Высокопроизводительный код на платформе .NET

Бен Уотсон

2-е издание

Writing High-Performance .NET Code

by Ben Watson

Высоко-производительный код на платформе .NET

Бен Уотсон

ПИТЕР®

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

2019

ББК 32.988.02-018
УДК 004.738.5
У65

Уотсон Бен

У65 Высокопроизводительный код на платформе .NET. 2-е изд. — СПб.: Питер, 2019. — 416 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0911-1

Хотите выжать из вашего кода на .NET максимум производительности? Эта книга развеивает мифы о CLR, рассказывает, как писать код, который будет просто летать. Воспользуйтесь ценнейшим опытом специалиста, участвовавшего в разработке одной из крупнейших .NET-систем в мире.

В этом издании перечислены все достижения и улучшения, внесенные в .NET за последние несколько лет, в нем также значительно расширен охват инструментов, содержатся дополнительные темы и руководства.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-0990583455 англ.
ISBN 978-5-4461-0911-1

Краткое содержание

Оглавление

Предисловие

Молодежь не понимает, как ей повезло! Рискуя прослыть старым ворчуном, должен заметить, что это не пустое утверждение, по крайней мере по отношению к анализу производительности. Самым очевидным доказательством является то, что в мои времена не было подобных книг, охватывающих сразу и важные основополагающие принципы анализа производительности, и практические сложности, с которыми сталкиваешься в реальном мире. Эта книга — золотая жила, и ее стоит не только прочитать, но и постоянно перечитывать в процессе работы над повышением производительности.

Уже более десяти лет я работаю архитектором производительности .NET Runtime. Проще говоря, моя задача — убедить людей, использующих C# и среду выполнения кода .NET, что их вполне устраивает производительность созданного ими кода. Часть этой задачи — поиск мест внутри .NET Runtime или библиотек этой среды, работающих неэффективно, и внесение в них исправлений, но это не самое трудное из того, чем приходится заниматься. Сложнее всего то, что в 90 % случаев производительность приложений определяется не особенностями реализации среды выполнения (например, качеством генерации кода, компиляцией ко времени применения, сборкой мусора или функционированием библиотек классов), а тем, что находится в ведении разработчика приложения (например, архитектурой приложения, выбором структур данных и алгоритмов и просто ошибками в коде). Таким образом, моя работа куда больше связана с *обучением*, чем с *программированием*.

Значительная часть работы заключается в проведении бесед и написании статей, и в основном я консультирую другие команды, которым нужна помощь в ускорении работы их программ. Именно в такой роли я шесть лет назад впервые встретился с Беном Уотсоном. Он был тем самым представителем Bing-команды, который всегда задавал необычные вопросы (и находил ошибки в нашем коде, а не в своем). Бен явно был нашим, из разряда борцов за высокую производительность. Вы не представляете, насколько это редкое явление. Наверное, 80 % программистов пройдут основную часть своего карьерного пути, имея весьма смутное представление о производительности создаваемого ими кода. Возможно, 10 % проявят достаточное внимание к вопросам производительности по мере освоения работы со средствами анализа, подобными профилировщикам. Тот факт, что вы читаете эту книгу (и это предисловие!), говорит о том, что вы относитесь

к небольшой группе людей, кто реально радеет за высокую производительность и действительно хочет, чтобы она постоянно повышалась. Бен идет еще дальше: его не только интересует все, что связано с производительностью, но он также заботится о ее достижении настолько серьезно, что нашел время для написания этой книги. Он относится к тем специалистам, которые составляют всего 0,0001 % от их общего количества. Вам предоставлена возможность учиться у самых лучших профессионалов.

Это весьма важная книга. На своем веку я сталкивался с множеством проблем, связанных с производительностью, и, как уже упоминалось, в 90 % случаев они возникали в самом приложении. Это означает, что решение таких проблем в *ваших* руках. В качестве предисловия к некоторым моим выступлениям о производительности я часто привожу следующую аналогию: представьте, что вы уже написали 10 000 строк нового кода для какого-то приложения и вам даже удалось этот код скомпилировать, но само приложение вы еще не запустили. Какова вероятность того, что код не содержит ошибок? Основная часть аудитории скажет: ноль. И будет совершенно права. *Все*, кто когда-либо занимался программированием, знают, что обрести уверенность в стабильной работе программы можно, лишь потратив много времени на эксплуатацию приложения и устранение проблем. Программирование — занятие *непростое*, и нужный результат получается путем последовательной доработки кода. Итак, представьте теперь, что вы потратили время на отладку программы в 10 000 строк, после чего она, казалось бы, заработала как надо. Но перед вашим приложением стоит весьма непростая цель — достичь высокой производительности. Какова вероятность того, что у него нет проблем с *производительностью*? Программисты — народ неглупый, следовательно, вы быстро поймете, что она стремится к нулю. Точно так же, как компилятор не в состоянии выявить множество проблем времени выполнения, обычное функциональное тестирование не позволяет отследить множество вопросов, связанных с производительностью. Поэтому *определенный* объем знаний о производительности необходим всем, что и предоставляет данная книга.

Еще одна тревожная истина, касающаяся производительности, гласит: самые трудноустранимые просчеты закладываются в приложение на ранней стадии проектирования. Дело в том, что именно тогда выбирается базовое представление обрабатываемых данных, и оно накладывает жесткие ограничения на производительность. Я сбился со счета, сколько раз приходилось консультировать людей, выбравших неудачное представление (например, XML, или JSON, или базу данных) для данных, играющих важную роль в достижении их приложениями высокой производительности. Они обращались ко мне за помощью на слишком позднем этапе производственного цикла, надеясь, что я сотворю чудо, устранив возникшие у них проблемы с производительностью. Разумеется, я помогал им измерить ее и обычно находил что-то, что можно исправить, но добиться существенных успехов это не позволяло, поскольку требовалось изменить базовое представление, что в конце производственного цикла было слишком затратно и рискованно. В результате конечный продукт работал далеко не так быстро, как мог бы, если бы проблемы с производительностью осознали своевременно.

Как же не дать этому произойти при разработке *ваших* приложений? У меня есть два простых правила написания высокопроизводительных приложений, которые совсем не случайно повторяют правила Бена.

1. Иметь план достижения высокой производительности.
2. Постоянно проводить измерения, измерения и еще раз измерения.

В реальности пункт «Иметь план достижения высокой производительности» сводится к стремлению серьезно относиться к вопросам производительности. Это означает, что нужно определить, какие метрики вы будете использовать (обычно это затраченное время, иногда что-то другое), и выявить основные операции, потребляющие наибольшее число ресурсов, охватываемых этой метрикой (обычно это операции с большими объемами данных, на выполнение которых, вероятно, уйдет значительная часть времени работы приложения). На самой ранней стадии проекта — перед тем как принять любое серьезное проектировочное решение — следует подумать о том, *какой* производительности предполагается достичь, и измерить этот показатель для какого-либо кода, например созданных ранее аналогичных приложений или прототипов вашего решения. Это или придаст вам уверенности в достижимости поставленных целей, или позволит понять, что добиться желаемого может оказаться нелегко и для поиска более удачного решения понадобятся глубже проработанные прототипы и проведение более тонких экспериментов. Я не говорю о каких-то космических технологиях. На реализацию некоторых планов повышения производительности уходят буквально минуты. Главное — проделать это на самой ранней стадии проектирования, чтобы планы достижения высокой производительности могли повлиять на принятие проектировочных решений, подобных форме представления данных, в самом начале разработки.

Пункт «Постоянно проводить измерения, измерения и еще раз измерения» просто подчеркивает, на что вы будете затрачивать основную часть времени (наряду с толкованием результатов). Как сказал бы Аластор Муди по прозвищу Бешеный Глаз (Alastor «Mad-Eye» Moody), нам нужна постоянная бдительность. Производительность может быть утрачена практически на *любом* этапе производственного цикла, от проектирования до поддержки готового продукта, и предотвратить это можно, только выполняя все новые и новые измерения, не позволяющие выбиться из колеи. Повторюсь: здесь не нужны космические технологии, должно быть просто желание делать это постоянно, предпочтительно автоматизировав необходимые действия.

Это ведь нетрудно, правда? Но есть загвоздка. *Как правило*, программы сложны и запускаются на непростом оборудовании с множеством абстракций (например, с использованием кэшей памяти, операционных систем, сред выполнения, сборщиков мусора и т. д.), и совсем не удивительно, что обеспечение высокой производительности в таких сложных обстоятельствах дается непросто. В деле ее повышения *может быть* множество важных деталей. Например, что делать с ошибками и как поступать при возникновении конфликтующих или, что бывает чаще, слишком изменчивых результатов измерений. Параллелизм — великолепный

способ повышения производительности многих приложений — также сильно усложняет анализ производительности и зависит от таких деталей, как диспетчеризация центрального процессора, что ранее никогда не бралось в расчет. В итоге задача обеспечения производительности становится чем-то вроде многослойной луковицы, которая все больше усложняется, когда снимают слои.

Ценность данной книги в том, что она поможет справиться с этой сложностью. Обеспечение производительности может показаться неподъемной задачей. Ведь столь многое может быть измерено и существует такое разнообразие инструментов для проведения этих измерений, что зачастую не вполне понятно, какие именно измерения наиболее полезны и как правильно соотнести их друг с другом. Для начала эта книга поможет с решением базовой задачи — обозначением *целей, важных именно для вас*. Вас также снабдят небольшим набором инструментов и метрик, ценность которых проверена временем, что позволит вам двигаться дальше в верном направлении. На этой прочной основе в книге начинается «раздевание луковицы», позволяющее вникнуть в детали рассматриваемых тем, играющих важную роль при решении проблем производительности для целого ряда приложений. К ним относятся управление памятью (сборка мусора), компиляция на лету (just in time, JIT) и асинхронное программирование. Таким образом, вы получите все необходимые детали (среды выполнения весьма сложны, и иногда эта сложность проявляется и серьезно влияет на производительность), к тому же в общей структуре, позволяющей связывать эти детали с тем, что представляет для вас истинный интерес, — с целями именно вашего приложения.

Теперь дадим слово Бену, и он умело растолкует вам все остальное. Мне же хотелось не просветить вас, а мотивировать на чтение книги. Исследование вопросов повышения производительности — весьма сложная область компьютерной науки, и без того непростой. Чтобы набраться опыта, нужны время и настойчивость в достижении поставленной цели. Я обращаюсь к читателям не для приукрашивания действительности, а чтобы сказать, что дело того стоит. Производительность *весьма* важна. Я могу практически гарантировать, что при широком использовании вашего приложения его производительность *будет* важна. Учитывая важность данного вопроса, можно считать чуть ли не преступлением то, что крайне мало сведущих в нем специалистов, обладающих достаточным мастерством для создания высокопроизводительных приложений на системной основе. Теперь вы читаете эти строки, чтобы влиться в ряды элитной группы нашего сообщества. И эта книга *существенно* упростит решение данной задачи.

Молодежь не понимает, как ей повезло!

Вэнс Моррисон (Vance Morrison),
Performance Architect
среды выполнения .NET, Microsoft

Об авторе

Бен Уотсон (Ben Watson) с 2008 года является программистом компании Microsoft. В команде, работающей над платформой Bing, он создал на основе платформы .NET высокопроизводительные серверные приложения, которые способны обрабатывать большие объемы информации и отличаются высокой скоростью реакции на запросы от тысяч машин и миллионов клиентов. Эти приложения можно отнести к самым совершенным в мире. На досуге Бен читает, слушает музыку, гуляет и общается с женой Летицией и детьми Эммой и Мэтью. Они живут недалеко от Сиэтла (штат Вашингтон, США).

Контактная информация

Бен Уотсон (Ben Watson)

Электронная почта: feedback@writinghighperf.net.

Веб-сайт: http://www.writinghighperf.net.

Блог: http://www.philosophicalgeek.com.

LinkedIn: https://www.linkedin.com/in/benmwatson.

Twitter: https://twitter.com/benmwatson.

Если вам понравится эта книга, пожалуйста, оставьте свой отзыв в вашем любимом интернет-магазине. Спасибо!

Благодарности

Спасибо моей жене Летиции и нашим детям Эмме и Мэтью за их терпение, любовь и поддержку в тот период, когда я тратил больше времени не на них, а на подготовку второго издания этой книги. Летиция вложила немало труда в редактирование и корректуру, придав книге ту последовательность изложения, которой не было бы, не участвуй она в работе.

Спасибо Клэр Уотсон (Claire Watson) за разработку красивой обложки для обоих изданий книги.

Я благодарен своему наставнику Майку Магрудеру (Mike Magruder), который, наверное, чаще, чем кто-либо другой, читал эту книгу. Он был техническим редактором первого издания, а уже будучи на пенсии, нашел время, чтобы вернуться к подробностям устройства среды .NET во втором издании.

Спасибо читателям предварительной версии книги, чье неоценимое внимание к ее формулировкам и темам позволило обнаружить опечатки, мои упущения и многие другие недочеты. Это Абхинав Джайн (Abhinav Jain), Майк Магрудер (Mike Magruder), Чад Парри (Chad Parry), Брайан Расмуссен (Brian Rasmussen) и Мэтт Уоррен (Matt Warren). Именно благодаря им книга стала еще лучше.

Благодарю Вэнса Моррисона (Vance Morrison), прочитавшего раннюю версию книги и написавшего замечательное предисловие к этому изданию.

И наконец, я благодарен всем читателям первого издания, чьи отзывы помогли сделать второе издание лучше.

От издательства

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

Мы будем рады узнать ваше мнение!

На сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Введение во второе издание

Со времени выхода первого издания книги *Writing High-Performance .NET Code* основы достижения высокой производительности в среде .NET особых изменений не претерпели. Правила оптимизации сборки мусора остались практически неизменными. JIT-компиляция, прибавив в производительности, в основном сохранила прежнее поведение. Но за это время вышло уже пять доработанных версий среды .NET, и они заслуживают упоминания там, где это уместно.

За прошедшие годы книга подверглась существенной переработке. В дополнение к рассмотрению новых функциональных возможностей, появившихся в среде .NET, в ней заполнены досадные пробелы, существовавшие в первом издании. Практически каждый раздел книги был тем или иным образом переработан: либо незначительно подкорректирован, либо чуть ли не переписан заново с включением новых примеров, материалов или объяснений. Изменений так много, что их полный список занял бы слишком много места, но несколько основных стоит перечислить.

- Объем увеличился на 50 %.
- Учтены отзывы сотен читателей.
- Появилось новое предисловие от Performance Architect .NET Вэнса Моррисона.
- Приведены десятки новых примеров и фрагментов учебного кода.
- Обновлены схемы и графики.
- Добавлен перечень улучшений производительности CLR, выполненных за это время.
- Рассмотрено большее количество инструментальных аналитических средств.
- Значительно расширена область использования Visual Studio для анализа производительности среды .NET.
- Добавлены многочисленные примеры анализа с использованием набора API Microsoft.Diagnostics.Runtime (CLR MD).
- В некоторые учебные проекты добавлен материал по эталонному тестированию (бенчмаркингу) и применению популярных сред такого тестирования.
- Появились новые разделы, посвященные особенностям CLR и .NET Framework, имеющим отношение к производительности.

- Больше внимания уделено сборке мусора, добавлена новая информация о составлении пула, `stack-alloc`, финализации, слабых ссылках, поиске утечек памяти и многом другом.
- Подробнее рассмотрены различные технологии предварительной подготовки кода.
- Добавлена информация о TPL и новый раздел о TPL Dataflow.
- Рассмотрены возвращения по ссылке и локальные переменные.
- Гораздо подробнее рассмотрены коллекции, включая исходный объем, сортировку и сравнение ключей.
- Дан подробный анализ затрат на LINQ.
- Приведены примеры SIMD-алгоритмов.
- Показаны приемы создания автоматизированных анализаторов кода и средств его исправления.
- Добавлено приложение, в котором даются советы по увеличению производительности на более высоком уровне для ADO.NET, ASP.NET и WPF.

...И многое другое!

Уверен, что даже тем, кто прочитал первое издание, стоит найти время и ознакомиться со вторым.

Введение

Цель этой книги

.NET — великолепная система для разработки программных продуктов. Она позволяет создавать функциональные взаимосвязанные приложения значительно быстрее, чем несколько лет назад. Многие из них исправно работают, и это здорово. Данная среда предлагает приложениям безопасность памяти и типов, мощную библиотеку функционала «из коробки», такие сервисы, как автоматическое выделение памяти, и многое другое.

Программы, написанные с помощью среды .NET, называются управляемыми приложениями, поскольку они зависят от среды выполнения и той среды, которая управляет многими жизненно важными для них задачами, обеспечивая основу безопасности операционного окружения. В отличие от неуправляемого или машинного кода, программ, написанных с непосредственным обращением к API операционной системы, управляемые приложения не могут свободно управлять своими процессами.

Эта управляющая прослойка между вашей программой и процессором компьютера может стать источником беспокойства для разработчиков, предполагающих, что она способна существенно повысить издержки. Эта книга поможет развеять тревоги, показав, что полученный результат оправдывает издержки и что предполагаемое снижение производительности практически всегда преувеличивают. Зачастую проблемы с производительностью, в которых разработчики обвиняют среду .NET, на самом деле связаны с неудачными шаблонами проектирования и недостатком знаний об оптимизации программ под эту среду. Годами наработанные навыки оптимизации программ, написанных на C++, Java или Python, не всегда применимы к управляемому коду в среде .NET, и некоторые советы на поверку оказываются вредными. Иногда ускоренная разработка, допускаемая средой .NET, может побудить к более быстрому, чем прежде, созданию раздутого, медленного, плохо оптимизированного кода. Разумеется, существуют и иные причины низкого качества кода: общий недостаток мастерства, дефицит времени, неудачное проектирование, недостаточный объем человеческих ресурсов в разработке, лень и т. д. Эта книга, несомненно, избавит от использования пробелов в знании среды в качестве оправдания, а также попробует помочь разобраться и с другими при-

чинами неудач. Применяя изложенные в ней принципы, вы научитесь создавать гибкие, быстрые и эффективные приложения. Истина одна для всех типов кода и на всех платформах: если требуется высокопроизводительный код, над этим нужно работать.

Работу над повышением производительности нельзя оставлять напоследок, особенно в макро- или архитектурном смысле. Чем больше и сложнее ваше приложение, тем на более ранней стадии следует уделять внимание производительности как его главной составляющей.

Я часто привожу как пример построение хижины в сравнении с возведением небоскреба. Если создается хижина, неважно, что именно захочется переделать в будущем. Хотите окно? Просто прорубите дыру в стене. Хотите провести электричество? Прикрутите провода. Вы абсолютно свободны, придавая строению завершенный вид, поскольку здесь все просто и почти ничто не зависит друг от друга.

А вот с небоскребом другая картина. Нельзя принять решение о применении стальных балок, после того как первые пять этажей уже построены из дерева. Следует разобраться с требованиями, а также характеристиками строительных материалов, прежде чем приступать к сооружению из них какой-то крупной конструкции.

Эта книга посвящена главным образом тому, чтобы довести до вас мысль о затратах на строительные материалы и получаемой от них выгоде, из чего вы должны извлечь уроки, применимые к любому создаваемому проекту.

Это не справочник по языку или руководство. И даже не подробное рассмотрение CLR. Эти темы раскрываются в других источниках (список заслуживающих внимание книг, блогов и статей приведен в конце издания). Чтобы получить от этой книги максимальную отдачу, вы должны обладать достаточным опытом работы в среде .NET.

Здесь приводится множество примеров кода, особенно основных деталей реализации на уровне IL или кода ассемблера. Я призываю не игнорировать эти разделы. Вам следует попробовать воспроизвести мои результаты, прорабатывая материалы книги, чтобы лучше понять происходящее.

Эта книга научит вас добиваться максимальной производительности управляемого кода, в идеале не жертвуя ни одним из преимуществ среды .NET, а в худшем случае жертвуя минимальным их числом. Вы освоите рациональные методы программирования, узнаете, чего следует избегать и, что, наверное, наиболее важно, как использовать инструментальные средства, находящиеся в свободном доступе, чтобы без особых затруднений измерить уровень производительности. В учебном материале будет минимум воды — только самое необходимое. В книге дается именно то, что нужно знать, она актуальна и лаконична, не содержит лишнего. Большинство глав начинается с общих сведений и предыстории, за которыми следуют конкретные советы, изложенные наподобие рецепта, а в конце — раздел проведения пошаговых измерений и отладки для множества разнообразных сценариев.

Попутно погрузимся в конкретные составляющие среды .NET, в частности в положенную в ее основу общеязыковую среду выполнения (Common Language Runtime (CLR)), и увидим, как происходит управление памятью вашей машины,

генерируется код, организуется многопоточное выполнение и делается многое другое. Вам будет показано, как архитектура .NET одновременно и ограничивает ваше программное средство, и предоставляет ему дополнительные возможности и как выбор путей программирования может существенно повлиять на общую производительность приложения. В качестве бонуса я поделюсь с вами историями из опыта создания в течение последних девяти лет очень крупных, сложных, высокопроизводительных .NET-систем в компании Microsoft. Скорее всего, вы заметите, что в этой книге я в основном обращаю внимание на серверные приложения, но практически все рассматриваемое в ней применимо и к приложениям для настольных систем, а также к веб- и мобильным приложениям. В нужных местах я поделюсь советами именно для этих конкретных платформ.

Понимание основ позволит вам осознать, почему приведенные советы по повышению производительности имеют смысл и работают. Вы станете разбираться в среде .NET и принципах создания высокопроизводительного кода настолько, что, столкнувшись с обстоятельствами, не нашедшими отражения в данной книге, сможете применить приобретенные знания и решить непредвиденные проблемы.

Программирование под управлением среды .NET не является чем-то совершенно иным по сравнению с вашим общим опытом программирования. Вам попрежнему понадобится знание алгоритмов, и большинство стандартных конструкций программирования будут выглядеть почти так же, как и раньше. Но здесь речь пойдет об оптимизации с целью повышения производительности и придется наблюдать совершенно иные вещи, отличающиеся от тех, что вы видели при создании неуправляемого программного кода. Возможно, больше не придется явно вызывать `delete` (ура!), но если требуется добиться наивысшей производительности, лучше сразу поверить в необходимость разобраться в том, как сборщик мусора повлияет на работу приложения.

Если цель — высокая доступность, то вам придется в известной степени заинтересоваться JIT-компиляцией. Используется богатая система типов? Потребует внимания диспетчеризация интерфейсов. А как насчет API в самой библиотеке классов .NET Framework? Может ли какой-нибудь из них негативно повлиять на производительность? Имеются ли более подходящие механизмы синхронизации? Принимали ли вы во внимание локальность памяти при выборе коллекций или алгоритмов?

Помимо чистого программирования, будут рассмотрены методы и процессы, позволяющие измерять вашу производительность с течением времени и формировать культуру достижения высокой производительности, как собственной, так и всей команды. Нельзя добиться высокой производительности однократно, а потом о ней забыть. Ее нужно постоянно поддерживать и сохранять, чтобы со временем она не снизилась. Вклад в высокопроизводительную инфраструктуру в дальнейшем принесет огромные дивиденды, позволяя автоматизировать основную часть нелегкой работы.

Суть в том, что уровень оптимальности производительности вашего приложения прямо пропорционален степени понимания не только собственного кода, но

и среды его выполнения, операционной системы и оборудования. Это касается любой платформы, которую вы предполагаете использовать.

Все примеры кода в этой книге написаны на C#, базовом IL, а иногда на ассемблерном коде x86 или x64, но все приведенные здесь принципы применимы к любому языку среды .NET. В этой книге предполагается, что вы работаете с .NET 4.5 или более свежей версией, а в некоторых примерах задействуются новые функции, доступные в более поздних версиях. Я настоятельно рекомендую перейти на использование самой последней версии, чтобы можно было воспользоваться новейшими технологиями, функциями, исправлениями ошибок и улучшениями производительности.

В книге уделяется немного внимания конкретным подразделам .NET, таким как WPF, WCF, ASP.NET, Windows Forms, Entity Framework, ADO.NET, и многим другим. Хотя у каждой из этих технологий есть свои проблемы и приемы повышения производительности, издание посвящено фундаментальным сведениям и методам, которые следует усвоить, чтобы разрабатывать код, пригодный для всех сценариев в среде .NET. Как только вы овладеете основами, появится возможность применять приобретенные знания в каждом проекте, над которым ведется работа, по мере накопления опыта узнавая больше о конкретных областях. В конце книги приведено небольшое приложение, которое даст вам ряд начальных рекомендаций по оптимизации приложений ASP.NET, ADO.NET или WPF.

В целом, я надеюсь показать, что инженерная работа над производительностью — это точно такая же инженерная работа, как и любая другая. Производительность не получить «из коробки» на любой платформе, даже на .NET.

В чем смысл выбора управляемого кода

Для выбора управляемого кода вместо неуправляемого есть множество причин.

- Безопасность — компилятор и среда выполнения могут обеспечить безопасность типов (объекты используются только такими, какие они в действительности), проверку границ, обнаружение переполнения чисел, гарантии защиты от взлома и многое другое. Куча больше не повреждается из-за нарушений доступа или неверных указателей.
- Автоматическое управление памятью — больше не нужны функция `delete` или подсчет ссылок.
- Более высокий уровень абстракции — более высокая производительность с меньшим количеством ошибок.
- Расширенные возможности языка — делегаты, безымянные методы, динамическая типизация и многое другое.
- Весьма обширная база кода — Framework Class Library, Entity Framework, Windows Communication Framework, Windows Presentation Foundation, Task Parallel Library и многое другое.

- Упрощенная расширяемость — благодаря возможностям отражения (reflection) значительно упростилось динамическое использование модулей с поздней привязкой, например, в архитектуре расширений.
- Впечатляющая отладка — исключения несут в себе множество полезной информации. У всех объектов имеются связанные с ними метаданные, обеспечивающие в отладчике тщательный анализ кучи и стека, зачастую без необходимости использования PDB (файлов символов).

Все это означает, что вы можете написать больше кода и сделать это быстрее и с меньшим количеством ошибок. Существенно упростилась диагностика ошибок. Учитывая все эти преимущества, управляемый код должен стать вашим выбором по умолчанию.

.NET также подталкивает к применению стандартных средств единой платформы. В мире неуправляемого кода довольно легко получить фрагментированные среды разработки с несколькими используемыми платформами (например, STL, Boost или COM) или несколькими разновидностями интеллектуальных указателей. В .NET смысл наличия такого разношерстного инструментария практически исчезает.

Хотя конечная цель — получить такой код, который можно, один раз создав, запускать везде, — вероятно, так и останется несбыточной мечтой, но к ее реальному воплощению мы все же приближаемся. Есть три основных варианта переносимости.

1. Переносимые библиотеки классов позволяют ориентироваться на Рабочий стол Windows, магазин Windows Store и другие типы приложений с единой библиотекой классов. Не все API доступны на всех платформах, но их вполне достаточно, чтобы сэкономить значительные усилия на разработке.
2. .NET Core — переносимая версия .NET, способная работать в Windows, Linux и MacOS. Она может ориентироваться на стандартные ПК, мобильные устройства, серверы дата-центров или устройства Интернета вещей (IoT) с гибкой минимизированной средой выполнения .NET. Этот вариант стремительно набирает популярность.
3. Использование Xamarin (набора инструментов и библиотек), которое позволяет нацелиться на платформы Android, iOS, MacOS и Windows с помощью единой кодовой базы .NET.

С учетом тех огромных преимуществ, которые дает управляемый код, считайте, что необходимость применения неуправляемого кода потребует серьезных доказательств, если таковые вообще возможно предъявить. Получите ли вы от этого ожидаемое повышение производительности? Неужели сгенерированный код станет реальным ограничивающим фактором? Можно ли будет написать быстрый прототип и доказать его состоятельность? Можно ли обойтись без всех возможностей .NET? Может оказаться, что в сложном неуправляемом приложении вы самостоятельно реализуете некоторые из имеющихся в .NET функций. Вряд ли вам хочется оказаться в неловком положении, дублируя чью-то работу.

И тем не менее есть веские причины для признания непригодности кода .NET.

- Необходимость доступа к полному набору инструкций процессора, особенно для расширенных приложений обработки данных, применяющих инструкции SIMD. Но ситуация меняется. SIMD-программирование, доступное в среде .NET, рассматривается в главе 6.
- Существование большой базы неуправляемого кода. В этом случае можно рассмотреть возможность использования интерфейса между новым и старым кодом. Если получится легко разработать понятный API для этого, то подумайте о возможности написания всего нового функционала на управляемом коде и добавления уровня взаимодействия между ним и неуправляемым кодом. Со временем можно будет перевести неуправляемый код в управляемый.
- Перекликающаяся с предыдущим пунктом опора на платформозависимые библиотеки или API в коде. Например, до появления управляемой оболочки последние версии функций Windows зачастую будут доступны только в Windows SDK для C/C++. Управляемых оболочек для некоторых функций просто не существует.
- Взаимодействие с оборудованием. Некоторые аспекты взаимодействия с оборудованием будут упрощаться благодаря прямому доступу к памяти и другим функциям языков более низкого уровня. К таким случаям можно отнести расширенные возможности видеокарты для игр.
- Необходимость строгого контроля над структурами данных. Возможности контроля размещения структур в памяти в C/C++ гораздо шире, чем в C#.

Даже если некоторые из перечисленных пунктов — ваш случай, это еще не означает, что все ваше приложение должно состоять из неуправляемого кода. В одном и том же приложении можно легко сочетать два типа кода, взяв самое лучшее из обоих миров.

Работает ли управляемый код медленнее нативного?

В мире существует множество нелепых стереотипов. К сожалению, один из них — то, что управляемый код не может быть быстрым. Это не так.

Ближе к истине то, что платформа .NET позволяет с необычайной легкостью создавать медленный код, если относиться к работе безалаберно и безответственно.

При создании кода на C#, VB.NET или любом другом языке управляемого кода компилятор переводит код на языке высокого уровня в код на промежуточном языке — Intermediate Language (IL) и в метаданные об использованных типах. Запущенный код проходит компиляцию времени выполнения (just-in-time), то есть JIT-компиляцию. Таким образом, при первом выполнении метода среда CLR вызовет для IL-кода JIT-компилятор, чтобы преобразовать его в ассемблерный код (например, x86, x64, ARM). Основная часть оптимизации происходит на

этой стадии. Так что при первом запуске производительность страдает, но после этого вы всегда получаете скомпилированную версию. Позже будет показано, что при необходимости можно найти способы обойти снижение производительности на старте.

Таким образом, устойчивый уровень производительности управляемого приложения определяется двумя факторами:

- качеством JIT-компилятора;
- объемом издержек сервисов среды .NET.

Как правило, качество сгенерированного кода не вызывает (за некоторыми исключениями) никаких нареканий и постоянно улучшается, особенно в последнее время.

В некоторых же случаях можно увидеть существенные преимущества использования управляемого кода.

- *Выделение памяти.* В отличие от нативных приложений отсутствует конкуренция при выделении памяти в куче. Часть сэкономленного времени затрачивается на сборку мусора, но в зависимости от настроек приложения даже эти временные затраты можно практически нивелировать. Сведения о поведении и конфигурации сборки мусора изложены в главе 2.
- *Фрагментация.* Усугубляющаяся со временем фрагментация памяти — весьма распространенная проблема в больших и долго работающих нативных приложениях. В приложениях .NET эта проблема выражена намного слабее, поскольку куча в меньшей степени подвержена фрагментации в принципе, но даже если куча фрагментируется, сборщик мусора уплотнит ее.
- *JIT-компиляция кода.* Поскольку код проходит JIT-компиляцию во время выполнения, его расположение в памяти может быть более оптимальным, чем у нативного кода. Связанные участки кода часто будут располагаться в одном месте и с большой долей вероятности помещаться на одной странице памяти или в одной строке кэша процессора. Это уменьшает количество отказов страницы.

На вопрос «Работает ли управляемый код медленнее нативного?» в большинстве случаев нужно отвечать решительным «нет». Разумеется, есть такие области, где управляемый код просто не может преодолеть ограничения безопасности, под которыми работает. Их гораздо меньше, чем можно себе представить, и большинству приложений отказ от управляемого кода не принесет существенных выгод. Зачастую разница в производительности преувеличена. В действительности же состав оборудования и архитектура часто влияют сильнее, чем выбор языка и платформы.

Гораздо чаще встречается код, управляемый или нативный, который просто плохо написан. Он, например, не справляется должным образом с управлением памятью, использует неудачные шаблоны, игнорирует стратегии кэширования в центральном процессоре или не позволяет достичь высокой производительности по какой-то другой причине.

Стоит ли овчинка выделки?

Как обычно, у каждого варианта есть свои издержки и выгоды. В большинстве случаев на практике я обнаруживал, что преимущества управляемого кода перевешивают затраты. Более того, подойдя к программированию с умом, обычно удается избежать наибольших затрат и при этом получить преимущества.

Сервисы, предоставляемые средой .NET, не бесплатны с точки зрения производительности, но затраты ниже ожидаемых. Их не нужно сводить к нулю (собственно, это и не получится), просто сделайте их настолько низкими, чтобы они не превышали значимость влияния других факторов в профиле производительности вашего приложения.

Функциональная особенность	**Преимущества**
Код, проходящий JIT-компиляцию	Лучшая локализация в памяти, ее пониженное потребление
Проверка границ	Безопасное обращение к памяти (меньше трудно обнаруживаемых ошибок)
Затраты на хранение метаданных типа	Более простая отладка, насыщенные метаданные, отражение, улучшенная обработка исключений, упрощение статического анализа
Сборка мусора	Быстрое выделение памяти, отсутствие ошибок, связанных с вызовом delete, безопасный доступ к указателям (невозможность нарушений доступа)

Все это может обеспечить и значительные дополнительные выгоды:

- более высокий уровень стабильности работы программных средств;
- уменьшение времени простоя;
- большую гибкость для разработчика.

Я что, теряю контроль?

В качестве аргумента против использования управляемого кода чаще всего приводят то соображение, что возникает ощущение утраты слишком большой доли контроля над выполнением программы. В частности, появляется страх перед сборкой мусора, которая, кажется, происходит в произвольные и неподходящие моменты. Но с точки зрения практического применения это не так. Сборка мусора — в большинстве случаев детерминированная операция, и можно существенно повлиять на частоту ее запуска, контролируя схемы выделения памяти и области видимости объектов, а также настраивая конфигурацию сборщика мусора. В нативном коде вы, конечно, контролируете другие аспекты выполнения программы, но возможность контроля как таковая определенно никуда не делась.

Работа с CLR, а не против нее

Люди, незнакомые с управляемым кодом, часто относятся к таким вещам, как сборщик мусора или JIT-компилятор, как к чему-то, с чем приходится «разбираться», что нужно «терпеть» или «обходить». Это совершенно непродуктивный взгляд. Какую бы платформу вы ни использовали, вам понадобится целенаправленно работать над производительностью, если хотите, чтобы производительность системы была на высоте. По этой и другим причинам не допускайте ошибок, считая сборку мусора и JIT-компиляцию проблемами, с которыми вам придется бороться.

Когда вы начнете ценить то, что CLR делает для управления выполнением вашей программы, вы поймете, что можно добиться существенного повышения производительности, просто выбрав работу с CLR, а не против нее. Любая платформа имеет определенные ожидания относительно того, как ее будут использовать, и .NET не исключение. К сожалению, многие из этих ожиданий заданы неявно, и API никак не запрещает, да и не может запретить вам принимать неверные решения, нарушающие эти ожидания.

Бóльшая часть этой книги посвящена объяснению работы CLR, для того чтобы решения, которые вы принимаете, как можно более точно соответствовали ее ожиданиям. Это особенно актуально, к примеру, для сборки мусора, где весьма четко очерчены принципы достижения оптимальной производительности. Если игнорировать их, может случиться катастрофа. Гораздо больше шансов добиться успеха, если подстроиться под среду, а не пытаться заставить ее соответствовать вашим представлениям или, что еще хуже, полностью отказаться от ее использования.

