк:

```
String answer = integerPart + "." + mantissa;
```

Специальные оптимизации в Java могут обработать эту конструкцию (хотя подробности зависят от версии).

В Java 8 компилятор `javac` преобразует такую команду в следующий код:

```
String answer = new StringBuilder(integerPart).append(".")
                        .append(mantissa).toString();
```

В JVM предусмотрен специальный код для конструкций такого вида (им управляет флаг `-XX:+OptimizeStringConcat`, по умолчанию равный `true`).

В Java 11 компилятор `javac` генерирует совершенно другой байт-код; этот код вызывает специальный метод JVM, оптимизирующий конкатенацию строк.

Это один из немногих случаев, в которых байт-код зависит от версии. Как правило, при переходе на новую версию перекомпилировать старый код не обязательно: байт-код останется неизменным. (Конечно, для использования новых языковых средств новый код должен быть откомпилирован новым кодом.) Однако эта конкретная оптимизация зависит от байт-кода. Если откомпилировать код, выполняющий конкатенацию строк в Java 8, и запустить его в Java 11, Java 11 JDK применит ту же оптимизацию, которая применялась в Java 8. Код будет оптимизирован и будет выполняться быстрее.

Но если перекомпилировать код для Java 11, байт-код будет использовать новые оптимизации и теоретически может работать еще быстрее.

Рассмотрим следующие три способа конкатенации двух строк:

```
@Benchmark
public void testSingleStringBuilder(Blackhole bh) {
    String s = new StringBuilder(prefix).append(strings[0]).toString();
    bh.consume(s);
}

@Benchmark
public void testSingleJDK11Style(Blackhole bh) {
    String s = prefix + strings[0];
    bh.consume(s);
}

@Benchmark
public void testSingleJDK8Style(Blackhole bh) {
    String s = new StringBuilder().append(prefix).append(strings[0]).toString();
    bh.consume(s);
}
```

Первый способ соответствует тому, как бы мы запрограммировали эту операцию вручную. Второй способ (при компиляции для Java 11) будет использовать новейшие оптимизации, а последний (независимо от используемого компилятора) будет одинаково оптимизироваться в Java 8 и Java 11.

Результаты этих операций представлены в табл. 12.3.

В данном случае старые (Java 8) и новые (Java 11) оптимизации конкатенации почти не отличаются; хотя `jmh` и сообщает, что различия статистически значимы, особой роли они не играют. Ключевое обстоятельство заключается в том, что обе оптимизации лучше ручного программирования этого простого случая.

И это немного странно, потому что случай ручного кодирования кажется самым простым: он содержит на один вызов метода `append()` меньше, чем случай для JDK 8, поэтому номинально он выполняет меньше работы. Однако оптимизация конкатенации строк в JVM не использует эту конкретную схему, так что в результате она работает медленнее.

Таблица 12.3. Производительность конкатенации

Режим	Время выполнения операции
Оптимизация JDK 11	47,7 ± 0,3 нс
Оптимизация JDK 8	42,9 ± 0,3 нс
Построитель строк	87,8 ± 0,7 нс

Впрочем, оптимизации Java 8 не переносятся на все конкатенации. Тесты можно слегка изменить:

```
@Benchmark
public void testDoubleJDK11Style(Blackhole bh) {
    double d = 1.0;
    String s = prefix + strings[0] + d;
    bh.consume(s);
}
@Benchmark
public void testDoubleJDK8Style(Blackhole bh) {
    double d = 1.0;
    String s = new StringBuilder().append(prefix).
                   append(strings[0]).append(d).toString();
    bh.consume(s);
}
```

Производительность изменяется, как показано в табл. 12.4.

Таблица 12.4. Производительность конкатенации со значением double

Режим	Время выполнения операции
Оптимизация JDK 11	49,4 ± 0,6 нс
Оптимизация JDK 8	77,0 ± 1,9 нс

Время для JDK 11 мало отличается от предыдущего примера, хотя мы и присоединяем новое значение и выполняем чуть больше работы. Однако время для JDK 8 намного хуже — приблизительно на 50%. Дело не в лишней конкатена-

ции, а в ее типе. Оптимизация JDK 8 хорошо работает со строками и целыми числами, но плохо справляется с `double` (и другими разновидностями данных). В таких случаях код JDK 8 пропускает специальную оптимизацию и повторяет поведение теста с ручной реализацией.

Ни одна из этих операций не действует при выполнении множественных операций конкатенации, особенно в цикле. Рассмотрим следующие тесты:

```
@Benchmark
public void testJDK11Style(Blackhole bh) {
    String s = "";
    for (int i = 0; i < nStrings; i++) {
        s = s + strings[i];
    }
    bh.consume(s);
}

@Benchmark
public void testJDK8Style(Blackhole bh) {
    String s = "";
    for (int i = 0; i < nStrings; i++) {
        s = new StringBuilder().append(s).append(strings[i]).toString();
    }
    bh.consume(s);
}

@Benchmark
public void testStringBuilder(Blackhole bh) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < nStrings; i++) {
        sb.append(strings[i]);
    }
    bh.consume(sb.toString());
}
```

На этот раз лучшие результаты показывает ручная реализация, и это логично. В частности, реализация Java 8 должна создавать новый объект `StringBuilder` при каждой итерации цикла, и даже в Java 11 затраты на создание строки в каждом цикле (вместо использования построителя строк) не проходят даром. Результаты приведены в табл. 12.5.

Таблица 12.5. Проект множественных конкатенаций

Режим	10 строк	1000 строк
Код JDK 11	613 ± 8 нс	2463 ± 55 мкс
Код JDK 8	584 ± 8 нс	2602 ± 209 мкс
Построитель строк	412 ± 2 нс	38 ± 211 мкс

Мораль: не бойтесь применять конкатенацию, когда она может выполняться в одной (логической) строке, но никогда не используйте конкатенацию строк в цикле, если только строка-результат не используется в следующей итерации цикла. В остальных случаях всегда явно используйте объект `StringBuilder` для достижения лучшей производительности. В главе 1 я указывал, что «преждевременная» оптимизация не всегда плоха, если это выражение используется в контексте, означающем попросту «написание хорошего кода». Вы увидели наглядный пример такого рода.

СОВЕТ

- Конкатенация строк в одну обеспечивает хорошую производительность.
- Для множественных операций конкатенации используйте `StringBuilder`.
- Конкатенация строк в одну, в которой задействованы некоторые типы, будет выполняться значительно быстрее при перекомпиляции в JDK 11.

Буферизованный ввод/вывод

Когда я вступил в группу Java Performance Group в 2000 году, мой начальник только что выпустил самую первую книгу о производительности Java, и одной из самых модных тем в те времена был буферизованный ввод/вывод. Через четырнадцать лет я почти решил, что эта тема устарела и ее нужно исключить из первого издания книги. Затем в ту неделю, когда я начал работать над черновиком первого издания книги, я обнаружил ошибки в двух не связанных проектах, в которых небуферизованный ввод/вывод заметно вредил производительности. Через несколько месяцев, работая над примерами для первого издания, я пытался понять, почему моя «оптимизация» работает так медленно. А потом понял: тупица, ты забыл правильно буферизовать ввод/вывод.

Потом наступило время подготовки второго издания. За две недели до того, как я вернулся к переработке этого раздела, ко мне обратились трое коллег. Они допустили ту же ошибку с буферизацией ввода/вывода, которую совершил я в примере для первого издания.

Итак, поговорим о производительности буферизованного ввода/вывода. Методы `InputStream.read()` и `OutputStream.write()` работают с отдельным символом. В зависимости от ресурса, к которому они обращаются, эти методы могут работать очень медленно. Объект `FileInputStream`, использующий метод `read()`, будет мучительно медленным: каждый вызов метода будет требовать

перехода в режим ядра для получения 1 байта данных. В большинстве операционных систем ядро буферизует ввод/вывод, так что этот сценарий (к счастью) не инициирует операцию чтения с диска для каждого вызова метода `read()`. Но этот буфер хранится в памяти ядра, а не приложения, и чтение данных по одному байту означает затратный вызов системной функции для каждого вызова метода.

Сказанное относится и к записи данных: использование метода `write()` для передачи одного байта `FileOutputStream` требует системного вызова для сохранения байта в буфере ядра. Со временем (при закрытии файла или очистке буфера) ядро запишет этот буфер на диск.

Для файлового ввода/вывода с использованием двоичных данных всегда используйте `BufferedInputStream` или `BufferedOutputStream` для представления файлового потока данных. Для файлового ввода/вывода с использованием символьных (строковых) данных всегда упаковывайте поток данных в `BufferedReader` или `BufferedWriter`.

И хотя этот аспект производительности проще всего понять на примере файлового ввода/вывода, это общая проблема, применимая почти к каждой разновидности ввода/вывода. Потоки данных, возвращаемые от сокета (вызовом `getInputStream()` или `getOutputStream()`), работают аналогично, и ввод/вывод с сокетом по одному байту будет достаточно медленным. Здесь также необходимо проследить за тем, чтобы потоки данных были упакованы в обертку буферизованных фильтрующих потоков.

С использованием классов `ByteArrayInputStream` и `ByteArrayOutputStream` связаны другие, более тонкие нюансы. Прежде всего, эти классы фактически представляют обычные большие буферы в памяти. Во многих случаях упаковка их в буферизованный фильтрующий поток данных означает, что данные копируются дважды: в буфер в фильтрующем потоке и в буфер в `ByteArrayInputStream` (или наоборот для выходных потоков). Если другие потоки не участвуют в этой схеме, буферизованного ввода/вывода в таком случае следует избегать.

При участии других фильтрующих потоков вопрос о том, нужно ли применять буферизацию, усложняется. Позднее в этой главе будет приведен пример сериализации объекта с участием нескольких фильтрующих потоков, использующих классы `ByteArrayOutputStream`, `ObjectOutputStream` и `GZIPOutputStream`.

Без выходного потока со сжатием фильтры для примера выглядят так:

```
protected void makePrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(prices);
    oos.close();
}
```

В этом случае упаковка потока данных `baos` в `BufferedOutputStream` потребует затрат на однократное лишнее копирование данных.

Однако после добавления сжатия этот код лучше всего записать в следующем виде:

```
protected void makeZippedPrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPOutputStream zip = new GZIPOutputStream(baos);
    BufferedOutputStream bos = new BufferedOutputStream(zip);
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(prices);
    oos.close();
    zip.close();
}
```

Теперь необходимо буферизовать выходной поток, потому что `GZIPOutputStream` работает с блоком данных более эффективно, чем с отдельными байтами данных. В любом случае `ObjectOutputStream` передает одиночные байты данных следующему потоку данных. Если следующий поток является итоговым получателем (`ByteArrayOutputStream`), буферизация не понадобится. Если между ними находится другой фильтрующий поток (такой, как `GZIPOutputStream` в этом примере), буферизация часто необходима.

Нет никаких общих правил относительно того, когда следует использовать буферизованный поток данных, занимающий промежуточное положение между двумя другими потоками. В конечном счете все зависит от типа потоков данных, но все типичные сценарии будут работать лучше, если им передается блок байтов (от буферизованного потока) вместо последовательности одиночных байтов (от `ObjectOutputStream`).

Для входных потоков ситуация выглядит аналогично. В этом конкретном случае `GZIPInputStream` будет более эффективно работать с блоком байтов; в общем случае потоки данных, размещенные между `ObjectInputStream` и исходным источником байтов, тоже будут более эффективными с блоком байтов.

Заметим, что этот случай также относится к потоковым кодировщикам и декодировщикам. Когда вы выполняете преобразование между байтами и символами, работа с максимально возможным блоком данных обеспечивает наилучшее быстродействие. Если передавать кодировщикам и декодировщикам одиночные байты или символы, быстродействие заметно ухудшится.

Стоит отметить, что отсутствие буферизации потоков gzip — именно та ошибка, которую я допустил при написании этого примера сжатия. Ошибка оказалась дорогостоящей, как видно из табл. 12.6.

Отсутствие буферизации ввода/вывода приводит к четырехкратному падению производительности.

Таблица 12.6. Время сериализации и десериализации объекта Stock со сжатием

Режим	Время
Сжатие/восстановление без буферизации	21,3 ± 8 мс
Сжатие/восстановление с буферизацией	5,7 ± 0,08 мс

РЕЗЮМЕ

- Проблемы буферизации ввода/вывода достаточно часто встречаются из-за стандартной реализации простых классов входного и выходного потока.
- Ввод/вывод должен быть буферизован для файлов и сокетов, а также для таких внутренних операций, как сжатие и кодирование строк.

Загрузка классов

Производительность загрузки классов наводит ужас на любого разработчика, пытающегося оптимизировать либо запуск программы, либо развертывание нового кода в динамической системе.

Это объясняется многими причинами. Во-первых, данные классов (то есть байт-код Java) обычно не находятся в быстрой доступности. Они должны загружаться с диска или из сети, они должны находиться в одном из нескольких файлов JAR в каталогах классов, и они должны находиться в одном из загрузчиков классов. Есть несколько способов упростить эту задачу: некоторые фреймворки кэшируют классы, прочитанные из сети, в скрытом каталоге, чтобы при следующем запуске того же приложения классы были загружены быстрее. Упаковка приложения в меньшее число файлов JAR также ускорит его загрузку. В этом разделе будет использоваться новая возможность Java 11 для ускорения загрузки классов.

Совместное использование данных классов

Совместное использование данных классов, или *CDS* (Class Data Sharing), — механизм, позволяющий совместно использовать метаданные классов на разных JVM. Это может быть полезно для экономии памяти при выполнении нескольких JVM: обычно каждая JVM содержит собственные метаданные класса, а отдельные копии расходуют физическую память. Если бы метаданные использовались совместно, в памяти было бы достаточно хранить только одну копию.

Однако механизм CDS также очень полезен для одиночных JVM, потому что он может ускорить процесс запуска.

Совместное использование данных классов доступно в Java 8 (и предыдущих версиях), но с тем ограничением, что оно применимо только к классам из rt.jar и только при использовании последовательного уборщика мусора с клиентской JVM. Иначе говоря, оно в какой-то степени полезно на 32-разрядных однопроцессорных настольных машинах с системой Windows.