Отдельные преимущества среды CLR могут в некотором смысле стать обоюдоострым мечом. Простота профилирования, обширная документация, насыщенные метаданные и инструментарий ETW-событий позволяют быстро найти источник проблем, но при этом проще становится и возложить на нее вину. У нативной программы могут быть подобные или еще бóльшие проблемы с выделением памяти в куче или неэффективным применением потоков, но, поскольку разглядеть их не так-то просто, нативная платформа избежит обвинений. При использовании как управляемого, так и нативного кода зачастую сбой дает сама программа, и ее нужно исправить, чтобы она лучше работала с базовой платформой. Ошибкой будет утверждать, что легкость обнаружения проблем свидетельствует о том, что они кроются в самой платформе.

Все это не означает, что CLR *никогда* не становится причиной проблем, но в первую очередь нужно искать их в приложении, а не в среде, операционной системе или оборудовании.

Уровни оптимизации

Оптимизация производительности может означать многое в зависимости от того, о какой части программного средства идет речь. В контексте .NET-приложений производительность представляется пятиуровневой (рис. 0.1).

Рис. 0.1. Уровни абстракций, они же приоритеты в работе

Верхний уровень — это архитектура вашей системы, будь то отдельное приложение или массив совместно работающих приложений, охватывающий весь дата-центр. Именно здесь начинается оптимизация производительности, поскольку архитектура оказывает наибольшее потенциальное влияние на общую производительность. Изменение архитектуры приводит к резкому изменению всех слоев, расположенных ниже, поэтому сначала следует убедиться, что у вас есть право на это изменение. Только после этого нужно перемещаться вниз по слоям.

Ниже располагается код, то есть алгоритмы, используемые для обработки данных. Здесь и происходят главные неприятности. На этом уровне встречается основная масса функциональных ошибок и просчетов, снижающих производительность. Это простое правило перекликается с аналогичным правилом отладки: опытный программист всегда будет считать, что ошибочен его собственный код, и не станет обвинять компилятор, платформу, операционную систему или оборудование. Это, безусловно, относится и к оптимизации производительности.

Ниже вашего кода находится среда .NET Framework — набор классов, предоставляемых компанией Microsoft или сторонними организациями и предлагающих стандартные функциональные возможности для обработки строк, коллекций и производства параллельных вычислений. Там же находятся такие полнофункциональные подплатформы для разработки определенных классов приложений, как

Windows Communication Framework, Windows Presentation Foundation и т. д. Вам не удастся избежать использования как минимум некоторой части среды, но применение большинства частей не обязательно. Бóльшая часть среды реализована с помощью управляемого кода, точно такой же управляемый код есть и в вашем собственном приложении. Код среды можно даже прочитать в Интернете по адресу http://referencesource.microsoft.com/ или из среды Visual Studio.

Под классами среды .NET Framework находится настоящая рабочая лошадка .NET — среда Common Language Runtime (CLR). Она представляет собой сочетание управляемых и неуправляемых компонентов, обеспечивающих такие сервисы, как сборка мусора, загрузка типов, JIT-компиляция и множество других деталей реализации среды .NET.

Еще ниже код, образно говоря, достигает «железа». После того как среда CLR выполнит JIT-компиляцию кода, вы фактически запускаете ассемблерный код процессора. Если пробраться в управляемый процесс с помощью отладчика машинного кода, можно увидеть исполняемый ассемблерный код. В этом заключается суть управляемого кода — используются обыкновенные машинные ассемблерные инструкции, выполняемые в контексте особо надежной среды.

Повторю еще раз: при проектировании высокопроизводительных приложений или их исследовании всегда следует начинать с верхнего уровня и двигаться вниз. Прежде чем ковыряться в деталях базового кода, убедитесь в целесообразности структуры и алгоритмов вашей программы. Макрооптимизация практически всегда выгоднее микрооптимизации.

В этой книге в первую очередь рассматриваются промежуточные уровни — .NET Framework и CLR. Они содержат «клей», скрепляющий вашу программу, и часто надежнее всего скрыты от программистов. Но многие из рассматриваемых инструментов применимы ко всем уровням. В конце книги будут кратко изложены некоторые практические и эксплуатационные приемы, позволяющие повысить производительность на всех уровнях системы.

Учтите, что хоть вся приводимая в книге информация и находится в открытом доступе, но в ней рассматриваются некоторые аспекты внутренних деталей реализации среды CLR. И все они могут быть изменены.

Коварная соблазнительность простоты

C# очень красивый язык. Он понятен, так как корнями уходит в C++ и Java. Его инновационность проявляется в заимствовании особенностей, присущих функциональным языкам, и в том, что вдохновение его разработчики черпают также из других источников, сохраняя при этом дух C#. Благодаря всему этому в нем удалось избежать сложностей, присущих большим языкам вроде C++. С ним легко начать работать, используя самый базовый синтаксис C# и постепенно повышая уровень своих знаний, чтобы применять более сложные функции.

Начать работать со средой .NET тоже просто. API в большинстве своем организованы в логические иерархические структуры, позволяющие найти именно то, что вы ищете. Модель программирования, развитые библиотеки функций и полезный

механизм IntelliSense в Visual Studio позволяют любому программисту довольно быстро создавать работающее программное обеспечение.

Но легкость порождает опасность. Мой бывший коллега однажды сказал: «Управляемый код позволяет посредственным разработчикам очень быстро создать много плохого кода».

Это можно показать на примере. Однажды мне попался код, который выглядел примерно так:

```
Dictionary <string , object > dict =
  new Dictionary <string , object >();
...
foreach(var item in dict)
{
    if (item.Key == "MyKey")
    {
        object val = dict["MyKey"];
        ...
    }
}
```

Впервые столкнувшись с ним, я был ошеломлен: как мог профессиональный разработчик не знать, как использовать словарь? Но, подумав, начал понимать, что, наверное, ситуация была не столь очевидной, как показалось вначале. Вскоре я выдвинул теорию, которая могла все объяснить. Проблема заключалась в конструкции `foreach`. Я считаю, что изначально в коде был применен `List<T>`, а чем можно воспользоваться для перебора элементов `List<T>` или любого перечисляемого типа коллекции? `foreach`. Простая гибкая семантика позволяет задействовать эту конструкцию практически для всех типов коллекции. Я предполагаю, что в какой-то момент разработчик понял, что структура словаря больше подойдет ему по смыслу, возможно, в других частях кода. Были внесены изменения, но сохранена конструкция `foreach`, поскольку, как бы то ни было, она все еще работала! За исключением того, что внутри цикла были уже не значения, а пары «ключ — значение». Ну, это хотя бы легко исправить...

Теперь понятно, как могла сложиться эта ситуация. Я, возможно, слишком мягок по отношению к первоначальному разработчику, но, если говорить начистоту, в данной ситуации его мало что оправдывает: код, несомненно, дефектный и свидетельствует о том, насколько несознательно разработчик отнесся к его написанию. Однако я считаю, что в данном случае синтаксис C# как минимум поспособствовал этому. Его легкость соблазнила разработчика менее критично отнестись к коду.

Есть множество других примеров совместной работы .NET и C#, значительно облегчающей действия среднего разработчика: очень просто инициировать выделение памяти, огромный объем кода, скрывающийся за многими функциями языка, весьма затратная реализация многих, казалось бы, простых API по причине их обобщенной, универсальной природы и т. д.

Цель этой книги — дать вам знания. Мы все начинаем как посредственные разработчики, но, имея хорошие инструкции, можем миновать эту стадию и начать действительно понимать создаваемый программный продукт.

Хронология совершенствования производительности среды .NET

Разработка сред CLR и .NET Framework продолжается по сей день, и со времени выхода в начале 2002 года версии 1.0 в них были внесены весьма существенные улучшения. В этом разделе задокументированы некоторые наиболее важные изменения, особенно те, что касаются повышения производительности.

- **1.0** (2002).
- **1.1** (2003).
  - Появление поддержки IPv6.
  - Появление возможности установки нескольких версий .NET на одном компьютере и выполнения программ на предопределенной версии платформы.
  - Повышение безопасности.
- **2.0** (2006).
  - Появление поддержки 64-разрядных систем (как x64, так и уже практически неактуальной IA-64).
  - Появление типов, допускающих значение `null`.
  - Появление анонимных методов.
  - Появление итераторов.
  - Появление обобщений и классов обобщенных коллекций.
  - Повышение производительности работы с кодировкой UTF-8.
  - Улучшение класса `Semaphore`.
  - Улучшения в сборщике мусора (GC):
    - уменьшение фрагментации, вызванной закреплением объектов;
    - уменьшение числа `OutOfMemoryExceptions`.
- **3.0** (2006).
  - Введение Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF).
- **3.5** (2007).
  - Появление LINQ и методов для работы с LINQ по всей библиотеке классов.
- **3.5 SP1** (2008).
  - Существенное повышение производительности WPF за счет, кроме всего прочего, аппаратной отрисовки, улучшения побитового отображения и отрисовки текстовых символов.
- **4.0** (2010).
  - Появление библиотеки параллельных задач (Task Parallel Library).
  - Появление Parallel LINQ (PLINQ).

- Появление диспетчеризации методов с помощью `dynamic`.
- Появление именованных и необязательных параметров.
- Усовершенствование фоновой сборки мусора в режиме рабочей станции.

❑ **4.5** (2012).
- Введение истечения срока ожидания разрешения регулярных выражений.
- Появление `async` и `await`.
- Усовершенствование GC:
  - появление фоновой сборки мусора в режиме сервера;
  - появление балансировки кучи больших объектов для сборки мусора в режиме сервера;
  - улучшение поддержки систем более чем с 64 процессорами;
  - появление устойчивого режима с низким уровнем задержки;
  - уменьшение фрагментации LOH;
  - появление поддержки наборов данных, превышающих 2 Гбайт.
- Появление многоядерной JIT-компиляции для сокращения времени запуска.
- Добавление `WeakReference<T>`.

❑ **4.5.1** (2013).
- Появление усовершенствованной поддержки отладки, особенно для кода x64.
- Автоматическое перенаправление привязки сборок.
- Появление явного уплотнения LOH.

❑ **4.5.2** (2014).
- Улучшение ETW.
- Улучшение поддержки профилирования.

❑ **4.6** (2015).
- Улучшение 64-разрядной JIT-компиляции (кодовое имя RyuJIT), появление поддержки инструкций SSE2 и AVX2.
- Добавление областей No-GC.

❑ **4.6.1** (2015).
- Повышение производительности сборщика мусора.
- Повышение производительности JIT-компилятора.

❑ **4.6.2** (2016).
- Разрешение использования имен путей длиннее 260 символов.
- Повышение производительности и надежности JIT-компилятора.
- Существенное исправление ошибок в `EventSource`.
- Улучшения в GC:

  - появление возможности сборки всех объектов, следующих за закрепленными объектами;
  - более эффективное использование свободного пространства gen 2.
- **4.7** (2017).
  - Повышение производительности JIT-компилятора.
  - Появление дополнительных вариантов конфигурации GC.
  - Появление типа `ValueTuple`.
- **4.7.1** (2017).
  - Улучшение сборки мусора за счет скорости выделения памяти в LOH. Выделение памяти в LOH больше не блокируется развертыванием полной фоновой сборки мусора.
- **4.7.2** (2018).
  - Повышение производительности `HashSet<T>` и `ConcurrentDictionary<TKey, TValue>`.
  - Повышение производительности `ReaderWriterLockSlim` и `ManualResetEventSlim`.
  - Повышение производительности сборки мусора.

.NET Core

.NET Core — это кросс-платформенная модульная версия .NET с открытым кодом. Microsoft выпустила версию 1.0 в июне 2016 года, а версия 2.0 вышла в августе 2017-го. .NET Core можно рассматривать в качестве поднабора полноценной среды .NET Framework, но эта версия содержит также дополнительные API, недоступные в стандартном выпуске. Используя .NET Core, можно создавать приложения для командной строки, приложения универсальной платформы Windows (UWP), веб-приложения ASP.NET Core и переносимые библиотеки кода. Хотя на .NET Core была перенесена основная часть стандартной библиотеки классов среды .NET Framework, в ней отсутствуют многие API. Если требуется выполнить переход с .NET Framework на .NET Core, может понадобиться существенная реструктуризация кода. Особо следует отметить отсутствие поддержки приложений Windows Forms или WPF[1].

Базовый код как для JIT, так и для Garbage Collector совпадает с тем, что имеется в полной версии среды .NET Framework. CLR-функции аналогичны в обеих системах.

Практически все проблемы с производительностью, рассматриваемые в этой книге, одинаково актуальны для обеих систем, и я не стану отмечать различия между этими платформами.

---

[1] Возможно, появится в .Net Core 3. В публично доступной бета-версии это уже есть. — *Примеч. науч. ред.*

И тем не менее сделаю несколько важных предупреждений.

- ASP.NET Core — это значительное улучшение ASP.NET, в которой используется .NET Framework. Если требуется получить высокопроизводительное веб-приложение, стоит освоить работу с ASP.NET Core.
- Поскольку .NET Core — это ПО с открытым кодом, данная среда получает улучшения гораздо быстрее среды .NET Framework. Некоторые из изменений переносятся обратно в .NET Framework, но гарантировать это невозможно.
- Работа многих отдельно взятых API была оптимизирована в плане повышения производительности:
  - улучшены, а в некоторых случаях полностью переделаны такие коллекции, как `List<T>`, `SortedSet<T>`, `Queue<T>` и др.;
  - в LINQ сократилось количество выделений памяти и инструкций;
  - быстрее стали обрабатываться регулярные выражения и строки;
  - ускорены математические операции над непримитивными типами данных;
  - повысилась эффективность кодирования строк;
  - быстрее стали работать сетевые API;
  - немного ускорилась работа примитивов для параллельных вычислений.

Изменения коснулись и многого другого...
Однако с .NET Core не работают многие конкретные технологии:

- приложения WPF;
- приложения Windows Forms;
- веб-формы ASP.NET;
- серверы WCF;
- C++/CLI (но в .NET Core поддерживается P/Invoke).

К .NET Core многие относятся с пристальным вниманием и любовью. По возможности эту среду следует использовать во всех новых разработках. Ее код открытый, поэтому и вы можете внести в нее свой вклад и сделать ее еще лучше.

Учебный исходный код

В книге часто встречаются ссылки на учебные проекты. Это небольшие изолированные проекты, предназначенные для демонстрации конкретного принципа. Примеры эти простые, поэтому они не будут адекватно отражать объем или масштаб проблем с производительностью, которые вы обнаружите в своих исследованиях. Рассматривайте их как стартовую позицию для приобретения технологического и исследовательского мастерства, а не как серьезные примеры типового кода.

Все примеры кода можно загрузить с веб-сайта книги, расположенного по адресу http://www.writinghighperf.net. Большинство проектов успешно проходят сборку в среде

.NET 4.5, но для некоторых требуется версия 4.7. Чтобы открыть большинство проектов, понадобится как минимум Visual Studio 2015.

В некоторых учебных проектах, инструментах и примерах, приводимых в книге, используются пакеты NuGet. Они должны автоматически обновляться средой Visual Studio, но ими можно управлять и индивидуально, щелкнув правой кнопкой мыши на названии проекта и выбрав пункт меню Manage NuGet References (Управление ссылками на NuGet).

1 Измерение производительности и инструменты

Прежде чем углубиться в специфику CLR и .NET, следует разобраться с измерением производительности в целом, а также с множеством доступных инструментов. Эффективность вашей работы обеспечивается исключительно арсеналом используемых инструментов. В этой главе я постараюсь дать вам прочную основу для освоения и начала применения множества инструментов, рассматриваемых в книге.

Выбор предмета измерения

Перед тем как принять решение о предмете измерения, следует усвоить набор требований к производительности. Эти требования должны носить довольно общий характер, чтобы не предписывать конкретную реализацию, но в то же время быть вполне определенными и поддаваться измерениям. Они должны основываться на реальных положениях, даже если пока вы не знаете, как добиться их выполнения. Требования станут управлять тем, какие показатели нужно будет собрать. Перед сбором числовых значений следует разобраться в том, что именно мы намерены измерять. Звучит это как само собой разумеющееся, но на самом деле вопрос гораздо шире, чем можно себе представить. Возьмем, к примеру, память. Разумеется, вам захочется измерить объем потребляемой памяти и свести его к минимуму. О какой именно памяти идет речь? Частном рабочем наборе? Запрошенном объеме памяти? Страничном пуле? Пиковом рабочем наборе? Объеме кучи .NET-среды? Объеме кучи крупных объектов? Об объемах куч отдельно взятых процессоров, обеспечивающих их сбалансированность? О каком-то ином варианте? Нужен ли для отслеживания объема потребляемой памяти по времени усредненный почасовой показатель или же необходим пиковый показатель? Коррелируется ли объем потребляемой памяти с объемом рабочей нагрузки при вычислениях? Как видите, навскидку легко набирается десяток и более показателей, относящихся к одной лишь памяти. А мы еще не касались частных куч или профилирования приложений с целью определения того, какие разновидности объектов потребляют память!

Характеризуя потребности в измерениях, нужно формулировать запросы как можно конкретнее.

ИСТОРИЯ

В крупном серверном приложении, за работу которого я отвечал, отслеживался объем частных байтов (дополнительные сведения о различных типах измерений потребляемой памяти найдете далее, в подразделе «Счетчики производительности» раздела «Полезные инструменты») в качестве наиболее важной метрики, чье значение учитывали, когда решали, необходимо ли принимать такие меры, как перезапуск процесса перед началом крупных, интенсивно потребляющих память операций. Оказалось, что излишний объем частных байтов со временем свопировался и не влиял на общее потребление памяти в системе, о чем, собственно, мы и беспокоились. В систему внесли изменение — стали измерять взамен объем рабочего набора. Преимущество выразилось в сокращении объема потребляемой памяти на несколько гигабайтов (как уже говорилось, это было довольно крупное приложение).

Определившись с предметом измерений, нужно получить четкое представление о конкретных целях использования каждой метрики. На ранних стадиях разработки эти цели могут быть не вполне устоявшимися, даже нереальными, и тем не менее они должны основываться на высокоуровневых требованиях. Главным вначале может быть не достижение поставленных целей, а принуждение к созданию системы, выполняющей автоматические измерения для достижения целей.

Цели должны иметь количественное выражение. В общем виде цель, намеченная для вашей программы, может задаваться так: она должна быть быстродействующей. Разумеется, должна. Но это далеко не самый удачный показатель, поскольку быстродействие субъективно и четко сформулированного способа определения средств достижения этой цели нет. У вас должна быть возможность придать ей числовое выражение и получить способ его измерения.

Плохой пример: «Пользовательский интерфейс должен быть отзывчивым».

Хороший пример: «Не должно быть операций, способных заблокировать поток пользовательского интерфейса более чем на 20 мс».

Но одного числового выражения недостаточно. Если рассмотреть предыдущий пример с памятью, становится понятно, что нужна предельная конкретизация цели.

Плохой пример: «Объем памяти должен быть менее 1 Гбайт».

Хороший пример: «Объем памяти, потребляемый рабочим набором, не должен превышать 1 Гбайт при пиковой нагрузке 100 запросов в секунду».

Во второй версии цели дается описание конкретных обстоятельств, определяющих условия ее достижения. Фактически в ней предлагается вполне приемлемый сценарий тестирования.

Еще один из основных определяющих факторов, характеризующих ваши цели, — разновидность создаваемого приложения. Программа, имеющая пользовательский интерфейс, должна неизменно оставаться отзывчивой в потоке этого интерфейса, независимо от того, что она делает еще. Серверная программа, обрабатывающая десятки, сотни или даже тысячи запросов в секунду, должна отличаться

невероятной эффективностью в управлении вводом-выводом и синхронизацией, обеспечивая тем самым максимальную пропускную способность и поддержку высокого уровня использования вычислительной мощности центрального процессора. Архитектура подобного сервера разрабатывается совершенно иначе, чем для других приложений. Если архитектура неудачно разработанного приложения имеет принципиальные изъяны с точки зрения эффективности, исправить ситуацию в процессе его эксплуатации будет крайне трудно.

Важную роль играет и планирование мощностей. При разработке вашей системы и планировании измерений производительности весьма полезно решить, какой будет ее оптимальная теоретическая производительность. Если можно исключить все источники издержек, в том числе сборку мусора, JIT-компиляцию, прерывания потока и все, что вы считаете таковыми, то что останется для выполнения работы, для которой приложение предназначено? Какие теоретические ограничения с точки зрения рабочей нагрузки, потребления памяти, использования центрального процессора и внутренней синхронизации вы можете определить? Зачастую это зависит от задействованного оборудования и операционной системы. Например, при наличии 16-процессорного сервера с 64 Гбайт оперативной памяти с двумя сетевыми каналами по 10 Гбайт возникает мысль о пороге распараллеливания и о том, сколько данных можно сохранить в памяти и сколько — передать по каналу в каждую секунду. Это поможет спланировать нужное количество машин данного типа, если одной окажется недостаточно.

Преждевременная оптимизация

Возможно, вы знакомы с высказыванием Дональда Кнута (Donald Knuth): «Преждевременная оптимизация — корень всех зол». Смысл этой цитаты заключается в определении тех областей вашей программы, которые действительно важно оптимизировать. Это отсылает нас к закону Амдала, в котором описывается теоретически максимальное ускорение работы программного средства путем его оптимизации, в частности способ применения оптимизации к последовательно выполняемым программам и выбор оптимизируемых частей программы. Время, потраченное на микрооптимизацию того кода, который не вносил существенного вклада в общую неэффективность, по большому счету считается затраченным впустую. Это положение верно для микрооптимизаций на уровне кода и может быть применимо и для более высоких уровней вашей архитектуры. Конечно, следует продумывать создаваемую архитектуру и по мере разработки разбираться в ее ограничениях, иначе можно упустить что-то важное и сильно затрудняющее работу приложения. Но многие из этих ограничений на поверку окажутся не столь важны для системы (или по крайней мере вы пока не сможете оценить их важность). Перепроектировать существующее приложение с нуля, конечно, можно, но это будет значительно дороже, чем изначально спроектировать его правильно. При разработке архитектуры крупной системы зачастую единственный способ избежать ловушки преждевременной

оптимизации — это применение опыта разработки и изучение архитектуры аналогичных или эталонных систем. В любом случае определение целей достижения производительности должно опережать проектные решения. Производительность, подобно безопасности и многим другим аспектам разработки программного обеспечения, не может играть второстепенную роль и с самого начала должна быть четко выраженной целью.

Анализ производительности, выполненный в самом начале работы над проектом, отличается от проводимого после написания программы и в ходе ее тестирования. Сначала нужно удостовериться, что ваша конструкция поддается масштабированию, технология теоретически может справиться с намеченными задачами и вы не наделали грубых архитектурных ошибок, которые будут вечно вас преследовать. Как только проект доберется до фаз тестирования, развертывания и сопровождения, больше времени будет отводиться на оптимизацию на микроуровне, анализ конкретных шаблонов в коде, попытки сокращения объемов потребляемой памяти и т. д.

У вас никогда не будет времени на всеобъемлющую оптимизацию, следовательно, начинать ее следует обдуманно. Сначала оптимизируйте самые неэффективные части программы, чтобы получить наибольшую выгоду. Именно поэтому решающее значение имеет наличие целей и совершенной системы измерений, в противном случае вы даже не поймете, с чего нужно начать.

Сравнение усредненных и процентных показателей

Рассматривая измеряемые числовые метрики, следует решить, какой вид статистических показателей будет наиболее подходящим. Большинство специалистов изначально полагаются на усредненные показатели. Конечно, в большинстве случаев они играют важную роль, но не следует сбрасывать со счетов и процентные показатели (процентили). При наличии требований к доступности вам почти наверняка понадобятся цели, выраженные в процентилях, например: «Среднее время задержки запросов к базе данных должно быть менее 10 мс. А 95%-ный процентиль задержек запросов к базе данных должен быть менее 100 мс».

Если вам неизвестно это понятие, не расстраивайтесь: в нем нет ничего сложного. Если взять 100 измерений чего угодно и отсортировать их, то 95%-ная запись в списке и является значением 95%-ного процентиля этого набора данных. 95%-ный процентиль свидетельствует о том, что 95 % значений в выборке имеет это или меньшее значение. Или же 5 % запросов имеют значение выше этого.

Общая формула для вычисления индекса отсортированного списка имеет следующий вид:

$$\frac{P}{100}N,$$

где P — процентиль; N — длина списка.

Рассмотрим серию измерений времени паузы при сборке мусора нулевого поколения в миллисекундах со следующими значениями (для удобства они были заранее отсортированы):

1, 2, 2, 4, 5, 5, 8, 10, 10, 11, 11, 11, 15, 23, 24, 25, 50, 87.

Для этой выборки из 18 значений усредненным является 17 мс, но 95%-ный процентиль намного выше — 50 мс. Если принять во внимание лишь усредненное значение, то задержки сборщика мусора могут вас и не обеспокоить, но, зная процентиль, вы получите более четкую картину происходящего и поймете, что при некоторых сборках мусора дела обстоят куда хуже.

Здесь также демонстрируется, что медианное значение (50%-ный процентиль) может значительно отличаться от среднего, которое подвергается весьма сильному влиянию со стороны значений более высоких процентилей.

Для сервисов с высокой доступностью процентильные значения зачастую оказываются гораздо важнее, чем для прочих. Чем выше требуемый уровень доступности, тем более высокий процентиль нужно отслеживать. Обычно для этого достаточно высок 99%-ный процентиль, но, если приходится иметь дело с по-настоящему огромным валом запросов, важную роль могут сыграть 99,99%-ный, 99,999%-ный или даже более высокий процентиль. Часто требуемый показатель определяется бизнес-необходимостью, а не техническими соображениями.

Ценность процентилей в том, что они дают представление об ухудшении показателей во всем контексте выполнения. Даже если усредненный пользовательский опыт работы или средние показатели обработки запросов вашим приложением представляются вполне приемлемыми, возможно, показатель 90%-ного процентиля высветит некоторые возможности для внесения усовершенствований. Он подскажет, что в 10 % случаев работа приложения подвергается влиянию более негативных обстоятельств, чем в остальных случаях. Отслеживание нескольких процентилей покажет, насколько быстро происходит деградация. Насколько важен этот процент пользователей или запросов, решается в конечном счете в бизнес-плоскости, и здесь на сцену возвращается закон убывающей доходности: оптимизация последнего процента может стать чрезвычайно сложным и затратным делом.

Я начал с того, что 95%-ный процентиль показанного ранее набора данных составлял 50 мс. Технически это так, но в данном случае польза от этой информации сомнительна, поскольку данных недостаточно, чтобы этот результат имел хоть какую-то статистическую значимость, и, в сущности, данный показатель может носить случайный характер. Для определения нужного числа значений в выборке воспользуйтесь следующим эмпирическим правилом: необходимо, чтобы размер выборки был на один порядок больше целевого процентиля. Для процентилей в диапазоне 0–99 нужны минимум 100 значений. Для 99,9%-ного процентиля — 1000, для 99,99%-ного процентиля — 10 000 и т. д. В большинстве случаев это правило работает, но если вы заинтересованы в определении фактического количества значений с математической точки зрения, то изучите науку определения объема выборки глубже.

Точнее говоря, потенциальная ошибка зависит от квадратного корня из числа значений в выборке. Например, 100 значений приводят к диапазону ошибок 90–100, или 10 % ошибок, 1000 — к диапазону 969–1031, или 3 % ошибок.

Не забудьте принять в расчет другие типы статистических значений: минимум, максимум, среднее значение, стандартные отклонения и многое другое, в зависимости от типа измеряемых показателей. Например, чтобы определить статистически значимые различия между двумя наборами данных, часто используются t-тесты. Стандартные отклонения применяют для определения вариативности набора данных.

Эталонное тестирование

Если нужно измерить производительность фрагмента кода, особенно в сравнении с альтернативным вариантом его реализации, понадобится эталонное тестирование (бенчмаркинг). Буквальное определение эталона (бенчмарка): это стандарт, с которым могут сравниваться результаты измерений. В мире разработки программного обеспечения это означает замеры точного времени, обычно усредняемые по результатам тысяч или миллионов итераций.

Эталонному тестированию можно подвергать множество разнообразных элементов программ на разных уровнях, от программ целиком до отдельных методов. Но чем больше варьируемость тестируемого кода, тем больше прогонов понадобится для получения результата приемлемой точности.

Проведение эталонного тестирования — задача непростая. Требуется определить метрики для вашего кода в реалистичных условиях, чтобы получить реалистичные данные, на основании которых можно предпринимать дальнейшие действия. Но создание таких условий для получения полезных данных может оказаться весьма сложным.

Эталонное тестирование наилучшим образом проявляет себя при тестировании отдельно взятого ресурса, за пользование которым не ведется соперничество. Классический пример такого ресурса — время центрального процессора. Конечно, можно протестировать что-нибудь вроде времени доступа по сети или считывания файлов с SSD, но для этого понадобится изолировать эти ресурсы от внешнего воздействия. Современные операционные системы для такого рода изоляции не приспособлены, но тщательное отслеживание среды окружения повышает вероятность получения приемлемых результатов.

Тестирование целых программ или подмодулей, скорее всего, потребует использования и таких ресурсов, за которые ведется соперничество. К счастью, широкоформатное тестирование требуется редко. Экспресс-профилирование приложения выявит точки наиболее интенсивного потребления ресурсов, позволяя сфокусировать внимание на этих областях.

Узкоформатное эталонное микротестирование чаще всего применяется для оценки времени центрального процессора, затрачиваемого на выполнение кода отдельно взятых методов, при этом тест зачастую перезапускается миллионы раз для сбора точной статистики.

Кроме аппаратной изолированности, следует рассмотреть и ряд других факторов.

- *Код должен быть предварительно JIT-компилирован.* Первоначальный запуск метода занимает гораздо больше времени, чем последующие запуски.
- *Иные скрытые инициализационные операции.* Существуют кэши операционной системы, кэши файловой системы, кэши CLR-среды, кэши оборудования, процесс генерации кода и масса других стартовых издержек, которые могут влиять на производительность кода.
- *Изоляция.* Если окажется, что в процессе эталонного тестирования запущены иные «тяжелые» процессы, измерения могут исказиться.
- *Резкие отклонения.* Нужно делать скидку на резкие статистические отклонения в измерениях, которые, вероятно, не следует брать в расчет. Определение того, что считать резким отклонением, а что — нормой, — задача непростая.
- *Узкая сфокусированность.* Время центрального процессора — важный показатель, но не менее важны метрики выделения памяти, ввода-вывода, блокировки потоков и многие другие.
- *Различие между кодом, предназначенным для выпуска (Release mode), и кодом для отладки (Debug mode).* Эталонное тестирование всегда должно проводиться в отношении кода, предназначенного для выпуска программного продукта, причем с включением всех механизмов оптимизации.
- *Влияние наблюдателя.* Само наблюдение обязательно вносит какие-то изменения в наблюдаемый процесс. Например, измерение времени центрального процессора или выделение памяти в .NET-среде не обходится без генерирования дополнительных ETW-событий, чего обычно не происходит в реальных условиях, и время на эти операции будет включено в общие результаты измерений.

Примеры кода, прилагаемые к книге, часто содержат разработанные на скорую руку эталонные тесты, но в силу перечисленных факторов их не следует считать истиной в последней инстанции.

Вместо написания собственных эталонных тестов практически всегда следует использовать существующую библиотеку, которая успешно справляется со многими названными нюансами. Несколько возможных вариантов будут рассмотрены в данной главе чуть позже.

Полезные инструменты

Если самый важный аспект данной книги выразить в одном-единственном правиле, оно будет звучать так:

«Измерения, измерения и еще раз измерения!»

Вы не узнаете, в чем именно заключаются проблемы с производительностью, если не проведете точные измерения. Вы наверняка наберетесь опыта, который

поможет обоснованно предполагать, где кроются проблемы с производительностью, исходя из изучения кода или на основе собственной интуиции. Вы даже можете оказаться правы, но ни в коем случае не поддавайтесь искушению проигнорировать измерения, кроме как решая самые примитивные проблемы. Аргументы в пользу этого утверждения носят двойственный характер.

Во-первых, представим, что вы правы и точно выявили причину снижения производительности. Но ведь вам, возможно, захочется выяснить, насколько ваше приложение улучшилось, не так ли? Бравировать своими навыками куда безопаснее, имея надежные данные для их подтверждения.

Во-вторых, я даже не могу припомнить, насколько часто ошибался. Вот наглядный пример: анализируя объем собственной памяти процесса в сравнении с управляемой памятью, мы до некоторых пор предполагали, что она формируется в одной конкретной области, куда загружается огромный набор данных. Но вместо того, чтобы поставить перед разработчиком задачу сокращения объема потребляемой таким образом памяти, мы провели ряд экспериментов по отключению загрузки данного компонента. Мы также воспользовались отладчиком для получения дампа данных обо всех кучах в процессе. К нашему удивлению, основной объем таинственно поглощаемой памяти приходился на издержки при загрузке сборок, а не на этот набор данных. Таким образом, мы избежали напрасной траты времени.

Оптимизация производительности бессмысленна, если не использовать специальные инструменты для проведения измерений. Измерение производительности — непрерывный процесс, который нужно сделать неотъемлемой частью вашего инструментария разработчика, процесса тестирования и средств мониторинга. И коль уж ваше приложение требует непрерывного мониторинга для контроля реализации своей функциональности, то и для контроля производительности, скорее всего, мониторинг потребуется.

Далее в этой главе рассматриваются различные инструменты, которыми можно воспользоваться для профилирования, мониторинга и отладки при решении проблем с производительностью. Основное внимание я уделил Visual Studio и программным средствам, находящимся в свободном доступе, но следует иметь в виду, что есть множество коммерческих предложений, которые в ряде случаев могут упростить решение различных аналитических задач. Если в бюджете есть средства на приобретение платных инструментов, воспользуйтесь ими. Но желаемого результата можно добиться и применяя более доступные инструменты, рассматриваемые в этой книге, или их аналоги. В первую очередь, их может быть проще запустить на машинах клиентов или в эксплуатационном окружении. Более того, будучи немного ближе к «железу», они дадут вам знание и понимание происходящего на очень глубоком уровне, что поможет как следует разобраться в данных, независимо от используемого инструментального средства.

Я даю описание основ применения каждого инструмента и базовые инструкции для начала работы с ним. В книге подробно расписаны шаги для действий по конкретным сценариям, но зачастую с опорой на уже имеющиеся у вас знания пользовательского интерфейса и основ работы с ним.

СОВЕТ

Прежде чем углубляться в освоение конкретных инструментов, примите совет по поводу работы с ними. При попытке использования незнакомого инструментального средства в большом и сложном проекте вам, возможно, это в полной мере не удастся, и вы будете расстроены или даже получите неверные результаты. Изучая принципы замера производительности с помощью нового средства, создайте тестовую программу с хорошо известным вам поведением и воспользуйтесь этим средством для подтверждения характеристик ее производительности. Таким образом вы разберетесь в том, как применять инструментальное средство в более сложной ситуации, и снизите вероятность совершения технических или оценочных ошибок.

Visual Studio

Visual Studio не единственная IDE, но именно ее использует большинство .NET-программистов, и если вы относитесь к их числу, то, вполне возможно, с нее и начнете анализ производительности. Различные редакции Visual Studio поставляются с различными инструментальными средствами. В этой книге предполагается, что у вас установлена как минимум профессиональная редакция (Professional), но я буду давать описание и некоторых средств, входящих в более высокие редакции. Можете смело пропустить такое описание и двигаться дальше, если у вас не окажется нужной редакции Visual Studio.

Если у вас установлена Visual Studio Professional или более высокая редакция, можно обратиться к инструментам замера производительности через меню Analyze (Анализ), в котором следует выбрать пункт Performance Profiler (Профайлер производительности) (или воспользоваться для этого сочетанием клавиш Alt+F2).

Для стандартных .NET-приложений будут показаны как минимум три варианта, а также дополнительные варианты, которые зависят от конкретного типа приложения (рис. 1.1).

- CPU Usage (Использование ЦП) — замеряет уровень потребления функцией ресурсов центрального процессора.
- Memory Usage (Использование памяти) — демонстрирует сборку мусора и позволяет получать отображения мгновенного состояния кучи.
- Performance Wizard (Мастер анализа производительности) — применяет VsPerf.exe для выполнения на основе ETW анализа использования центрального процессора (путем сбора выборок или инструментирования), выделения памяти средой .NET и взаимодействия потоков.

Если нужно только проанализировать использование центрального процессора или просмотреть содержимое кучи, следует воспользоваться первыми двумя инструментами. Мастер анализа производительности Performance Wizard также может провести анализ использования центрального процессора, но затратит на него чуть больше времени. Несмотря на то что он в некотором смысле устарел, с его помощью также можно отследить выделение памяти и распараллеливание потоков.

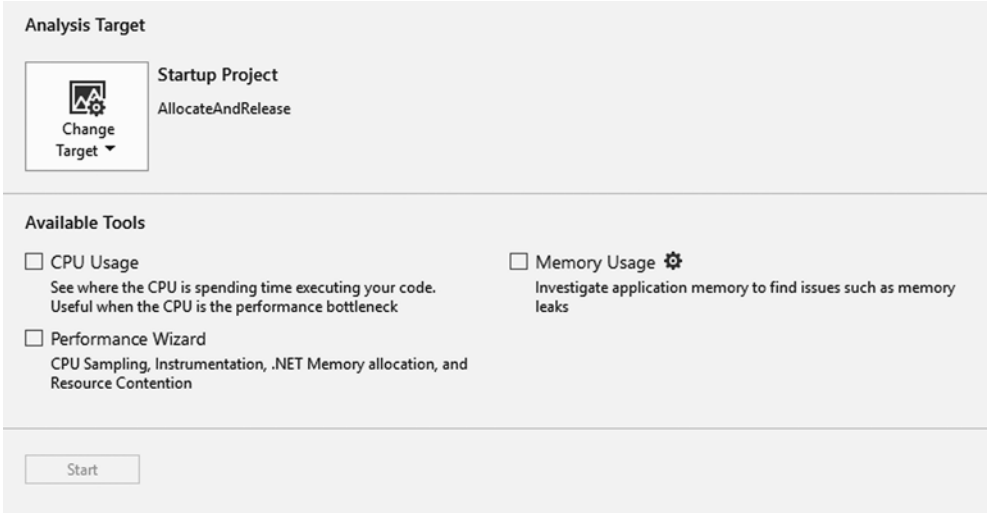

Рис. 1.1. Варианты профилирования в Visual Studio

Для более качественного анализа распараллеливания потоков следует установить имеющееся в свободном доступе инструментальное средство Concurrency Visualizer, доступное в качестве дополнительного расширения (меню Tools ▸ Extensions and Updates (Инструменты ▸ Расширения и обновления)).

Инструментальные средства Visual Studio относятся к простейшим в применении, но если у вас пока нет нужной версии Visual Studio, то обойдутся они весьма недешево. К тому же предлагаемые ими функции весьма ограничены и не отличаются особой гибкостью. Если воспользоваться Visual Studio нельзя или вы нуждаетесь в более широких возможностях, прочитайте далее описание свободно распространяемых альтернативных средств. Практически все современные средства измерения производительности используют в качестве основного механизма (по крайней мере в ядре Windows 8/Server 2012 и более высоких версиях) события ETW (Event Tracing for Windows — трассировка событий Windows). Этот способ позволяет операционной системе исключительно быстро и эффективно регистрировать все интересные события. Любое приложение может генерировать эти события с помощью очень простых API. В главе 8 рассматриваются механизмы применения ETW-событий в ваших программах путем определения собственных событий или интеграции с потоком системных событий. Некоторые инструменты, например PerfView, могут одновременно собирать произвольные ETW-события, все их вы можете анализировать по отдельности из одного сеанса сбора. Порой механизмы анализа производительности Visual Studio представляются мне предназначающимися для этапа разработки, в то время как другие инструменты — для реальной системы. Ваше мнение может отличаться от моего, и вам следует применять тот инструментарий, от которого ожидаете наибольшей отдачи.

Профилирование центрального процессора

В этом разделе будет представлен общий интерфейс профилирования при выборе опции профилирования ЦП. Другие варианты профилирования (например, для памяти) будут рассмотрены далее в соответствующих разделах.

Когда выбран вариант CPU Usage (Использование ЦП), появляется окно с графиком его применения и списком методов, наиболее активно потребляющих его ресурсы (рис. 1.2).

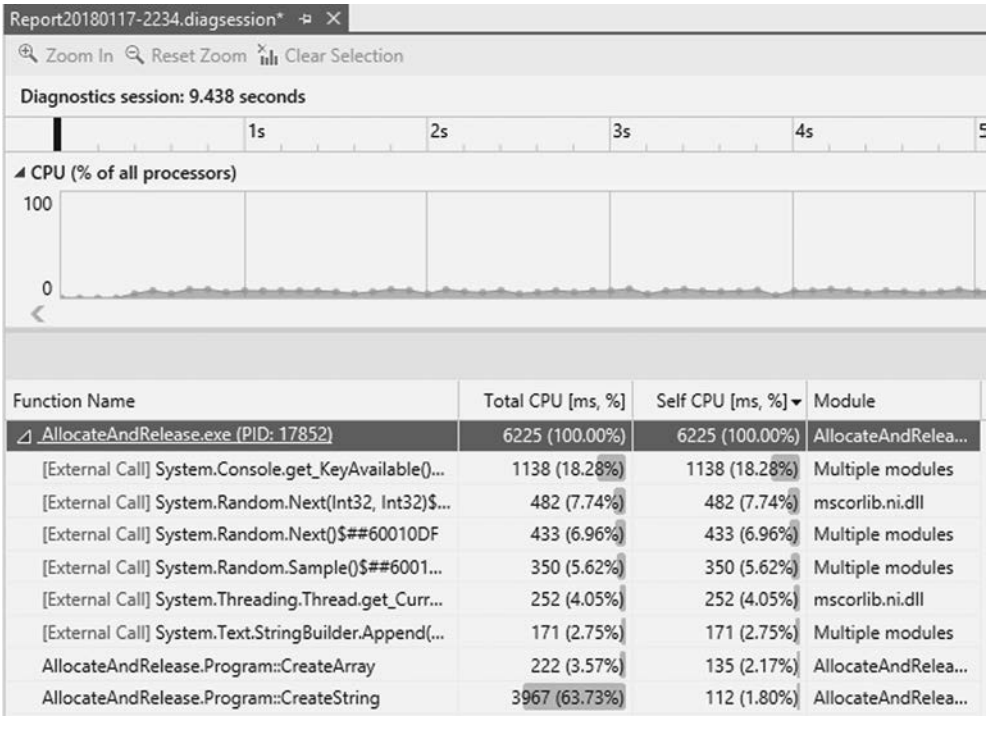

Рис. 1.2. Окно CPU Usage (Использование ЦП). Шкала времени, общий график использования и дерево методов, наиболее активно потребляющих ресурсы ЦП

Если нужно углубиться в показатели конкретного метода, следует на его названии сделать двойной щелчок кнопкой мыши — откроется соответствующее ему окно просмотра Call/Callee (Вызывающий/Вызываемый) (рис. 1.3).

Если этот вариант не даст достаточной информации, обратитесь к мастеру оценки производительности (рис. 1.4). Для сбора важных событий это средство ориентируется на приложение VsPerf.exe.

При выборе пункта CPU sampling (Опрос ЦП) выполняется сбор выборок без прерывания работы программы (рис. 1.5).

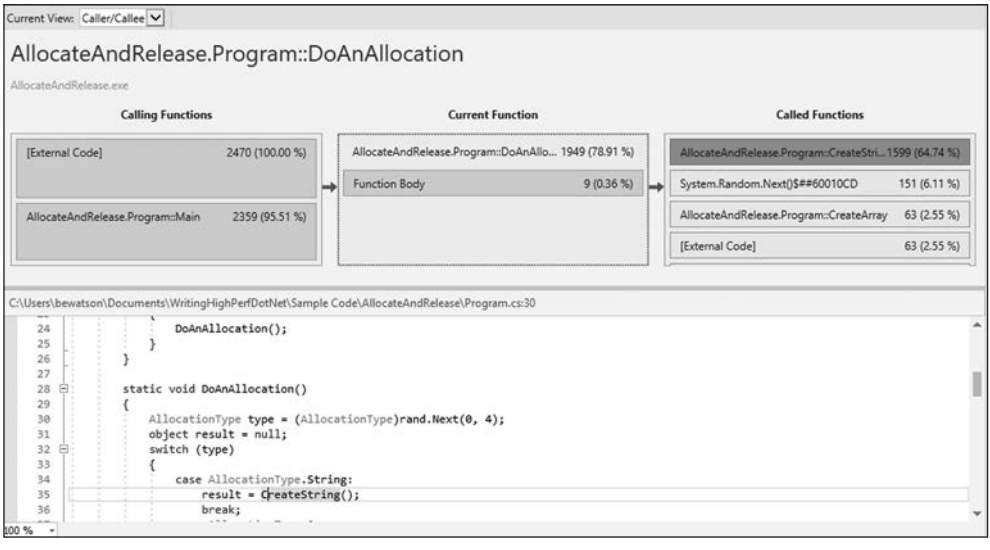

Рис. 1.3. Диаграмма Call/Callee (Вызывающий/Вызываемый) при исследовании метода в режиме анализа использования ЦП. Показаны те части метода, которые задействуют ЦП наиболее интенсивно

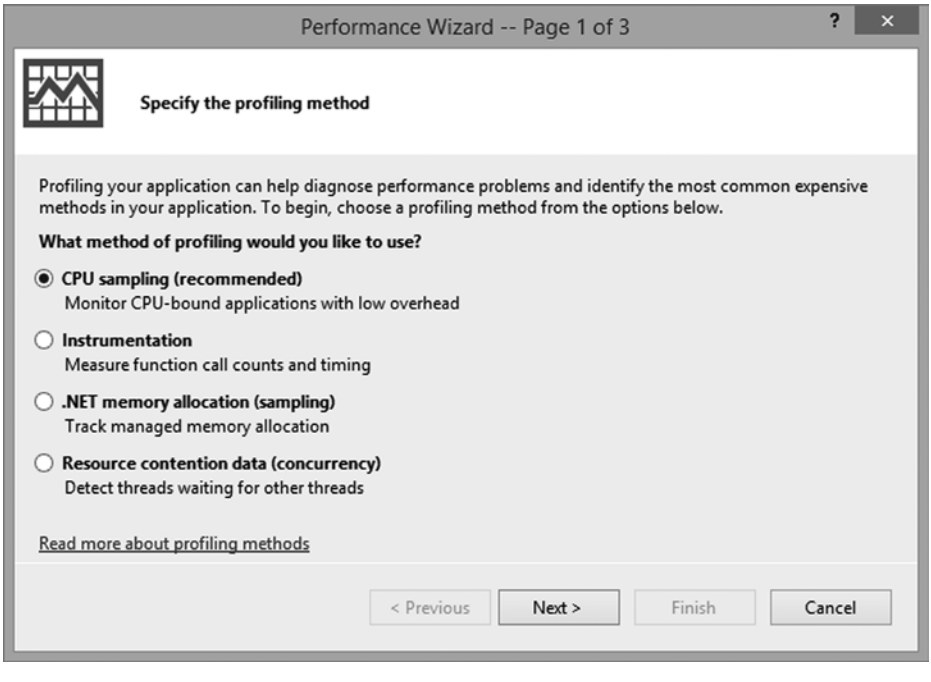

Рис. 1.4. Первый экран мастера оценки производительности Performance Wizard

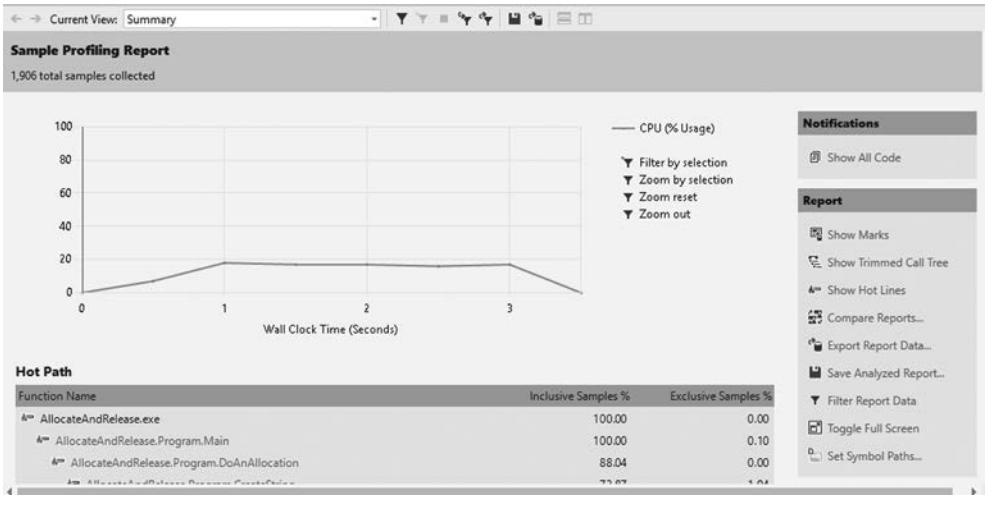

Рис. 1.5. Внешний вид отчета о выборке показателей использования ЦП в мастере оценки производительности Performance Wizard

Хотя внешне интерфейс отличается от того, который появился после выбора пункта CPU Usage (Использование ЦП), здесь вы также видите общую картину использования ЦП с течением времени, ниже которой расположено дерево методов, наиболее активно потребляющих ресурсы ЦП. Доступны также дополнительные отчеты. График можно укрупнить, в ответ на это обновится вся остальная аналитика. Щелчок на показанном в таблице имени метода приведет к переходу в уже знакомое окно просмотра функциональных подробностей Function Details (Функциональные подробности) (рис. 1.6).

Ниже сводки, относящейся к вызову функции, будет дан исходный код (если таковой доступен), где части метода с самым высоким потреблением ресурсов отмечены выделением строк.

Другие отчеты:

- Modules (Модули) — показывает, на какие сборки приходится наибольшее число выборок;
- Caller/Callee (Вызывающий/Вызываемый) — альтернатива окну просмотра Function Details (Функциональные подробности); здесь показываются таблицы выборок до и после позиции в стеке текущего метода;
- Functions (Функции) — предоставляет быстрый способ просмотра таблицы всех функций в процессе;
- Lines (Строки) — предлагает способ быстрого перехода к наиболее интенсивно потребляющим ресурсы строкам кода, выполняемого в процессе.

Вместо выборки можно снабдить код инструментальными вставками. При этом в исходный выполняемый код вносятся изменения путем добавления инструкций

вокруг каждого вызова метода с целью измерения расходуемого времени. Тем самым можно получить более точные отчетные показатели для очень маленьких быстро выполняемых методов, но при этом существенно возрастут издержки времени выполнения, а также увеличится объем производимых данных. За исключением отсутствия графика использования ЦП, отчет по внешнему виду и поведению напоминает отчет в режиме выборок. Основное отличие интерфейса заключается в указании времени измерения, а не количества выборок.

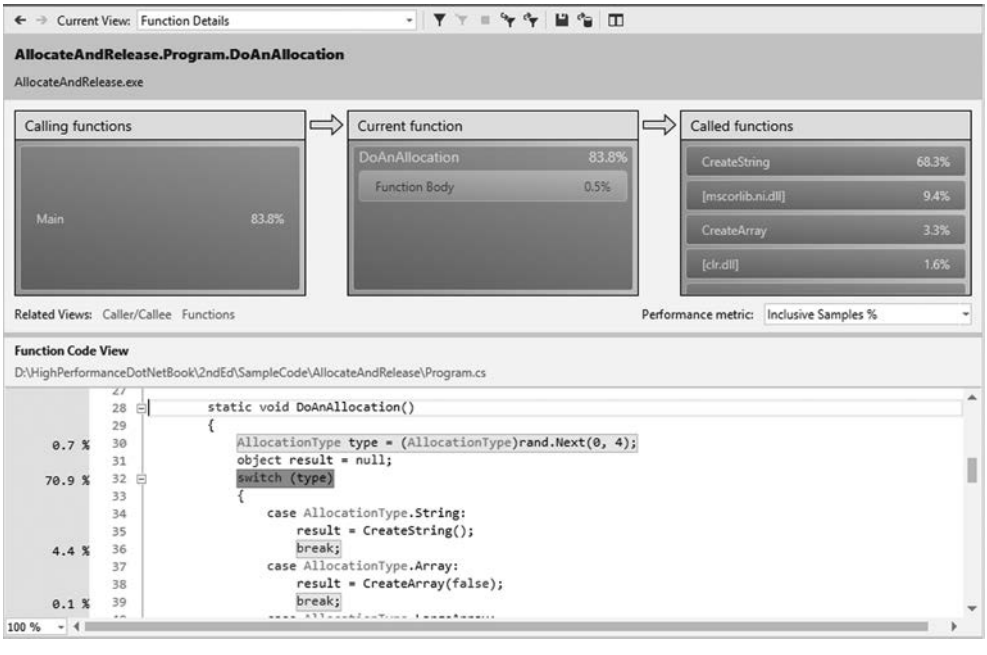

Рис. 1.6. Подробности использования ЦП

Профилирование с помощью командной строки

Visual Studio позволяет анализировать использование ЦП, выделение памяти и соперничество за ресурсы. Эта среда хорошо проявляет себя в ходе разработки или запуска комплексных тестов, с помощью которых тщательно проверяется работа программного продукта. Но для тестирования с целью получения точных срезов характеристик производительности крупного приложения, запущенного с реальными данными, она применяется крайне редко. Если нужно получить срез данных о производительности на машинах, не задействуемых для разработки, скажем, на машине клиента или в дата-центре, возникает потребность в инструменте, запускаемом вне среды Visual Studio.

Для этого существует автономный профилировщик Visual Studio Standalone Profiler, поставляемый с Visual Studio Professional или более высокими редакция-

ми Visual Studio. Вам придется установить этот инструмент с соответствующего носителя отдельно от Visual Studio. В моих ISO-файлах с образами дисков для 2012 Professional и 2015 Professional он находится в каталоге `Standalone Profiler`. Для Visual Studio 2017 исполняемый файл — `VsPerf.exe`, он расположен в каталоге C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\TeamTools\Performance Tools.

Чтобы с помощью этого инструмента собрать информацию из командной строки, нужно сделать следующее.

1. Перейти в папку установки (или добавить ее в PATH).
2. Запустить команду `VsPerfCmd.exe /Start:Sample /Output:outputfile.vsp`.
3. Запустить программу, профиль которой нужно получить.
4. Запустить команду `VsPerfCmd.exe /Shutdown`.

В результате будет создан файл `outputfile.vsp`, который можно открыть в Visual Studio.

В программе VsPerfCmd.exe имеется ряд других вариантов использования, включая все типы профилирования, предлагаемые полной версией Visual Studio. Кроме самых распространенных, можно также выбрать следующие.

- Coverage (Покрытие) — сбор информации о покрытии кода.
- Concurrency (Совместное выполнение) — сбор информации о соперничестве за ресурсы.
- Trace (Трассировка) — инструментирование кода для сбора данных о времени исполнения методов и числе их запусков.

Важно сделать правильный выбор между трассировкой и режимом выборок. Что именно использовать, зависит от того, что нужно измерить. Режим выборок должен применяться по умолчанию. При нем процесс прерывается каждые несколько миллисекунд и записываются стеки всех потоков. Это наилучший способ получения достоверной картины использования ЦП в вашем процессе. Однако он не дает хороших результатов для вызовов ввода-вывода, не отличающихся значительной степенью использования ЦП, но все-таки влияющих на общее время выполнения программы.

Режим трассировки требует внесения изменений в каждый вызов функции, имеющийся в процессе для записи меток времени. Он характеризуется более серьезным вторжением в код и существенно замедляет выполнение программы. Тем не менее он записывает фактическое время, затрачиваемое на выполнение каждого метода, следовательно, может обеспечить более высокую точность для менее объемных быстро выполняемых методов.

Режим покрытия не предназначен для анализа производительности, но пригодится, чтобы увидеть, какие строки кода выполнялись. Это свойство покажет себя с положительной стороны при запуске тестов для определения объема программного продукта, охватываемого тестом. Существуют коммерческие средства, выполняющие эту работу, но можно без особых усилий проделать все самостоятельно.

В режиме совместного выполнения записываются события, протекающие при конкурентном использовании ресурсов посредством блокировки или применения какого-то другого объекта синхронизации. Этот режим может сообщить вам, заблокированы ли ваши потоки, когда возникла конкуренция. Дополнительные сведения об асинхронном программировании и измерении количества конфликтов при блокировках в приложении можно получить в главе 4.

Счетчики производительности

Применение счетчиков производительности относится к самым простым способам отслеживания производительности вашего приложения и системы в целом. В Windows имеются сотни счетчиков десятков категорий, в том числе множество для .NET-технологии. Самый простой способ обращения к ним — через встроенную в Windows утилиту Performance Monitor (PerfMon.exe) (рис. 1.7).

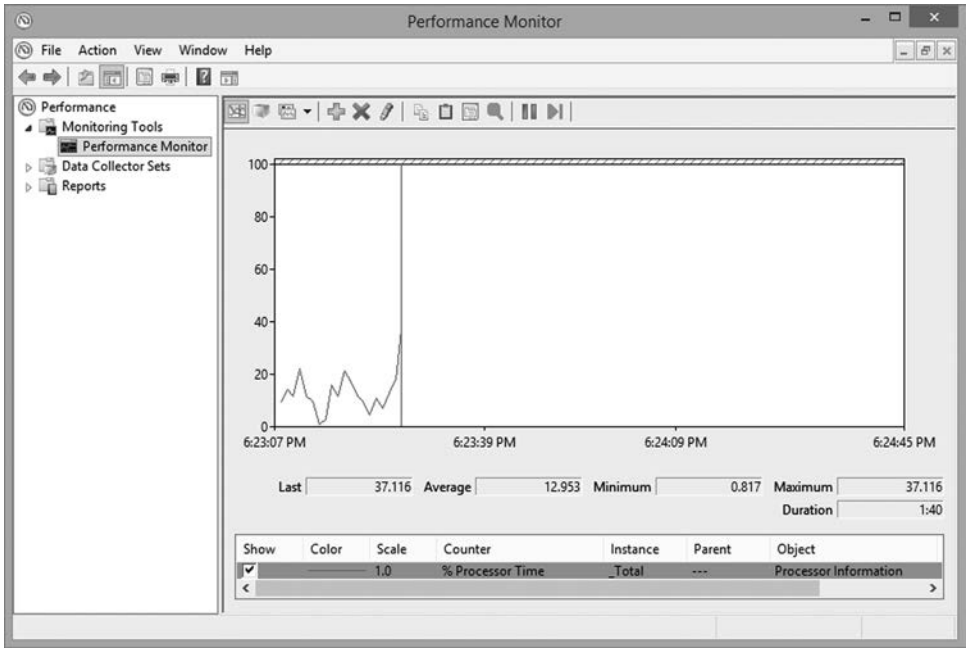

Рис. 1.7. Основное окно PerfMon, демонстрирующее показания счетчика процессора для небольшого отрезка времени. Вертикальная линия отображает текущее положение дел, а график в исходных настройках будет охватывать примерно 100 с

У каждого счетчика есть категория и имя. У многих счетчиков имеются также экземпляры выбранного счетчика. Например, для процентного счетчика % Processor Time в категории Process экземплярами являются различные процессы, для которых существуют значения. У некоторых счетчиков имеются также

метаэкземпляры, такие как _Total или <Global>, агрегирующие значения по всем экземплярам.

Соответствующие темам счетчики будут подробно описаны во многих следующих главах, но существуют и счетчики общего назначения, не относящиеся к технологии .NET, с которыми вам следует ознакомиться. Это счетчики производительности практически для любой подсистемы Windows, которые пригодны, как правило, в любой программе (рис. 1.8).

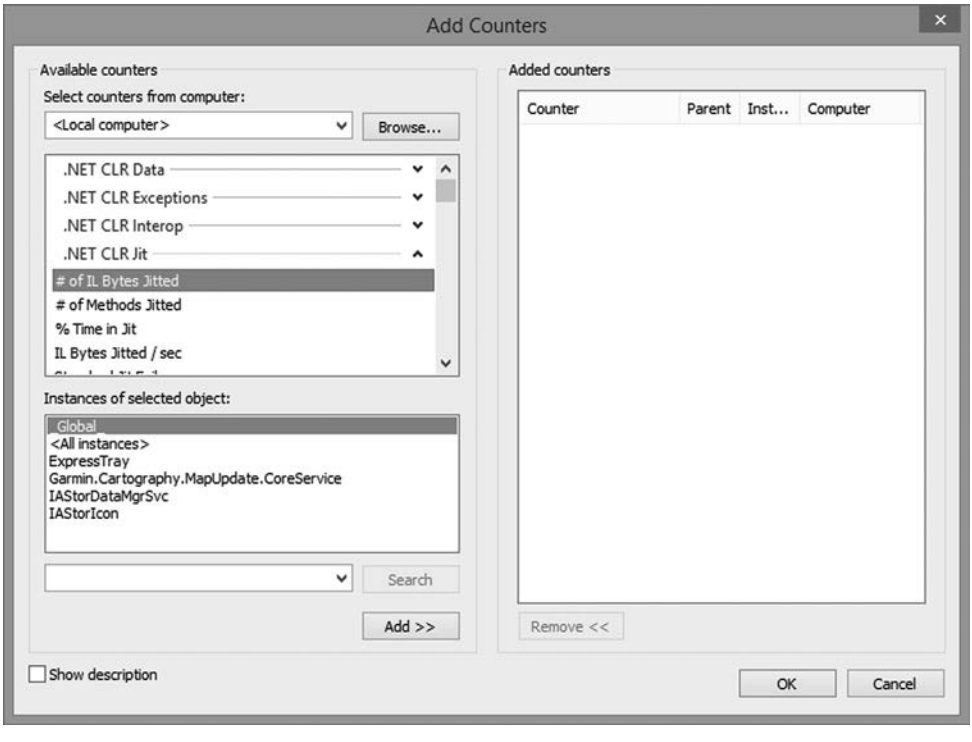

Рис. 1.8. Один из сотен счетчиков во многих категориях, показывающий все применимые экземпляры (в данном случае процессы)

Прежде чем продолжить, следует ознакомиться с основной терминологией операционной системы.

- *Физическая память (Physical Memory)* — реальные микросхемы памяти в компьютере. Напрямую управляет физической памятью только операционная система.
- *Виртуальная память (Virtual Memory)* — логическая организация памяти в заданном процессе. Виртуальная память может быть больше физической. Например, у 32-разрядных программ объем адресного пространства — 4 Гбайт, даже если у самого компьютера только 2 Гбайт оперативной памяти. Изначально

Windows позволяет программам иметь доступ только к 2 Гбайт, но если у исполняемой программы установлен флаг расширенного адресного пространства, возможен доступ ко всем 4 Гбайт (на 32-разрядных версиях Windows у программ с расширенным адресным пространством существует ограничение доступа, определяющее объем 3 Гбайт, а 1 Гбайт зарезервирован для операционной системы). С появлением Windows 8.1 и Server 2012 64-битные процессы обрели адресное пространство объемом 128 Тбайт, что существенно превышает имеющийся лимит физической памяти 4 Тбайт. Часть виртуальной памяти может находиться в оперативной памяти, а остальное хранится на диске в файле подкачки (или страничного обмена). Сплошные блоки виртуальной памяти могут быть не сплошными в физической памяти. Вся адресация памяти в процессе касается виртуальной памяти.

- *Зарезервированная память (Reserved Memory)* — область адресного пространства виртуальной памяти, зарезервированная для процесса, в силу чего она не будет выделяться другим запрашивающим память процессам. Зарезервированная память не может использоваться для удовлетворения запросов на выделение памяти, поскольку ее ничто не поддерживает, — это просто описание диапазона адресов памяти.
- *Выделенная память (Committed Memory)* — область памяти, имеющая физическое поддерживающее хранилище. Это может быть оперативная память или диск.
- *Страница (Page)* — организационная единица памяти. Блоки памяти выделяются в виде страницы объемом обычно несколько килобайт.
- *Страничный обмен (Paging)* — перемещение страниц между областями виртуальной памяти. Страницы перемещаются к другому процессу или от него (мягкий страничный обмен) или на диск и с диска (жесткий страничный обмен). Мягкий страничный обмен выполняется очень быстро путем отображения существующей памяти на виртуальное адресное пространство текущего процесса. Жесткий страничный обмен подразумевает относительно медленное перемещение данных на диск или с диска. Чтобы обеспечить хорошую производительность, программа должна любой ценой избегать такого обмена.
- *Загрузка страницы (Page In)* — перемещение страницы из другого места в текущий процесс.
- *Выгрузка страницы (Page Out)* — перемещение страницы из текущего процесса в другое место, например на диск.
- *Переключение контекста (Context Switch)* — процесс сохранения и восстановления состояния потока или процесса. Поскольку обычно число запущенных потоков превышает число доступных процессов, каждую секунду происходит множество переключений контекста. Это издержки в чистом виде, поэтому чем их меньше, тем лучше. Но определить, каким должно быть их оптимальное абсолютное значение, нелегко.

- *Режим ядра (Kernel Mode)* — режим, позволяющий операционной системе изменять низкоуровневые аспекты аппаратного состояния, например модифицировать конкретные регистры или разрешать/запрещать прерывания. Переход в режим ядра требует вызова операционной системы и может быть крайне затратным.
- *Пользовательский режим (User Mode)* — непривилегированный режим выполнения инструкций. В нем невозможно изменить низкоуровневые аспекты системы.

Некоторые из этих терминов будут использоваться по всей книге, особенно в главе 2, когда речь пойдет о сборке мусора. Дополнительную информацию по этим темам можно получить в изданиях, посвященных операционной системе, например в книге *Windows Internals* о ее внутреннем устройстве (см. библиографию в конце книги).

В категории счетчиков Process с помощью счетчиков с экземплярами для каждого процесса обрабатывается множество важной информации, включая:

- % Privileged Time (Процент работы в привилегированном режиме) — количество времени, затраченного на выполнение привилегированного кода (в режиме ядра);
- % Processor Time (Процент процессорного времени) — процент, относящийся к одному процессору, используемому приложением. Если ваше приложение задействует два логических процессорных ядра по 100 % каждое, показанием этого счетчика будет число 200;
- % User Time (Процент пользовательского времени) — количество времени, затраченного на выполнение непривилегированного кода (в режиме пользователя);
- IO Data Bytes/sec (Ввод-вывод данных в байтах в секунду) — объем ввода-вывода, выполняемого вашим процессом;
- Page Faults/sec (Количество отказов страниц в секунду) — общее количество отказов страниц в процессе. Отказ страницы возникает, когда страницу памяти невозможно найти в текущем рабочем наборе. Важно уяснить, что в это число входят отказы, произошедшие как по программным, так и по аппаратным причинам (соответственно soft page faults и hard page faults). Программный отказ страницы не наносит особого вреда и может быть вызван тем, что страница хранится в памяти, но находится за пределами текущего процесса (как, например, в случаях с общими DLL-библиотеками). Аппаратный отказ страницы создает более серьезную проблему и означает, что данные находятся на диске, но в текущий момент не в оперативной памяти. К сожалению, счетчики производительности не позволяют отслеживать количество аппаратных отказов страниц, приходящееся на каждый процесс, но, воспользовавшись счетчиком Memory\Page Reads/sec, можно увидеть их число для системы в целом. Вы можете выявить определенную взаимосвязь между общим количеством отказов страниц и общим количеством страниц, считанных системой (hard faults). Аппаратные отказы страниц можно четко отследить с помощью ETW-трассировки события Windows Kernel/Memory/Hard Fault;

- Pool Nonpaged Bytes (Пул невыгружаемых байтов, не разбиваемых на страницы) — обычно это выделенная операционной системой или драйверами память для структур данных, которые не могут быть выгружены (Paged Out), например для объектов операционной системы типа потоков или мьютексов, а также некоторых настраиваемых структур данных;
- Pool Paged Bytes (Пул выгружаемых байтов, разбиваемых на страницы) — создается также для структур данных операционной системы, с той лишь разницей, что они могут быть выгружены (Paged Out);
- Private Bytes (Приватные байты) — выделенная виртуальная память, принадлежащая конкретному процессу (не разделяемая с другими процессами);
- Virtual Bytes (Виртуальные байты) — выделенная память в адресном пространстве процесса, часть которой может быть поддержана страничным файлом, возможно используемая совместно с другими процессами или принадлежащая данному процессу;
- Working Set (Рабочий набор) — объем виртуальной памяти, который в настоящее время находится в физической памяти (обычно в оперативной памяти);
- Working Set-Private (Приватный рабочий набор) — объем приватных байтов, который в данное время находится в физической памяти;
- Thread Count (Счетчик потоков) — количество потоков в процессе. Может совпадать или не совпадать с количеством .NET-потоков. Сведения о счетчиках, относящихся к .NET-потокам, приводятся в главе 4.

Существует и еще несколько категорий, чья польза зависит от характера приложения. PerfMon можно применять для исследования имеющихся в них конкретных счетчиков.

- IPv4/IPv6 — счетчики для датаграмм и фрагментов, относящиеся к интернет-протоколу.
- Memory (Память) — общесистемные счетчики памяти, такие как счетчики общего страничного обмена, доступных байтов, выделенных байтов и многого другого.
- Objects (Объекты) — сведения об объектах, принадлежащих ядру, таких как события, мьютексы, процессы, потоки, семафоры и разделы.
- Processor (Процессор) — счетчики для каждого логического процессора в системе.
- System (Система) — переключатели контекста, исправления выравнивания данных, файловые операции, счетчик процессов, потоки и т. д.
- TCPv4/TCPv6 — сведения о TCP-подключениях и передачах сегментов.

Как ни странно, но найти в Интернете подробную информацию о счетчиках производительности весьма непросто, но, на наше счастье, они самодокументированы! В нижней части диалогового окна Add Counter (Добавить счетчик), имеющегося в PerfMon, можно установить флажок Show description (Демонстрация описания), в результате чего будут показаны подробности, относящиеся к выделенному счетчику.

В PerfMon имеется также возможность сбора конкретных счетчиков производительности в запланированное время и сохранения их данных в журнальных записях для последующего просмотра или даже выполнения пользователем какого-либо действия при преодолении счетчиком производительности определенного порога. Это делается с помощью наборов сборщиков данных — Data Collector Sets, которые не ограничиваются счетчиками производительности, но могут собирать также данные конфигурации системы и ETW-события.

Для установки набора сборщиков данных нужно в главном окне PerfMon выполнить следующие действия.

1. Раскрыть дерево Data Collector Sets (Наборы сборщиков данных).
2. Щелкнуть правой кнопкой мыши на пункте User Defined (Задаваемое пользователем).
3. Выбрать New (Создать).
4. Выбрать Data Collector Set (Набор сборщиков данных).
5. Задать имя набора, установить флажок Create manually (Advanced) (Создать вручную (Дополнительно)) (рис. 1.9) и нажать кнопку Next (Далее).