В Java 11 CDS механизм CDS доступен на всех платформах, хотя он не будет работать в исходном состоянии, потому что в системе нет общего архива метаданных класса по умолчанию. В Java 12 появился общий архив по умолчанию для основных классов JDK, так что все приложения по умолчанию пользуются некоторыми улучшениями по времени запуска (и памяти). В любом случае можно улучшить ситуацию, сгенерировав более полный общий архив для своего приложения, потому что в Java 11 CDS может работать с любым набором классов независимо от того, какой загрузчик их загружает и из какого файла JAR или модуля они загружаются. При этом действует одно ограничение: CDS работает только для классов, загруженных из модулей или файлов JAR. Нельзя обеспечить совместное использование (или быструю загрузку) классов из файловой системы или сетевых URL-адресов.

В определенном смысле это означает, что существуют две разновидности CDS: обычный CDS (для совместного использования стандартных классов JDK) и совместное использование классов приложения (для совместного использования произвольных наборов классов). Совместное использование данных классов приложений впервые появилось в Java 10 и работало не так, как обычный механизм CDS: программы должны были использовать различные аргументы командной строки. Теперь различия ушли в прошлое, и механизм CDS в Java 11 и выше работает одинаково независимо от того, какие именно классы используются.

Первое, что необходимо для использования CDS, — общий архив классов. Как упоминалось ранее, Java 12 поставляется с общим архивом классов по умолчанию, находящимся в файле $JAVA_HOME/lib/server/classes.jsa (или %JAVA_HOME%\bin\server\classes.jsa в Windows). Этот архив содержит данные для 12 000 классов JDK, так что он обеспечивает хорошее покрытие базовых классов. Чтобы сгенерировать собственный архив, для начала создайте список всех классов, для которых вы хотите включить совместное использование (а следовательно, быструю загрузку). Список может включать классы JDK и классы уровня приложения.

Такой список можно построить разными способами, но проще всего запустить приложение с флагом `-XX:+DumpLoadedClassList=`*имя_файла*. Этот флаг создаст (в заданном файле) список всех классов, загруженных приложением.

На втором этапе этот список используется для генерирования архива:

```
$ java -Xshare:dump -XX:SharedClassListFile=filename \
    -XX:SharedArchiveFile=myclasses.jsa \
    ... аргументы classpath ...
```

Команда создает новый файл общего архива с заданным именем (myclasses.jsa в данном случае) на основании списка файлов. Также необходимо настроить путь загрузки классов так же, как это делается для запуска приложения (то есть с аргументом `-cp` или `-jar`, который обычно используется при запуске приложений).

Команда генерирует множество предупреждений о классах, которые ей не удается найти. Этого следовало ожидать, потому что эта команда не может найти динамически сгенерированные классы: посредники, классы на базе отражения (reflection) и т. д. Если вы увидите предупреждение для класса, который должен был быть загружен, попробуйте скорректировать путь загрузки для этой команды. Даже если не будут найдены все классы, это не проблема; просто это означает, что они будут загружены нормально (по пути загрузки), а не из общего архива. Загрузка конкретного класса немного замедлится, но при нескольких таких классах задержка будет незаметной, так что особо не переживайте.

Наконец, общий архитектор используется для запуска приложения:

```
$ java -Xshare:auto -XX:SharedArchiveFile=myclasses.jsa ... другие аргументы ...
```

Несколько замечаний по поводу этой команды. Во-первых, команда `-Xshare` имеет три возможных значения:

- `off` — не использовать CDS;
- `on` — использовать CDS всегда;
- `aucto` — попытаться автоматически использовать CDS.

Работа CDS зависит от отображения общего архива в область памяти, и при некоторых (большей частью редких) обстоятельствах попытка отображения может завершиться неудачей. Если задан флаг `-Xshare:on`, то в такой ситуации приложение не будет выполняться. Поэтому по умолчанию используется значение `-Xshare:auto`, которое означает, что в нормальной ситуации CDS будет использоваться, но если по какой-то причине попытка отображения архива в память завершилась неудачей, приложение продолжит работу без него. Так как по умолчанию этот флаг содержит значение `auto`, указывать его в предыдущей команде не обязательно.

Кроме того, команда задает местоположение общего архива. Значением по умолчанию для флага `SharedArchiveFile` является путь classes.jsa, упоминав-

шийся ранее (в каталоге server JDK). Таким образом, в Java 12 (где этот файл присутствует) никакие аргументы командной строки задавать не нужно, если вы хотите использовать общий архив по умолчанию (только для JDK).

В одном распространенном случае загрузка общего архива может завершиться неудачей: путь загрузки классов, используемый для построения общего архива, должен быть подмножеством пути загрузки классов, используемого для запуска приложения, и файлы JAR не должны были измениться с момента создания общего архива. Следовательно, не следует генерировать общий архив для классов, не входящих в JDK, и размещать его в каталоге по умолчанию, так как путь загрузки классов для произвольных команд не совпадет.

Также будьте внимательны с изменением файлов JAR. Если вы используете настройку по умолчанию `-Xshare:auto`, а файл JAR изменился, приложение все равно запустится, хотя общий архив не используется. Что еще хуже, никаких предупреждений не будет; единственное, что вы заметите — что приложение стало запускаться медленнее. По этой причине можно рассмотреть возможность использования флага `-Xshare:on` вместо значения по умолчанию, хотя существуют и другие причины, по которым попытка использования общего архива может завершиться неудачей.

Чтобы убедиться в том, что классы загружаются из общего архива, включите журнал загрузки классов (`-Xlog:class+load=info`) в командную строку; вы увидите обычный вывод загрузки классов, а классы, загруженные из общего архива, будут отображаться примерно так:

```
[0.080s][info][class,load] java.lang.Comparable source: shared objects file
```

Преимущества совместного использования данных классов

Очевидно, выигрыш от совместного использования данных классов зависит от количества загружаемых классов. В табл. 12.7 приведено время, необходимое для запуска примера сервера с биржевыми котировками из примеров книги; оно требует загрузки 6314 классов.

Таблица 12.7. Время запуска приложения с CDS

Режим CDS	Время запуска
`-Xshare:off`	8,9 секунды
`-Xshare:on` (по умолчанию)	9,1 секунды
`-Xshare:on` (специальный набор)	7,0 секунды

В стандартном случае используется только общий архив для JDK; последняя строка соответствует общему архиву всех классов приложения. В данном случае CDS экономит 30% времени запуска.

CDS также обеспечит некоторую экономию памяти, потому что данные классов будут совместно использоваться процессами. В целом, как было показано в примерах главы 8, данные класса в низкоуровневой памяти занимают относительно немного памяти, особенно в сравнении с кучей приложения. В большой программе с множеством классов CDS будет экономить больше памяти, хотя большой программе, скорее всего, понадобится куча еще большего размера, так что относительная экономия остается незначительной. Тем не менее в среде, в которой вам особенно сильно не хватает ресурсов для выделения низкоуровневой памяти и запуска нескольких копий JVM, использующих значительное число тех же классов, CDS также обеспечит некоторую экономию памяти.

РЕЗЮМЕ

- Лучший способ ускорить загрузку классов — создание архива совместного использования данных классов для приложения. К счастью, никакие программные изменения для этого не потребуются.

Случайные числа

Следующая группа API, которую мы рассмотрим, относится к генерированию случайных чисел. Java поставляется с тремя стандартными классами генераторов случайных чисел: `java.util.Random`, `java.util.concurrent.ThreadLocalRandom` и `java.security.SecureRandom`. Между этими классами существуют важные различия из области производительности.

Классы `Random` и `ThreadLocalRandom` отличаются тем, что главная операция (метод `nextGaussian()`) класса `Random` синхронизирована. Этот метод используется любым методом, получающим случайное значение, так что конкуренция за блокировку возможна независимо от того, как используется генератор случайных чисел: если два потока используют один генератор случайных чисел конкурентно, одному придется дожидаться, пока другой завершит свою операцию. По этой причине есть потоково-локальная версия: если каждый поток работает со своим генератором случайных чисел, синхронизация класса `Random` уже не создает проблем. (Как упоминалось в главе 7, потоково-локальная версия также обладает значительными преимуществами по производительности, потому что она повторно использует объект, создание которого требует высоких затрат.)

Различия между этими классами и классом `SecureRandom` заключаются в используемом алгоритме. Класс `Random` (и класс `ThreadLocalRandom` посредством

наследования) реализует типичный алгоритм генерирования псевдослучайных чисел. Хотя эти алгоритмы достаточно сложны, в конечном итоге они детерминированы. Если известна затравка (исходное значение), можно определить точную последовательность сгенерированных чисел. Это означает, что хакеры могут просмотреть серию чисел конкретного генератора и (через какое-то время) определить, каким будет следующее число. Хотя хорошие генераторы псевдослучайных чисел могут генерировать серии чисел, которые выглядят полностью случайными (и даже удовлетворяющими вероятностным ожиданиям), действительно случайными они не являются.

С другой стороны, класс `SecureRandom` использует системный интерфейс для получения затравки своих случайных данных. Конкретный способ генерирования данных специфичен для операционной системы, но в общем случае этот источник предоставляет данные, основанные на действительно случайных событиях (например, перемещениях мыши). Такой механизм называется случайностью на базе энтропии, и он намного более безопасен для операций, зависящих от случайных чисел.

Java различает два источника случайных чисел: для генерирования затравок и для генерирования самих случайных чисел. Затравки используются для создания открытых и закрытых ключей — например, ключей, которые используются для обращения к системе средствами SSH или PuTTY. Эти ключи предназначены для долгосрочного хранения, поэтому для них необходим самый сильный возможный криптографический алгоритм. Безопасные случайные числа также используются для инициализации потоков случайных чисел, включая те, которые используются стандартными реализациями библиотек Java SSL.

В системах Linux этими двумя источниками являются `/dev/random` (для затравок) и `/dev/urandom` (для случайных чисел). Обе системы базируются на источниках энтропии в машине — по-настоящему случайных факторах вроде перемещений мыши или нажатий клавиш.

Энтропия ограничена и повторно генерируется случайным образом, чтобы она была независимой как истинный источник случайности. Две системы подходят к решению своей задачи по-разному: `/dev/random` блокируется, пока не накопит достаточно системных событий для генерирования случайных данных, а `/dev/urandom` возвращается к использованию генератора псевдослучайных чисел (PRNG). Генератор псевдослучайных чисел инициализируется по случайному источнику, поэтому обычно с позиций криптографии он не слабее потока данных `/dev/random`. Но энтропия для генерирования самой затравки может оказаться недоступной, в этом случае появляется теоретическая возможность взлома потока `/dev/urandom`. Относительно укрепления этого потока есть немало аргументов, но в Java было принято часто встречающееся решение — использовать `/dev/random` для затравок и `/dev/urandom` для всего остального.

Отсюда следует, что получение большого количества затравок для случайных чисел может занять много времени. Вызовы метода `generatedSeed()` класса `SecureRandom` занимают непредсказуемое время в зависимости от того, сколько неиспользованной энтропии доступно в системе. Если энтропия недоступна, то вызов «подвисает» — возможно, на несколько секунд, пока не появится необходимая энтропия. Это заметно усложняет хронометраж производительности: сама производительность становится случайной. С другой стороны, метод `generateSeed()` используется всего для двух операций. Во-первых, некоторые алгоритмы используют его для получения затравки для будущих вызовов метода `nextRandom()`. Обычно это должно делаться только один раз, или не более чем периодически на протяжении срока жизни приложения. Во-вторых, этот метод используется при создании долгоживущих ключей, а эта операция выполняется относительно редко.

Так как количество таких операций ограничено, в большинстве приложений энтропия не будет исчерпана. Но ограниченная энтропия может стать проблемой в приложениях, которые создают шифры на стадии запуска, особенно в облачных средах, где устройство для получения случайных чисел хостовой ОС совместно используется несколькими виртуальными машинами и/или контейнерами Docker. В этом случае хронометраж работы программы будет иметь очень большую дисперсию, а поскольку безопасные затравки чаще всего используются во время инициализации программ, запуск приложений в этой сфере может быть довольно медленным.

В такой ситуации есть несколько возможных путей. В крайнем случае (и если код может быть изменен) альтернатива заключается в выполнении тестов производительности с использованием класса `Random`, даже при том, что класс `SecureRandom` будет использоваться в рабочей версии. Если тесты производительности являются тестами модульного уровня, такое решение может иметь смысл: тестам потребуется больше случайных затравок, чем рабочей системе за тот же период времени. Однако со временем предполагаемую нагрузку нужно будет протестировать с классом `SecureRandom`, чтобы определить, сможет ли нагрузка в рабочей системе получить достаточное количество случайных затравок.

Второй вариант — настройка генератора безопасных случайных чисел Java, чтобы он использовал `/dev/urandom` как для затравок, так и для случайных чисел. Это можно сделать двумя способами: прежде всего, можно задать системное свойство `-Djava.security.egd=file:/dev/urandom`.[1]

Также можно изменить настройку в $JAVA_HOME/jre/lib/security/java.security:

```
securerandom.source=file:/dev/random
```

[1] Из-за ошибки в ранних версиях Java иногда рекомендуется присваивать этому флагу `/dev/./urandom` или другие значения. В Java 8 и более поздних JVM можно просто использовать `/dev/urandom`.

Этот параметр определяет интерфейс, используемый для инициализации операций. Ему можно присвоить значение `/dev/urandom`, если вы хотите, чтобы генератор безопасных случайных чисел никогда не блокировался.

Впрочем, более эффективное решение заключается в настройке операционной системы, чтобы она поставляла больше энтропии, что делается при помощи демона `rngd`. Обязательно проследите за тем, чтобы демон `rngd` был настроен для использования надежных аппаратных источников энтропии (например, `/dev/hwrng`, если он доступен), а не чего-то вроде `/dev/urandom`. К преимуществам этого метода следует отнести то, что он решает проблемы энтропии для всех программ на машине, а не только для программ Java.

РЕЗЮМЕ

- Инициализация класса Random в Java обходится дорого, но после инициализации он может использоваться повторно.
- В многопоточном коде лучше использовать класс `ThreadLocalRandom`.
- Иногда класс `SecureRandom` демонстрирует произвольную, полностью случайную производительность. Тесты производительности кода, использующие этот класс, должны быть тщательно спланированы.
- Проблем с блокированием класса `SecureRandom` можно избежать при помощи изменений конфигурации, но лучше решать их на уровне ОС добавлением энтропии в систему.

Интерфейс JNI

В рекомендациях производительности относительно Java SE (особенно на заре существования Java) часто утверждается, что если вам нужна действительно высокая производительность, то придется использовать низкоуровневый код. На самом деле если вы хотите написать самый быстрый код из возможных, держитесь подальше от интерфейса JNI (Java Native Interface).

Хорошо написанный код Java на современных версиях JVM как минимум не уступит по скорости соответствующему коду C или C++ (сейчас не 1996 год). Блюстители чистоты языка продолжают оспаривать относительные показатели производительности Java и других языков; конечно, можно найти пример приложения, написанного на другом языке, которое будет работать быстрее аналогичного приложения на Java (хотя часто такие примеры содержат плохо написанный код Java). При этом такие дебаты упускают ключевой момент этого

раздела: если приложение уже написано на Java, вызов низкоуровневого кода по соображениям производительности почти всегда нежелателен.

Однако в некоторых случаях интерфейс JNI будет полезен. Платформа Java предоставляет многие стандартные возможности операционных систем, но если вам нужен доступ к какой-то специфической функции операционной системы — используйте JNI. И зачем строить свою библиотеку для выполнения операции, если уже доступна коммерческая (низкоуровневая) версия кода? В этих и других случаях вопрос заключается в том, как написать наиболее эффективный код JNI.

Ответ: избегайте вызовов из кода Java в C, насколько это возможно. Переход границы KNI (межъязыковые вызовы) обходится дорого. Так как вызов функции из существующей библиотеки C требует написания связующего кода, выделите время на создание нового укрупненного интерфейса, работающего через этот связующий код: выполняйте множество обращений к библиотеке C за один вызов.

Интересно, что обратное не всегда верно: обращение из кода C в Java не сопряжено с большими потерями производительности (в зависимости от параметров). Для примера возьмем следующий фрагмент:

```
@Benchmark
public void testJavaJavaJava(Blackhole bh) {
    long l = 0;
    for (int i = 0; i < nTrials; i++) {
        long a = calcJavaJava(nValues);
        l += a / nTrials;
    }
    bh.consume(l);
}

private long calcJavaJava(int nValues) {
    long l = 0;
    for (int i = 0; i < nValues; i++) {
        l += calcJava(i);
    }
    long a = l / nValues;
    return a;
}

private long calcJava(int i) {
    return i * 3 + 15;
}
```

Этот (совершенно бессмысленный) код содержит два цикла: внутри тестового метода и внутри метода `calcJavaJava()`. Вместо этого можно было бы воспользоваться низкоуровневым интерфейсом и записать внешний метод на C:

```
@Benchmark
public void testJavaCC(Blackhole bh) {
    long l = 0;
    for (int i = 0; i < nTrials; i++) {
        long a = calcCC(nValues);
        l += 50 - a;
    }
    bh.consume(l);
}

private native long calcCC(int nValues);
```

Также можно было реализовать внутренний вызов на C (код вполне очевиден).

В табл. 12.8 приведены данные производительности для разных комбинаций с 10 000 проверок и 10 000 значений.

Таблица 12.8. Время вычисления через границу JNI

calculateError	Calc	Random	Переходы JNI	Общее время
Java	Java	Java	0	0,104 ± 0,01 секунды
Java	Java	C	10 000 000	1,96 ± 0,1 секунды
Java	C	C	10 000	0,132 ± 0,01 секунды
C	C	C	0	0,139 ± 0,01 секунды

Реализация только внутреннего метода на C приводит к наибольшему числу переходов границы JNI (`numberOfTrials` × `numberOfLoops`, или 10 миллионов). Сокращение количества переходов до `numberOfTrials` (10 000) значительно уменьшает затраты, а сокращение их до 0 обеспечивает наилучшую производительность.

Код JNI работает менее эффективно, если параметры не являются простыми примитивами. Снижение эффективности объясняется двумя причинами. Во-первых, для простых ссылок требуется преобразование адресов. Во-вторых, операции с данными на базе массивов должны особым образом обрабатываться в низкоуровневом коде. К этой категории относятся объекты `String`, поскольку строковые данные фактически представляют собой массивы символов. Для обращения к отдельным элементам этих массивов специальный вызов должен зафиксировать объект в памяти (а для объектов `String` — выполнить преобразование из кодировки Java UTF-16 в UTF-8 в JDK 8). Когда массив станет ненужным, он должен быть явно освобожден в коде JNI.

Пока массив остается зафиксированным в памяти, уборщик мусора работать не может — поэтому одна из самых дорогостоящих ошибок в коде JNI за-

ключается в фиксации строки или массива в долго выполняемом коде. Это не позволяет работать уборщику мусора, что фактически блокирует все потоки приложения до завершения кода JNI. Исключительно важно создать критическую секцию, в которой время фиксации массива в памяти остается по возможности коротким.

Часто в журналах уборки мусора встречается запись `GC Locker Initiated GC`. Она указывает на то, что уборщик мусора должен был запуститься, но не смог, потому что поток зафиксировал данные в вызове JNI. Как только фиксация данных будет снята, уборщик мусора запустится. Если эта запись встречается слишком часто, поищите возможности для ускорения кода JNI; другие потоки приложения сталкиваются с задержками, ожидая выполнения уборки мусора.

Иногда цель фиксации объектов на короткий период времени вступает в конфликт с сокращением числа вызовов, пересекающих границу JNI. В этом случае вторая цель важнее, хотя она и означает множественные переходы границы JNI; следовательно, секции, фиксирующие массивы и строки, должны быть как можно короче.

РЕЗЮМЕ

- JNI не решает проблем с производительностью. Код Java почти всегда работает быстрее, чем вызовы низкоуровневого кода.
- При использовании JNI постарайтесь ограничить число вызовов из кода Java в C; переход границы JNI обходится дорого.
- Код JNI, использующий массивы или строки, должен зафиксировать эти объекты в памяти. Ограничьте время, в течение которого они остаются зафиксированными, чтобы они не влияли на работу уборщика мусора.

Исключения

Известно, что обработка исключений Java обходится недешево. Она требует чуть больших затрат, чем обработка обычных управляющих конструкций, хотя в большинстве случаев дополнительные затраты не оправдывают попыток обойти их. С другой стороны, так как обработка исключений не бесплатна, она не должна использоваться как механизм общего назначения. Следует использовать исключения в соответствии с общими принципами хорошей архитектуры программ: в основном код должен выдавать исключения, только чтобы сообщить о том, что произошло нечто неожиданное. Соблюдение правил хорошей архитектуры означает, что ваш код Java не будет замедляться обработкой исключений.

Два фактора могут влиять на общую производительность обработки исключений. Первый — это сам блок `try/catch`: требует ли он высоких затрат? Возможно, когда-то дело действительно обстояло так, но ситуация уже давно изменилась. Но у интернета хорошая память, и иногда попадаются рекомендации, которые советуют избегать исключений просто из-за блока `try/catch`. Такие рекомендации устарели; код, генерируемый современными JVM, достаточно эффективно обрабатывает исключения.

Второй фактор связан с тем, что средства обработки исключений получают трассировку стека в точке исключения (хотя в этом разделе будет представлено исключение из этого правила). Эта операция может быть затратной, особенно при большой глубине трассировки стека.

Рассмотрим пример. Ниже приведены три реализации некоторого метода:

```
private static class CheckedExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            try {
                if ((i % pctError) == 0) {
                    throw new CheckedException("Failed");
                }
                Object o = new Object();
                al.add(o.toString());
            } catch (CheckedException ce) {
                // Продолжить
            }
        }
        bh.consume(al);
    }
}

private static class UncheckedExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            Object o = null;
            if ((i % pctError) != 0) {
                o = new Object();
            }
            try {
                al.add(o.toString());
            } catch (NullPointerException npe) {
                // Продолжить
            }
        }
        bh.consume(al);
    }
```

```
}

private static class DefensiveExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            Object o = null;
            if ((i % pctError) != 0) {
                o = new Object();
            }
            if (o != null) {
                al.add(o.toString());
            }
        }
        bh.consume(al);
    }
}
```

Каждый из приведенных методов создает массив произвольных строк из вновь созданных объектов. Размер этого массива зависит от количества исключений, которые должны выдаваться.

В табл. 12.9 приведено время выполнения каждого метода для 100 000 итераций в худшем случае — для значения `pctError`, равного 1 (каждый вызов генерирует исключение, а результатом является пустой список). Приведенный пример кода выполняется либо поверхностно (это означает, что метод вызывается, когда в стеке находятся всего 3 класса), либо глубоко (метод вызывается, когда в стеке находятся 100 классов).

Таблица 12.9. Время обработки исключений при 100%

Метод	Поверхностный вызов	Глубокий вызов
Проверяемое исключение	24 031 ± 127 мкс	30 613 ± 329 мкс
Непроверяемое исключение	21 181 ± 278 мкс	21 550 ± 323 мкс
Защитное программирование	21 088 ± 255 мкс	21 262 ± 622 мкс

В таблице есть три интересных момента. Во-первых, в случае проверяемых исключений между поверхностным и глубоким случаем заметны значительные различия во времени. Построение трассировки стека требует времени, которое зависит от глубины стека.

Но во втором случае задействованы непроверяемые исключения; JVM создает исключение при разыменовании `null`-указателя. В какой-то момент компилятор оптимизирует случай исключения, сгенерированного системой; JVM начинает

повторно использовать тот же объект исключения (вместо того, чтобы создавать новый объект каждый раз, когда потребуется). Объект повторно используется при каждом выполнении кода независимо от состояния вызывающего стека, и исключение не содержит стека вызовов (то есть метод `printStackTrace()` не возвращает данных). Эта оптимизация не применяется достаточно долго — до момента выдачи исключения с полным стеком, так что если ваш тест не включает достаточного цикла разогрева, вы не увидите ее эффекта.

Наконец, случай, в котором исключение не выдается, обладает практически такой же производительностью, как случай непроверяемого исключения. Этот случай служит контрольным для данного эксперимента: тест выполняет немалый объем работы для создания объектов. Случай защитного программирования отличается от любого другого временем, потраченным на создание, выдачу и перехват исключения. При усреднении по 100 000 вызовов различия по времени выполнения будут практически незаметны (напомню, что этот пример соответствует худшему случаю).

Таким образом, потери производительности при неосмотрительном использовании исключений меньше, чем можно было бы ожидать, а потери при множестве экземпляров одного системного исключения почти незаметны. Однако в некоторых случаях можно встретить код, который просто создает слишком много исключений. Так как потери производительности обусловлены заполнением трассировок стека, можно установить флаг `-XX:-StackTraceInThrowable` (по умолчанию `false`), для того чтобы отключить генерирование трассировок стека.

Как правило, этого делать не стоит: трассировка стека позволяет вам проанализировать, что же неожиданно пошло не так. При установке этого флага данная возможность теряется. Кроме того, есть код, который проверяет трассировку стека и определяет, как восстановить работоспособность после исключения, на основании полученной информации. Короче, мало того, что флаг создает проблемы сам по себе — отключение трассировки может привести к загадочному нарушению работоспособности кода.

В JDK существуют некоторые API, у которых обработка исключений может привести к проблемам производительности. Многие классы коллекций выдают исключение при попытке удаления несуществующего элемента. Например, класс `Stack` выдает исключение `EmptyStackException`, если стек был пуст на момент вызова метода `pop()`. Обычно лучше применить в таком случае защитное программирование, предварительно проверив длину стека. (С другой стороны, в отличие от многих классов коллекций, класс `Stack` поддерживает `null`-объекты, поэтому метод `pop()` просто не может вернуть `null` для обозначения пустого стека).

Самый известный пример сомнительного использования исключений в JDK — загрузка классов: метод `loadClass()` класса `ClassLoader` выдает исключение `ClassNotFoundException`, если ему не удается найти запрашиваемый класс.

Такая ситуация в действительности не должна считаться аномальной. Нельзя ожидать, что отдельный загрузчик классов будет знать, как загрузить каждый класс в приложении; именно по этой причине существуют иерархии загрузчиков.

В среде с десятками загрузчиков это означает, что в процессе поиска по иерархии загрузчика, который знает, как загрузить конкретный класс, будет создано множество исключений. На очень больших серверах приложений, с которыми я работал, отключение генерирования трассировки стеков может ускорить запуск до 3%. Такие серверы загружали более 30 000 классов из сотен файлов JAR; хотя, конечно, у вас все может быть по-другому.

РЕЗЮМЕ

- Обработка исключений не обязательно сопряжена с затратами, хотя они должны использоваться только тогда, когда это уместно.
- Чем глубже стек, тем дороже обходится обработка исключений.
- JVM оптимизирует часто создаваемые системные исключения.
- Отключение трассировки стека в исключениях иногда позволяет повысить производительность, хотя в процессе часто теряется критически важная информация.

Журналы

Журналы — одна из тех областей, к которым специалисты по производительности относятся либо с любовью, либо с ненавистью (либо сочетают эти два чувства). Каждый раз, когда меня спрашивают, почему программа плохо работает, я первым делом прошу предоставить все доступные журналы в надежде, что в них можно будет найти подсказки относительно того, чем занимается приложение. Каждый раз, когда меня просят проанализировать производительность рабочего кода, я немедленно рекомендую отключить все команды вывода в журнал.

Речь идет о разных журналах. JVM выводит собственные данные журналов, самым важным из которых является журнал уборки мусора (см. главу 6). Этот журнал можно направить в отдельный файл, размером которого может управлять JVM. Даже в рабочем коде журнал уборки мусора (даже с включением подробного вывода журнала) требует настолько низких затрат и принесет настолько большую потенциальную пользу, если что-то пойдет не так, что журналы всегда следует держать включенными.