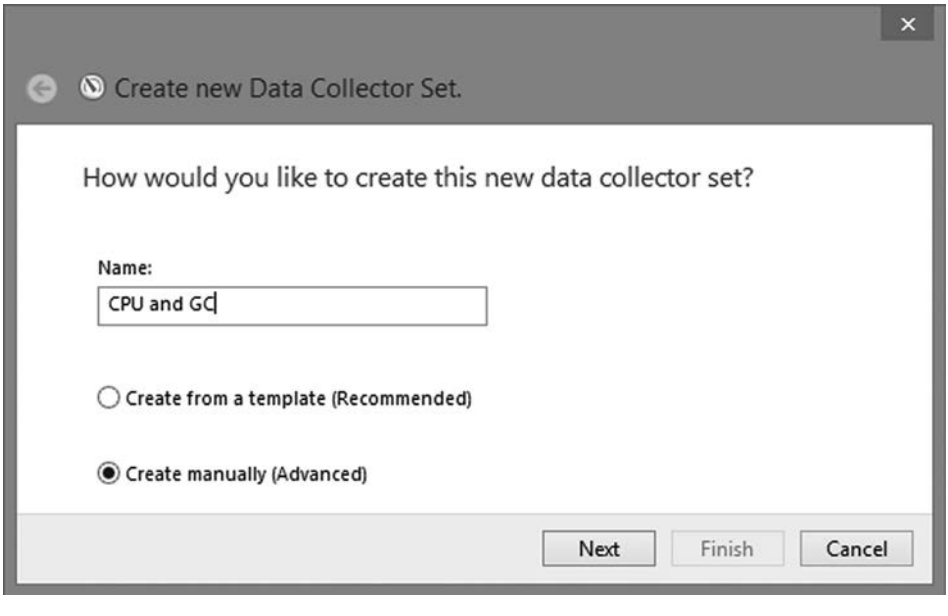

Рис. 1.9. Диалоговое окно конфигурации Data Collector Set (Набор сборщиков данных), предназначенное для создания обычных коллекций счетчиков

6. Установить флажок Performance counter (Счетчик производительности) в области Create data logs (Создать журналы регистрации данных) (рис. 1.10) и нажать кнопку Next (Далее).

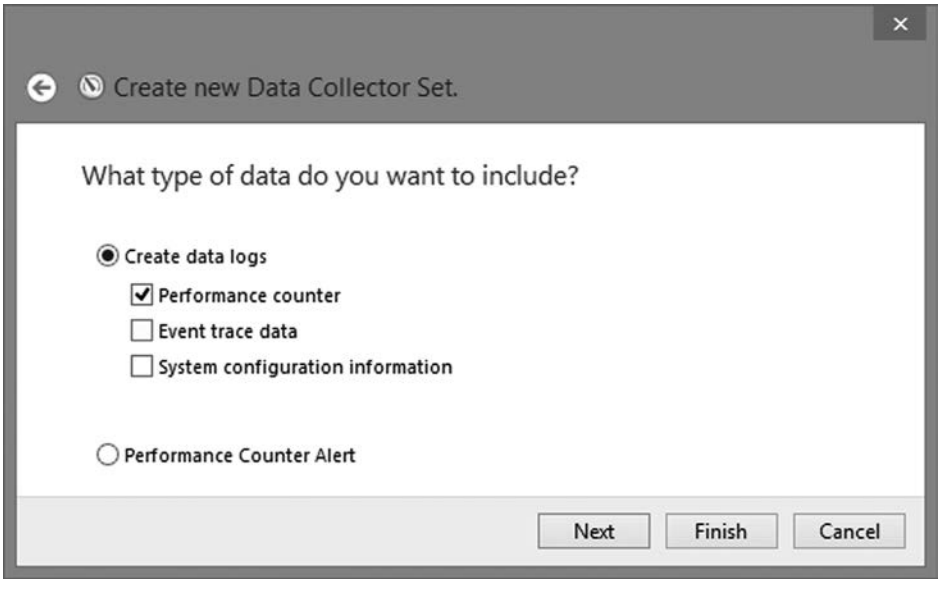

Рис. 1.10. Указания типа сохраняемых данных

7. Для выбора счетчиков, которые нужно добавить, нажать кнопку Add (Добавить) (рис. 1.11).

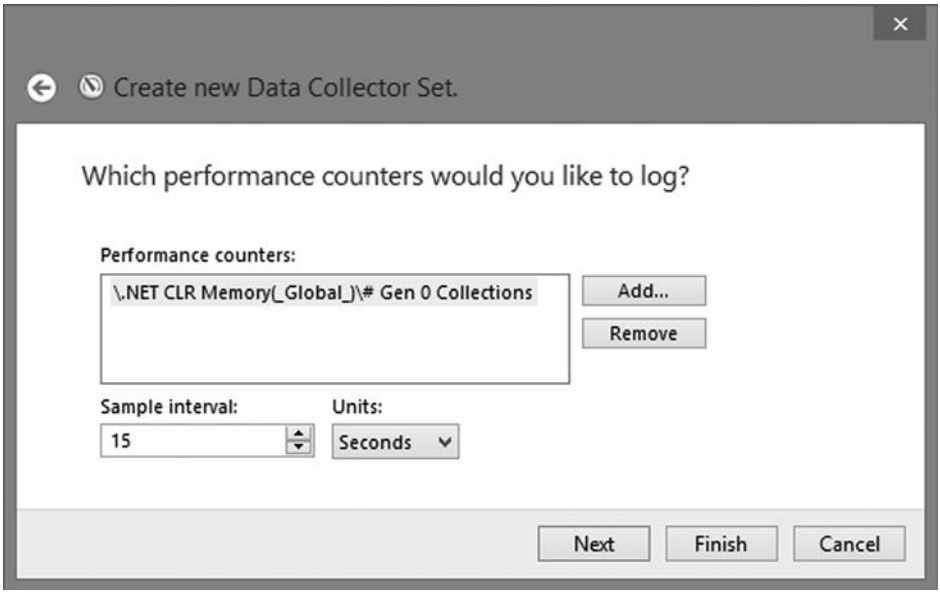

Рис. 1.11. Выбор собираемых счетчиков

8. Для задания пути сохранения журнальных записей нажать кнопку Next (Далее), для выбора информации, относящейся к безопасности, — еще раз кнопку Next (Далее) (рис. 1.12).

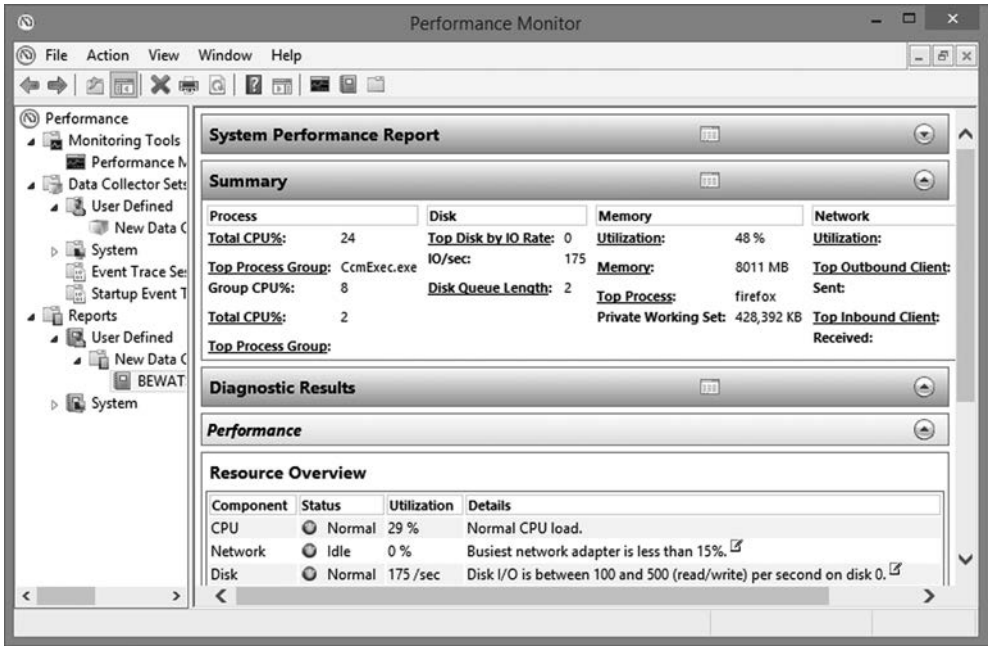

Рис. 1.12. Сохраненный файл отчета. Чтобы перейти к отображению зафиксированных данных счетчика в виде графика, следует воспользоваться кнопками инструментальной панели

После этого появится возможность открыть свойства набора собираемой информации и установить план сбора. Запуск можно произвести вручную, щелкнув правой кнопкой мыши на узле задания и выбрав в появившемся меню пункт Start (Начать). В результате будет создан отчет, просмотреть который можно будет после двойного щелчка на его узле, расположенном в основном дереве в списке Reports (Отчеты).

Для создания уведомления нужно проделать те же действия, но выбрать в мастере Wizard пункт Performance Counter Alert (Предупреждения счетчика производительности).

Вероятно, то, что вам нужно проделать со счетчиками производительности, можно выполнить с использованием рассмотренных здесь функциональных возможностей. Если требуется управлять ими программным способом или создать собственные счетчики, то за подробностями следует обратиться к главе 7. Анализ показаний счетчиков производительности можно рассматривать в качестве основы всей работы по повышению производительности приложения.

ETW-события

Механизм трассировки событий Windows (Event Tracing for Windows, ETW) является в Windows одним из основных строительных блоков для всего диагностического логирования и имеет отношение не только к производительности. В этом подразделе будет представлен обзор ETW-событий, а глава 8 научит вас тому, как создавать и отслеживать ваши собственные события.

События создаются поставщиками. Например, в CLR имеется поставщик Runtime, создающий большинство интересующих нас событий. Существуют поставщики практически для каждой подсистемы, имеющейся в Windows, к числу которых относятся центральный процессор, диск, сеть, брандмауэр, память и многое-многое другое. Подсистема ETW чрезвычайно эффективна и способна с минимальными издержками справиться с созданием невероятного количества событий.

С каждым событием связаны стандартные поля, такие как уровень события и ключевые слова.

Уровни имеют следующие значения:

- `0x5` — `Verbose` (подробный);
- `0x4` — `Informational` (информационный);
- `0x3` — `Warning` (предупреждение);
- `0x2` — `Error` (ошибка);
- `0x1` — `Critical` (критический);
- `0x0` — `LogAlways` (регистрировать всегда).

Каждый поставщик может определить собственные ключевые слова. Имеющийся в CLR поставщик Runtime оперирует ключевыми словами для сборщика мусора, компиляции времени исполнения, безопасности, взаимодействия с COM (Interop), соперничества за ресурсы и многого другого. Ключевые слова позволяют фильтровать события, которые нужно отслеживать.

У каждого события имеется также собственная структура данных, определенная его поставщиком, в которой дается описание состояния некоторого поведения. Например, поставляемые Runtime события сборщика мусора будут содержать сведения о поколении текущей сборки, о том, происходила ли она в фоновом режиме, и т. д.

Многие компоненты Windows создают огромное количество событий, описывающих чуть ли не каждый аспект выполнения приложения на всех уровнях архитектуры, поэтому вы можете лишь с помощью событий ETW выполнять существенный объем анализа производительности. Это делает ETW очень мощным инструментом.

Существует немало инструментов, обрабатывающих ETW-события и по-своему, исходя из своих целей, визуализирующих их. Фактически, начиная с Windows 8,

профилирование центрального процессора всегда выполняется с помощью ETW-событий (рис. 1.13).

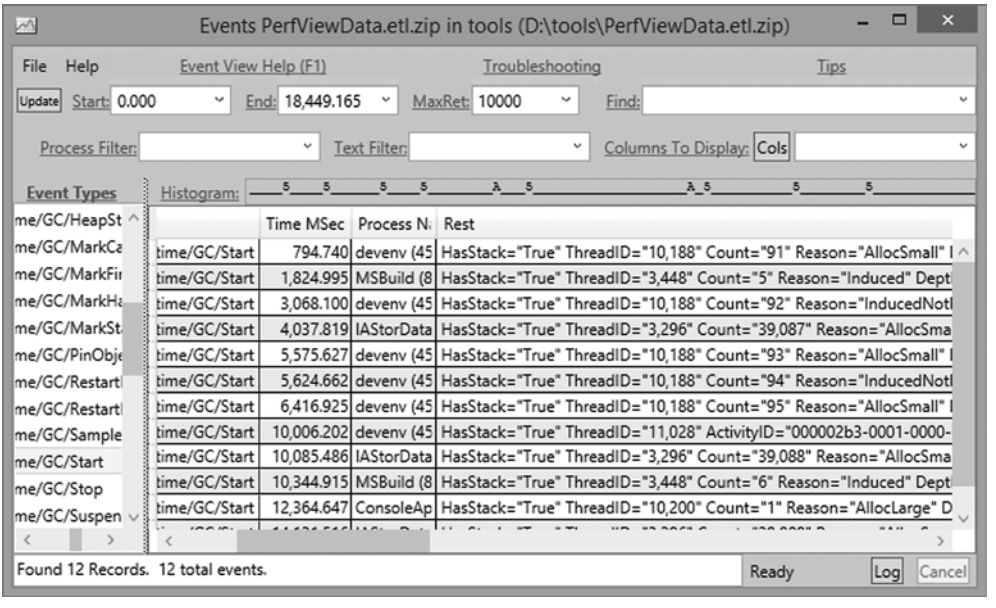

Рис. 1.13. Перечень всех событий GC Start (запуск сборки мусора), произошедших за 60 с. Обратите внимание на разнообразные данные, связанные с событием, например, на Reason (Причина) и Depth (Глубина)

Чтобы увидеть перечень всех ETW-поставщиков, зарегистрированных в вашей системе, откройте окно командной строки и запустите в нем следующую команду:

```
logman query providers
```

В результате появится весьма объемный поток информации, похожий на следующий:

```
Provider                                          GUID
-------------------------------------------------------------------------------
.NET Common Language Runtime                      {E13C0D23-CCBC-4E12-931B...
ACPI Driver Trace Provider                        {DAB01D4D-2D48-477D-B1C3...
Active Directory Domain Services: SAM             {8E598056-8993-11D2-819E...
Active Directory: Kerberos Client                 {BBA3ADD2-C229-4CDB-AE2B...
Active Directory: NetLogon                        {F33959B4-DBEC-11D2-895B...
ADODB.1                                           {04C8A86F-3369-12F8-4769...
ADOMD.1                                           {7EA56435-3F2F-3F63-A829...
Application Popup                                 {47BFA2B7-BD54-4FAC-B70B...
Application-Addon-Event-Provider                  {A83FA99F-C356-4DED-9FD6...
...
```

Можно также получить подробности относительно ключевых слов поставщика:

```
D:\>logman query providers "Windows Kernel Trace"

Provider                                GUID
-------------------------------------------------------------------
Windows Kernel Trace                    {9E814AAD-3204-11D2-9A82...

Value               Keyword             Description
-------------------------------------------------------------------
0x0000000000000001  process             Process creations/deletions
0x0000000000000002  thread              Thread creations/deletions
0x0000000000000004  img                 Image load
0x0000000000000008  proccntr            Process counters
0x0000000000000010  cswitch             Context switches
0x0000000000000020  dpc                 Deferred procedure calls
0x0000000000000040  isr                 Interrupts
0x0000000000000080  syscall             System calls
0x0000000000000100  disk                Disk IO
0x0000000000000200  file                File details
0x0000000000000400  diskinit            Disk IO entry
0x0000000000000800  dispatcher          Dispatcher operations
0x0000000000001000  pf                  Page faults
0x0000000000002000  hf                  Hard page faults
0x0000000000004000  virtalloc           Virtual memory allocations
0x0000000000010000  net                 Network TCP/IP
0x0000000000020000  registry            Registry details
0x0000000000100000  alpc                ALPC
0x0000000000200000  splitio             Split IO
0x0000000000800000  driver              Driver delays
0x0000000001000000  profile             Sample based profiling
0x0000000002000000  fileiocompletion    File IO completion
0x0000000004000000  fileio              File IO
```

К сожалению, хорошего онлайн-ресурса, объясняющего, какие именно события есть у того или иного поставщика, не существует. К числу наиболее распространенных ETW-событий для всех процессов Windows относятся следующие, относящиеся к категории трассировки ядра операционной системы (Windows Kernel Trace):

- Memory/Hard Fault (Память/Аппаратный отказ страницы);
- DiskIO/Read (Дисковый ввод-вывод/Чтение);
- DiskIO/Write (Дисковый ввод-вывод/Запись);
- Process/Start (Процесс/Запуск);
- Process/Stop (Процесс/Остановка);
- TcpIp/Connect (TcpIp/Подключение);
- TcpIp/Disconnect (TcpIp/Отключение);
- Thread/Start (Поток/Запуск);
- Thread/Stop (Поток/Остановка).

Для того чтобы увидеть другие события от этого или же других поставщиков, можно самостоятельно собрать ETW-события и проанализировать полученный набор.

Далее я буду упоминать важные события, на которые необходимо обратить внимание при анализе набора записей ETW, в частности создаваемые CLR-поставщиком Runtime. Подробную документацию по CLR ETW можно найти по адресу https://docs.microsoft.com/dotnet/framework/performance/etw-events-in-the-common-language-runtime.

PerfView

Собирать и анализировать ETW-события способны многие инструментальные средства, но PerfView, созданное разработчиком архитектуры производительности Microsoft .NET (и человеком, написавшим предисловие к данной книге) Вэнсом Моррисоном (Vance Morrison), одно из самых эффективных. Все ранее показанные копии экрана с ETW-событиями взяты именно из него.

Средство PerfView создано на основе механизма обработки ETW под названием TraceEvent, которым вы можете воспользоваться и самостоятельно (см. главу 8). Реальная польза от применения PerfView кроется в его исключительно эффективном механизме группировки и свертки стека данных, позволяющем углубиться в события на нескольких уровнях абстракции.

Из других инструментов, позволяющих проводить анализ ETW, также можно извлечь немалую пользу, но я зачастую отдаю предпочтение средству PerfView, и вот почему.

1. Оно не требует установки, поэтому легко запускается на любом компьютере.
2. У него огромное количество настроек, и можно легко создавать и сохранять типовые сценарии (скрипты) работы с ним.
3. Можно выбрать, за какими событиями следить, с весьма высокой степенью гранулярности, что, к примеру, позволяет провести многочасовое отслеживание всего лишь нескольких категорий событий.
4. Обычно оно весьма незначительно влияет на отслеживаемые машину или процесс.
5. У него имеются несравненные аналитические возможности, проявляющиеся посредством расширенной группировки и свертки стека данных.
6. Можно настроить PerfView под себя, создавая расширения для собственной аналитики и используя при этом преимущества встроенных средств группировки и свертки стека данных.
7. У него имеется встроенная возможность просмотра исходного кода, включая исходный код среды .NET Framework.
8. Оно обладает сложной аналитикой асинхронных вызовов, задействующих библиотеку параллельно выполняемых задач (Task Parallel Library).
9. У него есть поддержка для IIS и ASP.NET.

А вот как звучат наиболее распространенные вопросы, на которые я обычно нахожу ответы с помощью PerfView.

- На что расходуются ресурсы моего ЦП?
- На что выделяется основной объем памяти?
- Какие типы памяти выделяются чаще всего?
- Чем вызвана сборка мусора в Gen 2?
- Какова протяженность среднестатистической сборки Gen 0?
- Каков объем JIT-компиляции, приходящийся на мой код?
- Какие блокировки чаще всего вызывают соперничество за ресурсы?
- На что похожа моя управляемая куча?

Для сбора и анализа событий с помощью PerfView нужно выполнить следующие основные действия.

1. Выбрать из меню Collect (Сбор) одноименный пункт.
2. В появившемся диалоговом окне указать нужные параметры:
   - раскрыть пункт Advanced Options (Дополнительные параметры), чтобы сузить круг типов событий, данные которых нужно отследить;
   - установить флажок No V3.X NGEN Symbols (Не использовать символы V3.X NGEN), если не применяется .NET 3.5;
   - для автоматической остановки сбора по наступлении заданного времени дополнительно указать значение параметра Max Collect Sec.
3. Нажать кнопку Start Collection (Приступить к сбору).
4. Если значение Max Collect Sec не используется, по завершении сбора нажать кнопку Stop Collection (Остановить сбор).
5. Дождаться обработки событий.
6. Выбрать нужное представление из созданного в результате обработки дерева.

При сборе событий PerfView захватывает ETW-события для всех процессов. По завершении сбора есть возможность отфильтровать события для каждого процесса.

Сбор событий не обходится без издержек. Собирать определенные категории событий более накладно, чем другие. Например, профилирование ЦП создает огромную массу событий, поэтому время профилирования нужно строго ограничивать (до 1–2 мин), или же вы получите файлы объемом несколько гигабайт, которые не сможете проанализировать.

Интерфейс и представления данных в PerfView

Большинство вариантов представления данных в PerfView — это вариации одного-единственного типа, поэтому будет полезно разобраться в том, как он работает.

Средство PerfView является по большому счету агрегатором стека данных и просмотрщиком. При записи ETW-событий для каждого из них записывается стек. В PerfView эти стеки анализируются и показываются в таблице, имеющей общий

вид для ЦП, выделения памяти, конфликтов блокировки, выдачи исключений и большинства других типов событий. Принципы, усвоенные вами при выполнении одного типа исследования, применимы и к другим типам, поскольку анализ стека одинаков.

Вам также нужно усвоить концепции группировки и свертки. Группировка превращает несколько источников в единый объект. Например, имеется несколько DLL-библиотек .NET Framework, а в каждой DLL — конкретная функция, обычно не представляющая интереса для профилирования. Используя группировку, можно определить ее шаблон, например `System.*!=>LIB`, который объединяет все сборки `System.*.dll` в одну группу под названием `LIB`. Это один из исходных шаблонов группировки, применяемый PerfView. Если, к примеру, нужно свернуть все вызовы методов в классе `TimeZoneInfo`, можно воспользоваться группой, которая определена следующим образом:

```
mscorlib.ni!System.TimeZoneInfo*->TIMEZONE
```

Это заставит появляться записи `TIMEZONE` во всем вашем стеке вместо любых методов `TimeZoneInfo`.

Свертка позволяет скрывать некоторые несущественные подробности располагающихся ниже уровней кода путем подсчета затрат на его выполнение в вызывающих его узлах. В качестве простого примера вспомните, что выделение памяти всегда происходит через какой-нибудь внутренний CLR-метод, вызываемый оператором `new`. По сути, вам нужно узнать, какие типы несут наибольшую ответственность за это выделение. Свертка позволяет приписать такие внутренние затраты порождающим их типам, то есть коду, который вы можете реально контролировать. Например, в большинстве случаев вам все равно, на какие внутренние операции затрачивается время внутри `String.Format`, — вас в первую очередь интересует вопрос, какие области кода вызывают `String.Format`. PerfView может свернуть такие операции в источник вызова (caller), чтобы дать вам более понятное представление о производительности вашего кода (рис. 1.14–1.17).

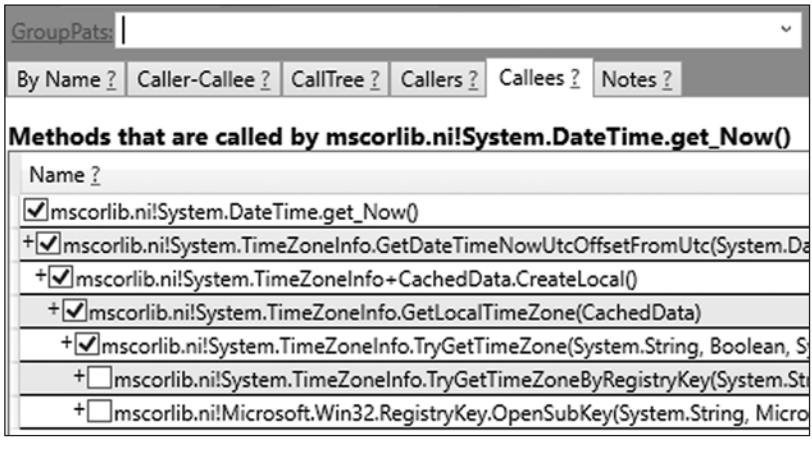

Рис. 1.14. Без группировки внутри класса TimeZoneInfo получается несколько уровней вызовов

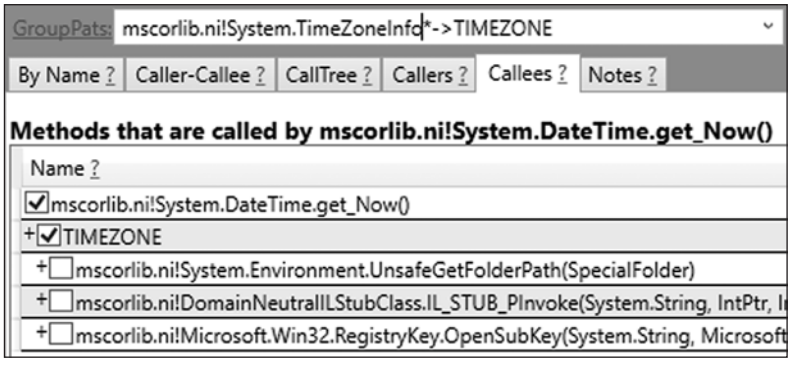

Рис. 1.15. Группировка позволяет скрыть подробности и представить все в виде одной записи в стеке

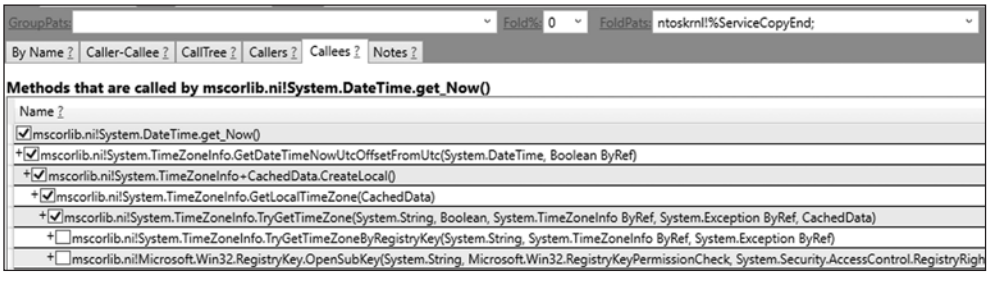

Рис. 1.16. Вызов DateTime.Now включает глубокую цепь вызовов методов TimeZoneInfo

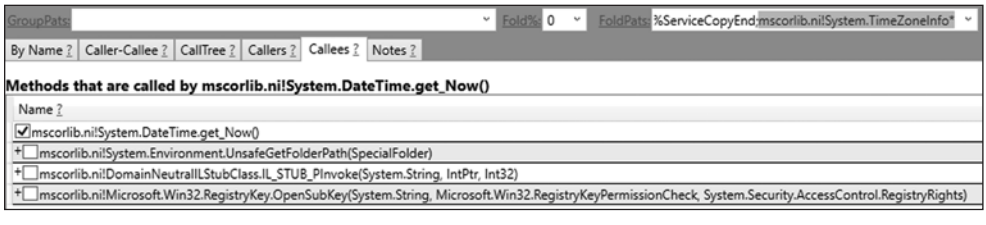

Рис. 1.17. Путем свертки с использованием шаблона mscorlib.ni!System.TimeZoneInfo* все затраты на вызовы таких методов будут представлены как затраты на вызов DateTime.Now

В шаблонах свертки могут использоваться группы, определенные вами для группировки. Например, можно просто указать шаблон свертки `LIB`, обеспечивающий то, что все методы в `System.*` будут отнесены к методу, который их вызывает из-за пределов `System.*`.

Пользовательский интерфейс просмотрщика стека также требует некоторых кратких пояснений (рис. 1.18).

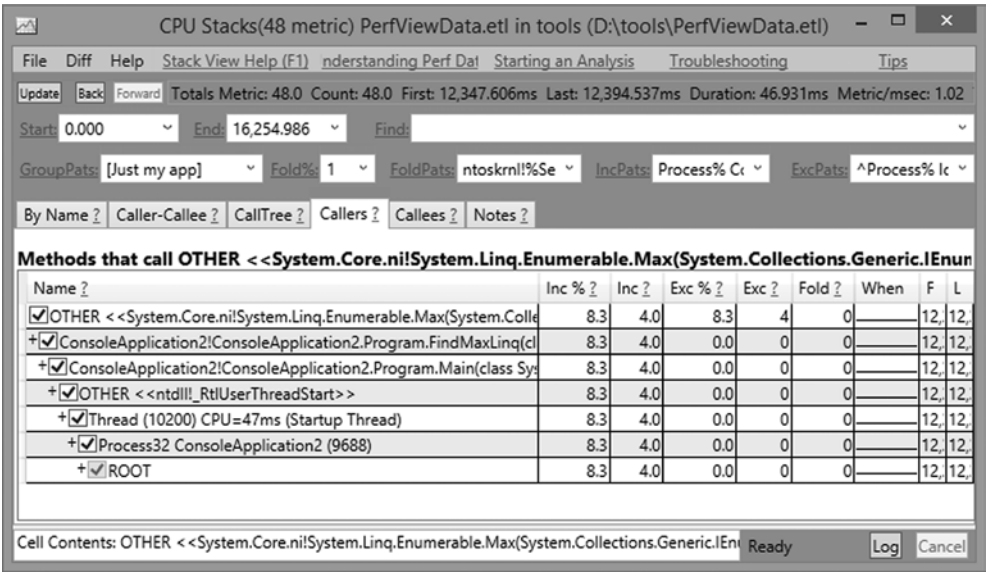

Рис. 1.18. Обычное представление стека в PerfView. В пользовательском интерфейсе имеется множество параметров фильтрации, сортировки и поиска

Расположенные в верхней части окна элементы управления позволяют упорядочить представление стека несколькими способами. Далее приводится краткая информация об их использовании, но вы можете щелкнуть на символе ? в заголовках столбцов, вызвав файл справки, содержащий дополнительные сведения:

- Start (Начало) — время начала исследуемого периода (в микросекундах);
- End (Конец) — время завершения исследуемого периода (в микросекундах);
- Find (Найти) — текст для поиска;
- GroupPats (Шаблоны группировки) — список шаблонов группировки с точкой с запятой в качестве разделителя;
- Fold% (Процент свертки) — любой стек, на который приходится доля потребления, меньшая, чем указанный процент, будет свернут в свой родительский элемент;
- FoldPats (Шаблоны свертки) — список шаблонов свертки с точкой с запятой в качестве разделителя;
- IncPats (Шаблоны включения) — в анализ должны попасть только стеки, соответствующие данному шаблону. Обычно там содержится имя процесса;
- ExcPats (Шаблоны исключения) — из анализа будут исключены стеки, соответствующие данному шаблону. Изначально в этом поле содержится только процесс `Idle`.

Существует несколько разных вкладок просмотра:

- By Name (По имени) — показывает каждый узел, будь то тип, метод или группа. Хорошо подходит для анализа, проводимого снизу вверх;
- Caller-Callee (Вызывающий — вызываемый) — фокусируется на одном узле, показывая вызывающие и вызываемые этого узла;
- CallTree (Дерево вызовов) — показывает дерево всех узлов в профиле, начиная с ROOT. Хорошо подходит для проведения анализа сверху вниз;
- Callers (Вызывающие) — показывает все вызывающие конкретный узел;
- Callees (Вызываемые) — показывает все вызванные методы конкретного узла;
- Notes (Заметки) — позволяет сохранять заметки по исследованию в самих ETL-файлах.

В табличном представлении имеется несколько столбцов. Для получения дополнительной информации следует щелкать кнопкой мыши на их названиях. Далее представлена краткая информация о наиболее важных столбцах:

- Name (Имя) — тип, метод или заданное пользователем имя группы;
- Exc % (Исключающий процент) — процент исключительных затрат. Для отслеживания памяти учитывается объем памяти, относящийся только к данному типу или методу. Для отслеживания ЦП учитывается количество времени ЦП, относящееся к данному методу;
- Exc (Исключающее количество) — сумма выборочных измерений только в данном узле, исключая дочерние узлы. Для отслеживания памяти учитывается количество байтов, относящихся исключительно к данному узлу. Для отслеживания ЦП учитывается количество затраченного здесь времени (в миллисекундах);
- Exc Ct (Исключающее количество выборок) — количество выборок только в данном узле;
- Inc % (Включающий процент) — процент затрат на данный тип или метод и на все его дочерние элементы. Всегда не меньше Exc %;
- Inc (Включающее количество) — затраты на данный узел, включая все дочерние элементы. При замере использования ЦП учитывается количество времени ЦП, затраченное в данном узле и во всех его дочерних элементах;
- Inc Ct (Включающее количество выборок) — количество выборок в данном узле и во всех его дочерних элементах.

В последующих главах будут даны инструкции по решению конкретных проблем с помощью различных типов исследований производительности. Полный обзор PerfView заслуживает отдельной книги или по крайней мере очень подробного справочного файла, который, по счастливой случайности, поставляется вместе с PerfView. Я настоятельно рекомендую прочитать представленное

в нем руководство, как только будет освоено проведение нескольких простых анализов.

Может сложиться впечатление, что средство PerfView предназначено главным образом для анализа памяти или ЦП, но не стоит забывать, что фактически это просто универсальная программа структуризации и объединения стеков данных, а такие стеки могут поступать от любого ETW-события. Она способна анализировать ваши источники конфликтов при блокировках, дисковый ввод-вывод или любое произвольное событие приложения, сохраняя при этом мощь группировок и сверток.

Профилировщик CLR Profiler

Средство CLR Profiler — это потенциальная альтернатива возможностям анализа памяти, предоставляемым PerfView, когда требуется получить графическое представление кучи и взаимоотношений между объектами. CLR Profiler способен продемонстрировать множество подробностей, например:

- график выделения памяти программой и цепочку методов, ведущих к конкретному выделению;
- гистограммы выделенных, перемещенных и финализированных объектов по размеру и типу;
- гистограмму объектов по времени существования;
- хронологию выделения объектов и сборок мусора, показывающую изменения кучи во времени;
- графическое представление объектов по их адресам в виртуальной памяти, что довольно легко может продемонстрировать фрагментацию.

Я редко пользуюсь средством CLR Profiler из-за некоторых присущих ему ограничений и возраста, но временами оно еще может принести определенную пользу. У него уникальные средства визуализации, которых нет у других свободно распространяемых инструментов. Это средство поставляется в виде двоичных файлов для 32- и 64-разрядных систем в сопровождении документации и исходного кода.

Вот базовая инструкция по использованию CLR Profiler.

1. Выберите правильную версию для запуска — 32- или 64-разрядную — в зависимости от целевой программы. Профилирование 32-разрядной программы невозможно выполнить с помощью 64-разрядного профилировщика, и наоборот.
2. Установите флажок Profiling active (Активизация профилирования).
3. Дополнительно можно установить флажки Allocations (Распределения) и Calls (Вызовы).

4. При необходимости перейдите к настройкам, которые становятся доступными при выборе пункта меню File ▸ Set Parameters (Файл ▸ Установка параметров), чтобы настроить параметры командной строки, рабочий каталог, каталог для файлов с логами и т. п.
5. Нажмите кнопку Start Application (Запустить приложение).
6. Перейдите к приложению, профиль которого нужно получить, и нажмите кнопку Open (Открыть).

Это позволит запустить приложение с активизированным профилировщиком. После профилирования следует выйти из программы или выбрать в CLR Profiler действие Kill Application (Закрыть приложение). Оно остановит выполнение профилируемого приложения и запустит обработку собранных при логировании записей. Обработка займет довольно много времени, которое зависит от продолжительности профилирования (иногда мне приходилось дожидаться ее окончания более часа).

В ходе профилирования в CLR Profiler можно нажать кнопку Show Heap now (Показать кучу сейчас). Это заставит профилировщик получить дамп кучи и открыть результаты в визуальном представлении взаимодействий объектов. Профилирование продолжится без какого-либо прерывания, то есть дампы можно получать множество раз в различные моменты выполнения приложения.

Как только профилирование завершится, будет показан экран основных результатов (рис. 1.19).