Серверы HTTP генерируют журнал обращений, который обновляется при каждом запросе. Обычно этот журнал заметно влияет на производительность:

его отключение определенно ускорит тест, выполняемый с сервером приложения. С точки зрения диагностируемости эти журналы (по моему опыту) не принесут особой пользы, если что-то пойдет не так. Тем не менее этот журнал часто критичен в контексте бизнес-требований, и тогда его следует оставить включенным.

Хотя это не входит в стандарт Java, многие серверы HTTP поддерживают конвенцию Apache `mod_log_config`, которая позволяет точно указать, какая информация должна регистрироваться для каждого запроса (а серверы, не поддерживающие синтаксис `mod_log_config`, обычно поддерживают другие средства настройки журналов). Главное — сохранять как можно меньше информации, соблюдая при этом бизнес-требования. Производительность протоколирования зависит от объема записываемых данных.

Прежде всего в журналах обращений HTTP (и в любых журналах вообще) желательно регистрировать всю информацию в числовом виде: IP-адреса вместо имен хостов, временные метки (например, количество секунд от начала эпохи) вместо строковых данных (вида «Monday, June 3, 2019 17:23:00 -0600») и т. д. Сведите к минимуму любые преобразования данных, вычисление которых потребует времени и памяти, чтобы также свести к минимуму их воздействие на систему. Журналы всегда можно подвергнуть пост-обработке для получения преобразованных данных.

Следует помнить три основных принципа, относящихся к журналам приложений. Во-первых, следует поддерживать баланс между сохраняемыми данными и уровнем протоколирования. JDK поддерживает семь стандартных уровней протоколирования, и журналы по умолчанию настраиваются для вывода данных трех из этих уровней (INFO и выше). Часто это приводит к недоразумениям в проектах: создается впечатление, что сообщения уровня INFO относительно обычны и должны описывать ход выполнения приложения («перехожу к выполнению задачи A», «перехожу к выполнению задачи B» и т. д.). Особенно для приложений с высокими уровнями многопоточности и масштабируемости такой объем журнальной информации окажет отрицательное воздействие на производительность (не говоря уже о риске того, что вывод окажется слишком длинным, чтобы принести реальную пользу). Не бойтесь использовать более низкие уровни протоколирования.

Аналогичным образом при сохранении кода в групповом репозитории следует учитывать потребности пользователя проекта, а не только ваши потребности как разработчика. Нам всем хотелось бы иметь хорошую обратную связь относительно того, как работает наш код, после того как он будет интегрирован в большую систему и пройдет серию тестов, но если сообщение не имеет смысла для конечного пользователя или системного администратора, включать его по умолчанию не стоит. Оно только будет замедлять систему (и сбивать с толку пользователя).

Второй принцип — использование узкоспециализированных средств вывода протоколирования. Возможно, отдельный диспетчер протоколирования в каждом классе будет довольно утомительно настраивать, но более высокая степень контроля за отладочным выводом часто оправдывает эти усилия. Совместное использование диспетчера протоколирования для набора классов в небольшом модуле может стать хорошим компромиссом. Учтите, что проблемы в рабочем экземпляре приложения (и особенно проблемы в рабочем экземпляре, происходящие под высокой нагрузкой или иным образом связанные с производительностью) трудно воспроизвести при значительном изменении условий среды.

Включение слишком высокого уровня детализации журнала часто изменяет среду так, что исходная проблема уже не проявляется. Следовательно, вы должны иметь возможность включить протоколирование только для небольшой части кода (и по крайней мере в исходном состоянии небольшая группа команд протоколирования работает на уровне FINE, а за ней следуют другие команды на уровнях FINER и FINEST), чтобы производительность кода не пострадала.

Необходимо иметь возможность активизировать небольшие подмножества сообщений в рабочей среде без вреда для производительности системы. Впрочем, обычно это можно считать одним из требований: вероятно, администраторы рабочей системы не станут включать журнал, если это приведет к замедлению системы, а если система замедлится — это снизит вероятность воспроизведения проблемы.

Третий принцип, который необходимо учитывать при внедрении журнала в код: помните, что очень легко написать код протоколирования с непредвиденными побочными эффектами, даже если вывод журнала не включен. Это еще один пример ситуации, в которой в «преждевременной» оптимизации кода нет ничего плохого: как показывает пример из главы 1, не забывайте использовать метод `isLoggable()` каждый раз, когда протоколируемая информация содержит вызов метода, конкатенацию строк или любую другую разновидность выделения памяти (например, выделение массива `Object` для аргумента `MessageFormat`).

РЕЗЮМЕ

- Код должен содержать большой объем протоколирования, чтобы пользователи могли разобраться в происходящем, но оно не должно быть включено по умолчанию.
- Не забудьте протестировать уровень протоколирования перед обращением к диспетчеру протоколирования, если аргументы требуют вызова методов или выделения памяти для объектов.

API коллекций Java

API коллекций Java весьма обширен; он содержит не менее 58 классов коллекций. Правильный выбор класса коллекции — а также правильное использование классов коллекций — является важным фактором производительности при написании приложений.

Первым правилом использования классов коллекций должен стать выбор коллекции, подходящей для алгоритмических потребностей вашего приложения. Этот совет относится не только к Java, а скорее является темой вводного учебного курса структур данных. Класс `LinkedList` плохо подходит для поиска; если вам нужен доступ к произвольному элементу данных, храните данные в `HashMap`. Если данные должны оставаться отсортированными, используйте `TreeMap`, вместо того чтобы пытаться сортировать данные в приложении. Используйте `ArrayList`, если обращение к данным будет происходить по индексу — но только не в том случае, если данные часто будут вставляться в середину массива. И так далее... Алгоритмический выбор класса коллекции критичен, но выбор в Java не сильно отличается от выбора в любом другом языке программирования.

При этом у коллекций Java есть свои особенности, которые также необходимо учитывать.

Синхронизированные и несинхронизированные коллекции

По умолчанию практически все коллекции Java не синхронизируются (основные исключения — `Hashtable`, `Vector` и связанные с ними классы).

В главе 9 был представлен микробенчмарк для сравнения защиты на базе CAS и традиционной синхронизации. В многопоточном случае это оказалось непрактичным, но что, если все обращения к данным всегда осуществляются из одного потока — что будет, если вообще не использовать никакой синхронизации? Результаты сравнения приведены в табл. 12.10.

Таблица 12.10. Производительность синхронизированного и несинхронизированного доступа

Режим	Одиночное обращение	10 000 обращений
Операция CAS	22,1 ± 11 нс	209 ± 90 мкс
Синхронизированный метод	20,8 ± 9 нс	179 ± 95 мкс
Несинхронизированный метод	15,8 ± 5 нс	104 ± 55 мкс

СИНХРОНИЗИРОВАННЫЕ КЛАССЫ КОЛЛЕКЦИЙ

Если вам интересно, почему `Vector` и `Hashtable` (и связанные с ними классы) синхронизированы, — немного истории.

На заре существования Java в JDK существовали только эти классы коллекций. В те времена (до Java 1.2) формального определения Collections Framework еще не было; были полезные классы, которые предоставлялись исходной платформой Java.

При первом выпуске Java многие разработчики неправильно понимали многопоточность, и создатели Java старались помочь разработчикам преодолеть некоторые проблемы программирования в многопоточной среде. Из-за этого классы были реализованы как потоково-безопасные.

К сожалению, синхронизация — даже без конкуренции — создавала огромные проблемы с производительностью в ранних версиях Java, так что при выходе следующей основной версии платформы в Collection Framework был выбран прямо противоположный подход: все новые классы коллекций по умолчанию были несинхронизированными. И хотя производительность синхронизации с тех пор была существенно улучшена, она все еще не обходится без затрат. Несинхронизированные коллекции помогают всем писать более быстрые программы (с встречающимися время от времени ошибками, вызванными параллельным изменением несинхронизированных коллекций).

Так как мы никак не пытаемся моделировать конкуренцию, микробенчмарк в этом примере действителен именно в этом конкретном случае: при отсутствии конкуренции и если вас интересует вопрос о том, с какими затратами связана «излишняя синхронизация» обращений к ресурсу.

Использование любого метода защиты данных (вместо простого несинхронизированного доступа) сопряжено с небольшими потерями. Как это обычно бывает с микробенчмарками, различия незначительны: порядка 5–8 наносекунд. Если операция достаточно часто выполняется в приложении, потери производительности будут более заметными. В большинстве случаев различия будут скомпенсированы намного большими неэффективностями в других областях приложения. Помните, что абсолютное значение полностью определяется целевой машиной, на которой выполнялся тест (в данном случае это была моя домашняя машина с процессором AMD); для получения более реалистичных измерений тест необходимо проводить на оборудовании, близком к целевой среде.

Таким образом, если приходится выбирать между синхронизированным списком (например, возвращенным синхронизированным методом `List()` класса

Collections) и несинхронизированным ArrayList, какую из коллекций следует использовать? Обращения к ArrayList будут чуть быстрее, и в зависимости от частоты обращений к списку может быть достигнут существенный выигрыш по производительности. (Как упоминалось в главе 9, лишние вызовы синхронизированного метода могут плохо отразиться на производительности некоторых аппаратных платформ.)

С другой стороны, этот выбор предполагает, что к коду никогда не будут обращаться сразу несколько потоков. Возможно, сегодня это так, но как насчет завтра? Если ситуация изменится, лучше использовать синхронизированные коллекции сегодня и поработать над всеми возможными последствиями для производительности. Это решение из области проектирования, и ответ на вопрос о том, оправдаются ли будущие затраты времени и усилий на проверку потоковой безопасности, зависит от особенностей разрабатываемого приложения.

Определение размера коллекции

Классы коллекций проектируются для хранения произвольного количества элементов данных. При добавлении новых элементов они должны расширяться по мере надобности. Правильный выбор размера коллекции может быть важным для их общей производительности.

И хотя набор типов данных, предоставляемых классами коллекций в Java, достаточно широк и разнообразен, на базовом уровне эти классы должны хранить свои данные с использованием только примитивных типов данных Java; числа (integer, double и т. д.), ссылки на объекты и массивы этих типов. Следовательно, ArrayList содержит фактический массив данных:

```
private transient Object[] elementData;
```

Элементы, добавляемые и удаляемые из ArrayList, сохраняются в нужной позиции массива elementData (что может привести к сдвигу других элементов массива). Аналогичным образом HashMap содержит массив внутреннего типа данных HashMap$Entry, который связывает каждую пару «ключ — значение» с позицией в массиве, определяемой хеш-кодом ключа.

Не все коллекции используют массив для хранения своих элементов; например, LinkedList хранит каждый элемент данных в объекте внутреннего класса Node. Но при выборе размера классов коллекций, которые используют массив для хранения своих элементов, приходится учитывать особые обстоятельства. Чтобы понять, относится ли конкретный класс к этой категории, достаточно взглянуть на его конструкторы: если у класса есть конструктор, который позволяет задать исходный размер коллекции, значит, в его внутреннем представлении для хранения элементов используется массив.

Для таких классов коллекций важно точно задать исходный размер. Возьмем простой пример `ArrayList`: массив `elementData` (по умолчанию) начнет с исходного размера 10. Когда в `ArrayList` вставляется 11-й элемент, список должен расширить массив `elementData`. Это означает выделение нового массива, копирование исходного содержимого в новый массив и последующее включение нового элемента. Структура данных и алгоритм, используемые, скажем, классом `HashMap`, намного сложнее, но принцип остается тем же: в какой-то момент размеры этих внутренних структур должны быть изменены.

Класс `ArrayList` изменяет размер массива, увеличивая текущий размер приблизительно на 50%, так что размер массива `elementData` сначала будет равен 10, потом 15, 22, 33 и т. д. Какой бы алгоритм ни использовался для изменения размера массива (см. врезку), это приведет к потерям памяти (а это в свою очередь повлияет на время, проведенное приложением за уборкой мусора). Кроме того, при каждом изменении размера массива должна выполняться затратная операция копирования массива для переноса содержимого из старого массива в новый.

Чтобы свести к минимуму эти потери производительности, проследите за тем, чтобы коллекция создавалась с максимально точной оценкой ее окончательного размера.

РАСШИРЕНИЕ ДАННЫХ В ДРУГИХ КЛАССАХ

Многие классы, не относящиеся к категории коллекций, также хранят большое количество данных во внутренних массивах. Например, класс `ByteArrayOutputStream` должен хранить все данные, записанные в поток, во внутреннем буфере; классы `StringBuilder` и `StringBuffer` тоже должны хранить все свои символы во внутреннем символьном массиве.

Многие из этих классов используют тот же алгоритм для изменения размеров своего внутреннего массива: он увеличивается вдвое каждый раз, когда потребуется изменить его размеры. Это означает, что в среднем размер внутреннего массива будет на 25% больше данных, содержащихся в нем в текущий момент.

Факторы производительности похожи: объем используемой памяти больше, чем в примере с `ArrayList`, а количество копирований данных меньше, но принцип остается тем же. Каждый раз, когда вам предоставляется возможность задать размер объекта при его создании и вы можете оценить, сколько данных будет храниться в объекте, используйте конструктор, который получает параметр с исходным размером.

Коллекции и эффективность использования памяти

Мы уже видели один пример ситуации, в которой эффективность использования памяти коллекциями может быть субоптимальной: во вспомогательном хранилище, в котором хранятся элементы коллекции, часть памяти нередко остается неиспользуемой.

Такие потери особенно проблематичны для разреженных коллекций (то есть содержащих один-два элемента). При интенсивном использовании разреженные коллекции могут терять большое количество памяти. Одно из возможных решений — определение размера коллекции при ее создании. Другой вариант — подумать, так ли необходима коллекция в данном случае. Если предложить разработчикам быстро отсортировать произвольный массив, они обычно предлагают быструю сортировку. Хороший специалист по производительности захочет узнать размер массива: в достаточно малых массивах самым быстрым способом сортировки может оказаться сортировка методом вставки[1]. Размер имеет значение.

РАЗМЕРЫ КОЛЛЕКЦИЙ В ПАМЯТИ

Так как недостаточное использование коллекций создает проблемы во многих приложениях, в Java (начиная с Java 8 и поздних версий Java 7) были оптимизированы реализации `ArrayList` и `HashMap`. По умолчанию (то есть когда при вызове конструктора не задан параметр размера) эти классы не выделяют памяти для хранения данных. Вспомогательное хранилище создается только при добавлении первого элемента в коллекцию.

Это пример отложенной инициализации, описанной в главе 7. Когда это изменение появилось в Java 7, тестирование некоторых распространенных приложений продемонстрировало повышение производительности из-за сокращения необходимости в уборке мусора. В таких приложениях нередко встречались многочисленные коллекции, которые никогда не использовались; отложенное выделение памяти для таких коллекций обеспечивало прирост производительности. Так как размер вспомогательного хранилища уже приходилось проверять при каждом обращении, проверка того, была ли выделена память для хранилища, уже не приводила к потерям производительности (хотя время, необходимое для создания исходного вспомогательного хранилища, сместилось от момента создания объекта к моменту вставки первых данных в объект).

[1] Реализации быстрой сортировки обычно используют сортировку методом вставки для малых массивов; в случае Java реализация метода Arrays.sort() предполагает, что массив, содержащий менее 47 элементов, будет отсортирован методом вставки быстрее, чем алгоритмом быстрой сортировки.

Аналогичным образом HashMap обеспечивает самый быстрый поиск элементов по ключу, но если ключ только один, использование HashMap обернется лишними затратами по сравнению с использованием простой ссылки на объект. Даже если ключей несколько, хранение нескольких ссылок на объект потребляет намного меньше памяти, чем полноценный объект HashMap, что положительно отразится на уборке мусора.

РЕЗЮМЕ

- Тщательно продумайте, как вы будете обращаться к коллекциям, и выберите для них подходящий тип синхронизации. Однако затраты при бесконфликтном доступе к коллекции с защищенной памятью (особенно использующей защиту на базе CAS) будут минимальными; иногда лучше перестраховаться.
- Определение размеров коллекций может оказать большое влияние на производительность: либо замедлить уборку мусора, если коллекция слишком велика, либо породить множество операций копирования и изменения размеров, если коллекция слишком мала.

Лямбда-выражения и анонимные классы

Для многих разработчиков самым интересным новшеством Java 8 стало появление лямбда-выражений. Безусловно, лямбда-выражения оказывают огромное положительное влияние на эффективность разработчиков Java, хотя, конечно, это влияние трудно выразить в числовом виде. Однако производительность кода можно анализировать при помощи лямбда-конструкций.

Основной вопрос по поводу производительности лямбда-выражений — какие результаты они показывают по сравнению со своей заменой, то есть анонимными классами? Как выясняется, различия незначительны.

Типичный пример использования лямбда-класса начинается с кода, который создает анонимные внутренние классы (в типичном примере часто используется Stream вместо итератора, приведенного ниже; информация о классе Stream будет приведена далее в этом разделе):

```
public void calc() {
    IntegerInterface a1 = new IntegerInterface() {
        public int getInt() {
            return 1;
        }
    };
```

```
    IntegerInterface a2 = new IntegerInterface() {
        public int getInt() {
            return 2;
        }
    };
    IntegerInterface a3 = new IntegerInterface() {
        public int getInt() {
            return 3;
        }
    };
    sum = a1.get() + a2.get() + a3.get();
}
```

Сравните со следующим кодом, использующим лямбда-выражения:

```
public int calc() {
   IntegerInterface a3 = () -> { return 3; };
   IntegerInterface a2 = () -> { return 2; };
   IntegerInterface a1 = () -> { return 1; };
    return a3.getInt() + a2.getInt() + a1.getInt();
}
```

Здесь критически важно тело лямбда-выражения или анонимного класса: если в нем выполняются какие-либо значительные операции, время, потраченное на операцию, затмит любые незначительные различия в реализациях лямбда-выражения или анонимного класса. Тем не менее даже в этом минимальном случае время, потраченное на операцию, фактически останется тем же, как показывает табл. 12.11, хотя с ростом количества выражений (то есть классов/лямбда-выражений) начинают проявляться небольшие различия.

Таблица 12.11. Время выполнения метода calc() с использованием лямбда-выражений и анонимных классов

Реализация	1024 выражения	3 выражения
Анонимные классы	781 ± 50 мкс	10 ± 1 нс
Лямбда-выражения	587 ± 27 мкс	10 ± 2 нс
Статические классы	734 ± 21 мкс	10 ± 1 нс

Один интересный момент: код, использующий анонимный класс, создает новый объект при каждом вызове метода. Если метод вызывается многократно (как это должно быть в эталонном тесте для измерения его производительности), вы создаете его быстро и теряете множество экземпляров этого анонимного класса. Как было показано в главе 5, такое использование часто не оказывает особого влияния на быстродействие. Выделение памяти (и что

еще важнее — инициализация) для объектов требует очень небольших затрат, а из-за быстрого освобождения они почти не замедляют уборку мусора. Но при измерении в наносекундном диапазоне эти мелкие затраты накапливаются. В последней строке таблицы используются заранее сконструированные объекты вместо анонимных классов:

```
private IntegerInterface a1 = new IntegerInterface() {
    public int getInt() {
        return 1;
    }
};
... Аналогично для других интерфейсов....
public void calc() {
       return a1.get() + a2.get() + a3.get();
   }
}
```

Типичное применение лямбда-выражений не создает новый объект при каждой итерации цикла — из-за чего в этой области некоторые граничные случаи могут сыграть в пользу производительности лямбда-выражений. Впрочем, даже в этом случае различия минимальны.

ЗАГРУЗКА КЛАССОВ С ЛЯМБДА-ВЫРАЖЕНИЯМИ И АНОНИМНЫМИ КЛАССАМИ

Один из граничных случаев, в которых проявляются эти различия, — запуск и загрузка классов. Было бы соблазнительно рассмотреть код лямбда-выражения и заключить, что он представляет собой «синтаксическое удобство» для создания анонимного класса. Однако на самом деле он работает не так — код лямбда-выражения создает статический метод, который вызывается через специальный вспомогательный класс. Анонимный класс представляет собой реальный класс Java; он имеет отдельный файл класса и загружается из загрузчика класса.

Как было показано ранее в этой главе, производительность загрузки классов может сыграть важную роль, особенно при длинном списке `classpath`. Таким образом, запуск программ с множеством анонимных классов (в отличие от множества лямбда-выражений) может продемонстрировать более значительные различия, чем показано здесь.

РЕЗЮМЕ

- Выбор между использованием лямбда-выражений и анонимного класса должен определяться простотой программирования, потому что в области производительности особых различий нет.

- Лямбда-выражения не реализуются в виде анонимных классов, так что одним исключением из этого правила становятся среды, в которых важно поведение загрузки классов; в этом случае лямбда-выражения работают чуть быстрее.

Производительность потоков данных и фильтров

Еще одной ключевой особенностью Java 8, которая часто используется в сочетании с лямбда-выражениями, является новый API `Stream`. Одной важной характеристикой потоков данных является автоматическая параллелизация кода. Информация о параллельных потоках данных приводилась в главе 9; в этом разделе обсуждаются общие характеристики производительности потоков данных и фильтров.

Отложенный перебор

Первое преимущество потоков данных заключается в том, что они реализуются в виде отложенных структур данных. Допустим, имеется список биржевых обозначений акций и вы хотите найти первое обозначение в списке, которое не содержит буквы A. Код решения этой задачи с использованием потока данных выглядит примерно так:

```
@Benchmark
public void calcMulti(Blackhole bh) {
    Stream<String> stream = al.stream();
    Optional<String> t = stream.filter(symbol -> symbol.charAt(0) != 'A').
        filter(symbol -> symbol.charAt(1) != 'A').
        filter(symbol -> symbol.charAt(2) != 'A').
        filter(symbol -> symbol.charAt(3) != 'A').findFirst();
    String answer = t.get();
    bh.consume(answer);
}
```

Очевидно, есть более эффективная реализация с использованием одиночного фильтра, но мы еще вернемся к ее обсуждению в этом разделе. А пока посмотрим, что означает отложенная реализация потока в данном примере. Каждый метод `filter()` возвращает новый поток, поэтому фактически здесь задействованы четыре логических потока.

Как выясняется, метод `filter()` не делает ничего, кроме настройки серии указателей. В результате при вызове метода `findFirst()` для потока никакая обработка данных не выполняется — никакие сравнения данных с символом `A` еще не были сделаны.

Вместо этого `findFirst()` запрашивает элемент у предыдущего потока данных (возвращаемого фильтром 4). Поток данных еще не содержит элементов, поэтому он обращается к потоку, произведенному фильтром 3, и т. д. Фильтр 1 берет первый элемент из списка (формально — из потока) и проверяет, начинается ли он с символа `A`. В таком случае он завершает обратный вызов и возвращает этот элемент далее по цепочке; в противном случае он продолжает перебирать массив, пока не найдет подходящий элемент (или не завершит перебор всего массива). Фильтр 2 ведет себя аналогично — при возвращении обратного вызова к фильтру 1 он проверяет, отличен ли второй символ от `A`. В таком случае обратный вызов завершается, а символ передается далее по цепочке; если нет — он выдает еще один обратный вызов к фильтру 1 для получения следующего биржевого обозначения.

На первый взгляд кажется, что схема с таким количеством обратных вызовов неэффективна, но рассмотрим альтернативу. Алгоритм с немедленной обработкой потока данных будет выглядеть примерно так:

```
private <T> ArrayList<T> calcArray(List<T> src, Predicate<T> p) {
    ArrayList<T> dst = new ArrayList<>();
    for (T s : src) {
        if (p.test(s))
            dst.add(s);
    }
    return dst;
}

@Benchmark
public void calcEager(Blackhole bh) {
    ArrayList<String> al1 = calcArray(al, 0, 'A');
    ArrayList<String> al2 = calcArray(al1, 1, 'A');
    ArrayList<String> al3 = calcArray(al2, 2, 'A');
    ArrayList<String> al4 = calcArray(al3, 3, 'A');
    String answer = al4.get(0);
    bh.consume(answer);
}
```

Такой подход уступает по эффективности отложенной реализации, используемой в Java, по двум причинам. Во-первых, он требует создания множества временных экземпляров класса `ArrayList`. Во-вторых, в отложенной реализации обработка может остановиться сразу же при получении элемента методом `findFirst()`. Это означает, что реально через эти фильтры должно пройти только подмножество элементов. С другой стороны, немедленная реализация должна обработать весь список несколько раз, пока не будет создан последний список.

А значит, не приходится удивляться тому, что отложенная реализация заметно превосходит по производительности альтернативу из этого примера. В данном

случае тест обрабатывает список из 456 976 четырехбуквенных биржевых обозначений, отсортированных по алфавиту. Немедленная реализация обработает только 18 278 из этих обозначений, прежде чем столкнется с обозначением BBBB; в этой точке она может остановиться. Итератору для нахождения этого ответа потребуется на два порядка больше времени, как видно из табл. 12.12.

Таблица 12.12. Время обработки с отложенными и немедленными фильтрами

Реализация	Необходимое время
Фильтр/findfirst	0,76 ± 0,047 мс
Итератор/findfirst	108,4 ± 4 мс

Следовательно, одна из причин, по которым фильтры могут работать намного быстрее итераторов, заключается в том, что они могут воспользоваться алгоритмическими возможностями оптимизации: сделав то, что требовалось сделать, отложенная реализация фильтра может завершить обработку, что приводит к сокращению объема обрабатываемых данных.

А если требуется обработать весь набор данных? Как соотносится базовая производительность фильтров и итераторов в этом случае? Для этого примера тест будет слегка изменен. Предыдущий пример продемонстрировал одну поучительную особенность работы множественных фильтров, но вполне очевидно, что код будет лучше работать с одним фильтром:

```
@Benchmark
public void countFilter(Blackhole bh) {
    count = 0;
    Stream<String> stream = al.stream();
    stream.filter(
        symbol -> symbol.charAt(0) != 'A' &&
        symbol.charAt(1) != 'A' &&
        symbol.charAt(2) != 'A' &&
        symbol.charAt(3) != 'A').
          forEach(symbol -> { count++; });
    bh.consume(count);
}
```

В этом примере также изменен итоговый код подсчета обозначений акций, чтобы обработан был весь список. С другой стороны, немедленная реализация теперь может использовать итератор напрямую:

```
@Benchmark
public void countIterator(Blackhole bh) {
    int count = 0;
    for (String symbol : al) {
```

```
            if (symbol.charAt(0) != 'A' &&
                symbol.charAt(1) != 'A' &&
                symbol.charAt(2) != 'A' &&
                symbol.charAt(3) != 'A')
                count++;
            }
        bh.consume(count);
    }
```

Даже в этом случае отложенная реализация с фильтром быстрее реализации с итератором (табл. 12.13).

Таблица 12.13. Время обработки с одним фильтром и итератором

Реализация	Необходимое время
Фильтры	7 ± 0,6 мс
Итератор	7,4 ± 3 мс

РЕЗЮМЕ

- Фильтры обеспечивают значительный выигрыш по производительности, позволяя завершить обработку в середине перебора данных.
- Даже если обрабатывается весь набор данных, одиночный фильтр слегка превосходит итератор по производительности.
- Использование множественных фильтров создает дополнительные затраты; проследите за тем, чтобы фильтры были написаны качественно.

Сериализация объектов

Сериализация объектов представляет собой механизм сохранения двоичного состояния объекта для его последующего восстановления. JDK предоставляет стандартный механизм сериализации объектов, реализующих интерфейс `Serializable` или `Externalizable`. Производительность сериализации практически каждого возможного объекта может быть улучшена по сравнению с кодом сериализации по умолчанию, но это определенно одна из тех областей, в которых преждевременное выполнение этой оптимизации будет неразумным. Написание специального кода сериализации и десериализации объектов требует времени, и такой код создает больше проблем с сопровождением, чем код, использующий

сериализацию по умолчанию. Кроме того, правильно написать код сериализации тоже непросто, так что любые попытки оптимизировать его повышают риск написания некорректного кода.