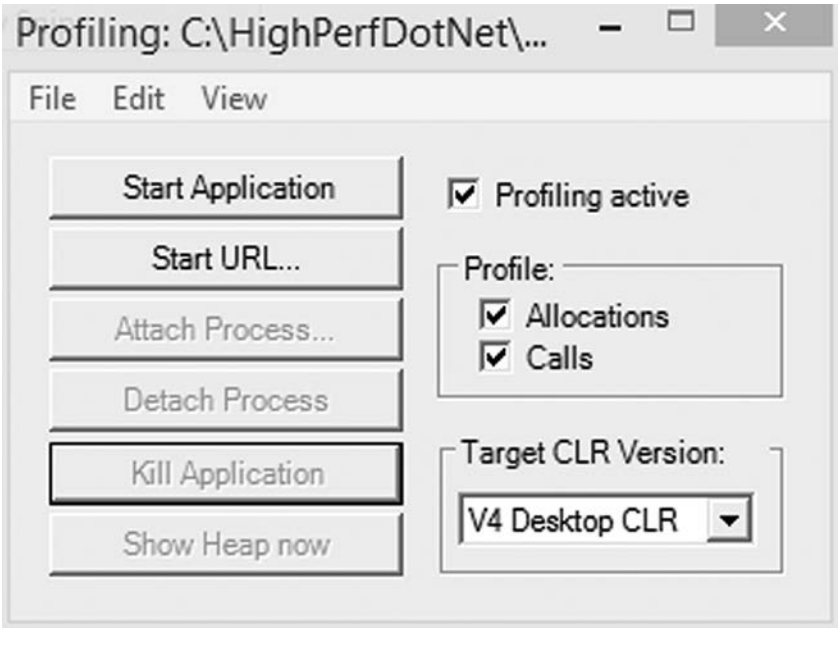

Рис. 1.19. Главное окно CLR Profiler

С этого экрана можно получить доступ к различным представлениям данных из кучи. Чтобы увидеть некоторые наиболее важные возможности, начните с графика выделения памяти (Allocation Graph) и хронологии (Time Line) (рис. 1.20). Как только вы освоитесь с анализом управляемого кода, представления в виде гистограмм также станут весьма ценным ресурсом.

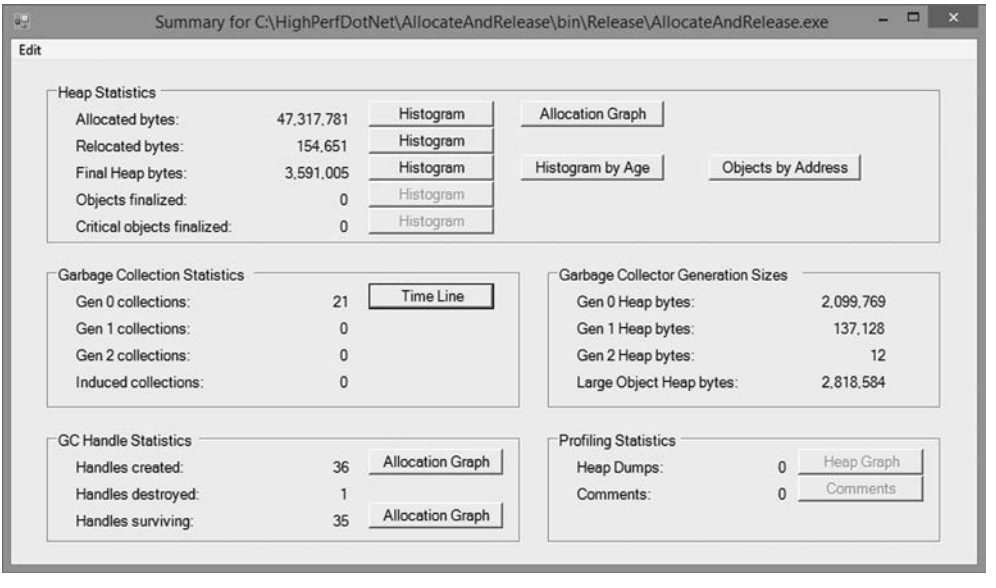

Рис. 1.20. Представление получившихся в CLR Profiler сводных данных, собранных в процессе отслеживания

ПРИМЕЧАНИЕ

В целом CLR Profiler проявляет себя неплохо, однако при его использовании я столкнулся с рядом серьезных проблем. Этот профайлер довольно привередлив. При некорректной настройке перед запуском профилирования он может выдавать исключения или внезапно прекращать работу. Например, для получения хоть каких-то данных мне всегда приходилось устанавливать флажки Allocations (Распределения) и Calls (Вызовы). Кнопку Attach to Process (Прикрепить к процессу) следует проигнорировать, так как она работает ненадежно. Похоже, что CLR Profiler плохо справляется с работой с очень большими приложениями с огромными кучами или большим количеством сборок. Если и вы столкнетесь с проблемами, лучше, наверное, будет воспользоваться PerfView, так как он тщательно проработан и обладает исключительной гибкостью благодаря весьма подробным параметрам командной строки, позволяющим управлять практически всеми аспектами его поведения. Конечно, это только мое мнение. В то же время CLR Profiler поставляется с собственным исходным кодом, следовательно, вы можете исправить его недостатки!

Анализатор производительности Windows Performance Analyzer

Windows Assessment and Deployment Kit (Windows ADK, являющийся частью Windows SDK) содержит несколько инструментов, помогающих развертывать операционные системы и приложения на компьютерах. В его составе имеются инструменты Windows Performance Recorder (средство записи параметров производительности) и Windows Performance Analyzer (анализатор производительности). Они обрабатывают ETW-события аналогично тому, как это делает PerfView. Однако анализатор Windows Performance Analyzer преуспевает в отображении информации уровня аппаратного обеспечения и операционной системы. Он может показать также .NET-события, но менее удобен в использовании, чем PerfView.

Для получения результатов отслеживания нужно вызвать Windows Performance Recorder и запустить запись (рис. 1.21).

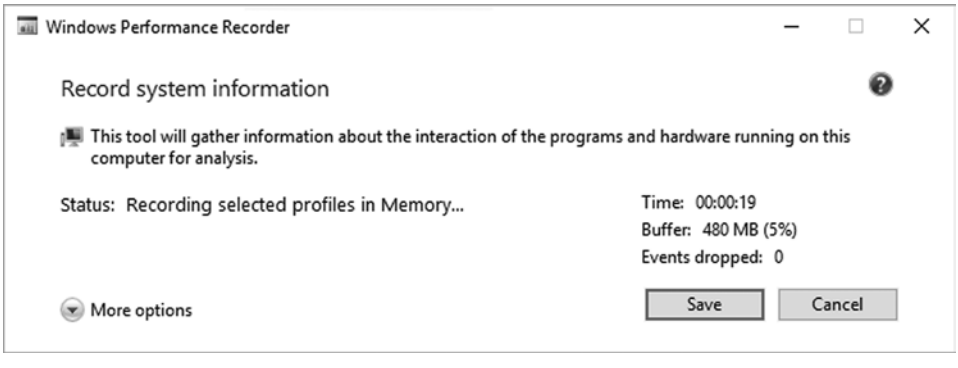

Рис. 1.21. Главный интерфейс Windows Performance Recorder. Для настройки на определенные виды записываемых событий нужно нажать кнопку More Options (Дополнительные настройки)

После записи событий нужно нажать кнопку Save (Сохранить), в результате чего появится интерфейс, где вы сможете ввести дополнительную информацию о текущей сессии профилирования, в то время как WPR будет обрабатывать записанные данные в фоновом режиме (рис. 1.22).

Файл записанных данных можно открыть в любом инструментальном средстве, способном проанализировать ETW-события, но здесь есть удобная кнопка, позволяющая сделать это непосредственно в анализаторе Windows Performance Analyzer.

Слева на экране Windows Performance Analyzer показан список категорий ресурсов. Двойной щелчок на ресурсе позволяет открыть детализированное представление с графиком и таблицей, содержащей подробности, относящиеся к выбранному ресурсу (рис. 1.23). Например, детальная информация об использовании памяти скомпонована таким образом, что вы сможете узнать о соотношении активной и переданной памяти, пуле страничной памяти, приватных страницах и многом другом.

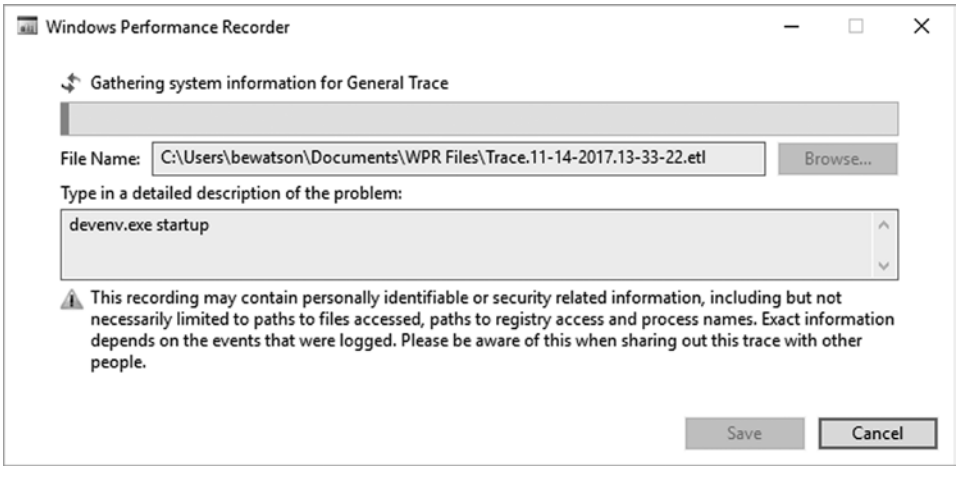

Рис. 1.22. На завершение процесса может понадобиться несколько минут

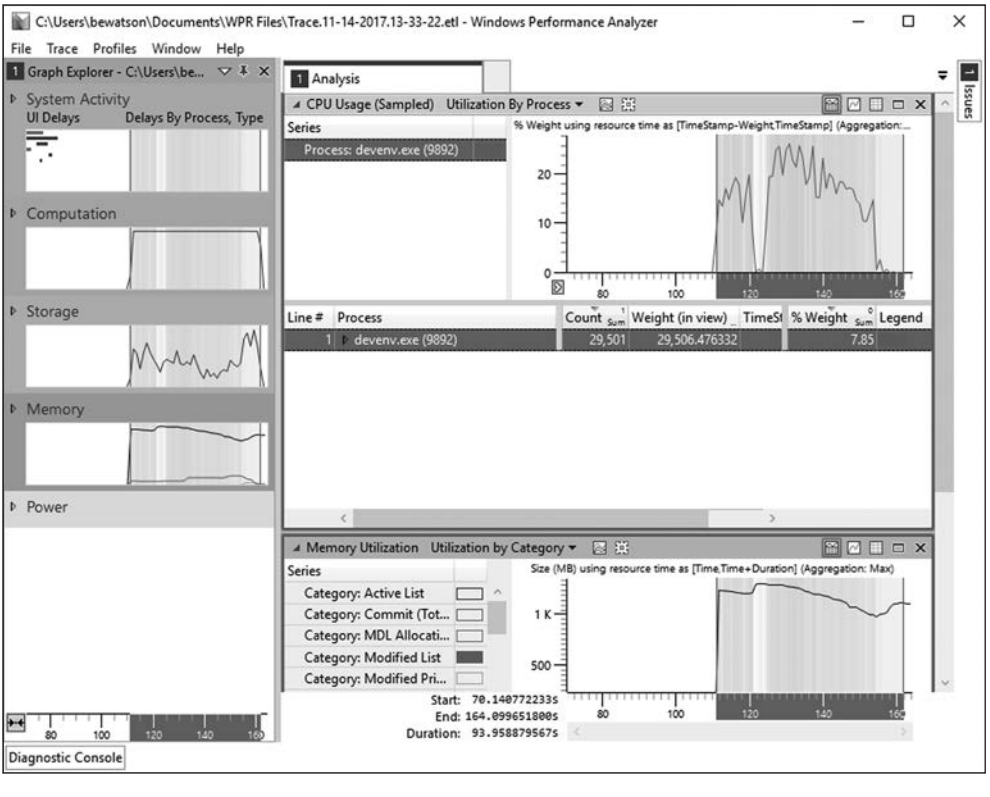

Рис. 1.23. Главный интерфейс анализатора Windows Performance Analyzer, отображающий записанные показатели операционной системы и аппаратного оборудования

Поскольку данный инструментарий больше ориентирован на использование основных ресурсов операционной системы, а не на вопросы, связанные с .NET, я не стану продолжать его обсуждение, но его возможности следует учесть, столкнувшись с некоторыми разновидностями проблем с производительностью.

WinDbg

Средство WinDbg представляет собой универсальный отладчик Windows Debugger, свободно распространяемый компанией Microsoft. Если вы привыкли пользоваться Visual Studio в качестве основного отладчика, применение этого не отягощенного излишествами текстового отладчика может показаться довольно сложным. Не поддавайтесь подобному настроению. Изучив всего несколько команд, вы освоитесь и вскоре практически откажетесь от использования для отладки Visual Studio, кроме как в процессе активной разработки приложений.

WinDbg намного мощнее Visual Studio и позволит вам исследовать процесс множеством недоступных без его применения способов. Он также не отличается особым потреблением ресурсов и намного легче развертывается на эксплуатационных серверах или клиентских машинах. Это весьма серьезный аргумент для того, чтобы освоить WinDbg. Но для управляемого кода сам по себе отладчик WinDbg не так интересен. Для эффективной работы с управляемыми процессами нужно будет воспользоваться имеющимися в .NET расширениями SOS, поставляемыми с каждой версией среды .NET Framework. Удобный краткий вариант справочника по SOS можно найти по адресу https://docs.microsoft.com/dotnet/framework/tools/sos-dll-sos-debugging-extension. Из Visual Studio тоже можно воспользоваться библиотекой SOS.dll, но разобраться в этом будет сложнее, а в знакомстве с WinDbg есть еще и другие преимущества, поэтому я остановлюсь именно на сценарии использования WinDbg вместе с SOS.

Применяя WinDbg совместно с SOS, можно довольно быстро получить ответы на следующие вопросы.

- Сколько объектов каждого типа находится в куче и насколько они велики?
- Насколько велика каждая из используемых мною куч и какую их долю составляет свободное пространство (какова степень фрагментации)?
- Какие объекты пережили сборку мусора?
- Какие объекты закреплены?
- Какие потоки расходуют больше всего времени ЦП? Имеется ли среди них поток, застрявший в бесконечном цикле?

Обычно WinDbg не первый используемый мной инструмент (зачастую в этой роли выступает PerfView), но нередко он становится вторым или третьим, позволяя увидеть то, что сложно заметить, работая с другими инструментами. Поэтому в данной книге показывать применение WinDbg буду довольно часто, чтобы вы

увидели, как нужно исследовать действия вашей программы, даже если другие инструменты справляются с этим быстрее или лучше (волноваться не стоит, такие инструменты тоже рассмотрим).

Не пугайтесь текстового интерфейса WinDbg. После использования нескольких команд, позволяющих заглянуть в ваш процесс, вы быстро освоитесь и по достоинству оцените скорость, с которой сможете анализировать программу. Изучение данной книги и разбор конкретных сценариев позволят понемногу приобрести нужные знания и навыки.

Чтобы получить WinDbg, следует установить Windows SDK. По желанию можно выбрать установку только отладчиков.

Чтобы начать работать с WinDbg, разберем в качестве обучающего примера небольшую программу. Она довольно проста — в ней будет очевидная и легко отлавливаемая утечка памяти. Программу можно найти в сопровождающем книгу исходном коде в проекте MemoryLeak (доступен по адресу http://www.writinghighperf.net).

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace MemoryLeak
{
  class Program
  {
    static List<string > times = new List<string >();

    static void Main(string[] args)
    {
      Console.WriteLine("Press any key to exit");
      while (!Console.KeyAvailable)
      {
        times.Add(DateTime.Now.ToString());
        Console.Write('.');
        Thread.Sleep(10);
      }
    }
  }
}
```

Запустите эту программу и дайте ей поработать несколько минут.

Запустите WinDbg из того места, куда вы его установили. Если это сделано через Windows SDK, то он должен быть в меню Пуск. Выберите правильную версию для запуска: x86 (для 32-разрядных процессов) или x64 (для 64-разрядных). Чтобы появилось диалоговое окно прикрепления к процессу, выберите в меню File ▸ Attach to Process (Файл ▸ Прикрепить к процессу) или нажмите клавишу F6 (рис. 1.24).

Найдите в этом окне процесс `MemoryLeak` (возможно, легче будет сделать это, выбрав опцию сортировки по имени исполняемого файла By Executable). Нажмите кнопку OK.

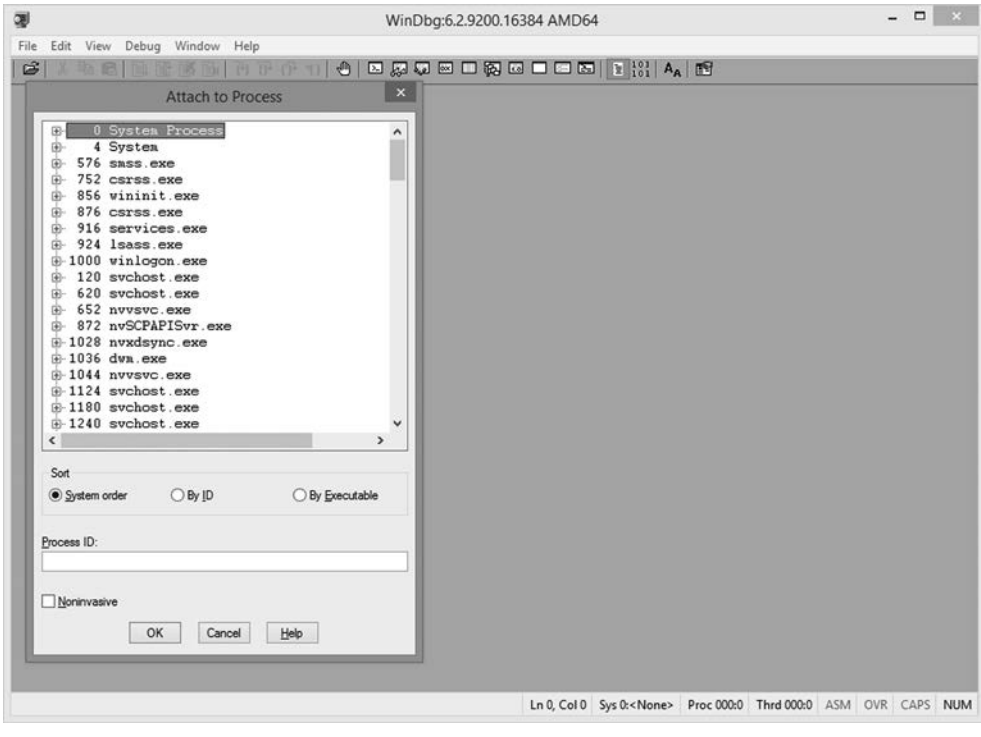

Рис. 1.24. Окно прикрепления к процессу, появляющееся в WinDbg

WinDbg приостановит процесс (об этом важно знать при отладке реального производственного процесса) и покажет все загруженные модули. На данном этапе отладчик станет дожидаться вашей команды. Обычно в первую очередь требуется загрузить отладочные расширения CLR. Введите следующую команду:

```
.loadby sos clr
```

Если она будет выполнена успешно, на экране ничего не появится.

Если же вы получите сообщение об ошибке следующего содержания: Unable to find module 'clr' (Невозможно найти модуль 'clr'), то, скорее всего, среда CLR еще не загружена. Так бывает, если программа запущена из WinDbg и тут же вошла в режим отладки. В таком случае сначала следует установить контрольную точку на загрузке модуля CLR:

```
sxe ld clr
g
```

Первая команда устанавливает контрольную точку на загрузке модуля CLR. А команда `g` заставляет отладчик продолжить выполнение. После еще одной остановки CLR должен быть загружен, и теперь вы сможете загрузить SOS с помощью команды `.loadby sos clr` в соответствии с предыдущим описанием.

С этого момента появится возможность проделать множество действий. Рассмотрим несколько пробных команд.

```
!ProcInfo
```

В результате выполнения этой команды будет выведена отладочная информация общего характера о процессе в целом, включая установленные переменные среды окружения:

```
---------------------------------------
Environment
=::=::\
=C:=C:\WINDOWS\system32
...множество переменных среды окружения
---------------------------------------
Process Times
Process Started at: 2017 Nov 7 22:5:49.44
Kernel CPU time   : 0 days 00:00:00.01
User CPU time     : 0 days 00:00:00.01
Total CPU time    : 0 days 00:00:00.02
---------------------------------------
Process Memory
WorkingSetSize:   26572 KB    PeakWorkingSetSize:   26572 KB
VirtualSize:     717972 KB    PeakVirtualSize:     717972 KB
PagefileUsage:   566560 KB    PeakPagefileUsage:   566560 KB
---------------------------------------
44 percent of memory is in use.

Memory Availability (Numbers in MB)

                      Total      Avail
Physical Memory        4095       4095
Page File              4095       4095
Virtual Memory         4095       3783
```

А теперь более полезные команды.

```
g
```

Эта команда означает go (запуск) и приводит к продолжению выполнения кода. В ходе выполнения программы никакие команды вводить нельзя.

```
<Ctrl-Break>
```

Нажатие этого сочетания клавиш поставит выполняемую программу на паузу. Ее применяют после команды g (go) для возвращения контроля над отладкой.

```
.dump /ma d:\memorydump.dmp
```

Данная команда приведет к созданию полного дампа процесса в выбранном файле. Это позволит заняться отладкой состояния процесса позже, но, поскольку это моментальный снимок, разумеется, вы не сможете выполнять отладку хода дальнейшего выполнения программы.

```
!DumpHeap -stat
```

DumpHeap показывает сводку данных обо всех управляемых объектах в куче, включая их размер (только для самого объекта, но не для объектов, на которые он ссылается), количество и др. Если нужно увидеть в куче каждый объект типа `System.String`, наберите команду `!DumpHeap -type System.String`. Более подробно она будет рассматриваться при изучении сборки мусора.

```
~*kb
```

Это обычная команда WinDbg, не имеющая отношения к SOS. Она выводит текущий стек для всех потоков, имеющихся в процессе.

Для переключения с текущего потока на другой поток используется команда:

```
~32s
```

Она приведет к замене текущего потока на поток 32. Обратите внимание на то, что номера потоков в WinDbg не совпадают с идентификаторами потоков. WinDbg нумерует все потоки в процессе, чтобы легче было ссылаться на них, не обращая внимания на идентификатор, присвоенный операционной системой Windows или средой .NET.

```
!DumpStackObjects
```

Можно также воспользоваться сокращенной версией `!dso`. Эта команда приводит к выводу адреса и типа каждого объекта из всех стековых фреймов для текущего потока.

Заметьте, что все команды, находящиеся в отладочном расширении SOS, предназначенном для управляемого кода, предваряются символом `!`.

Для эффективной работы с отладчиком нужно выполнить еще одно действие — установить используемый вами путь поиска символов, чтобы загрузить общедоступные символы для DLL-библиотек от Microsoft, что даст возможность увидеть все происходящее на системном уровне. Установите для своей переменной среды окружения `NT SYMBOL PATH` следующее значение:

```
symsrv*symsrv.dll*c:\sym*http://msdl.microsoft.com/download/symbols
```

Замените фрагмент `c:\sym` предпочтительным путем локального символьного кэша и удостоверьтесь в том, что указанный каталог действительно создан. После установки значения для переменной среды окружения и WinDbg, и Visual Studio станут использовать этот путь для автоматической загрузки и кэширования общедоступных символов для системных DLL-библиотек. В ходе начальной загрузки визуализация символов может идти довольно медленно, но как только произойдет их кэширование, скорость существенно возрастет. Для автоматической установки пути к символам непосредственно на символьный сервер Microsoft и локальный кэш можно также воспользоваться командой `.symfix`:

```
.symfix c:\sym
```

Если вам не доводилось ранее пользоваться WinDbg, осваивайте его без страха и сомнений. Запомнив весьма скромный набор команд, вы в кратчайшие сроки добьетесь высокой продуктивности. Истинное мастерство использования WinDbg придет со временем и с накоплением опыта работы, но, пройдя этот путь, вы ни о чем

не пожалеете. Отладчик WinDbg позволяет выполнять разные виды анализа, которые очень трудны или вовсе невозможны в других отладчиках. Обратите внимание на раздел «Исследование памяти и сборки мусора» главы 2, в котором рассматривается решение проблем с памятью и приведено множество примеров применения WinDbg.

CLR MD

Приобретя начальный опыт использования WinDbg и оценки доступных широких возможностей этого отладчика, вы можете подумать: «Жаль, что я не могу получить доступ ко всему этому программным путем». К счастью, такая возможность есть! И ее дает библиотека Microsoft.Diagnostics.Runtime с открытым кодом (называемая также CLR MD), расположенная по адресу https://github.com/microsoft.clrmd (рис. 1.25). Она дает доступ ко многим функциональным возможностям, имеющимся в SOS.dll, путем задействования весьма удобного и легкого в применении API. Библиотека CLR MD разработана как довольно низкоуровневый API, позволяющий легко создавать надстройки над ним для предоставления расширенных функциональных возможностей. Фактически ряд функциональных возможностей PerfView разработан поверх CLR MD, поэтому, если PerfView не предоставляет вам именно того, в чем вы нуждаетесь, можете, как говорится, заглянуть под капот этой библиотеки и «допилить» все, что нужно, самостоятельно.

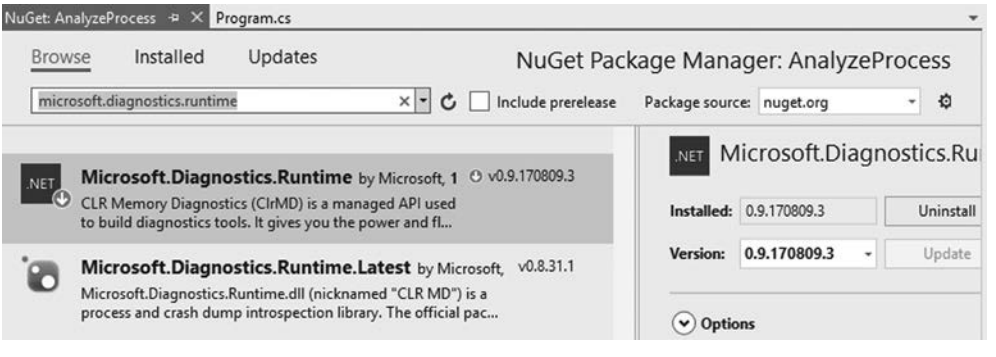

Рис. 1.25. Проще всего получить библиотеку через пакет NuGet с адреса http://NuGet.org. Там можно выполнить поиск по строке либо Microsoft.Diagnostics.Runtime, либо CLR MD

В данном подразделе мы кратко рассмотрим инструмент и порядок его использования, а конкретные решения проблем можно будет найти в соответствующих разделах книги.

ПРИМЕЧАНИЕ

Библиотека находится в весьма активной разработке, и между документацией и текущей реализацией можно будет заметить некоторые расхождения. Вероятно, API и в дальнейшем продолжит изменяться.

Эту библиотеку можно использовать как для прикрепления к действующим процессам (в качестве отладчика), так и для открытия файлов дампа кучи на диске. Оба примера будут показаны далее.

Для прикрепления к действующему процессу нужно лишь предоставить идентификатор процесса. В данном примере для удобства явным образом будет запущен новый процесс. Большинство примеров, приводимых в издании и относящихся к CLR MD, взяты из проекта учебного кода AnalyzeProcess, сопровождающего книгу.

```
static void Main(string[] args)
{
   // Создадим собственный процесс для тестирования
   var startInfo = new ProcessStartInfo(TargetProcessName);
   startInfo.CreateNoWindow = true;
   startInfo.WindowStyle = ProcessWindowStyle.Hidden;
   var targetProcess = Process.Start(startInfo);
   Thread.Sleep(1000);
   using (DataTarget target = DataTarget.AttachToProcess(
       targetProcess.Id,
       10000, // timeout
       AttachFlag.Invasive))
   {
       PrintDumpInfo(target);

       var clr = target.ClrVersions[0].CreateRuntime();
   }
}

private static void PrintDumpInfo(DataTarget target)
{
   PrintHeader("Target Info");

   Console.WriteLine($"Architecture: {target.Architecture}");
   Console.WriteLine($"Pointer Size: {target.PointerSize}");
   Console.WriteLine("CLR Versions:");
   foreach(var clr in target.ClrVersions)
   {
       Console.WriteLine($"\t{clr.Version}");
   }
}
```

Эта программа выведет на экран следующую информацию:

```
Target Info
===========
Architecture: X86
Pointer Size: 4
CLR Versions:
        v4.7.2115.00
```

Объект `clr`, полученный после вызова `PrintDumpInfo`, представляет собой основной интерфейс для большинства интересующих нас команд. Благодаря его

использованию можно, к примеру, осуществить последовательный перебор всех объектов, имеющихся в куче:

```
var heap = clr.Heap;
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
    Console.WriteLine(
      $"0x{obj.Address:x} - {obj.Type.Name}" +
      $" - Generation: {generation}");
}
```

В результате чего на экран будет выведена следующая информация:

```
0x30ec8ac - System.Byte[] - Generation: 0
0x30ecca0 - LargeMemoryUsage.B - Generation: 1
```

Кроме кучи, можно исследовать и код:

```
foreach(var module in clr.Modules)
{
    foreach (var type in module.EnumerateTypes())
    {
       foreach(var method in type.Methods)
       {
           Console.WriteLine(method.Name);
       }
    }
}
```

В результате будет выведена следующая информация:

```
Main
GetNewObject
.cctor
ToString
ToString
Equals
```

Можно также открыть аварийные дампы. Это дается чуть сложнее, так как приходится получать файл `mscordacwks.dll`, соответствующий версии (или версиям) CLR, присутствующим в дампе. При прикреплении к действующему процессу это не составляет труда, поскольку ее наличие на машине гарантировано. При работе с дампом с другой машины с потенциально совершенно другой версией CLR ее нужно получить с этой машины или загрузить с символьного сервера Microsoft. Как это сделать, показывает следующий код:

```
{
    ...
    string dacFile =
      GetDacFile(
        dataTarget.ClrVersions[0],
        dataTarget);
```

```
    var clr = dataTarget.ClrVersions[0].CreateRuntime(dacFile);
    ...
}

private static string GetDacFile(ClrInfo clrInfo ,
                                 DataTarget target)
{
    string location = clrInfo.LocalMatchingDac;
    if (string.IsNullOrEmpty(location) || !File.Exists(location))
    {
       // попытка загрузки с символьного сервера
       ModuleInfo dacInfo = clrInfo.DacInfo;
       try
       {
          location = target.SymbolLocator.FindBinary(dacInfo);
       }
       catch (WebException)
       {
          return null;
       }
    }
    return location;
}
```

Этот метод — эквивалент вызова `CreateRuntime` без аргументов, однако полезно будет узнать, как это сделать самостоятельно, на случай возникновения специфических потребностей.

Примеры эффективности использования данной библиотеки еще будут встречаться в дальнейших главах, но вкратце она способна сделать следующее:

- перечислить все объекты в куче и дать информацию об их создании, о том, закреплены ли они, и т. д.;
- предоставить инструменты для поиска корней и размеров объектов;
- перечислить сегменты памяти;
- выполнить последовательный обход всех методов в процессе;
- вычислить размер кода на промежуточном языке (IL) и кода аппаратной платформы.

ПРИМЕЧАНИЕ

При использовании этой библиотеки для исследования кода по-настоящему большой DLL я столкнулся с парой проблем. API в Microsoft.Diagnostics.Runtime зависят от внутренних API .NET, реализации которых могут быть не максимально эффективными. В одном случае я задействовал файл дампа для вычисления количества JIT-компиляций, произошедших в DLL объемом 500 Мбайт, имеющей 80 000 типов а также сотни тысяч методов. Я нажал Ctrl+Break по прошествии примерно 36 часов. Но это единственная DLL, с которой у меня были проблемы.

Анализаторы IL

Существует множество бесплатных и платных продуктов, способных получать скомпилированную сборку и выполнять ее декомпиляцию в IL, C#, VB.NET или какой-нибудь другой .NET-язык. К числу наиболее популярных можно отнести Reflector, ILSpy и dotPeek, но ими список не исчерпывается.

Их ценность заключается в способности показывать внутренние подробности кода, созданного другими людьми, что представляет интерес при качественном анализе производительности (рис. 1.26). Я пользуюсь ими чаще всего для изучения среды .NET Framework, когда хочу увидеть потенциальное влияние на производительность со стороны различных API.

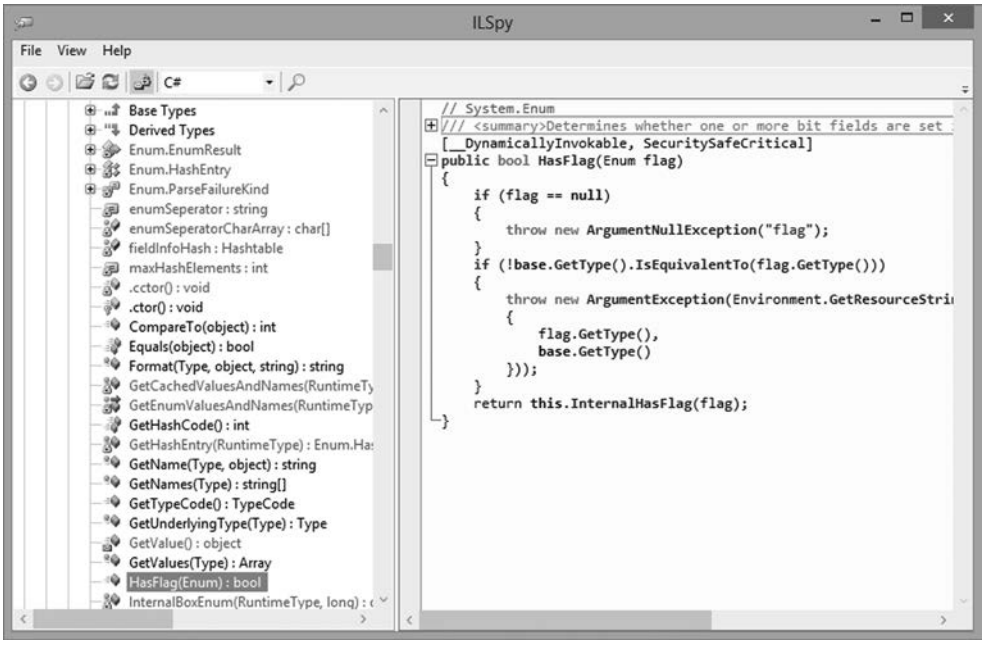

Рис. 1.26. ILSpy с результатом декомпиляции Enum.HasFlag в C#. Декомпиляторы — весьма эффективный инструмент для изучения работы и производительности кода, созданного сторонними разработчиками

Преобразование вашего собственного кода в читаемый IL также представляет определенную ценность, поскольку может показать многие операции, например упаковку (boxing), которые не видны в языках высокого уровня.

Код среды .NET Framework рассматривается в главе 6, где приводятся аргументы в пользу внимательного изучения каждого используемого вами API. Инструменты типа ILSpy, dotPeek и Reflector для этого просто жизненно необходимы, и вам довольно часто придется прибегать к ним, когда более тесно познакомитесь с имеющимся кодом. Объем работы, который придется выполнить с применением,

казалось бы, простых методов, зачастую будет для вас сюрпризом. Анализ сборок других разработчиков и компаний может научить вас работать более рационально на примере более или менее удачного выбора организации, проектного решения и практики программирования.

Эти инструменты способны показать вам и некоторые другие вещи:

- ссылки сборки;
- метаданные сборки, например целевую платформу, архитектуру процессора;
- IL-код (эта возможность будет использоваться в книге довольно часто);
- размер кода.

Многие инструменты обладают также поисковыми возможностями, позволяющими искать типы, методы, поля или инструкции кода.

MeasureIt

MeasureIt — весьма удобный инструмент для проведения эталонных микротестов, созданный Вэнсом Моррисоном (Vance Morrison), который является также автором PerfView. Он показывает относительные затраты ресурсов на различные .NET API во многих категориях, включая вызовы методов, массивы, делегаты, итерирование, рефлексию (отражение), P/Invoke и многое другое. Он сравнивает все затраты с вызовом в качестве эталона пустой статической функции.

Средство MeasureIt в первую очередь полезно для демонстрации влияния проектных решений на производительности на API-уровне. Например, в категории блокировок оно покажет вам, что `ReaderWriteLock` работает приблизительно в четыре раза медленнее, чем обычная инструкция `lock`.

К коду MeasureIt нетрудно добавить собственные эталонные тесты. Это средство поставляется с находящимся внутри пакетом кода, для его извлечения просто запустите команду `MeasureIt /edit`. Изучая этот код, можно усвоить принципы написания довольно точных эталонных тестов. В комментариях к коду содержатся подробные объяснения относительно проведения высококачественного анализа, на которые вам следует обратить пристальное внимание, особенно если вы собираетесь самостоятельно создать простой эталонный тест.

Например, следующий код не позволяет компилятору производить инлайнинг (inlining):

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void AnyEmptyFunction()
{
}
```

Используются в нем и другие трюки, например обход кэшей процессора и выполнение достаточного количества итераций для создания статистически значимых результатов.