Временные поля

В общем случае улучшение затрат на сериализацию объектов сводится к сериализации меньшего количества данных. Для этого поля объекта помечаются как *временные*, чтобы они не сериализовались по умолчанию. Тогда класс может предоставить специальные методы `writeObject()` и `readObject()` для обработки этих данных. Если данные не нужны для восстановления, достаточно пометить их как временные.

Переопределение сериализации по умолчанию

Методы `writeObject()` и `readObject()` позволяют взять процесс сериализации данных под полный контроль. Впрочем, с большим контролем приходит большая ответственность: допустить ошибку очень легко.

Чтобы понять, почему при оптимизации сериализации возникает столько проблем, возьмем простой объект `Point`, представляющий точку на плоскости:

```
public class Point implements Serializable {
    private int x;
    private int y;
    ...
}
```

Для выполнения сериализации можно написать такой код:

```
public class Point implements Serializable {
    private transient int x;
    private transient int y;
    ....
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        oos.writeInt(x);
        oos.writeInt(y);
    }
    private void readObject(ObjectInputStream ois)
                          throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        x = ois.readInt();
        y = ois.readInt();
    }
}
```

В этом простом примере более сложный код быстрее работать не будет, но при этом останется функционально правильным. Но будьте осторожны при применении этого метода в общем случае:

```
public class TripHistory implements Serializable {
    private transient Point[] airportsVisited;
    ....
    // ЭТОТ КОД ФУНКЦИОНАЛЬНО НЕПРАВИЛЕН
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        oos.writeInt(airportsVisited.length);
        for (int i = 0; i < airportsVisited.length; i++) {
            oos.writeInt(airportsVisited[i].getX());
            oos.writeInt(airportsVisited[i].getY());
        }
    }

    private void readObject(ObjectInputStream ois)
                          throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        int length = ois.readInt();
        airportsVisited = new Point[length];
        for (int i = 0; i < length; i++) {
            airportsVisited[i] = new Point(ois.readInt(), ois.readInt();
        }
    }
}
```

Здесь в поле `airportsVisited` хранится массив аэропортов, в которые или из которых я вылетал, в порядке их посещения. Таким образом, некоторые аэропорты (такие, как `JFK`) часто встречаются в массиве; `SYD` встречается только один раз (пока).

Из-за того что запись ссылок на объекты обходится дорого, этот код будет работать определенно быстрее, чем механизм сериализации этого массива по умолчанию: на моей машине сериализация массива из 100 000 объектов `Point` занимает 15,5 ± 0,3 мс, а десериализация — 10,9 ± 0,5 мс. При использовании такой «оптимизации» сериализация занимает 1 ± 0,600 мс, а десериализация — 0,85 ± 0,2 мкс.

Однако этот код неправилен. До сериализации один объект представляет JFK, а ссылка на этот объект встречается в массиве несколько раз. После того как массив пройдет сериализацию с последующей десериализацией, появится сразу несколько объектов, представляющих JFK. Таким образом изменяется поведение приложения.

При десериализации массива в этом примере ссылки на JFK преобразуются в отдельные объекты. Теперь при изменении одного из таких объектов будет

изменен только этот конкретный объект, и он будет содержать другие данные, чем остальные объекты, относящиеся к JFK.

Это важный принцип, о котором необходимо помнить, потому что суть оптимизации сериализации часто заключается в особой обработке ссылок на объекты. При правильном исполнении вы сможете радикально повысить производительность кода сериализации; при неправильном — создадите коварные ошибки.

Учитывая этот факт, рассмотрим сериализацию класса `StockPriceHistory` и посмотрим, как ее можно оптимизировать. Этот класс включает следующие поля:

```
public class StockPriceHistoryImpl implements StockPriceHistory {
    private String symbol;
    protected SortedMap<Date, StockPrice> prices = new TreeMap<>();
    protected Date firstDate;
    protected Date lastDate;
    protected boolean needsCalc = true;
    protected BigDecimal highPrice;
    protected BigDecimal lowPrice;
    protected BigDecimal averagePrice;
    protected BigDecimal stdDev;
    private Map<BigDecimal, ArrayList<Date>> histogram;
    ....
    public StockPriceHistoryImpl(String s, Date firstDate, Date lastDate) {
        prices = ....
    }
}
```

Когда для заданного обозначения `s` строится история цен, объект создает и сохраняет отсортированную карту `prices`, ключом которой представляет дату всех цен между `start` и `end`. Код также сохраняет `firstDate` и `lastDate`. Конструктор не заполняет другие поля; они инициализируются в отложенном режиме. При вызове get-метода любого из этих полей get-метод проверяет `needsCalc`. Если переменная содержит `true`, вычисляются значения остальных полей (все конкурентно).

Вычисление включает создание гистограммы, которая показывает, сколько дней при закрытии торгов действовала конкретная цена. Гистограмма содержит те же данные (в отношении объектов `BigDecimal` и `Date`), которые содержатся в карте `prices`; она всего лишь обеспечивает другой способ представления данных.

Из-за того что все поля с отложенной инициализацией могут быть вычислены по массиву `prices`, все они могут быть помечены как временные; для их сериализации и десериализации никакая дополнительная работа не потребуется. В данном случае пример выглядит просто, потому что код уже выполнял отложенную инициализацию полей; он может повторить отложенную инициализацию при получении данных. Даже если бы код немедленно инициализировал эти поля,

он мог бы пометить все вычисленные поля как временные и заново вычислить их значения в методе `readObject()` класса.

Также обратите внимание на то, что при этом сохраняется объектное отношение между объектами `prices` и `histogram`: при повторном вычислении гистограммы существующие объекты просто будут вставлены в новую карту.

Такие оптимизации почти всегда положительны, но в некоторых случаях они могут ухудшить производительность. В табл. 12.14 приведено время, необходимое для сериализации и десериализации для временного и постоянного объекта `histogram`, а также размера сериализованных данных в каждом случае.

Таблица 12.14. Время сериализации и десериализации объектов с временными полями

Объект	Время сериализации	Время десериализации	Размер данных
Без временных полей	19,1 ± 0,1 мс	16,8 ± 0,4 мс	785 395 байт
Временное поле histogram	16,7 ± 0,2 мс	14,4 ± 0,2 мс	754 227 байт

На данный момент пример экономит около 15% общего времени сериализации и десериализации объекта. Но в этом тесте объект `histogram` не воссоздавался на стороне получения. Объект будет создан тогда, когда получающий код впервые обратится к нему.

В некоторых случаях объект `histogram` не понадобится; может оказаться, что получателя интересуют только цены за определенные дни, а не гистограмма. Здесь встречается более необычный случай: если поле `histogram` должно быть постоянно доступным, а его вычисление занимает более 2,4 мс, то поля с отложенной инициализацией на самом деле приведут к общему снижению производительности.

В данном случае вычисление гистограммы не относится к этой категории — эта операция выполняется очень быстро. В общем случае может быть трудно найти ситуацию, в которой повторное вычисление некоторых данных обходилось бы дороже сериализации и десериализации этих данных. Этот фактор следует учитывать при оптимизации кода.

В этом тесте фактическая передача данных не выполняется; данные записываются и читаются в предварительно выделенные байтовые массивы, чтобы измерялось только время сериализации и десериализации. Также заметим, что объявление поля `histogram` временным также сэкономило около 13% размера данных. Это может быть важно, если данные должны передаваться по сети.

Сжатие сериализованных данных

Производительность сериализации кода можно повысить еще одним способом: сжатием сериализованных данных, чтобы они быстрее передавались. В классе, представляющем историю цен, для этого можно провести сжатие карты `prices` в процессе сериализации:

```
public class StockPriceHistoryCompress
        implements StockPriceHistory, Serializable {

    private byte[] zippedPrices;
    private transient SortedMap<Date, StockPrice> prices;

    private void writeObject(ObjectOutputStream out)
        throws IOException {
        if (zippedPrices == null) {
            makeZippedPrices();
        }
        out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        unzipPrices();
    }

    protected void makeZippedPrices() throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream zip = new GZIPOutputStream(baos);
        ObjectOutputStream oos = new ObjectOutputStream(
                new BufferedOutputStream(zip));
        oos.writeObject(prices);
        oos.close();
        zip.close();
        zippedPrices = baos.toByteArray();
    }

   protected void unzipPrices()
       throws IOException, ClassNotFoundException {
        ByteArrayInputStream bais = new ByteArrayInputStream(zippedPrices);
        GZIPInputStream zip = new GZIPInputStream(bais);
        ObjectInputStream ois = new ObjectInputStream(
                new BufferedInputStream(zip));
        prices = (SortedMap<Date, StockPrice>) ois.readObject();
        ois.close();
        zip.close();
    }
}
```

Метод `zipPrices()` сериализует карту в байтовый массив и сохраняет полученные байты, которые затем нормально сериализуются в методе `writeObject()` при вызове метода `defaultWriteObject()`. (Собственно, если процесс сериализации настраивается, будет чуть лучше сделать массив `zippedPrices` временным и записать его длину и байты напрямую. Но приведенный пример кода чуть понятнее, а чем проще — тем лучше.) При десериализации выполняется обратная операция. Если целью является сериализация байтового потока (как в исходном примере), это гиблое дело. И это не удивительно: время, необходимое для сжатия байтов, намного больше времени их записи в локальный байтовый массив. Время выполнения операций приведено в табл. 12.15.

Таблица 12.15. Время сериализации и десериализации 10 000 объектов со сжатием

Сценарий использования	Время сериализации	Время десериализации	Размер данных
Без сжатия	16,7 ± 0,2 мс	14,4 ± 0,2 мс	754 227 байт
Сжатие/восстановление	43,6 ± 0,2 мс	18,7 ± 0,5 мс	231 844 байта
Только сжатие	43,6 ± 0,2 мс	0,720 ± 0,3 мс	231 844 байта

Самое интересное в этой таблице — последняя строка. В этом тесте данные сжимаются перед сжатием, но метод `unzipPrices()` не вызывается в методе `readObject()`. Вместо этого он вызывается тогда, когда потребуется, то есть при первом вызове метода `getPrice()` получающей стороной. Без этого вызова нужно десериализовать несколько объектов `BigDecimal`, а это делается довольно быстро.

В этом примере получателю вообще могут не понадобиться фактические цены: возможно, ему будет достаточно вызвать `getHighPrice()` и аналогичные методы для получения сводной информации о данных. Если ничего, кроме этих методов, не понадобится, отложенное восстановление информации цен может сэкономить немало времени. Отложенное восстановление также может пригодиться, если соответствующий объект помещается в долгосрочное хранилище (например, если это состояние сеанса HTTP, которое сохраняется в резервной копии на случай отказа сервера приложения). Отложенное восстановление данных экономит как процессорное время (за счет отказа от восстановления), так и память (так как сжатые данные занимают меньше места).

Следовательно — особенно если целью является экономия памяти, а не времени — сжатие данных для сериализации и их отложенное восстановление могут принести пользу.

Если сериализация проводится для передачи данных по сети, мы имеем стандартные плюсы и минусы, зависящие от скорости сети. В быстрой сети время сжатия легко может превысить время, сэкономленное за счет передачи меньшего

объема данных; в более медленных сетях может быть справедливо обратное. В данном случае мы передаем приблизительно на 500 000 меньше байт, поэтому мы можем вычислить потери или экономию на основании среднего времени передачи такого объема данных. В нашем примере на сжатие данных тратится около 40 мс. Медленная общедоступная Wi-Fi-сеть выиграет от применения сжатия, а более быстрые сети — нет.

Отслеживание дубликатов

Раздел «Сериализация объектов» начинается с примера того, как не нужно сериализовать данные, содержащие ссылки на объекты, иначе ссылки могут быть изменены при десериализации данных. Одной из более мощных оптимизаций, которые могут применяться в методе `writeObject()`, становится предотвращение записи дубликатов ссылок на объекты. В классе `StockPriceHistoryImpl` это означает отказ от записи дубликатов ссылок в карту `prices`. Так как в примере для карты используется стандартный класс JDK, беспокоиться об этом не нужно: классы JDK уже написаны с расчетом на оптимальную сериализацию своих данных. Однако будет поучительно посмотреть, как эти классы выполняют свои оптимизации, чтобы лучше представить существующие возможности.

В классе `StockPriceHistoryImpl` ключевой структурой является `TreeMap`. Упрощенная версия этой карты показана на рис. 12.2. С сериализацией по умолчанию JVM записала бы примитивные поля данных для узла A; затем она рекурсивно вызовет метод `writeObject()` для узла B (а затем для узла C). Код узла B запишет примитивные поля данных, а затем рекурсивно запишет данные своего родительского поля.

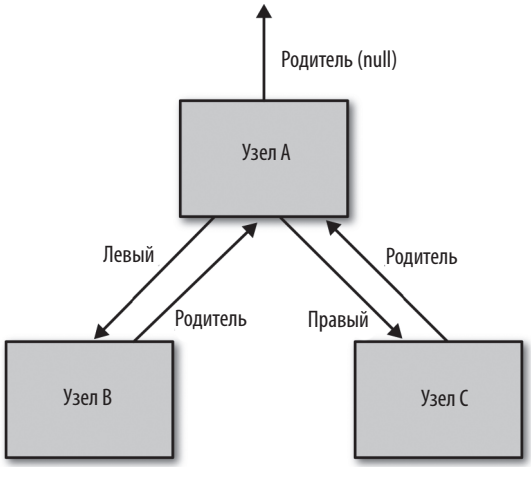


Рис. 12.2. Простая структура TreeMap

Стоп — этим родительским полем является узел A, который уже был записан. Код сериализации объектов достаточно умен, чтобы это понять: он не станет повторно записывать данные узла A. Вместо этого он просто добавит ссылку на объект для уже записанных данных.

Отслеживание набора ранее записанных объектов, а также вся рекурсия немного повышают затраты на сериализацию объектов. Но как было продемонстрировано в примере с массивом объектов `Point`, потери неизбежны: код должен отслеживать ранее записанные объекты и реконструировать правильные ссылки на объекты. Однако можно выполнить нетривиальные оптимизации за счет подавления ссылок на объекты, которые могут быть легко воссозданы при десериализации объекта.