Средство MeasureIt удобно наличием в нем ряда встроенных измерений самой среды CLR, которые могут дать вам хорошее представление о базовых затратах. Если вас интересует проведение эталонного тестирования своего кода, изучите следующий подраздел.

BenchmarkDotNet

Стандартом в эталонном тестировании .NET может, наверное, послужить проект с открытым кодом под названием BenchmarkDotNet. Это библиотека, которая способна справиться как с множеством обычных задач, связанных с эталонным микротестированием, так и с более развернутыми задачами за счет:

- атрибутов, облегчающих выбор кода для бенчмаркинга;
- создания изолированных проектов для каждого тестируемого метода;
- автоматического вычисления количества итераций, позволяющих достичь нужной точности;
- разогрева кода;
- выполнения статистического анализа;
- сравнения производительности в нескольких разных средах выполнения кода, таких как x86, x64, различных версиях JIT, конфигурациях сборщика мусора и т. д.;
- анализа работы ЦП, сборки мусора, выделения памяти, JIT и различных аппаратных счетчиков.

Приступить к работе с этим средством очень легко. Рассмотрим простой пример, в котором сравнивается производительность циклов `foreach` для массива и для `IEnumerable`. Используя простой атрибут, можно позволить библиотеке выполнять практически всю работу.

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System.Collections.Generic;

namespace BenchmarkTest
{
    public class LoopBenchmarks
    {
        static int[] arr = new int[100];

        public LoopBenchmarks()
        {
            for (int i = 0; i < arr.Length; i++)
            {
```

```
                arr[i] = i;
            }
        }

        [Benchmark]
        public int ForEachOnArray()
        {
            int sum = 0;
            foreach (int val in arr)
            {
                sum += val;
            }
            return sum;
        }

        [Benchmark]
        public int ForEachOnIEnumerable()
        {
            int sum = 0;
            IEnumerable <int> arrEnum = arr;
            foreach (int val in arrEnum)
            {
                sum += val;
            }
            return sum;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var summary = BenchmarkRunner.Run<LoopBenchmarks >();
        }
    }
}
```

Все это можно запустить самостоятельно с помощью кода из примера BenchmarkTest.

В результате будет получен следующий вывод:

```
Total time: 00:00:43 (43.64 sec)

// * Summary *

BenchmarkDotNet=v0.10.9, OS=Windows 10 Redstone 2 (10.0.15063)
Processor=Intel Core i7-3930K CPU 3.20GHz (Ivy Bridge),
  ProcessorCount=12
Frequency=14318180 Hz, Resolution=69.8413 ns, Timer=HPET
  [Host] : .NET Framework 4.7 (CLR 4.0.30319.42000),
    32bit LegacyJIT-v4.7.2102.0
```

```
  DefaultJob : .NET Framework 4.7 (CLR 4.0.30319.42000),
    32bit LegacyJIT-v4.7.2102.0
               Method |      Mean |     Error |    StdDev |
--------------------- |----------:|----------:|----------:|
      ForEachOnArray  | 53.32 ns  | 0.2083 ns | 0.1846 ns |
ForEachOnIEnumerable  | 561.69 ns | 7.2943 ns | 6.8231 ns |

// * Hints *
Outliers
  ForEachTest.ForEachOnArray: Default -> 1 outlier was removed

// * Legends *
  Mean : Arithmetic mean of all measurements
  Error : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 ns : 1 Nanosecond (0.000000001 sec)

// ***** BenchmarkRunner: End *****
// * Artifacts cleanup *
```

Обратите внимание: на выполнение эталонного теста даже такого простого кода было затрачено 43 с.

Разумеется, есть возможность контролировать выполнение этих тестов с помощью конфигурационных параметров.

Для получения дополнительных сведений можно обратиться по адресу http://benchmarkdotnet.org. Добавьте это средство к своему проекту непосредственно из Visual Studio, установив пакет BenchmarkDotNet NuGet.

Оснащение кода инструментами

Привычную отладку, выполняемую через консольный вывод, игнорировать не стоит. Но я рекомендую вместо этого воспользоваться ETW-событиями, подробно рассмотренными в главе 8.

Временами может принести пользу и строгое хронометрирование выполнения кода. Никогда не задействуйте для измерения производительности свойство `DateTime.Now`. Оно слишком медленно работает для этих целей. Вместо него для замера временных интервалов малых или больших событий с предельной надежностью измерений, низкой погрешностью и минимальными издержками используйте класс `System.Diagnostics.Stopwatch`.

```
var stopwatch = Stopwatch.StartNew();
...do work...
stopwatch.Stop();
TimeSpan elapsed = stopwatch.Elapsed;
long elapsedTicks = stopwatch.ElapsedTicks;
```

Более подробно работа с временем и его измерениями в .NET рассматривается в главе 6.

Если нужно обеспечить точность и повторяемость результатов ваших собственных эталонных тестов, изучите исходный код и документацию MeasureIt, где демонстрируются самые удачные технологии в этой области. Вопреки возможным ожиданиям, зачастую добиться желаемого весьма непросто и некорректное проведение эталонного тестирования может оказаться куда вреднее полного отказа от него, поскольку повлечет за собой напрасную трату времени на неверные действия. Лучше воспользоваться библиотекой стороннего производителя вроде BenchmarkDotNet.

Утилиты SysInternals

Без этого великолепного набора инструментов вряд ли обойдется разработчик, системный администратор и даже простой любитель. Созданный Марком Руссиновичем (Mark Russinovich) и Брюсом Когсуэллом (Bryce Cogswell), а ныне принадлежащий Microsoft, этот набор предназначен для управления компьютером, инспектирования процессов, анализа сети и многого другого. В перечень наиболее предпочтительных, с моей точки зрения, входят следующие инструменты:

- **ClockRes** — показывает разрешение системных часов (что также является максимальным разрешением таймера);
- **CoreInfo** — связывает логические процессы с физическими процессорами, сокетами, кэшами и многим другим;
- **Diskmon** — отслеживает всю дисковую активность;
- **DiskView** — утилита посекторной работы с жесткими дисками;
- **Handle** — показывает, какие файлы открыты какими процессами;
- **ListDLLs** — выводит список загруженных DLL-библиотек;
- **NTFSInfo** — выдает подробную версию о томах NTFS;
- **PsInfo** — показывает операционную систему, диск, пользователя и программную информацию о системе;
- **ProcDump** — гибко настраиваемый создатель дампа процесса;
- **Process Explorer** — более удачное средство, чем диспетчер задач Task Manager, с широким спектром подробностей о каждом процессе (рис. 1.27);
- **Process Monitor** — отслеживает в реальном времени активности, связанные с файлами, реестром и процессами (рис. 1.28);
- **RAMMap** — анализирует использование физической памяти во всей системе;
- **SDelete** — безопасная утилита удаления файлов;
- **Strings** — ищет строки в двоичных файлах;
- **VMMap** — анализирует адресное пространство процесса.

В набор входят и десятки других утилит. Сам набор утилит можно загрузить (по одной или целиком) с сайта по адресу https://docs.microsoft.com/sysinternals/.

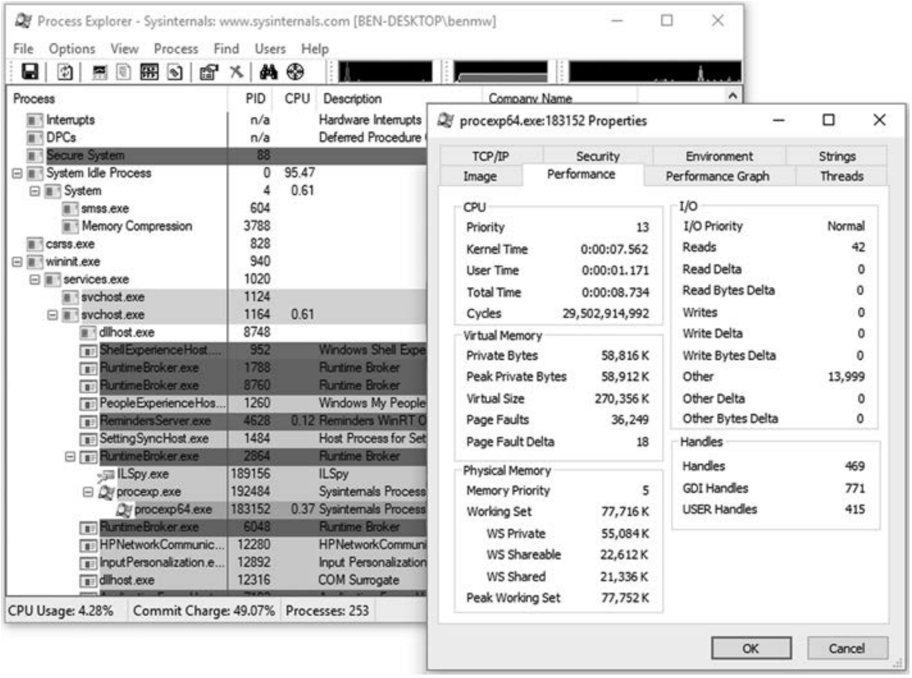

Рис. 1.27. Process Explorer — усовершенствованная версия диспетчера задач Task Manager, сообщающая множество подробностей о каждом процессе, а также взаимоотношениях процессов

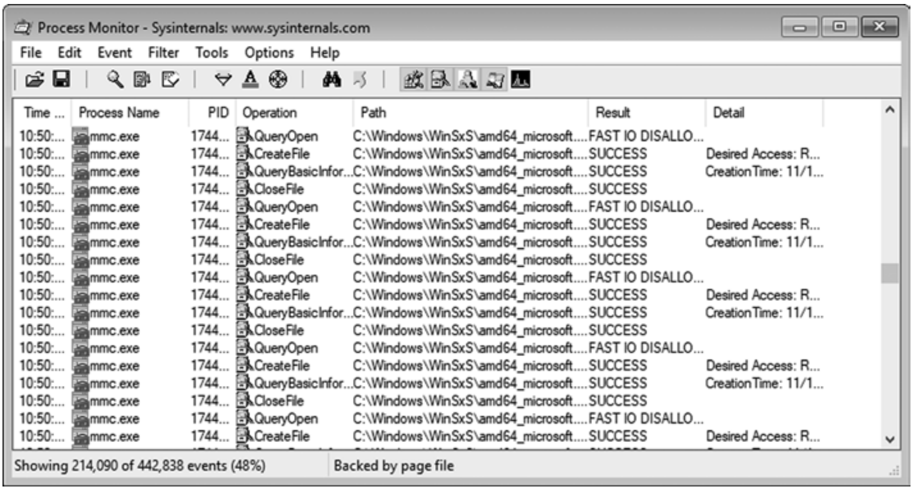

Рис. 1.28. Process Monitor показывает в реальном времени события, касающиеся файлов, реестра, процессов, потоков и сети в масштабе всей системы. Эта утилита может пригодиться, к примеру, чтобы установить: процесс выполняет чтение из конкретного файла и происходит это в такое-то время

База данных

Финальным средством исследования производительности довольно общего плана является простая база данных — нечто, позволяющее отслеживать производительность на протяжении длительного времени. В качестве метрик используйте все, что имеет смысл для вашего проекта, а формат не обязательно должен быть полномасштабной реляционной базой данных типа SQL Server (хотя у такой системы есть определенные преимущества). Он может представлять собой коллекцию отчетов, сохраняемых в определенный период времени в том формате, который проще читается, или просто в CSV-файлах с заголовками и значениями. Смысл в том, что вы должны записать, сохранить его и предусмотреть возможность создания на его основе отчета.

Когда вам задают вопрос, обладает ли ваше приложение более высокой производительностью, каков будет наилучший ответ?

Да.

Или таков: за последние шесть месяцев загруженность ЦП мы сократили на 50 %, потребление памяти — на 25 %, а задержки при запросах — на 15 %. Частота сборки мусора снизилась до одной каждые 10 с (а была каждую секунду!), и время запуска полностью зависит от загрузки конфигурации (35 с).

Как уже упоминалось, куда лучше хвастаться повышением производительности, опираясь на надежные данные!

Другие инструменты

Вам может встретиться и множество других инструментов. Существует большое количество статических анализаторов кода, сборщиков и анализаторов ETW-событий, декомпиляторов сборок, профайлеров производительности и многого другого.

Перечень, приведенный в данной главе, можно рассматривать как отправную точку, но при этом иметь в виду, что представленные в нем инструментальные средства позволяют выполнить весьма значительный объем работы. Иногда может помочь хорошо продуманная визуализация проблем производительности, но постоянной потребности в ней не будет.

Также со временем вы поймете, что по мере освоения таких технологий, как счетчики производительности (performance counters) или ETW-события, станет легче создавать собственные инструменты для выдачи настраиваемых отчетов или рационального анализа. Многие из рассматриваемых в книге инструментов в определенной степени поддаются автоматизации.

Издержки измерений

Несмотря на все ваши усилия, измерения производительности не обойдутся без издержек. Профилирование ЦП замедляет программу, счетчикам производительности потребуется память и/или дисковое пространство. ETW-события, какими бы быстрыми они ни были, также не обходятся без издержек.

Вам придется отслеживать и оптимизировать эти издержки в своем коде точно так же, как и все другие аспекты программы. Затем нужно будет решить, стоят ли затраты на измерения в некоторых сценариях того потенциального прироста производительности, которым придется пожертвовать.

Если нельзя позволить себе вести постоянные измерения, придется довольствоваться каким-нибудь профилированием. Пока этого в большинстве случаев будет достаточно для выявления причин возникновения проблем, менять, видимо, ничего не придется. Но не стоит недооценивать затраты людских ресурсов на измерение производительности вручную: зачастую они могут существенно превысить стоимость создания системы, способной выполнять измерения в автоматическом режиме.

Также вы можете иметь специальные сборки своих программных продуктов, но они способны представлять собой определенную опасность. Такие сборки не должны превратиться со временем в нечто не отражающее сущность реального продукта.

Как всегда, когда имеешь дело с программными средствами, нужно находить баланс между наличием всех желаемых данных и достижением оптимальной производительности.

Резюме

Самое важное правило достижения высокой производительности — *измерения, измерения и еще раз измерения*!

Выясните, какие показатели важны для вашего приложения. Выработайте точные количественные цели для каждого показателя. Средние значения, конечно, хороши, но обратите внимание и на процентили, особенно для сервисов, которые должны быть высокодоступными. Сначала убедитесь, что включаете в свою архитектуру разумные цели достижения производительности и правильно учитываете влияние архитектуры на производительность. Сперва оптимизируйте те части программы, которые сильнее всего влияют на производительность. Перед тем как перейти к микрооптимизациям, сконцентрируйтесь на макрооптимизациях на алгоритмическом или системном уровне. Если не уверены в достижении высокой производительности при реализации выбранного алгоритма, воспользуйтесь для его тестирования средами проведения эталонных тестов.

Обзаведитесь для своей программы высококачественной базой из счетчиков производительности и ETW-событий. Для анализа и отладки применяйте подходящий рабочий инструмент. Для быстрого решения проблем научитесь пользоваться наиболее эффективными средствами, например WinDbg и PerfView.

2 Управление памятью

Сборка мусора и управление памятью станут тем, с чего вы начнете и чем закончите. Эти два аспекта — бесспорный источник большинства наиболее очевидных проблем с производительностью, от которых проще всего избавиться, и чтобы все держать под контролем, за ними нужно постоянно наблюдать. Я говорю «бесспорный источник», потому что, как будет показано далее, многие проблемы связаны с неверным пониманием поведения сборщика мусора и неоправданными ожиданиями результатов его работы. Производительности, зависящей от задействования памяти, нужно уделять ничуть не меньше внимания, чем производительности, зависящей от использования центрального процессора. Это утверждение распространяется и на производительность неуправляемого кода, но для технологии .NET оно более характерно и в ее среде с соответствующими проблемами справляться проще. Этот аспект настолько важен для бесперебойной работы .NET, что наиболее значительная часть этой книги посвящена именно ему.

Многих сильно беспокоят издержки от сборки мусора. Однако, разобравшись в том, как работает сборщик мусора, вы сможете довольно просто оптимизировать свою программу, подстроившись под него. Во введении было показано, что сборщик мусора во многих случаях способен повысить общую производительность кучи, поскольку он лучше справляется с выделением и фрагментацией памяти. В большинстве случаев применяемая в технологии .NET стратегия управления памятью, включая работу сборщика мусора, действительно может принести вашему приложению не вред, а пользу.

Сборка мусора рассматривается в начале этой книги, поскольку слишком много понятий, приводимых далее, будет завязано на материале этой главы. Осознание влияния сборщика мусора на работу вашей программы играет настолько важную роль в достижении высокой производительности, что затрагивает практически все остальное.

Выделение памяти

Между характером работы обычных типовых куч и куч, подвергаемых сборке мусора в среде CLR, имеется существенная разница. В обычной куче Windows, чтобы знать, где размещать вновь выделяемые области памяти, ведутся списки свободных участков памяти. Многие надолго запускаемые приложения с обычным кодом страдают от фрагментации. Время, затрачиваемое на выделение памяти, постепенно

увеличивается, поскольку система выделения тратит все больше и больше времени на просмотр списков незанятых участков в поисках свободного места. Потребление памяти нарастает, и становится неизбежным перезапуск процесса, чтобы начать цикл заново. Некоторые обычные программы справляются с данной проблемой за счет замены исходной реализации `malloc` своими собственными, усердно старающимися сократить возникающую фрагментацию. В Windows также предоставляются кучи с низким уровнем фрагментации, используемые внутри среды CLR.

В среде .NET выделение памяти осуществляется довольно просто, поскольку обычно происходит в конце сегмента памяти и представляет собой не что иное, как несколько простых инструкций: сложение, уменьшение на единицу и сравнение. В этих простых случаях нет просматриваемых списков свободных участков и весьма невелика вероятность фрагментации. Кучи со сборщиком мусора действительно могут быть более эффективными, поскольку объекты, для которых память выделялась в один и тот же момент, скорее всего, будут располагаться в куче недалеко друг от друга, повышая локальность памяти.

По умолчанию небольшой фрагмент кода сравнит размер объекта, для которого требуется выделить память, с размером свободного места в небольшом буфере для выделения памяти. Если окажется, что объект помещается в этот буфер, выделение будет очень быстрым и пройдет без соперничества за ресурсы. Если же буфер окажется переполненным, запустится система выделения памяти со сборкой мусора и найдет место для объекта (в данном случае может понадобиться использование списков свободных участков памяти). После этого будет зарезервирован новый буфер для выделения памяти для будущих запросов на выделение.

Ассемблерный код этого процесса состоит всего из нескольких инструкций, которые полезно изучить.

На языке C# это можно продемонстрировать, взяв простейшее выделение памяти:

```
class MyObject {
  int x;
  int y;
  int z;
}
static void Main(string[] args)
{
  var x = new MyObject();
}
```

Так выглядит код, вызывающий выделение памяти:

```
; Копирование указателя на таблицу методов для класса
; в ecx в качестве аргумента для new().
; Для просмотра этого значения можно воспользоваться командой !dumpmt
mov ecx,3F3838h

; Вызов new
call 003e2100

; Копирование возвращенного значения (адреса объекта) в регистр
mov edi,eax
```

А вот фактическое выделение:

```
; ПРИМЕЧАНИЕ: большинство адресов в коде удалено из соображений форматирования.
;
; Присваивание eax значения 0x14, размер объекта, для которого выделяется
; память. Значение берется из таблицы методов
mov eax,dword ptr [ecx+4] ds:002b:003f383c=00000014

; Помещение в edx информации о буфере для выделений
mov edx,dword ptr fs:[0E30h]

; edx+40 содержит адрес следующего доступного для выделения байта.
; Прибавление этого значения к желаемому размеру.
add eax,dword ptr [edx+40h]

; Сравнение предполагаемого размера выделения с концом
; буфера для выделений.
cmp eax,dword ptr [edx+44h]

; При выходе за пределы буфера для выделений
; переход к медленному пути
ja 003e211b
; Обновление указателя до следующего свободного байта
; (0x14 байт после старого значения)
mov dword ptr [edx+40h],eax

; Вычитание размера объекта из указателя
; для получения адреса начала нового объекта
sub eax,dword ptr [ecx+4]

; Помещение указателя на таблицу методов
; в первые четыре байта объекта.
; eax теперь указывает на новый объект
mov dword ptr [eax],ecx

; Возвращение к вызывающему коду
ret

; Медленный путь – вызов метода CLR
003e211b jmp clr!JIT_New (71763534)
```

В целом все вылилось в один непосредственный вызов метода и всего лишь в девять инструкций во вспомогательном методе. Данный рекорд побить весьма трудно.

При использовании некоторых конфигурационных параметров, например серверного режима работы сборщика мусора, конкуренции за ресурсы нет как для быстрого, так и для медленного пути выделения памяти, поскольку куча имеется для каждого процессора. В .NET простота выделения памяти достигается за счет более сложного механизма высвобождения памяти, но сталкиваться напрямую

с этой сложностью вы не обязаны. Нужно только научиться оптимизировать под нее свой код, что мы и сделаем в данной главе.

Существует немало способов вынудить систему выделения опуститься до медленного пути. Если буфер для выделений недостаточно большой или достигнут конец сегмента, будет вызван медленный путь. Кроме того, если в распределяемом объекте реализуется финализатор, сборщику мусора потребуется выполнить много вспомогательных действий для отслеживания времени существования объекта, в силу чего он также вызовет медленный путь.

Операция сборки мусора

В механизме принятия решений сборщиком мусора постоянно происходят точечные улучшения, особенно в условиях более широкого распространения технологии .NET в высокопроизводительных системах. Поэтому приведенное далее объяснение может содержать детали, которые перестанут быть актуальными в результате изменений в последующих версиях .NET, но общая картина вряд ли существенно изменится в ближайшем будущем.

В управляемом процессе два типа куч: неуправляемые и управляемые. Неуправляемые кучи выделяются с помощью API Windows под названием VirtualAlloc и используются операционной системой и CLR для неуправляемой памяти. Такова, например, память для Windows API, для структур данных операционной системы и даже во многом для самой CLR. Память для всех управляемых .NET-объектов выделяется CLR в управляемой куче, которая называется также GC-кучей, поскольку находящиеся в ней объекты являются объектами сборки мусора.

Управляемая куча делится на два типа куч: кучу малых объектов и кучу больших объектов (large object heap, LOH). Каждой из куч назначаются собственные сегменты, представляющие собой принадлежащие ей блоки памяти. Как куча малых, так и куча больших объектов могут иметь несколько назначенных им сегментов. Размеры сегментов могут варьироваться в зависимости от конфигурации и аппаратной платформы.

Конфигурация	**Размер сегмента в 32-разрядной системе**	**Размер сегмента в 64-разрядной системе**
GC рабочей станции	16 Мбайт	256 Мбайт
GC сервера	64 Мбайт	4 Гбайт
GC сервера, имеющего более четырех логических процессоров	32 Мбайт	2 Гбайт
GC сервера, имеющего более восьми логических процессоров	16 Мбайт	1 Гбайт

Сегменты кучи малых объектов далее делятся на поколения. Существует три поколения с условными названиями gen 0, gen 1 и gen 2. Gen 0 и gen 1 всегда находятся в одном и том же сегменте, а gen 2 может распространяться на несколько сегментов, как и куча больших объектов. Сегмент, содержащий gen 0 и gen 1, называется эфемерным сегментом.

Поначалу куча малых объектов состоит из одного сегмента, а куча больших объектов представляет собой еще один сегмент (рис. 2.1). Сначала размер gen 2 и gen 1 всего несколько байтов, так как пока что они пустые.

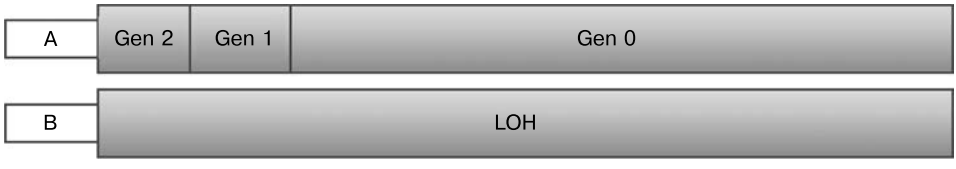

Рис. 2.1. Начальная структура кучи

Жизненный цикл объектов, распределяемых в кучу малых объектов, требует некоторых пояснений. Среда CLR распределяет все объекты размером менее 85 000 байт в куче малых объектов. Они всегда распределяются в gen 0, обычно в конец использованного на данный момент пространства. Поэтому, как было показано в начале этой главы, выделение памяти в .NET происходит очень быстро. Если добиться быстрого распределения не удастся, объекты могут быть размещены там, куда они способны поместиться в пределах границ поколения gen 0. Если они не поместятся в имеющееся свободное пространство, система распределения расширит текущие границы gen 0 для приема нового объекта. Расширение происходит в конце использованного пространства по направлению к концу сегмента. Если в результате этого произойдет сдвиг конца сегмента, может запуститься сборка мусора. Пространство gen 1 остается нетронутым.

Малые объекты (размером менее 85 000 байт) всегда начинают свой жизненный цикл в gen 0. До тех пор пока они существуют, при каждой сборке сборщик мусора станет продвигать их в следующие поколения. Сборку мусора в gen 0 и gen 1 иногда называют эфемерной сборкой.

При сборке мусора может произойти уплотнение — процесс, при котором GC физически перемещает объекты на новое место, чтобы освободить место в сегменте. Если уплотнения не случилось, границы просто переопределяются (рис. 2.2).

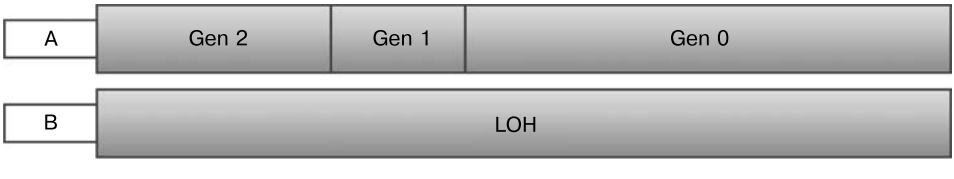

Рис. 2.2. Структура кучи после сборки мусора

Переместились не отдельные объекты, а линии границ.

Уплотнение может произойти при сборке мусора в любом поколении, и это довольно затратный процесс, поскольку GC должен исправить все ссылки на имеющиеся объекты, чтобы они указывали на новое место, для чего может потребоваться пауза в работе всех управляемых потоков. Из-за такой затратности сборщик мусора станет выполнять уплотнение, только когда это будет продуктивным действием, что оценивается на основе некоторых внутренних показателей.

Добравшись до gen 2, объект остается там на весь период своего существования. Это не означает, что gen 2 постоянно разрастается: если объекты в gen 2 в конечном итоге уничтожаются и в целом сегменте не будет живых объектов, сборщик мусора сможет возвратить сегмент операционной системе или просто придержать его для дальнейшего использования. Уменьшение рабочего множества памяти процесса в ходе сборки мусора не гарантируется.

А что означает понятие «живой объект»? Если GC может добраться до объекта через любые известные корни GC, следуя по графу ссылок объектов, значит, объект живой. Корнем могут быть статические переменные в программе, потоки, имеющие стеки из всех запущенных методов (таким образом ссылающиеся на локальные переменные), «сильные» дескрипторы сборщика мусора (например, закрепленные дескрипторы) и очередь финализатора. Следует учесть, что могут существовать и такие объекты, к которым уже не ведут никакие корни, но если они находятся в gen 2, то сборка мусора в поколении gen 0 их не очистит. Им придется дождаться полной сборки мусора.

Если gen 0 начинает заполнять сегмент, а сборка мусора не может выполнить достаточное уплотнение, GC выделит новый сегмент. В нем будут помещаться новое поколение gen 1 и gen 0 до тех пор, пока старый сегмент не окажется преобразован в gen 2. Все в старом поколении 0 станет частью нового поколения 1, а старое поколение 1 будет повышено до поколения 2, которое, что удобно, не придется копировать.

Если поколение gen 2 продолжит разрастаться, оно может распространиться на несколько сегментов. LOH также может распространиться на несколько сегментов. Независимо от того, как много в них сегментов, поколения 0 и 1 всегда будут существовать в одном и том же сегменте. Эти знания о сегментах пригодятся чуть позже, при попытке выяснить, какие объекты и где находятся в куче.

Куча больших объектов живет по другим правилам. Любой объект размером не менее 85 000 байт автоматически распределяется в LOH и не проходит через модель поколений (рис. 2.3). Если перефразировать, он распределяется прямо в gen 2. Единственные типы объектов, размер которых обычно больше 85 000 байт, — это массивы и строки. Из соображений производительности в LOH в ходе сборки мусора автоматическое уплотнение не выполняется, поэтому данная куча легко фрагментируется. Но начиная с версии .NET 4.5.1 появилась возможность уплотнить ее по требованию. Как и в gen 2, если в LOH память больше не нужна, ее можно освободить для других частей кучи. Чуть позже будет показано, что в идеале желания подвергать память в куче больших объектов сборке мусора вообще не возникает.

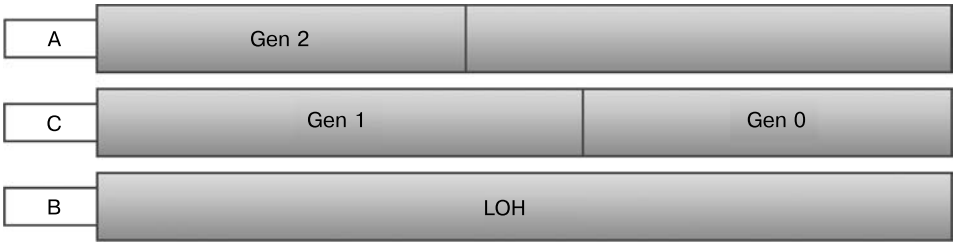

Рис. 2.3. Структура кучи после дальнейшего распределения и сборки мусора, вызвавших распределение новых сегментов

Для определения самого подходящего места для объектов сборщик мусора в LOH всегда использует список свободных участков памяти. Некоторые технологии сокращения уровня фрагментации в данной куче будут рассмотрены в этой главе.

ПРИМЕЧАНИЕ

Если придется разбираться с объектами в LOH в отладчике, станет видно, что не только она сама может целиком быть меньше 85 000 байт, но и в нее могут быть распределены объекты меньшего размера. Эти объекты обычно распределяются средой CLR, и их можно игнорировать.

Сборка мусора запускается в отношении конкретного поколения и всех нижестоящих поколений. Если сборка проводится в gen 1, то она будет выполнена и в gen 0. Если мусор собирается в gen 2, то сборка выполняется во всех поколениях и в куче больших объектов. Когда происходит сборка в gen 0 или gen 1, программа становится на паузу. Сборка в gen 2 может частично происходить в фоновом потоке, что зависит от параметров конфигурации.

Сборку мусора можно разделить на четыре этапа.

1. **Приостановка.** Перед сборкой все управляемые потоки в приложении делают вынужденную паузу. Стоит отметить, что приостановка может происходить только в определенных безопасных точках кода, например на инструкции `ret`. Обычные потоки не приостанавливаются и продолжают работу, пока не перейдут в управляемый код, тогда они тоже будут приостановлены. При наличии множества потоков существенная часть времени, отведенного на сборку мусора, может затрачиваться просто на приостановку потоков.
2. **Установка маркеров.** Стартуя из каждого корня, сборщик мусора следует по каждой ссылке объекта и маркирует замеченные объекты. В число корней входят стеки потоков, закрепленные GC-описатели и статические объекты.
3. **Уплотнение.** Сокращение фрагментации памяти путем перемещения объектов и расположения их рядом друг с другом, а также обновления всех ссылок, чтобы они указывали на новые места. Все это по мере надобности происходит

в куче малых объектов, и проконтролировать эти действия невозможно. В куче больших объектов автоматического уплотнения вообще не бывает, но есть возможность проинструктировать сборщик мусора, чтобы он выполнил уплотнение по требованию.

4. **Возобновление.** Разрешение управляемым потокам возобновить выполнение.

На самом деле на этапе установки маркеров обращаться к каждому объекту в куче не нужно — обход будет выполняться только в ее целевой части. Например, при сборке мусора в gen 0 рассматриваются только объекты из поколения gen 0, а при сборке мусора в gen 1 маркеры будут устанавливаться как в gen 0, так и в gen 1. А что касается сборки в поколении gen 2 или в куче целиком, то тут понадобится обход каждого живого объекта в ней, что превращает это действие в потенциально весьма затратную работу.

Здесь есть еще одна особенность: объект в более высоком поколении может послужить корнем для объекта в менее высоком поколении. Для отслеживания объектов между поколениями в GC используется таблица карточек (card table), представляющая собой двоичный массив, каждому разряду которого соответствует некоторый диапазон в куче. Разряд устанавливается в «грязный» (dirty) при записи в память в соответствующем диапазоне. При сборке мусора GC будет также рассматривать в качестве корней любые объекты, расположенные в «грязном» диапазоне. Таким образом, GC позволяется просматривать только поднабор объектов в более высоком поколении, затраты на это меньше, чем на полную сборку для данного поколения.

У рассмотренного поведения есть два важных последствия.

Во-первых, время, затрачиваемое на сборку мусора, практически полностью зависит от количества живых объектов в подвергаемом сборке поколении, а не от того, скольким объектам вы выделили память. Следовательно, если выделить память под дерево из миллиона объектов и отрезать корневую ссылку перед следующей сборкой мусора, этот миллион объектов абсолютно ничего не добавит к времени, затраченному на сборку.

Во-вторых, частота сборки мусора определяется в первую очередь объемом памяти, выделенным конкретному поколению. Как только этот объем пройдет внутренний порог, для данного поколения будет выполнена сборка мусора. Порог постоянно изменяется, и GC подстраивается под поведение вашего процесса. Если сборка мусора в отношении конкретного поколения продуктивна (при ней происходит продвижение множества объектов), она будет происходить чаще, и наоборот. Еще одной не зависящей от вашего приложения причиной сборки мусора является общий объем доступной памяти на машине. Если он падает ниже конкретного порога, сборка мусора может происходить чаще — предпринимается попытка сократить общий объем кучи.

На основе приведенного описания может сложиться мнение, что сборка мусора не поддается контролю. Но это далеко не так. Обычно можно манипулировать поведением GC, управляя применяемыми вами схемами выделения памяти. Для этого нужно понимать, как работает сборщик мусора, как часто вы выделяете

память, насколько хорошо контролируете время существования объектов и какие параметры конфигурации вам доступны. Давайте более пристально приглядимся к соответствующим параметрам конфигурации.

Подробная структура кучи. После изучения концепций, на которых основана работа памяти, обратимся к более подробным схемам кучи, полученным в ходе сеанса работы с отладчиком посредством команды `!eeheap -gc` (рис. 2.4–2.6).

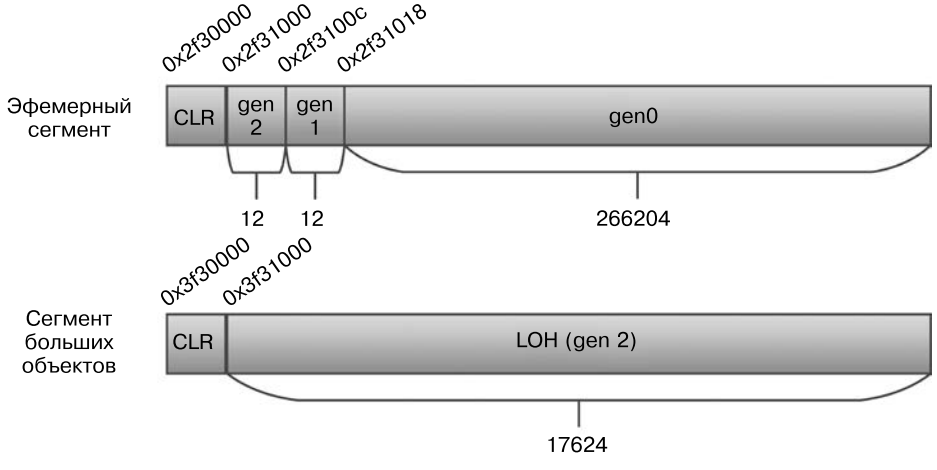

Рис. 2.4. Исходная структура кучи некоего приложения с эфемерным сегментом и сегментом больших объектов. Общий объем кучи составляет около 258 Кбайт, из которых 4 Кбайт используются непосредственно средой CLR. Поколения gen 1 и gen 2 занимают всего по 12 байт каждое

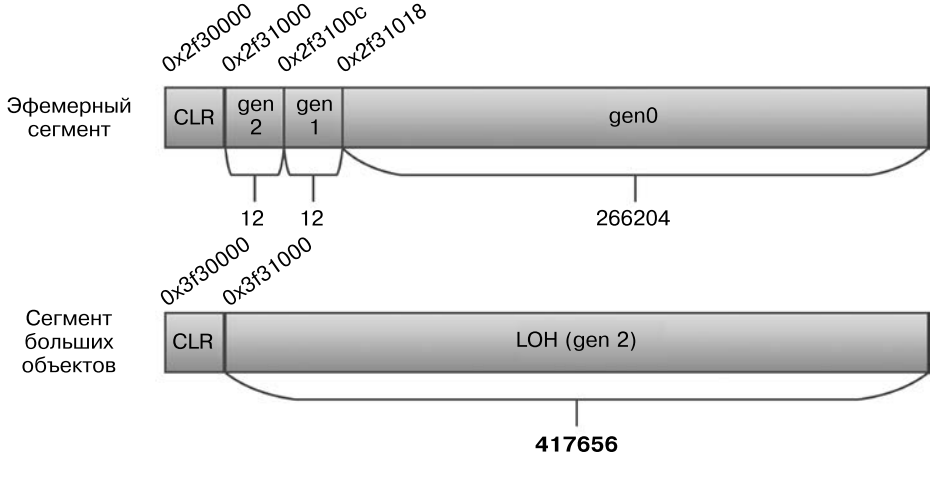

Рис. 2.5. После выделения множества небольших участков и большого массива памяти эфемерная куча сохранилась неизменной — все границы остались прежними. А вот LOH расширилась с 17 Кбайт до более чем 400 Кбайт. Затем произойдет GC

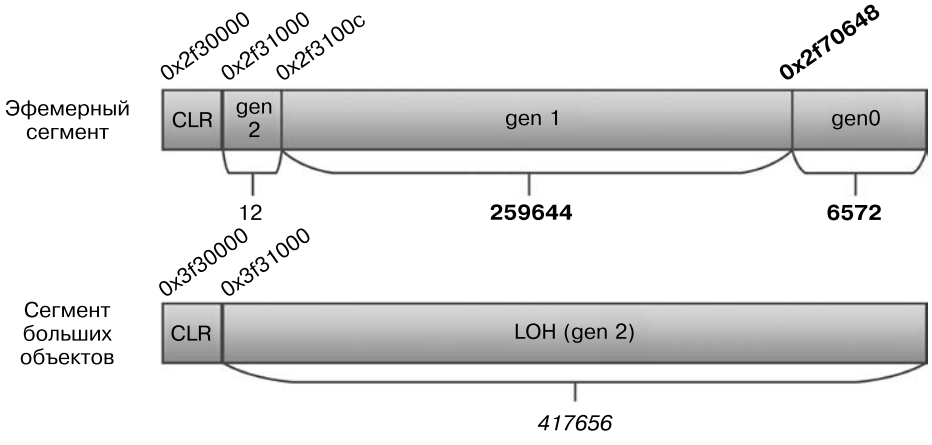

Рис. 2.6. После сборки мусора куча больших объектов осталась неизменной, а в эфемерной куче произошли существенные модификации. Поколение gen 1 сильно увеличилось, а поколение gen 0 соответственно сжалось

Параметры конфигурации

Среда .NET Framework дает не слишком много готовых способов конфигурирования сборщика мусора. Можно счесть это меньшим количеством патронов, чтобы выстрелить себе в ногу. Сборщик мусора выполняет конфигурирование и настройку в основном самостоятельно на основе конфигурации вашего оборудования, доступных ресурсов и поведения приложения. Несколько предоставляемых параметров предназначены для самого высокоуровневого поведения и определяются главным образом типом разрабатываемой вами программы.

Сравнение сборки мусора в режиме рабочей станции и в режиме сервера

Важнее всего выбрать, что использовать для сборки мусора — режим рабочей станции или серверный режим.

Сборка в режиме рабочей станции устанавливается по умолчанию. В этом случае вся сборка мусора происходит в том же самом потоке, который его инициировал, и выполняется с тем же уровнем приоритета. Для несложных приложений, особенно запускаемых на интерактивных рабочих станциях, где работает множество управляемых процессов, такой вариант наиболее рационален. Для компьютеров с одним процессором это единственный возможный вариант, и попытки настроить какую-либо иную конфигурацию окажутся безрезультатными.

Сборка в режиме сервера приводит к созданию специально выделенного потока для каждого логического процессора или ядра. Такие потоки запускаются

с наивысшим уровнем приоритета (THREAD PRIORITY HIGHEST), но всегда содержатся в приостановленном состоянии, пока не понадобится сборка мусора. Вся сборка мусора происходит в этих потоках, а не в потоках приложения. После нее они снова переходят в спящее состояние.

Кроме того, среда CLR создает для каждого процессора отдельную кучу. Внутри этой кучи имеются кучи малых и больших объектов. С точки зрения вашего приложения это логически одна и та же куча — коду неизвестно, какой куче принадлежат объекты, и ссылки на них существуют между всеми кучами (все они разделяют одно и то же адресное пространство).

Наличие нескольких куч дает два преимущества.

- Сборка мусора происходит в режиме параллельной работы. Каждый GC-поток выполняет сборку в одной из куч. Тем самым сборка мусора может проходить значительно быстрее, чем при сборке в режиме рабочей станции.
- В некоторых случаях выделение памяти может выполняться быстрее, особенно в куче больших объектов, где память может выделяться сразу во всех кучах.

Есть и другие внутренние различия, например более крупные размеры сегментов, что может означать более длительные промежутки между сборками мусора.

Конфигурация сборок мусора в режиме сервера находится в файле app.config внутри элемента <runtime>:

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

Какой из режимов нужно применять — рабочей станции или сервера? Если приложение запускается на машине с несколькими процессорами, предназначенной только для него, несомненно, следует выбрать серверный режим. Он в большинстве ситуаций обеспечит самую короткую задержку. Однако сборка в режиме сервера предполагает также наличие значительно большего рабочего множества, следовательно, приближение к ограничениям физической памяти. Чем больше объектов в памяти, тем больше времени может занять сборка мусора, сводя на нет преимущества.

В то же время при потребности делить машину между несколькими управляемыми процессорами выбор не столь очевиден. Сборка в режиме сервера создает множество потоков с высоким уровнем приоритета, и если ее выполняют несколько приложений, они могут отрицательно повлиять друг на друга, вплоть до возникновения конфликтов при диспетчеризации потоков. В таком случае, возможно, стоит воспользоваться сборкой мусора в режиме рабочей станции.

Если действительно требуется задействовать серверный режим сборки мусора при нескольких приложениях на одной и той же машине, еще одним вариантом будет привязка конкурирующих приложений к конкретным процессорам. Среда CLR создаст кучи только для процессоров, разрешенных для конкретного приложения.

Большинство советов в данной книге применимы к обоим режимам сборки мусора.

Сборка мусора в фоновом режиме

При сборке мусора в фоновом режиме изменяется порядок обработки сборщиком мусора поколения gen 2 за счет разрешения ему чаще делать свою работу в фоновом режиме наряду с выполнением других потоков. Сборка мусора в поколениях gen 0 и gen 1 по-прежнему происходит на первом плане с блокировкой выполнения всех потоков приложения.

Сборка мусора в фоновом режиме работает за счет наличия выделенного потока для сборки мусора в поколении 2. Для сборки в режиме сервера будут использоваться еще по одному потоку для каждого логического процессора в дополнение к потоку, изначально созданному для серверной сборки. Конечно, это означает, что при использовании сборки в режиме сервера и сборки в фоновом режиме у вас будет задействовано по два потока на процессор, выделенный для сборки мусора, но это не вызывает особых опасений. В том, что у процесса множество потоков, нет ничего особенного, тем более когда большинство из них основную часть времени ничего не делают. Один поток предназначен для сборки мусора на первом плане и запускается с наивысшим уровнем приоритета, но основную часть времени он находится в приостановленном состоянии. Поток для сборки в фоновом режиме запускается с более низким уровнем приоритета одновременно с потоками вашего приложения и будет приостановлен при активизации потоков сборки мусора, выполняемых на первом плане, поэтому между одновременно выполняемыми режимами сборки мусора нет никакой конкуренции.

При использовании сборки мусора в режиме рабочей станции фоновая сборка включена всегда. Начиная с версии .NET 4.5, она по умолчанию включена и при серверном режиме сборки мусора, но вы можете ее выключить.

Фоновый режим сборки мусора выключается с помощью следующего параметра конфигурации:

```
<configuration>
  <runtime>
    <gcConcurrent enabled="false"/>
  </runtime>
</configuration>
```

Исходя из своей практики, могу отметить, что причина для выключения сборки в фоновом режиме вряд ли появится. Отключение обычно приводит к ухудшению производительности и более частой сборке мусора на первом плане. Если хотите, чтобы на фоновые потоки сборки мусора вообще не тратилось процессорное время приложения, но не возражаете против потенциального увеличения времени полной блокирующей сборки или более частого ее проведения, можете выключить данный режим. Влияние принятого решения на производительность должно быть тщательно измерено.

Режимы задержки

У сборщика мусора имеется несколько режимов задержки, доступ к большинству можно получить через свойство `GCSettings.LatencyMode`. Частая смена режима крайне нежелательна, но временами она может оказаться полезной.

- **Интерактивный** режим является исходным при сборке мусора и используется при включенном режиме параллельной сборки мусора (который активируется по умолчанию). Этот режим позволяет проводить сборку мусора в фоновом режиме.
- **Пакетный** режим выключает всю одновременную сборку мусора и заставляет выполнять ее в едином пакете. Он сильно влияет на исполнение программы, поскольку заставляет ее полностью останавливаться на время сборки мусора. Регулярно применять его не следует, особенно в программах с пользовательским интерфейсом.

Есть два режима с низкой задержкой, которыми можно пользоваться в течение ограниченного времени. Если есть отрезки времени, требующие особенно высокой производительности, можно заставить сборку мусора не выполняться в поколении gen 2, где на нее уходит много времени.

- **LowLatency** (с низкой задержкой) — только для сборки мусора в режиме рабочей станции, сборка в поколении gen 2 будет подавляться.
- **SustainedLowLatency** (стабильный с низкой задержкой) — для сборки мусора в режиме рабочей станции и серверном режиме. Он будет полностью подавлять сборку в поколении gen 2, но при этом разрешит фоновую сборку в этом поколении. Чтобы этот режим оказался эффективным, нужно включить сборку мусора в фоновом режиме.

Оба режима существенно увеличат размер управляемой кучи, поскольку уплотнения не будет. Если процесс потребляет много памяти, этого следует избегать.

Непосредственно перед входом в один из рассмотренных режимов стоит принудительно выполнить последнюю полную сборку мусора, вызвав метод `GC.Collect(2, GCCollectionMode.Forced)`. Как только ваш код выйдет из режима с низкой задержкой, следует провести еще один сеанс полной принудительной сборки мусора.

Никогда не нужно пользоваться режимами с низкой задержкой по умолчанию. Они разработаны для приложений, которые должны исполняться без серьезных прерываний долгое время, но не 100 % времени. Хорошим примером может послужить биржевая торговля. Не хочется, чтобы в часы, когда рынок открыт, происходила полная сборка мусора. Когда рынок закрывается, этот режим выключается и до нового открытия рынка выполняется полная сборка мусора.

Включать режим с низкой задержкой допустимо только при выполнении следующих условий.

- В обычном режиме работы приложения задержка при полной сборке мусора недопустима.

- Приложение использует намного меньше памяти, чем доступно в целом (если нужен режим с низкой задержкой, следует максимально задействовать физическую память).
- Программа способна продержаться до тех пор, пока не выключит режим низкой задержки, или не перезапустит саму себя, или вручную не выполнит полную сборку мусора.
- И наконец, запуская код в среде .NET 4.6, можно объявить области, где сборка мусора запрещена, воспользовавшись для этого режимом `NoGCRegion`. Тем самым будет предпринята попытка ввода сборщика мусора в такой режим, при котором сборка будет полностью запрещена. Но установить его через это свойство невозможно. Вместо этого следует воспользоваться методом `TryStartNoGCRegion`.

Тут есть некоторые существенные предостережения.

- При запуске необходимо задать ожидаемый полный объем выделяемой памяти.
- Запрашиваемый объем не должен превышать размер эфемерного сегмента (информация о размере сегментов уже приводилась в данной главе).
- Среда CLR должна иметь возможность немедленно выделить запрашиваемую память как для кучи малых объектов, так и для кучи больших объектов.

Существует несколько перезагрузок `TryStartNoGCRegion`. В следующем примере демонстрируется одна из них со всеми параметрами:

```
bool success = GC.TryStartNoGCRegion(
    totalSize: 2000000,
    lohSize: 1000000,
    disallowFullBlockingGC: true);

if (success)
{
    try
    {
        //  выделение памяти
    }
    finally
    {
        if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion)
        {
            GC.EndNoGCRegion();
        }
    }
}
```

Параметр `totalSize` содержит общее количество байтов, которое предполагается выделить в этом регионе. Параметр `lohSize` показывает количество, которое предполагается выделить в куче больших объектов. Разница между `totalSize` и `lohSize` составляет количество, которое предполагается выделить в эфемерной куче. Оно должно быть меньше или равно размеру эфемерной кучи (приведен

в начале этой главы). По умолчанию, если память не может быть выделена средой CLR, она выполнит полную блокирующую сборку мусора в попытке освободить пространство памяти. Параметр `disallowFullBlockingGC` может выключить эту функциональность.

Только если вызов `TryStartNoGCRegion` завершится успехом, вы сможете вызвать `EndNoGCRegion`. Вкладывать вызовы `TryStartNoGCRegion` друг в друга нельзя.

Если выделенная память превысит зарезервированный объем, гарантия больше не будет действовать и может произойти сборка мусора.

ПРИМЕЧАНИЕ

Абсолютной гарантии работы в режимах с низкой задержкой или без сборки мусора дать нельзя. Если при дефиците памяти в системе сборщик вынужден выбирать между выполнением полной сборки и выдачей исключения `OutOfMemoryException`, будет выполнена полная сборка, независимо от установленного режима.

Альтернативные режимы задержек используются довольно редко, и следует дважды подумать, прежде чем применять их: последствия могут оказаться непредсказуемыми. Если вы считаете, что они полезны, проделайте тщательные измерения, чтобы убедиться в своей правоте. Варьируя режимы задержек, можно вызвать другие проблемы производительности, например выполнение большего количества эфемерных сборок в поколениях 0 и 1 в попытке справиться с нехваткой полных сборок мусора. Может получиться так, что вы поменяете один набор проблем на другой.

Большие объекты

По умолчанию количество элементов массивов ограничивается величиной `UInt32.MaxValue`, а фактический размер — объемом 2 Гбайт. Используя параметры конфигурации, можно допустить создание более крупных массивов, но максимальное количество элементов при этом останется прежним:

```
<configuration>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

Это позволит 64-разрядным процессам использовать массивы, превышающие по размеру 2 Гбайт. Тем не менее:

- максимальное количество элементов по-прежнему определяется значением параметра `UInt32.MaxValue` (4 294 967 295);
- максимальный индекс в любом измерении составляет 2 147 483 591 для массивов однобайтовых элементов или 2 146 435 071 для других типов;
- максимальный размер других объектов не изменяется.

Дополнительные параметры

Некоторые параметры сборки мусора должны быть настроены перед запуском процесса, поскольку они требуются в ходе инициализации среды CLR. Как правило, необходимость в этих настройках будет возникать крайне редко, и каждый такой случай требует строгого разбирательства.

Эти настройки конфигурируются через переменные среды окружения, значения которых устанавливаются в командной строке перед запуском процесса, который получит копию текущей среды окружения.

Ограничение количества куч

В серверном режиме сборки мусора для каждого процессора создаются куча и как минимум один поток. Временами вам может потребоваться применить для сборки мусора меньшее количество процессоров, возможно, в сочетании с изменением маски привязки процессоров для приложения:

```
// Ограничение на использование первых 16 процессоров
Process currentProcess = Process.GetCurrentProcess();
long mask = (long)currentProcess.ProcessorAffinity;
mask &= 0xFFFF;
currentProcess.ProcessorAffinity = (IntPtr)mask;
```

Если приложение запускается с уже примененной маской привязки процессоров, сборщик мусора, запущенный в серверном режиме, будет автоматически ограничивать количество создаваемых им куч и потоков для сборки мусора.

Однако этим также ограничивается количество процессоров, доступных приложению для выполнения основной работы. Если нужно, чтобы приложение использовало все процессоры для собственной работы, а сборщик мусора запускался только на поднаборе таких процессоров, следует установить соответствующее значение для переменной `GCHeapCount`, введенной в CoreCLR в середине 2016 года, или в среде .Net Framework 4.7.

```
SET COMPLUS_GCHeapCount=<n>
```

Этот параметр применим только при использовании серверного режима сборки мусора. Замените `<n>` значением, меньшим, чем количество задействованных логических процессоров.

Этим параметром можно воспользоваться, если требуется получить преимущества, предлагаемые серверным режимом сборки мусора, но необходимо ограничить количество ЦП, используемых в ходе сборки. Поскольку при серверном режиме сборка мусора выполняется с высоким уровнем приоритета, наличие потока, приходящегося на каждое ядро, будет останавливать все остальные процессы, запущенные на машине. Обычно это соответствует архитектуре и предполагается, что приложение с серверной сборкой мусора «владеет» машиной, но если некоторые процессоры требуется освободить, применяется этот параметр. Например, при задействовании 64-процессорного сервера может возникнуть желание использовать параллелизм и быстрые выделенные потоки сборки мусора, но если нужно быть экономными

и не держать другие процессы при сборке мусора на голодном пайке, то наличие 64 куч — это за гранью разумного. Кроме того, если суммарные требования к памяти более скромны, уменьшится объем издержек на применение памяти.

Отключение привязки потока сборщика мусора

В обычных обстоятельствах при серверном режиме сборки мусора каждый поток сборщика мусора привязан по запуску к конкретному логическому процессору. Это означает, что в ходе сборки мусора, по сути, гарантировано следующее: поток сборки мусора захватит процессор как поток с наивысшим приоритетом.

Используя настройку:

```
SET COMPLUS_GCNoAffinitize=1
```

можно отключить привязку, что позволит потокам сборки мусора запускаться на любом доступном процессоре. Она обеспечит лучшее взаимодействие процесса сборки мусора в режиме сервера с другими процессами.

Эта настройка спроектирована для хорошей работы совместно с `COMPLUS_GCHeapCount`, когда оптимизируется взаимодействие между приложением со сборкой мусора в режиме сервера и другими процессами, запущенными на машине.

Включение данного режима явно объявляет о желании достичь более высокого уровня взаимодействия при меньшем уровне эксклюзивности. Это означает, что улучшить производительность приложения за счет данной настройки не получится, но она может повысить общую производительность системы.

Проверка кучи

При оптимизации кода с целью получения наивысших показателей производительности, к сожалению, предпринимаются попытки срезать угол, которые могут привести к ошибкам наподобие повреждения состояния программы или даже самой структуры кучи. Повреждение кучи в .NET-приложениях — это практически всегда результат выполнения дефектного неуправляемого кода в том же самом процессе. Впрочем, такое возможно и в приложениях исключительно с управляемым кодом и может служить признаком ошибки внутри самой среды CLR. В таком случае отладка может быть крайне затруднена, поскольку сбой не будет происходить в каком-то определенном месте.

Для проверки кучи в отладчике можно воспользоваться командой `!VerifyHeap`:

```
0:006> !VerifyHeap
object 04b05980: bad member 00000066 at 04B05984
Last good object: 04B057E4.
0:006> !do 04B057E4
Name:        System.Int32[]
MethodTable: 62281938
EEClass:     61e09600
Size:        412(0x19c) bytes
Array:       Rank 1, Number of elements 100, Type Int32
Fields:
None
```

Кроме того, специально довести кучу до состояния, когда будут четко обозначены ее проблемы, может оказаться нелегко. При сборке мусора куча способна находиться в промежуточном состоянии, поэтому нужно гарантировать выполнение проверки в тот момент, когда не идет сборка мусора.

Существует несложный способ добиться этого вне отладчика. Можно включить режим, заставляющий кучу подвергаться проверке до или после каждой сборки мусора:

```
SET COMPLUS_HeapVerify=1
```

Включение режима проверки кучи приведет к снижению производительности, поскольку каждая сборка мусора теперь будет принудительно проверять кучу, то есть выполнять процесс, занимающий больше времени в зависимости от размера используемой кучи. При обнаружении повреждения будет выдано исключение, а процесс прекращен.

Советы по повышению производительности

Сокращайте размеры выделяемой памяти

Практически само собой разумеется, что уменьшение количества выделяемой памяти влечет за собой уменьшение рабочей нагрузки на сборщик мусора. Можно также сократить фрагментацию памяти и степень использования ЦП. Для достижения этой цели нужно проявить творческий подход, но он может конфликтовать с другими целями разработки.

Дайте критическую оценку каждому объекту и задайте себе следующие вопросы.

- Нужен ли вообще этот объект?
- Если ли у него поля, от которых можно избавиться?
- Можно ли сократить размер массивов?
- Можно ли сократить размер базовых элементов, например, с Int64 до Int32?
- Используются ли некоторые объекты исключительно в крайне редко складывающихся обстоятельствах, в силу чего могут быть проинициализированы только по мере надобности?
- Можно ли преобразовать некоторые классы в структуры, чтобы они размещались в стеке или существовали в виде части другого объекта и не создавали издержек, связанных с созданием каждого экземпляра?
- Не выделяется ли слишком много памяти, притом что задействуются только небольшие ее части?
- Нельзя ли получить эту же информацию, но каким-нибудь иным способом?
- Нельзя ли выделить память заранее?

ИСТОРИЯ

На сервере, обрабатывающем запросы пользователей, обнаружилось, что один из типов частых запросов вызывает выделение памяти, превышающей по размеру сегмент кучи. Поскольку среда CLR ограничивает максимальный размер сегментов, а поколение gen 0 должно располагаться в одном сегменте, мы гарантированно получали сборку мусора при каждом запросе. Ситуация сложилась не из приятных, поскольку для решения данной проблемы существует немного альтернатив, помимо сокращения размера выделяемой памяти.

Самое важное правило

Существует основное правило для высокопроизводительного программирования, касающееся сборщика мусора. По сути, сборщик мусора был явно разработан с прицелом на то, чтобы **проводить сборку мусора в поколении gen 0 или не проводить ее вообще**.

Иными словами, нужно стремиться к созданию объектов с экстремально коротким временем существования, чтобы сборщик мусора никогда их не касался, а если добиться этого невозможно, чтобы они как можно скорее попадали в поколение gen 2 и оставались там навсегда, никогда не подвергаясь сборке мусора. Это означает неизменное задействование ссылок на долгоживущие объекты, а зачастую также создание пула переиспользуемых объектов (особенно это относится к чему-либо в куче больших объектов).

Сборка мусора становится более затратной в каждом поколении. Нужно, чтобы она чаще проходила в поколениях gen 0/1 и лишь изредка — в поколении gen 2. Даже при сборке мусора в поколении gen 2 в фоновом режиме все равно возникнут затраты процессорного времени, которых могло бы и не быть: процессор должен применяться для остальной части вашей программы.

ПРИМЕЧАНИЕ

Возможно, вы слышали миф о том, что у вас должно быть десять сборок мусора в поколении gen 0, приходящихся на каждую сборку в поколении gen 1, и десять сборок мусора в поколении gen 1, приходящихся на каждую сборку в поколении gen 2. Это неправда. Поймите: вам нужно, чтобы было множество быстрых сборок в поколении gen 0 и очень мало весьма затратных сборок — в поколении gen 2.

Следует избегать перехода объектов в поколение gen 1, поскольку со временем они станут кандидатами на переход в поколение gen 2. Поколение gen 1 служит для них своеобразным буфером, прежде чем они попадут в поколение gen 2.

В идеале каждый выделяемый объект к моменту появления следующего gen 0 выходит из области видимости. Можно измерить продолжительность этого интервала и сравнить полученный результат с продолжительностью жизни данных

в вашем приложении. Порядок использования инструментальных средств для обнаружения этой информации будет показан в конце главы.

Если вы не привыкли руководствоваться этим правилом, то его выполнение потребует основательно изменить свои взгляды. Им будет обосновываться практически каждый аспект вашего приложения, поэтому выработайте привычку применять его как можно раньше и вспоминайте о нем как можно чаще.

Сокращайте время существования объекта

Чем короче время существования объекта, тем меньше вероятность его продвижения в следующее поколение при сборке мусора. Как правило, не нужно выделять память под объекты, пока в них не возникнет насущная необходимость. Исключением могут стать такие высокие затраты на создание объекта, что целесообразнее будет создать его на ранней стадии, когда он не будет мешать другой работе.

На другом конце жизненного цикла объектов нужно обеспечить, чтобы они как можно скорее вышли из области видимости. Для локальных переменных этот момент может настать после их последнего локального использования, даже до завершения метода. Можно лексически сузить его область видимости, воспользовавшись фигурными скобками `{ }`, но, вероятнее всего, это не будет иметь никакого практического значения, поскольку компилятор обычно сам распознает, когда логический объект больше не применяется. Если в вашем коде операции над объектом разнесены по разным участкам, постарайтесь сократить время между первым и последним использованием, чтобы сборщик мусора смог подобрать объект как можно раньше.

Изредка может возникнуть потребность в явном исключении ссылки на временный объект путем присвоения ей значения `null`, если он является элементом или статическим полем долгоживущего объекта. Это нужно делать только в том случае, когда требуется воспрепятствовать сборщику мусора в продвижении этого объекта в следующее поколение. Сначала попробуйте изменить конструкцию, чтобы ссылка стала локальной переменной, где время существования объекта — не слишком большая проблема. Если будет принято решение избавиться от поля, присвоив ему значение `null`, код может несколько усложниться, поскольку появится больше проверок на `null` по всему коду. Кроме того, это может создать противоречия между эффективностью и постоянной доступностью полного состояния, особенно при отладке. Один из вариантов обхода данной проблемы заключается в преобразовании объекта, от которого нужно избавиться путем присвоения значения `null`, в иную форму. Например, сериализовать иерархию XML-документа в строку или преобразовать временный объект состояния в сообщение в логе, которое может более эффективно записать состояние для последующей отладки. Необходимость в применении этого метода обычно возникает только в отношении больших временных графов объектов, существующих в полях для удобства.

Еще один способ достижения желаемого баланса кроется в переменном поведении: запустите программу (или ее определенную часть, скажем, только для конкретного запроса) в режиме, который не присваивает ссылкам значение `null`, а сохраняет их как можно дольше для облегчения отладки.

Сбалансируйте выделение

Как уже упоминалось в начале главы, сборщик мусора работает, переходя по ссылкам объекта. При сборке мусора в режиме сервера он делает это сразу в нескольких потоках. Параллелизм стоит использовать как можно чаще, но если один поток попадает на слишком длинную цепочку вложенных объектов, весь процесс сборки мусора не завершится, пока не закончит свою работу этот долгоиграющий поток. Кроме того, если некий поток выделит больше памяти, чем остальные, он будет инициировать сборку мусора чаще, чем в случае, если бы такие же выделения выполнялись в нескольких кучах.

Нам помогут алгоритмы балансировки нагрузки. Когда сборщик мусора обнаружит, что кучи стали разбалансированными, он приступит к принудительному выделению памяти в разных кучах. Такая функциональная возможность в отношении куч малых объектов имеется для многих версий среды CLR, но балансировка куч больших объектов появилась только в версии 4.5. Те ядра, которые закончили работу по сборке мусора, могут взять на себя работу из других куч.

С появлением у сборщика мусора этих функциональных возможностей проблемы с разбалансированностью куч стали менее актуальными. Но если есть подозрения, что сборка выполнятся слишком часто или в работе сборщика возникают длительные паузы, возможно, стоит проверить код на наличие глубоких древовидных структур объектов или потока, склонного к более частому выделению памяти.

Если найдется какой-нибудь поток, несущий ответственность за подавляющую часть случаев выделения памяти, найдите способы, позволяющие разделить эту ответственность между потоками. Убедитесь в использовании объектов `Task` или пула потоков, чтобы уравнять возможности обработки разными потоками разных запросов. Избегайте схемы, при которой один поток обрабатывает очередь запросов и выполняет основную часть выделений памяти, прежде чем передать работу другим потокам для завершения обработки.

Сократите количество ссылок между объектами

Сборщик мусора потратит больше времени на обход объектов, содержащих множество ссылок на другие объекты. Большая пауза в работе сборщика зачастую является признаком большого и сложного графа объектов.

Еще одна опасность заключается в существенном усложнении предсказания сроков существования объектов, если нельзя легко определить все возможные ссылки на них. В данном случае упрощение не только является достойной целью для применения разумных приемов программирования, но и облегчает отладку и решение проблем производительности.

Следует также понимать, что сделать работу сборщика мусора неэффективной могут ссылки между объектами разных поколений, в особенности от более старых объектов к более новым. Если объект, относящийся к поколению 2, ссылается на объект в поколении 0, то при каждой сборке мусора в поколении 0 часть объектов

поколения 2 также будет просканирована, чтобы убедиться в том, что ими по-прежнему удерживается ссылка на объект поколения 0. Это, конечно, менее затратная процедура, чем полная сборка мусора, но все же ненужная работа, если подобных ссылок можно избежать.

Избегайте закреплений

Закрепление объекта фиксирует его на месте, не позволяя сборщику мусора его перемещать. Закрепление существует для того, чтобы можно было без опасений передавать ссылки на управляемую память неуправляемому коду. Чаще всего оно задействуется для передачи неуправляемому коду массивов или строк, а также получения непосредственного зафиксированного доступа к структурам данных или полям. При отсутствии взаимодействия с неуправляемым кодом и небезопасного кода потребности в закреплении вообще не возникает. Но даже если избегать явного закрепления вашего собственного кода, есть много часто используемых API, которые без него все равно не обходятся.

Хотя сами по себе операции закрепления не особо затратны, они отрицательно влияют на сборку мусора, увеличивая вероятность фрагментации. Сборщик мусора отслеживает закрепленные объекты, чтобы можно было задействовать свободные пространства между ними, но слишком большое количество закреплений все же способно вызвать фрагментацию и разрастание кучи.

Закрепление может быть явным или неявным. Явное закрепление выполняется с использованием `GCHandle` типа `GCHandleType.Pinned` или ключевого слова `fixed` и должно находиться внутри кода, помеченного как небезопасный — `unsafe`. Разница между применением `fixed` или `GCHandle` аналогична разнице между `using` и явным вызовом `Dispose`. Ключевое слово `fixed` удобнее в использовании, но не может применяться в асинхронных ситуациях, в то время как `GCHandle` можно передавать между функциями и избавиться от него в функции обратного вызова.

Неявное закрепление встречается чаще, но его может быть труднее обнаружить и сложнее удалить. Наиболее явным источником закрепления будут любые объекты, переданные неуправляемому коду через вызов неуправляемого кода — Platform Invoke (P/Invoke). Это не только ваш собственный код, это могут сделать и зачастую делают управляемые API, вызывающие обычный, неуправляемый код, требующий закрепления.

В среде CLR также будут существовать закрепленные объекты в ее собственной структуре данных, но обычно это не должно вызывать никаких опасений.

В идеале нужно избегать закреплений настолько, насколько это возможно. Если же это невозможно, придерживайтесь тех же правил, что и для сборки мусора: максимально сокращайте время существования. Если объекты закреплены ненадолго, меньше вероятность, что они повлияют на следующую сборку мусора. Нужно также избегать одновременного наличия слишком большого количества закрепленных объектов. Обычно закрепление объектов, находящихся в поколении gen 2 или в LOH, вполне приемлемо, поскольку эти объекты вряд ли будут

куда-нибудь перемещены. Это может привести к стратегии, предусматривающей либо выделение больших буферов в куче больших объектов и выделение части из них по мере надобности либо выделение небольших буферов в куче малых объектов (но тогда перед закреплением надо обеспечить продвижение закрепляемых объектов в поколение gen 2). Для этого от вас понадобятся определенные управляющие действия, которые могут позволить полностью избежать присутствия закрепленных буферов в ходе сборки мусора в поколении gen 0.

Избегайте финализаторов

Создавать финализатор без особой надобности не стоит. Под финализаторами понимается код, запускаемый на выполнение сборщиком мусора с целью очищения неуправляемых ресурсов. Финализаторы вызываются из одного потока поочередно и только после того, как сборщик мусора, закончив работу, объявит объект неиспользуемым. Это означает, что при условии реализации в вашем классе финализатора вы гарантируете, что он будет оставаться в памяти даже после сборки мусора, при которой сборщик должен был от него избавиться. Также при этом предусматривается дополнительное выполнение учета при каждой сборке мусора, поскольку список финализаторов нуждается в постоянном обновлении при перемещении объекта. Все это снижает общую эффективность сборки мусора и гарантирует, что ваша программа затратит более весомые ресурсы на очистку объекта.

Но это еще не все. Под объект с финализатором память выделяется медленнее. Вместо того чтобы пустить распределитель по «быстрому пути», системе приходится заниматься дополнительным учетом, обеспечивающим отслеживание времени существования объекта со стороны сборщика мусора.