Разные классы коллекций по-разному решают эту проблему. Например, класс `TreeMap` просто обходит дерево и записывает только ключи и значения; сериализация отбрасывает всю информацию об отношениях между ключами (то есть их порядок сортировки). Когда данные будут десериализованы, метод `readObject()` снова сортирует эти данные для построения дерева. И хотя может показаться, что повторная сортировка объектов обойдется дорого, на самом деле это не так: для набора из 10 000 объектов этот процесс выполняется приблизительно на 20% быстрее сериализации объектов по умолчанию, которая отслеживает все ссылки на объекты.

Класс `TreeMap` также выигрывает от этой оптимизации, потому что он может сохранять меньше объектов. Узел (в терминологии JDK — `Entry`) в этой карте содержит два объекта: ключ и значение. Так как карта не может содержать два идентичных узла, коду сериализации не нужно беспокоиться о защите ссылок объектов на узлы. В данном случае он может пропустить запись самого объекта узла и просто записать объекты ключа и значения. В итоге метод `writeObject()` будет выглядеть примерно так (синтаксис адаптирован для удобства чтения):

```
private void writeObject(ObjectOutputStream oos) throws IOException {
    ....
    for (Map.Entry<K,V> e : entrySet()) {
        oos.writeObject(e.getKey());
        oos.writeObject(e.getValue());
    }
    ....
}
```

Код очень похож на тот, который не работал в примере с `Point`. Отличие заключается в том, что код все равно записывает объект, когда эти объекты могут быть идентичными. `TreeMap` не может содержать два идентичных узла, так что записывать ссылки на узлы не нужно. `TreeMap` *может* содержать два

совпадающих значения, поэтому значения должны записываться как ссылки на объекты.

Мы возвращаемся к началу: как я упоминал в начале этого раздела, правильная реализация оптимизаций сериализации объектов может быть непростой задачей. Но когда сериализация объектов создает серьезное узкое место в приложении, правильная оптимизация может обеспечить заметный выигрыш.

КАК НАСЧЕТ EXTERNALIZABLE?

Другой подход к оптимизации сериализации объектов основан на реализации интерфейса `Externalizable` вместо интерфейса `Serializable`.

На практике эти два интерфейса различаются тем, как они обрабатывают поля, которые не являются временными. Интерфейс `Serializable` записывает невременные поля при вызове метода `defaultWriteObject()` методом `writeObject()`. Класс `Externalizable` должен явно записать все поля — как временные, так и другие, — которые он намерен передавать. Даже если все поля объекта являются временными, лучше реализовать интерфейс `Serializable` и вызвать метод `defaultWriteObject()`. Этот путь ведет к коду, который будет намного проще в сопровождении при добавлении (и удалении) полей в коде. У интерфейса `Externalizable` нет никаких внутренних преимуществ с точки зрения производительности: в конечном итоге все определяется объемом записываемых данных.

РЕЗЮМЕ

- Сериализация данных может стать важным узким местом в производительности.
- Пометка переменных экземпляров как временных ускорит сериализацию и сократит объем передаваемых данных. Оба фактора обычно обеспечивают значительный прирост производительности, если только повторное создание данных на стороне получателя не занимает много времени.
- Другие оптимизации с использованием методов `writeObject()` и `readObject()` могут значительно ускорить сериализацию. Будьте осторожны с ними, потому что вы легко можете создать в программе трудноуловимую ошибку.
- Сжатие сериализованных данных часто приносит пользу, даже если данные не будут передаваться по медленной сети.

Итоги

Краткое описание ключевых областей Java SE JDK завершает описание темы производительности Java. У большинства тем этой главы есть одна интересная особенность: они демонстрируют эволюцию производительности самого JDK. По мере развития и становления Java как платформы ее разработчики обнаружили, что повторно генерируемым исключениям не обязательно тратить время на сбор данных стека; что использование потоково-локальных переменных для предотвращения синхронизации генератора случайных чисел может принести пользу; что размер по умолчанию коллекции `ConcurrentHashMap` был слишком большим; и т. д.

Непрерывный процесс внесения последовательных улучшений — в этом заключается вся суть настройки производительности Java. От настройки компилятора и уборщика мусора до эффективного использования памяти и понимания ключевых различий производительности в API — инструменты и процессы, описанные в книге, позволят вам реализовать аналогичные улучшения в вашем собственном коде.

ПРИЛОЖЕНИЕ

Список флагов настройки

В приложении перечислены часто используемые флаги и даются рекомендации по их использованию. Среди них встречаются флаги, которые часто использовались в старых версиях Java и в настоящее время не рекомендованы к использованию. Эти флаги могут упоминаться в документации и рекомендациях по старым версиям Java, поэтому я привожу их здесь.

Таблица П.1. Флаги настройки JIT-компилятора

Флаг	Назначение	Когда используется	См. также
`-server`	Флаг ни на что не влияет и игнорируется	–	«Многоуровневая компиляция», с. 123
`-client`	Флаг ни на что не влияет и игнорируется	–	«Многоуровневая компиляция», с. 123
`-XX:+TieredCompilation`	Использует многоуровневую компиляцию	Всегда, кроме ситуаций с крайней нехваткой памяти	«Многоуровневая компиляция», с. 123, «Плюсы и минусы многоуровневой компиляции», с. 145
`-XX:ReservedCodeCacheSize=<MB>`	Резервирует память для кода, компилируемого JIT-компилятором	Когда вы запускаете большую программу и получаете предупреждение о нехватке памяти в кэше команд	«Настройка кэша команд», с. 125
`-XX:InitialCodeCacheSize=<MB>`	Выделяет исходную память для кода, компилируемого JIT-компилятором	Если вам потребовалось заранее выделить память для кэша команд (что бывает нечасто)	«Настройка кэша команд», с. 125

Флаг	Назначение	Когда используется	См. также
`-XX:CompileThreshold=<N>`	Задает количество выполнений метода или цикла перед его компиляцией	Использовать этот флаг не рекомендуется	«Пороги компиляции», с. 137
`-XX:+PrintCompilation`	Предоставляет журнал операций JIT-компилятора	Если вы подозреваете, что важный метод не компилируется, или вас интересует, чем вообще занимается компилятор	«Анализ процесса компиляции», с. 127
`-XX:CICompilerCount=<N>`	Задает количество потоков, используемых JIT-компилятором	Когда запускается слишком много потоков. Это актуально прежде всего для больших машин, на которых работают несколько JVM	«Пороги компиляции», с. 137
`-XX:+DoEscapeAnalysis`	Включает агрессивные оптимизации компилятора	В редких случаях этот флаг может приводить к сбоям, поэтому в некоторых источниках рекомендуют его отключать. Не делайте этого, если только вы не знаете, что он породил проблему	«Анализ локальности», с. 143
`-XX:UseAVX=<N>`	Задает набор команд для процессоров Intel	Присвойте значение 2 в ранних версиях Java 11; в поздних версиях значение 2 используется по умолчанию	«Код для конкретного процессора», с. 144
`-XX:AOTLibrary=<path>`	Использует конкретную библиотеку для предварительной компиляции	В отдельных ситуациях может ускорить начальное выполнение программы	«Предварительная компиляция», с. 149

Таблица П.2. Флаги выбора алгоритма уборки мусора

Флаг	Назначение	Когда используется	См. также
`-XX:+UseSerialGC`	Использует простой, однопоточный алгоритм уборки мусора	Для однопоточных виртуальных машин и контейнеров, для малого размера кучи (100 Мбайт)	«Последовательный уборщик мусора», с. 161
`-XX:+UseParallelGC`	Использует несколько потоков для уборки как в молодом, так и в старом поколении при остановленных потоках приложения	Используйте для настройки с расчетом на максимальную производительность вместо скорости отклика; используется по умолчанию в Java 8	«Параллельный уборщик мусора», с. 192
`-XX:+UseG1GC`	Использует несколько потоков для уборки в молодом поколении при остановленных потоках приложения, и фоновый поток(-и) для удаления мусора в старом поколении с минимальными паузами	Если доступны ресурсы процессора для фонового потока(-ов) и вы хотите избежать долгих пауз при уборке мусора. Используется по умолчанию в Java 11	«Уборщик мусора G1», с. 200
`-XX:+UseConcMarkSweepGC`	Использует фоновый поток(-и) для уборки мусора в старом поколении с минимальными паузами	Не рекомендуется; используйте G1	«Уборщик мусора CMS», с. 215
`-XX:+UseParNewGC`	Использует в CMS несколько потоков для уборки в молодом поколении при остановленных потоках приложения	Не рекомендуется; используйте G1	«Уборщик мусора CMS», с. 215
`-XX:+UseZGC`	Использует экспериментальный уборщик мусора Z (только в Java 12)	Чтобы сократить паузы при уборке в молодом поколении, которая выполняется в конкурентном режиме	«Конкурентное сжатие: ZGC и Shenandoah», с. 241
`-XX:+UseShenandoahGC`	Использует экспериментальный уборщик мусора Shenandoah (только в Java 12 OpenJDK)	Чтобы сократить паузы при уборке в молодом поколении, которая выполняется в конкурентном режиме	«Конкурентное сжатие: ZGC и Shenandoah», с. 241
`-XX:+UseEpsilonGC`	Использует экспериментальный уборщик мусора Epsilon (только в Java 12)	Если ваше приложение никогда не должно выполнять уборку мусора	«Фиктивный эпсилон-уборщик», с. 245

Таблица П.3. Флаги, общие для всех алгоритмов уборки мусора

Флаг	Назначение	Когда используется	См. также
`-Xms`	Задает исходный размер кучи	Если исходный размер по умолчанию слишком мал для вашего приложения	«Определение размера кучи», с. 175
`-Xmx`	Задает максимальный размер кучи	Если максимальный размер по умолчанию слишком мал (или слишком велик) для вашего приложения	«Определение размера кучи», с. 175
`-XX:NewRatio`	Задает соотношение между молодым и старым поколением	Увеличьте значение, чтобы сократить пропорцию относительно молодого поколения; уменьшите, чтобы увеличить пропорцию относительно молодого поколения. Значение определяет только начальное соотношение; пропорция изменится, если только адаптивное определение размера не отключено. С уменьшением размера молодого поколения уборка мусора в молодом поколении будет происходить чаще, а полная уборка — реже (и наоборот)	«Определение размеров поколений», с. 178
`-XX:NewSize`	Задает исходный размер молодого поколения	Если вы проводите точную настройку для требований своего приложения	«Определение размеров поколений», с. 178
`-XX:MaxNewSize`	Задает максимальный размер молодого поколения	Если вы проводите точную настройку для требований своего приложения	«Определение размеров поколений», с. 178
`-Xmn`	Задает исходный и максимальный размер молодого поколения	Если вы проводите точную настройку для требований своего приложения	«Определение размеров поколений», с. 178
`-XX:MetaspaceSize=`*`N`*	Задает исходный размер метапространства	В приложениях, использующих большое количество классов, увеличьте значение	«Определение размера метапространства», с. 181

Флаг	Назначение	Когда используется	См. также
`-XX:MaxMetaspaceSize=`*N*	Задает максимальный размер метапространства	Уменьшите, чтобы ограничить объем низкоуровневой памяти, используемой для метаданных классов	«Определение размера метапространства», с. 181
`-XX:ParallelGCThreads=`*N*	Задает количество потоков, используемых уборщиками мусора для операций переднего плана (например, уборки мусора в молодом поколении и уборки в старом поколении для параллельного уборщика)	Уменьшите это значение в системах с множеством JVM или в контейнерах Docker в Java 8 перед обновлением 192. Рассмотрите возможность увеличения для JVM с очень большими кучами в очень больших системах	«Управление параллелизмом», с. 184
`-XX:+UseAdaptiveSizePolicy`	При установке этого флага JVM будет изменять параметры размера кучи, пытаясь выполнить цели уборщика мусора	Отключите, если размеры кучи были точно настроены	«Адаптивное определение размеров», с. 180
`-XX:+PrintAdaptiveSizePolicy`	Включает в журнал уборки мусора информацию об изменении размеров поколений	Используйте этот флаг, чтобы понять, как работает JVM. При использовании G1 проверьте этот вывод, чтобы узнать, не вызвана ли полная уборка мусора выделением памяти для огромных объектов	«Адаптивное определение размеров», с. 180
`-XX:+PrintTenuringDistribution`	Включает в журнал уборки мусора информацию порога хранения	Используйте информацию для определения того, как следует отрегулировать параметры порога хранения	«Порог хранения и области выживших», с. 225
`-XX:InitialSurvivorRatio=`*N*	Задает долю молодого поколения, зарезервированную для областей выживших	Увеличьте, если объекты с коротким сроком жизни слишком часто переводятся в старое поколение	«Порог хранения и области выживших», с. 225
`-XX:MinSurvivorRatio=`*N*	Задает адаптивную долю молодого поколения, зарезервированную для областей выживших	Уменьшение этого значения сокращает максимальный размер областей выживших (и наоборот)	«Порог хранения и области выживших», с. 225

Флаг	Назначение	Когда используется	См. также
`-XX:TargetSurvivorRatio=`*N*	Доля свободной памяти, которую JVM пытается поддерживать в областях выживших	Увеличение этого значения сокращает размер областей выживших (и наоборот)	«Порог хранения и области выживших», с. 225
`-XX:InitialTenuringThreshold=`*N*	Исходное количество циклов уборки мусора, в которых JVM пытается удерживать объект в областях выживших	Увеличьте это значение, чтобы объекты дольше оставались в областях выживших (но учтите, что оно будет изменяться JVM)	«Порог хранения и области выживших», с. 225
`-XX:MaxTenuringThreshold=`*N*	Максимальное количество циклов уборки мусора, в которых JVM пытается удерживать объект в областях выживших	Увеличьте это значение, чтобы объекты дольше оставались в областях выживших; JVM будет настраивать фактический порог в диапазоне между этим значением и исходным порогом	«Порог хранения и области выживших», с. 225
`-XX:+DisableExplicitGC>`	Блокирует вызовы System.gc()	Используйте, чтобы запретить некорректно работающим приложениям явно выполнять уборку мусора	«Включение и выключение принудительной уборки мусора», с. 164
`-XX:-AggressiveHeap`	Активизирует набор флагов настройки, «оптимизированных» для машин с большим объемом памяти, на которых работает одна JVM с большой кучей	Использовать этот флаг не рекомендуется; лучше использовать отдельные флаги по мере надобности	«Флаг AggressiveHeap», с. 237

Таблица П.4. Флаги, управляющие журналом уборки мусора

Флаг	Назначение	Когда используется	См. также
`-Xlog:gc*`	Управляет журналом уборки мусора в Java 11	Протоколирование уборки мусора должно быть включено всегда, даже в рабочей версии. В отличие от предыдущего набора флагов Java 8, в Java 11 этот флаг управляет всеми параметрами журнала уборки мусора; соответствие между этими флагами и флагами Java 8 описано в тексте	«Инструменты уборки мусора», с. 185
`-verbose:gc`	Включает базовое протоколирование уборки мусора в Java 8	Протоколирование уборки мусора должно быть включено всегда, но обычно стоит использовать другие, более подробные журналы	«Инструменты уборки мусора», с. 185
`-Xloggc:<path>`	В Java 8 направляет журнал уборки мусора в специальный файл вместо стандартного вывода	Информацию, содержащуюся в журнале, всегда лучше сохранять на будущее	«Инструменты уборки мусора», с. 185
`-XX:+PrintGC`	Включает базовое протоколирование уборки мусора в Java 8	Протоколирование уборки мусора должно быть включено всегда, но обычно стоит использовать другие, более подробные журналы	«Инструменты уборки мусора», с. 185
`-XX:+PrintGCDetails`	Включает подробное протоколирование уборки мусора в Java 8	Всегда, даже в рабочей версии (затраты на вывод журнала минимальны)	«Инструменты уборки мусора», с. 185
`-XX:+PrintGCTimeStamps`	Выводит относительную временную метку в каждой записи журнала уборки мусора в Java 8	Всегда, если только не включен вывод метки даты	«Инструменты уборки мусора», с. 185
`-XX:+PrintGCDateStamps`	Выводит метку даты в каждой записи журнала уборки мусора в Java 8	Требует чуть больших затрат по сравнению с временными метками, но может упростить обработку	«Инструменты уборки мусора», с. 185

Флаг	Назначение	Когда используется	См. также
`-XX:+PrintReferenceGC`	Выводит информацию об обработке мягких и слабых ссылок в ходе уборки мусора в Java 8	Если в программе используется много таких ссылок, добавьте этот флаг для определения их влияния на затраты уборки мусора	«Мягкие, слабые и другие ссылки», с. 280
`-XX:+UseGCLogFileRotation`	Включает ротацию журнала уборки мусора для экономии файлового пространства в Java 8	В рабочих системах, которые работают неделями, можно ожидать, что журналы уборки мусора будут занимать много места	«Инструменты уборки мусора», с. 185
`-XX:NumberOfGCLogFiles=`*N*	При включении ротации журналов в Java 8 обозначает количество хранимых журнальных файлов	В рабочих системах, которые работают неделями, можно ожидать, что журналы уборки мусора будут занимать много места	«Инструменты уборки мусора», с. 185
`-XX:GCLogFileSize=`*N*	При включении ротации журналов в Java 8 определяет размер каждого журнального файла перед ротацией	В рабочих системах, которые работают неделями, можно ожидать, что журналы уборки мусора будут занимать много места	«Инструменты уборки мусора», с. 185

Таблица П.5. Флаги параллельного уборщика мусора

Флаг	Назначение	Когда используется	См. также
`-XX:MaxGCPauseMillis=`*N*	Рекомендации параллельному уборщику мусора по продолжительности пауз; размер кучи динамически изменяется в попытках достижения этой цели	Как первый шаг настройки параллельного уборщика мусора, если вычисленный размер по умолчанию не удовлетворяет целям приложения	«Настройка адаптивного и статического определения размеров кучи», с. 196
`-XX:GCTimeRatio=`*N*	Рекомендации параллельному уборщику мусора по времени, которое должно проходить за уборкой мусора; размер кучи динамически изменяется в попытках достижения этой цели	Как первый шаг настройки параллельного уборщика мусора, если вычисленный размер по умолчанию не удовлетворяет целям приложения	«Настройка адаптивного и статического определения размеров кучи», с. 196

Таблица П.6. Флаги уборщика мусора G1

Флаг	Назначение	Когда используется	См. также
`-XX:MaxGCPauseMillis=N`	Рекомендации уборщику мусора G1 по продолжительности пауз; алгоритм G1 автоматически корректируется в попытках достижения этой цели	Как первый шаг настройки уборщика мусора G1; увеличьте значение, чтобы предотвратить полную уборку мусора	«Настройка уборщика мусора G1», с. 211
`-XX:ConcGCThreads=N`	Задает количество потоков, используемых для фонового сканирования G1	Доступны значительные ресурсы процессора, а уборщик мусора G1 испытывает сбои конкурентного режима	«Настройка уборщика мусора G1», с. 211
`-XX:InitiatingHeapOccupancyPercent=N`	Задает точку, в которой начинается фоновое сканирование G1	Уменьшите это значение, если уборщик G1 испытывает сбои конкурентного режима	«Настройка уборщика мусора G1», с. 211
`-XX:G1MixedGCCountTarget=N`	Задает количество смешанных уборок мусора, в которых G1 пытается освободить области, ранее идентифицированные как содержащие в основном мусор	Уменьшите это значение, если уборщик G1 испытывает сбои конкурентного режима; увеличьте его, если смешанные циклы уборки мусора занимают слишком много времени	«Настройка уборщика мусора G1», с. 211
`-XX:G1MixedGCCountTarget=N`	Задает количество смешанных уборок мусора, в которых G1 пытается освободить области, ранее идентифицированные как содержащие в основном мусор	Уменьшите это значение, если уборщик G1 испытывает сбои конкурентного режима; увеличьте его, если смешанные циклы уборки мусора занимают слишком много времени	«Настройка уборщика мусора G1», с. 211
`-XX:G1HeapRegionSize=N`	Задает размер области G1	Увеличьте это значение для очень больших куч, или если приложение создает очень, очень большие объекты	«Размеры областей уборщика G1», с. 235
`-XX:+UseStringDeduplication`	Позволяет G1 устранять дубликаты строк	Используйте в программах с множеством дубликатов строк, если интернирование неприемлемо по практическим соображениям	«Дублирование и интернирование строк», с. 429

Таблица П.7. Флаги уборщика мусора CMS

Флаг	Назначение	Когда используется	См. также
`-XX:CMSInitiating OccupancyFraction=N`	Определяет, когда уборщик CMS должен начать фоновое сканирование старого поколения	Если CMS сталкивается со сбоями конкурентного режима, уменьшите это значение	«Уборщик мусора CMS», с. 215
`-XX:+UseCMSInitiat ingOccupancyOnly`	Заставляет CMS использовать только `CMSInitiating OccupancyFraction` для определения того, когда следует начать фоновое сканирование CMS	Всегда, когда задается `CMSInitiating OccupancyFraction`	«Уборщик мусора CMS», с. 215
`-XX:ConcGCThreads=N`	Задает количество потоков, используемых для фонового сканирования CMS	Если доступны значительные ресурсы процессора, а CMS сталкивается со сбоями конкурентного режима	«Уборщик мусора CMS», с. 215
`-XX:+CMSIncremental Mode`	Запускает CMS в инкрементном режиме	Более не поддерживается	–

Таблица П.8. Флаги управления памятью

Флаг	Назначение	Когда используется	См. также
`-XX:+HeapDumpOnOutOf MemoryError`	Генерирует дамп кучи, когда JVM выдает ошибку нехватки памяти	Установите этот флаг, если приложение выдает ошибку нехватки памяти в куче или постоянном поколении, чтобы кучу можно было проанализировать на утечку памяти	«Ошибки нехватки памяти», с. 255
`-XX:HeapDumpPath=<path>`	Задает имя файла, в котором должны сохраняться автоматические дампы кучи	Чтобы задать путь, отличный от `java_pid<pid>.hprof`, для дампов кучи, генерируемых для ошибок нехватки памяти или событий уборки мусора (при включении этих параметров)	«Ошибки нехватки памяти», с. 255
`-XX:GCTimeLimit=<N>`	Задает время, которое JVM может провести за выполнением уборки мусора, без выдачи исключения `OutOfMemoryException`	Уменьшите это значение, чтобы JVM быстрее выдавала исключение нехватки памяти, если в программе выполняется слишком много циклов уборки мусора	«Ошибки нехватки памяти», с. 255

Флаг	Назначение	Когда используется	См. также
`-XX:HeapFreeLimit=<N>`	Задает объем памяти, которую JVM должна освободить для предотвращения `OutOfMemoryException`	Уменьшите это значение, чтобы JVM быстрее выдавала исключение нехватки памяти, если в программе выполняется слишком много циклов уборки мусора	«Ошибки нехватки памяти», с. 255
`-XX:SoftRefLRUPolicy MSPerMB=N`	Управляет длительностью существования мягких ссылок после использования	Уменьшите это значение, чтобы мягкие ссылки быстрее освобождались, особенно в условиях нехватки памяти	«Мягкие, слабые и другие ссылки», с. 280
`-XX:MaxDirectMemorySize=N`	Управляет объемом памяти, который может выделяться методом `allocateDirect()` класса `ByteBuffer`	Рассмотрите возможность настройки этого параметра, если вы хотите ограничить объем памяти, которая может выделяться программой. В новых версиях уже не требуется устанавливать этот флаг для выделения более 64 Мбайт памяти	«Низкоуровневые буферы NIO», с. 311
`-XX:+UseLargePages`	Приказывает JVM выделять страницы из подсистемы больших страниц ОС, если она поддерживается	Если большие страницы поддерживаются ОС, этот параметр обычно повышает производительность	«Большие страницы», с. 313
`-XX:+StringTableSize=N`	Задает размер хеш-таблицы, используемой JVM для хранения интернированных строк	Увеличьте это значение, если приложение часто выполняет интернирование строк	«Дублирование и интернирование строк», с. 429
`-XX:+UseCompressedOops`	Эмулирует 35-разрядные указатели для ссылок на объекты	Используется по умолчанию для куч, размер которых меньше 32 Гбайт; отключение не дает никаких преимуществ	«Сжатые OOP», с. 296
`-XX:+PrintTLAB`	Выводит сводную информацию о TLAB в журнале уборки мусора	Если вы используете JVM без поддержки JFR, этот флаг поможет вам проверить правильность выделения TLAB	«TLAB», с. 230
`-XX:TLABSize=N`	Задает размер TLAB	Если приложение выполняет значительное выделение памяти за пределами TLAB, используйте это значение для увеличения размера TLAB	«TLAB», с. 230
`-XX:-ResizeTLAB`	Запрещает изменение размеров TLAB	Если флаг TLABSize установлен, этот флаг необходимо сбросить	«TLAB», с. 230

Таблица П.9. Флаги контроля за низкоуровневой памятью

Флаг	Назначение	Когда используется	См. также
`-XX:NativeMemoryTracking=X`	Включает контроль низкоуровневой памяти	Если вы хотите знать, какую память JVM использует за пределами кучи	«Контроль за низкоуровневой памятью», с. 304
`-XX:+PrintNMTStatistics`	Выводит статистику контроля за низкоуровневой памятью при завершении программы	Если вы хотите знать, какую память JVM использует за пределами кучи	«Контроль за низкоуровневой памятью», с. 304

Таблица П.10. Флаги работы с потоками

Флаг	Назначение	Когда используется	См. также
`-Xss<N>`	Задает размер низкоуровневого стека для потоков	Уменьшите этот размер, чтобы освободить память для других частей JVM	«Настройка размеров стеков потоков», с. 357
`-XX:-BiasedLocking`	Отключает алгоритм смещенной блокировки для JVM	Может улучшить производительность приложений, использующих пулы потоков	«Смещенная блокировка», с. 358

Таблица П.11. Разные флаги JVM

Флаг	Назначение	Когда используется	См. также
`-XX:+CompactStrings`	По возможности использует 8-разрядные представления строк (только в Java 11)	По умолчанию; используйте всегда	«Компактные строки», с. 428
`-XX:-StackTraceInThrowable`	Запрещает собирать данные трассировки стека при выдаче исключений	В системах с очень глубокими стеками при частой выдаче исключений (если отсутствует возможность исправления кода для сокращения числа исключений)	«Исключения», с. 454
`-Xshare`	Управляет совместным использованием данных классов	Флаг используется для создания новых архивов CDS для кода приложений	«Совместное использование данных классов», с. 444

Таблица П.12. Флаги Java Flight Recorder

Флаг	Назначение	Когда используется	См. также
`-XX:+FlightRecorder`	Включает Java Flight Recorder	Механизм Flight Recorder рекомендуется включать всегда, потому что он обходится почти без затрат, кроме времени фактического сохранения данных (в этом случае затраты зависят от используемой функциональности, но все равно будут относительно небольшими)	«Java Flight Recorder», с. 102
`-XX:+FlightRecorderOptions`	Задает параметры сохранения данных по умолчанию из командной строки (только в Java 8)	Управляет параметрами сохранения данных для JVM по умолчанию	«Java Flight Recorder», с. 102
`-XX:+StartFlightRecorder`	Запускает JVM с заданными параметрами Java Flight Recorder	Управляет параметрами сохранения данных для JVM по умолчанию	«Java Flight Recorder», с. 102
`-XX:+UnlockCommercialFeatures`	Позволяет JVM использовать коммерческие средства (не относящиеся к открытому коду)	Если у вас имеется соответствующая лицензия, установка этого флага необходима для включения Java Flight Recorder в Java 8	«Java Flight Recorder», с. 102

Скотт Оукс

Эффективный Java. Тюнинг кода на Java 8, 11 и дальше

Перевели с английского Е. Матвеев, И. Сигайлюк

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Петруненко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, М. Молчанова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.01.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 39,990. Тираж 400. Заказ 0000.