При реализации финализатора нужно также реализовать интерфейс `IDisposable`, чтобы позволить выполнять явную очистку, и сделать в методе `Dispose` вызов `GC.SuppressFinalize(this)`, чтобы удалить объект из очереди на финализацию. При условии вызова `Dispose` до следующей сборки он вычистит объект должным образом, исключая необходимость запуска финализатора. Правильное применение этой схемы показано в следующем примере. Заметьте, что реализовать паттерн `Dispose` можно, а зачастую и нужно без реализации финализатора.

```
class Foo : IDisposable
{
  private bool disposed = false;
  private IntPtr handle;
  private IDisposable managedResource;
  ~Foo()     // Финализатор
  {
    Dispose(false);
  }

  public void Dispose()
  {
    Dispose(true);
```

```
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
     if (this.disposed)
     {
       return;
     }
     if (disposing)
     {
       // Из финализатора это делать небезопасно
       this.managedResource.Dispose();
     }

    // Очистка неуправляемых ресурсов,
    // которую безопасно проводить в финализаторе
    UnsafeClose(this.handle);

    // Если базовый класс является объектом типа IDisposable,
    // необходимо также вызвать base.Dispose(disposing);

    this.disposed = true;
  }
}
```

Вся логика очистки сконцентрирована в методе `Dispose(bool)`. Весь остальной код просто вызывает это метод. Переменная `disposing` показывает, вызвал ли разработчик `Dispose` явно. Если так и было, значит, использовать `Dispose` будет безопасно для всех ресурсов. Но если этот метод вызван через финализатор, значит, нет никаких гарантий того, что любые объекты, на которые есть ссылки, все еще валидны. Следовательно, безопасно в этом методе могут быть очищены только те неуправляемые ресурсы, которые принадлежат этому объекту явно. В контексте финализатора о состоянии объектов, на которые ссылается данный объект, может быть сделано очень мало предположений. Код должен быть простым и затрагивать только ту память, которая гарантированно принадлежит только этому объекту и валидна. Обычно это означает, что нельзя работать с любым другим финализируемым (Finalizable) или освобождаемым (Disposable) объектом, кроме случаев, когда можно гарантировать их валидность.

Сделайте виртуальной только `protected`-версию `Dispose` и позвольте ей быть переопределенной дочерними типами. В поле `disposed` отслеживается, был ли объект уже освобожден, позволяя методу `Dispose` быть вызванным более одного раза.

Методы `Dispose` и финализаторы никогда не должны выдавать исключения. Если исключение произойдет в ходе выполнения кода финализатора, процесс будет прерван. Финализаторы также должны быть очень осторожны, выполняя любого рода операции ввода-вывода, даже такие простые, как логирование.

Важно реализовать этот паттерн правильно, чтобы гарантировать его корректную работу с полиморфными типами. Нужно будет принимать решение, следует ли

применять на практике финализаторы в отношении базовых типов, которые сами не имеют неуправляемых ресурсов, но могут иметь производные типы с такими ресурсами. В некоторых случаях может потребоваться принести производительность в жертву корректности, но этого следует избегать.

Любой тип, содержащий экземпляры других `IDisposable`-типов, должен сам реализовывать `IDisposable`. Вследствие этого `IDisposable` может распространиться по вашим структурам данных. При правильной реализации все ресурсы могут быть легко освобождены простым вызовом `Dispose` для корневого `IDisposable`.

ПРИМЕЧАНИЕ

Наверное, вам уже приходилось слышать, что финализаторы гарантированно выполнятся. В целом это верно, но есть исключения. При принудительном завершении программы никакой код больше не выполняется и процесс немедленно прекращается. Поток финализатора активизируется сборкой мусора, следовательно, при ее отсутствии финализаоры запущены не будут. Существует также лимит времени, ограничивающий продолжительность выполнения всех финализаторов при завершении процесса. Если ваш финализатор находится в конце списка, он может быть пропущен. Более того, поскольку финализаторы выполняются последовательно, если внутри следующего финализатора имеется ошибочно допущенный бесконечный цикл, все последующие финализаторы никогда не будут запущены. Это может привести к утечкам памяти. Исходя из всех этих причин, не стоит полагаться на финализаторы в надежде достичь состояния завершенной очистки внешних по отношению к вашему процессу ресурсов.

Избегайте выделения больших объектов

Не все выделенные объекты попадают в одну и ту же кучу. Объекты, превышающие определенный размер, попадают в кучу больших объектов и тут же оказываются в поколении gen 2. Для понятия «большой объект» после статистического анализа программ того времени была установлена граница 85 000 байт. Любой объект этого и большего размера считается большим и попадает в отдельную кучу.

Выделения объектов в большую кучу следует избегать, насколько это возможно. И не только из-за того, что сборка мусора из этой кучи обходится дороже, — с ней выше вероятность возникновения фрагментации, вызывающей со временем безграничный рост расхода памяти. Постоянное выделение в кучу больших объектов посылает сборщику мусора четкий сигнал о необходимости постоянно выполнять сборки, а в такую ситуацию попадать крайне нежелательно.

Во избежание этих проблем нужен строгий контроль над тем, что именно ваша программа помещает в кучу больших объектов. Все, что туда попадает, должно жить в течение всей жизни программы и многократно использоваться по мере необходимости с помощью пула.

Куча больших объектов не подвергается автоматическому уплотнению, но начиная с версии .NET 4.5.1 уплотнение можно задать программным путем. Однако задействовать эту возможность можно только в крайнем случае, поскольку уплотнение вызовет слишком большую паузу в работе. Прежде чем пояснить, как это делается, в следующих нескольких разделах поговорим о том, как не попасть в такую ситуацию.

Избегайте копирования буферов

Копирования данных нужно избегать, где только можно. Предположим, к примеру, что файловые данные считаны в поток данных в память `MemoryStream` (предпочтительно с пулом, если нужны большие буферы). После соответствующего выделения памяти эту ее область нужно рассматривать как предназначенную только для чтения, и любой компонент, которому нужен доступ к ней, будет читать из одной и той же копии данных.

Часто требуется возможность получить доступ к поддиапазонам буфера, массива или диапазона памяти. Сейчас среда .NET предоставляет два способа выполнения этого требования.

Первый вариант, доступный только для массивов, заключается в использовании структуры `ArraySegment<T>` для представления лишь части исходного массива. Этот `ArraySegment` может быть роздан API независимо от исходного потока данных, можно даже прикрепить именно к этому сегменту новый `MemoryStream`. В ходе этого процесса никакого копирования данных не происходит.

```
var memoryStream = new MemoryStream(2048);
var segment = new ArraySegment <byte >(memoryStream.GetBuffer(),
                                       100,
                                       1024);
...
var blockStream = new MemoryStream(segment.Array,
                   segment.Offset,
                   segment.Count);
```

Самая большая проблема при копировании памяти связана не с центральным процессором, а со сборкой мусора. Если все же потребуется скопировать буфер, попробуйте скопировать его в другой буфер из пула или уже существующий буфер во избежание нового выделения памяти.

Более новый вариант представления частей уже существующих буферов предусматривает использование структуры `Span<T>`. Чтобы воспользоваться соответствующей библиотекой, следует обзавестись NuGet-пакетом System.Memory и задействовать среду Visual Studio 2017.

Структура `Span<T>` похожа на массив в том смысле, что она представляет непрерывный блок памяти. А разница в том, что можно заключить управляемую память, неуправляемую память и память стека в одну и ту же абстракцию. Что касается неуправляемой памяти, эту структуру можно рассматриваться как смарт-оболочку, выполняющую арифметические действия над указателями.

Следующие примеры применения Span<T> взяты из проекта Span в примере кода, сопровождающего книгу.

В первом примере создаются стандартный байтовый массив в управляемой куче и извлечение, охватывающее некоторую часть массива (так же легко оно может охватывать и весь массив).

```
{
...
    byte[] array = new byte[] {0, 1, 2, 3};
    Span<byte> byteSpan = new Span<byte >(array , 1, 2);
    PrintSpan(byteSpan);
    ...
}

private static void PrintSpan <T>(Span<T> span)
{
    for (int i = 0; i < span.Length; i++)
    {
        ref T val = ref span[i];
        Console.Write(val);
        if (i < span.Length - 1) { Console.Write(", "); }
    }
    Console.WriteLine();
}
```

Этот код выведет такие данные:

```
1, 2
```

В следующем примере для создания оболочки массива, размещенного в стеке, используется Span<T>:

```
Unsafe
{
    int* stackMem = stackalloc int[4];
    Span<int> intSpan = new Span<int>(stackMem , 4);
    for (int i=0;i<intSpan.Length;i++)
    {
        intSpan[i] = 13 + i;
    }
    PrintSpan(intSpan);
}
```

Как видите, для заключения данного массива в оболочку используется точно такая же семантика и тот же самый вспомогательный метод может быть применен для вывода значений. Он выведет следующие данные:

```
13, 14, 15, 16
```

Очередной пример немного сложнее. Когда происходит выделение из обычной кучи, нужно указать количество выделяемых байтов, а когда в оболочку Span<T> заключается неуправляемая память, назначаются типы этой памяти, следовательно, длина извлечения определяется количеством объектов, а не байтов. Здесь данное

обстоятельство учитывается перед выделением путем умножения размера желаемых объектов на их количество:

```
unsafe
{
    const int ObjectCount = 4;
    int memSize = sizeof(int) * ObjectCount;
    IntPtr hNative = Marshal.AllocHGlobal(memSize);
    Span<int> unmanagedSpan = new Span<int>(hNative.ToPointer(),
                                            ObjectCount);
    for (int i = 0; i < unmanagedSpan.Length; i++)
    {
        unmanagedSpan[i] = 100 + i;
    }
    PrintSpan(unmanagedSpan);
    Marshal.FreeHGlobal(hNative);
}
```

Этот код выведет следующие данные:

```
100, 101, 102, 103
```

В заключительном примере используется один из методов расширения, включенный в библиотеку для преобразования строки в тип `ReadOnlySpan<char>`. К сожалению, взаимосвязи между `Span<T>` и `ReadOnlySpan<T>` нет, поскольку в `Span<T>` для предотвращения копирования значений применяется семантика возвращений по ссылке. Это означает, что для вывода значений нужен отдельный вспомогательный метод.

```
{
...
    ReadOnlySpan <char> subString =
      "NonAllocatingSubstring".AsSpan().Slice(13);
    PrintSpan(subString);
...
}

private static void PrintSpan <T>(ReadOnlySpan <T> span)
{
    for (int i = 0; i < span.Length; i++)
    {
        T val = span[i];
        Console.Write(val);
        if (i < span.Length - 1) { Console.Write(", "); }
    }
    Console.WriteLine();
}
```

Вывод этого кода будет выглядеть следующим образом:

```
S, u, b, s, t, r, i, n, g
```

Существуют также вспомогательные методы для преобразования массивов и структур `ArraySegment` в структуры `Span<T>`.

Объединяйте долгоживущие и большие объекты в пулы

Следует помнить о ранее сформулированном главном правиле: объекты существуют либо недолго, либо всегда. Они должны или уходить в поколении gen 0, или навсегда оставаться в поколении gen 2. Некоторые объекты статичны по своей сути, они создаются и сохраняются в течение всего жизненного цикла программы естественным образом. Остальным объектам, очевидно, не нужно оставаться вечными, но их естественное время существования в контексте вашей программы гарантирует им жизнь более длительную, чем период сборки мусора в поколении gen 0 (и, может быть, gen 1). Эти типы объектов — кандидаты на объединение в пулы. Еще одним серьезным кандидатом на объединение в пулы является любой объект, размещаемый в куче больших объектов, — обычно имеются в виду коллекции.

Единого способа объединения в пулы не существует, как не существует и стандартного API такого объединения, на который можно было бы положиться. По сути, вы сами должны разработать способ, работающий для вашего приложения, и выбрать конкретные объекты для объединения в пул.

Один из способов, позволяющих отнести объекты к разряду объединяемых в пул, заключается в превращении управляемого обычным порядком ресурса (памяти) в нечто, чем придется управлять явным образом. В среде .NET уже есть шаблон для работы с конечными управляемыми ресурсами, который называется `IDisposable`. Правильная реализация этого шаблона нам уже встречалась. Разумная конструкция предполагает выведение нового типа с реализацией в нем интерфейса `IDisposable`, в котором метод `Dispose` помещает объединяемый объект обратно в пул. Для пользователей данного типа это будет убедительным сигналом о необходимости особого отношения к этому ресурсу.

Реализация разумной стратегии объединения в пул дается нелегко и может всецело зависеть от того, как ваша программа должна его использовать, а также какие типы объектов нуждаются в объединении. Рассмотрим код, показывающий один из примеров простого класса объединения в пул, чтобы вы смогли понять, что для этого нужно. Этот код взят из учебной программы PooledObjects:

```
interface IPoolableObject : IDisposable
{
  int Size { get; }
  void Reset();
  void SetPoolManager(PoolManager poolManager);
}

class PoolManager
{
  private class Pool
  {
    public int PooledSize { get; set; }
    public int Count { get { return this.Stack.Count; } }
    public Stack <IPoolableObject > Stack { get; private set; }
    public Pool()
```

```
  {
    this.Stack = new Stack <IPoolableObject >();
  }

}
const int MaxSizePerType = 10 * (1 << 10); // 10 MB

Dictionary <Type, Pool> pools =
  new Dictionary <Type, Pool >();

public int TotalCount
{
  get
  {
    int sum = 0;
    foreach (var pool in this.pools.Values)
    {
      sum += pool.Count;
    }
    return sum;
  }
}

public T GetObject <T>()
  where T : class , IPoolableObject , new()
{
  Pool pool;
  T valueToReturn = null;
  if (pools.TryGetValue(typeof(T), out pool))
  {
    if (pool.Stack.Count > 0)
    {
      valueToReturn = pool.Stack.Pop() as T;
    }
  }
  if (valueToReturn == null)
  {
    valueToReturn = new T();
  }
  valueToReturn.SetPoolManager(this);
  pool.PooledSize -= valueToReturn.Size;
  return valueToReturn;
}

public void ReturnObject <T>(T value)
  where T : class , IPoolableObject , new()
{
  Pool pool;
  if (!pools.TryGetValue(typeof(T), out pool))
  {
    pool = new Pool();
```

```
      pools[typeof(T)] = pool;
    }

    if (value.Size + pool.PooledSize <= MaxSizePerType)
    {
      pool.PooledSize += value.Size;
      value.Reset();
      pool.Stack.Push(value);
    }
  }
}

class MyObject : IPoolableObject
{
  private PoolManager poolManager;
  public byte[] Data { get; set; }
  public int UsableLength { get; set; }

  public int Size
  {
     get { return Data != null ? Data.Length : 0; }
  }

  void IPoolableObject.Reset()
  {
    UsableLength = 0;
  }

  void IPoolableObject.SetPoolManager(
    PoolManager poolManager)
  {
    this.poolManager = poolManager;
  }

  public void Dispose()
  {
    this.poolManager.ReturnObject(this);
  }
}
```

Заставлять объединяемые в пул объекты иметь реализацию специализированного интерфейса может показаться обременительным, но, помимо удобства, этим подчеркивается весьма важный факт: чтобы воспользоваться объединением в пул и повторно применять объекты, вам нужно хорошо в них разобраться и взять их под контроль. Всякий раз перед возвращением их в пул код должен сбрасывать их в известное безопасное состояние. Это означает, что бездумно объединять в пулы сторонние объекты напрямую нельзя. Реализуя собственные объекты со специализированным интерфейсом, вы подаете весьма четкий сигнал о том, что объекты имеют особый характер. В первую очередь следует опасаться объединения в пулы объектов из среды .NET Framework.

Очень сложно объединять в пулы коллекции из-за их особенностей — не хочется разрушать существующее хранилище данных (в конце концов, весь смысл объединения в пул и заключается в том, чтобы не допустить этого), но при этом нужно обеспечить возможность обозначения пустой коллекции с доступным пространством. К счастью, в большинстве типов коллекций реализуются оба свойства, обозначающие эти различия, — `Length` и `Capacity`. Учитывая опасность объединения существующих типов коллекций среды .NET в пул, будет лучше, если вы реализуете собственные типы коллекций, используя стандартные интерфейсы коллекций, такие как `IList<T>`, `ICollection<T>` и т. д. Основное руководство по созданию ваших собственных типов коллекций изложено в главе 6.

Дополнительной стратегией становится обладание собственными объединяемыми типами, реализующими в качестве механизма безопасности финализатор. Если запускается финализатор, это означает, что метод `Dispose` никогда не вызывался, что является ошибкой. Может быть выбран вариант записи какой-нибудь информации в лог, аварийного завершения работы или другого оповещения о возникшей проблеме. Но подавать сигналы нужно с особой осторожностью, поскольку обращение к памяти, которую сборщик мусора признал аннулированной, приведет к аварии или зависанию.

Следует понимать, что пул, никогда не сбрасывающий объекты, неотличим от утечки памяти. У вашего пула должен быть конечный размер (либо в байтах, либо в количестве объектов), при превышении которого он должен сбрасывать объекты для их очистки сборщиком мусора. В идеале пул должен быть достаточно большим, чтобы выполнять обычные операции без сброса чего бы то ни было и чтобы сборщик мусора был необходим только после коротких всплесков необычной активности. Сброс объектов, содержащихся в пуле, может приводить к продолжительным, полным сборкам мусора, что зависит от размера и количества объектов. Важно обеспечить настраиваемость пула под вашу ситуацию.

Обычно я не использую объединение в пул в качестве решения по умолчанию. Как универсальный этот механизм неэффективен, в нем часто возникают ошибки. Но может оказаться, что от объединения небольшого количества типов в пулы ваше приложение получит вполне определенные преимущества.

Практический пример: RecyclableMemoryStream. Однажды я работал над приложением, управляющим объединением (federation) тысяч сетевых серверных ресурсов в секунду. В основном оно занималось чтением байтов из сети или записью их в сеть. Около 90 % выделяемой памяти приходилось на объекты `MemoryStream`, которые постоянно то создавались — и под них выделялась память, — то изменялись в размерах: выполнялись кодировка строк, маршализация, демаршализация, создание временных буферов и многое другое. В результате на простую сборку мусора затрачивалось невероятное количество процессорного времени — почти 25 %! Профилирование памяти и ЦП быстро выявили необходимость найти более приемлемый способ работы с байтами, чем тот, что использовался в `MemoryStream`.

Здесь мы рассмотрим конструкцию и некоторые подробности реализации класса `RecyclableMemoryStream` для объединения объектов `MemoryStream` в пул. Код можно

загрузить по адресу https://github.com/Microsoft/Microsoft.IO.RecyclableMemoryStream или же воспользоваться им непосредственно из Visual Studio с пакетом NuGet.

Требования к новому способу работы были следующими.

- Полное исключение выделения памяти в куче больших объектов.
- Сокращение затрат времени на сборку мусора, особенно проводимую в поколении gen 2.
- Предотвращение утечек памяти с помощью ограничения размера пула.
- Предотвращение фрагментации памяти.
- Простота отладки.
- Возможность оснащения инструментами для выполнения измерений и ведения логов.
- Соблюдение семантики `Dispose`.
- Максимальная простота процесса замены `MemoryStream`.
- Обеспечение потокобезопасности.

Все эти требования были соблюдены, что дало следующий перечень функциональных свойств и особенностей реализации.

- Вместо объединения в пулы самих потоков данных объединялись базовые буферы. Это позволило нам лучше оснащать потоки инструментами и обнаруживать аномальные шаблоны использования, такие как повторное применение и утечка потоков (что особенно важно в сценариях с пулами), а также более просто повторно задействовать массивы. Также мы смогли предотвратить фрагментацию, сделав буферы одинакового размера.
- Потоки данных стали абстракционной надстройкой над собранными в цепочку буферами, что позволило им выглядеть как один большой буфер.
- Хотя сами потоки не потокобезопасны, потокобезопасность обеспечивалась при выделении и освобождении потока из пула.
- У каждого потока данных имелся идентификационный тег, который может помочь при отладке неправильного использования пула.
- Обеспечена возможность получения информации о стеке при выделении потока для облегчения отладки утечек пула.
- Разрешены гибкие и настраиваемые ограничения пула, накладывающие ограничения на использование памяти при обработке пиковых нагрузок.
- Представлены детализированные события и показатели для отслеживания использования с течением времени.

Как говорится, дьявол кроется в деталях, поэтому углубимся в подробности реализации.

Для того чтобы уметь выделять `RecyclableMemoryStream`, нужно создать диспетчер пула, то есть объект `RecyclableMemoryStreamManager`. Он представляет собой класс, на самом деле управляющий буферными пулами и отслеживающий

использование ресурсов. Думайте о нем как о миниатюрной куче внутри кучи среды CLR. В этом классе устанавливаются все настройки конфигурации, такие как исходные размеры буферов, максимальный размер кучи и многое другое. Обычно на каждый процесс имеется один объект-диспетчер, который существует во время жизни процесса. Но в некоторых особых случаях при возникновении реальной необходимости не вызовет проблем и применение нескольких объектов `RecyclableMemoryStreamManager`.

`RecyclableMemoryStreamManager` поддерживает две категории буферов: малый пул (Small Pool) и большой пул (Large Pool). Small Pool состоит из большого количества буферов одинакового размера. Слово Small в этом названии относится к размеру отдельно взятого буфера, а не всего пула. Буферы в Small Pool называются блоками, поскольку они объединены для формирования более длинного потока данных. В Large Pool содержатся более крупные буферы, но их значительно меньше и они сконструированы с прицелом на менее частое использование — только когда вызывается `GetBuffer`. В обоих пулах размеры буфера делают одинаковыми, чтобы уменьшить вероятность фрагментации кучи (рис. 2.7).

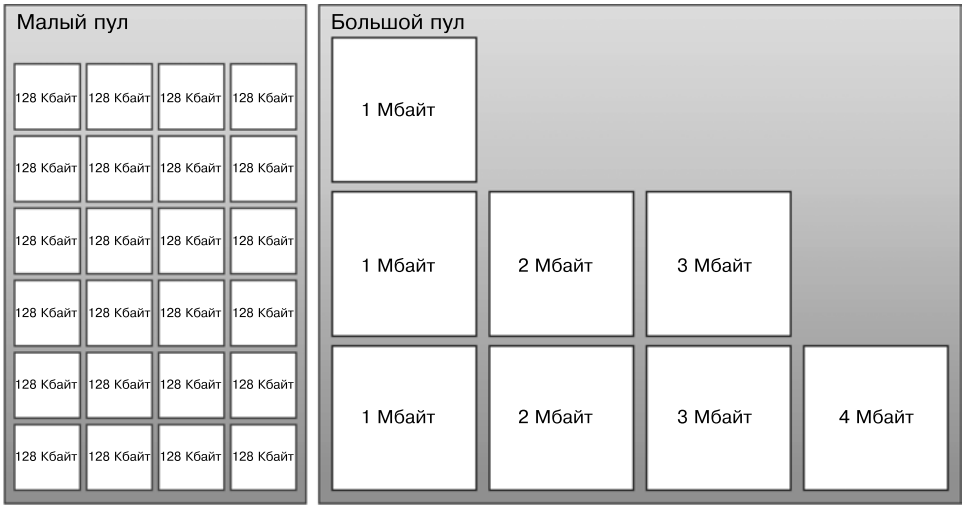

Рис. 2.7. Пулы в RecyclableMemoryStreamManager с размером блока 128 Кбайт, большой буфер, кратный 1 Мбайт, и максимальный размер буфера 4 Мбайт

Воспользоваться этой библиотекой довольно просто:

```
var sourceBuffer = new byte[]{0,1,2,3,4,5,6,7};
var manager = new RecyclableMemoryStreamManager();
using (var stream = manager.GetStream("Test"))
{
    stream.Write(sourceBuffer , 0, sourceBuffer.Length);
}
```

Этот код создает `RecyclableMemoryStreamManager` с исходными установками, захватывает поток данных, записывает в него несколько байтов, а затем возвращает пулу блоки этого потока с помощью вызова метода `Dispose`. В этом примере конструктору потока данных передается тег `Test`.

Этот тег не уникален для каждого потока данных, но служит для идентификации местоположения в коде, где он был выделен, что может помочь при отладке. Применять теги не обязательно, но они приносят пользу. Внутри системы каждому потоку данных также присваивается уникальный GUID, который служит для идентификации отдельно взятого потока и может пригодиться при отслеживании одновременного использования нескольких потоков данных.

Под капотом `RecyclableMemoryStream` захватит блок у диспетчера. Чем больше данных записано в поток, тем больше блоков будут объединены в цепочку, а потоковые API заставят их выглядеть как один непрерывный блок памяти. По мере роста длины потока данных общее потребление памяти возрастает только на размер блока (и это если предположить, что блоки еще не были сохранены в пул). Это отличается от реализации `MemoryStream`, где по мере роста длины потока его емкость удваивается, что делает возможной обширную потерю памяти, с чем можно мириться в небольшом масштабе, но нельзя в большом.

Пока используются лишь методы `Read` и `Write`, задействоваться будут только блоки. Но иногда возникает необходимость получения одного непрерывного буфера. Для этого служит API `GetBuffer`, унаследованный от `MemoryStream`. Когда вызывается `GetBuffer`, должен быть возвращен непрерывный блок. Если на данный момент применяется только один блок, возвращается ссылка на него. Если используются несколько блоков, для удовлетворения запроса задействуется большой пул и байты копируются из блоков в большой буфер. Если запрошенный буфер больше максимального размера буфера пула, то для его удовлетворения происходит выделение памяти.

Следует заметить, что возвращенный буфер имеет как минимум такой же размер, что и содержащиеся в нем данные, но может быть и значительно больше. Для определения фактического объема содержащихся в нем данных нужно воспользоваться свойством потока данных `Length`. Иногда недальновидные пользователи библиотеки игнорируют это обстоятельство и записывают огромные буферы в сеть или файлы. После преобразования потока в буфер со связанной с ним длиной данных может оказаться полезно обернуть эти данные в структуру `ArraySegment<byte>`.

При использовании пулов метод `ToArray` менее полезен. Он требует возвращения массива точного размера, следовательно, произойдет выделение памяти (возможно, в куче больших объектов), а также будет выполнено копирование памяти. Из-за такого неэффективного поведения нужно избегать применения `ToArray`.

Я настоятельно рекомендую изучить код, находящийся по ранее приведенной ссылке: полезно разобраться в том, как библиотека пытается избежать выделения памяти, не забывая при этом о выполнении остальных требований.

После реализации этой библиотеки в производственном коде наблюдалось уменьшение выделения памяти в куче больших объектов на 99 %. Беспокойство по поводу весьма затратных сборок мусора в поколении gen 2 ушло в прошлое. Время, затрачиваемое на сборку мусора, снизилось с 25 % до менее чем 1 %.

Сокращайте степень фрагментации кучи больших объектов

Если полностью избежать выделения памяти в куче больших объектов невозможно, значит, нужно приложить все силы для того, чтобы избежать фрагментации.

Если вы не проявите должной осмотрительности, куча больших объектов может расти до бесконечности, что компенсируется списком свободных участков памяти. Чтобы получить реальную выгоду от пользования этим списком, нужно увеличить вероятность того, что выделение памяти может быть выполнено за счет промежутков в куче.

Один из путей достижения этой цели — обеспечение одинакового размера всех выделяемых участков памяти, касающихся LOH, или по крайней мере обеспечение их кратности какому-то стандартному размеру. Нужно избежать разброса размеров буферов, сделав их все одинаковыми или кратными известному числу, например 1 Мбайт. Тогда, если в какой-то момент сборщик мусора должен будет собрать один из буферов, высока вероятность того, что при последующем выделении памяти под новый буфер будет заполнен образовавшийся в результате сборки мусора промежуток, а не выделено новое место в конце кучи.

При определенных обстоятельствах выполняйте принудительную полную сборку мусора

Почти никогда не следует выполнять принудительную сборку мусора вне обычного расписания, определенного самим сборщиком мусора. Это нарушит его автоматическую настройку и в целом может существенно ухудшить его поведение. Но для высокопроизводительных систем есть некоторые соображения, способные заставить пересмотреть этот совет в особых ситуациях.

Полезно бывает провести принудительную сборку мусора в более оптимальный момент времени, чтобы избежать ее выполнения в неблагоприятный момент в будущем. Следует отметить, что речь идет только о весьма затратных и в идеале проводимых крайне редко полных сборках мусора. Сборка мусора в поколениях gen 0 и gen 1 может и должна происходить довольно часто во избежание слишком большого увеличения размера поколения gen 0.

Принудительная сборка мусора может быть оправдана в следующих ситуациях.

1. Использование режима сборки мусора с низкой задержкой. В этом режиме размер кучи может расти, и необходимо выбрать подходящий момент для проведения полной сборки. Режим сборки мусора с низкой задержкой рассматривался ранее в этой главе.

2. Образование в расписании работы приложения естественного простоя или периода без пиковых нагрузок. Это может означать использование режима с низкой задержкой в условиях, когда этого не требуется.
3. Выделение время от времени большого количества памяти с большим временем существования (в идеале вечной). Есть смысл как можно скорее переместить эти объекты в поколение gen 2. Если эти объекты заменят другие объекты, которые теперь станут мусором, от них можно сразу избавиться в ходе принудительной сборки мусора.
4. Необходимость в уплотнении кучи больших объектов из-за фрагментации. Такое уплотнение рассмотрено в специальном разделе.

В ситуациях 1 и 3 не допускается полная сборка мусора в течение конкретного времени, вместо этого она принудительно проводится в другое время. Ситуация 4 призвана сократить общий размер кучи при существенной фрагментации в LOH. Если ваш сценарий не относится ни к одной из этих категорий, его следует считать вредным.

Для проведения полной сборки нужно вызвать метод `GC.Collect` с поколением, к которому его нужно применить. Дополнительно можно указать значение аргумента перечисления `GCCollectionMode`, чтобы сообщить сборщику мусора, должен ли он самостоятельно принять решение о необходимости сборки. Используются три значения:

- `Default` (По умолчанию) — сейчас эквивалентно значению `Forced` (Принудительно);
- `Forced` (Принудительно) — команда сборщику мусора немедленно запустить сборку;
- `Optimized` (Оптимально) — разрешение сборщику мусора принимать решение о том, насколько сейчас благоприятное время для запуска сборки.

```
GC.Collect(2);
// эквивалент следующей инструкции:
GC.Collect(2, GCCollectionMode.Forced);
```

ИСТОРИЯ

Рассматриваемая ситуация сложилась на сервере, принимающем запросы пользователей. Каждые несколько часов нам требовалась перезагрузка более чем 1 Гбайт данных, заменяющих существующие данные. Поскольку эта операция требовала немалых затрат и у нас уже было сокращено количество получаемых машиной запросов, мы после перезагрузки принудительно провели две полные сборки мусора. Тем самым были удалены старые данные и гарантировано, что все распределенное в поколении gen 0 либо подверглось сборке, либо перешло в поколение gen 2, где оно и должно было оказаться. При возобновлении полной загруженности запросами уже не проводилась обширная полная сборка мусора, которая могла бы повлиять на первые запросы.

Уплотняйте кучу больших объектов по требованию

Даже при объединении в пулы сохраняется вероятность возникновения неподконтрольного вам выделения памяти, и со временем куча больших объектов становится фрагментированной. С выпуском среды .NET 4.5.1 появилась возможность заставить сборщик мусора уплотнить кучу больших объектов при следующей полной сборке.

```
GCSettings.LargeObjectHeapCompactionMode =
GCLargeObjectHeapCompactionMode.CompactOnce;
```

В зависимости от размера кучи больших объектов это может быть весьма затратная операция, занимающая до нескольких секунд. Может потребоваться перевести вашу программу в состояние, когда она приостанавливает работу, для которой предназначена, и инициировать принудительную сборку мусора с помощью метода `GC.Collect`.

Эта настройка будет действовать только при выполнении следующей полной сборки мусора. Как только она произойдет, `GCSettings.LargeObjectHeapCompactionMode` автоматически переключится на `GCLargeObjectHeapCompactionMode.Default`.

Из-за высокой затратности данной операции я рекомендую сократить количество случаев выделения в куче больших объектов до минимально возможного и объединить в пулы те из них, которые все же приходится выполнять. Это существенно сократит потребность в уплотнении. Рассматривайте возможность уплотнения в качестве крайней меры и только в случае, когда фрагментация и очень большой размер кучи действительно являются для вас проблемой.

Получайте уведомление о намечающейся сборке мусора

Если ваше приложение ни в коем случае не должно подвергаться воздействию сборок мусора в поколении gen 2, можно заставить сборщик мусора уведомлять о приближении полной сборки. Это даст вам возможность приостановить обработку данных, возможно, за счет увода запросов с машины или какого-либо иного перевода приложения в наиболее благоприятное состояние.

Может показаться, что механизм уведомлений является ответом на все проблемы, касающиеся сборки мусора, но я рекомендую проявлять предельную осмотрительность. Реализовать подобные уведомления можно, лишь выполнив максимально возможную оптимизацию в других областях. Получить преимущества от уведомлений от сборщика мусора можно, только если будут справедливы следующие утверждения.

1. Полная сборка мусора настолько затратна, что вы не в состоянии позволить себе выдержать одну такую сборку в ходе обычной работы приложения.
2. У вас есть возможность полностью отключить выполнение полезной работы приложением (возможно, в это время данную работу смогут выполнять другие компьютеры или процессы).

3. У вас есть возможность быстро отключить работу приложения (то есть на остановку затрачивается меньше времени, чем на фактическое выполнение сборки мусора).
4. Сборка мусора в поколении gen 2 происходит настолько редко, чтобы игра стоила свеч.

Сборка мусора в поколении gen 2 будет происходить редко только при условии сведения к минимуму выделения в кучу больших объектов и перемещения в следующие поколения за пределами gen 0. Поэтому, чтобы выяснить, при каких именно обстоятельствах можно действительно извлечь пользу от уведомлений от сборщика мусора, нужно приложить немало усилий.

К сожалению, из-за неопределенности, присущей моменту запуска сборки мусора, при подписке на уведомление о сборке можно указать лишь приблизительное время, оставшееся до нее, в которое должно сработать уведомление. Для этого используется число в диапазоне 1–99. При слишком низком значении показателя уведомление будет получено незадолго до выполнения сборки мусора, но возникнет риск, что она начнется еще до того, как вы сможете отреагировать на уведомление. При слишком высоком значении показателя момент сборки мусора может быть сильно отдален и вы будете получать уведомления слишком часто, что сделает режим работы крайне неэффективным. Все зависит от частоты выделения памяти и ее общей загруженности. Заметьте, что указываются два числа: по одному для порогового значения в поколении gen 2 и порогового значения кучи больших объектов. Наряду с другими функциями это уведомление выдается сборщиком по мере возможности. Сборщик не дает никаких гарантий того, что вы сможете избежать сборки мусора.

Чтобы воспользоваться данным механизмом, выполните следующие общие действия.

1. Вызовите метод `GC.RegisterForFullGCNotification` с двумя пороговыми значениями.
2. Выполните опрос сборщика мусора с помощью метода `GC.WaitForFullGCApproach`. Он может находиться в режиме бесконечного ожидания или принять значение истечения срока ожидания.
3. Если метод `WaitForFullGCApproach` возвратил значение `Success`, переведите приложение в состояние, допускающее полную сборку мусора (например, выключите на машине режим приема запросов).
4. Самостоятельно запустите полную сборку мусора путем вызова метода `GC.Collect`.
5. Вызовите метод `GC.WaitForFullGCComplete` (тут опять можно дополнительно указать значение истечения срока ожидания), чтобы дождаться завершения полной сборки мусора, прежде чем приложение продолжит работу.
6. Включите режим приема запросов.
7. Когда надобность в получении уведомлений о предстоящей полной сборке мусора отпадет, вызовите метод `GC.CancelFullGCNotification`.

Поскольку в данном сценарии задействован механизм опроса, нужно будет запустить поток, который сможет периодически проводить проверку. У многих приложений уже имеются своего рода служебные потоки, выполняющие различные действия по расписанию. Он вполне мог бы справиться и с этой задачей, или же можно создать специально выделенный поток.

Рассмотрим полный пример из учебного проекта GCNotification, в котором показано соответствующее поведение в простом тестовом приложении, непрерывно выделяющем память. Обратитесь к коду, сопровождающему книгу, чтобы самим попробовать запустить этот пример.

```
class Program
{
  static void Main(string[] args)
  {
    const int ArrSize = 1024;
    var arrays = new List<byte[]>();

    GC.RegisterForFullGCNotification(25, 25);

    // Запуск отдельного потока для ожидания уведомлений от сборщика мусора
    Task.Run(()=>WaitForGCThread(null));

    Console.WriteLine("Press any key to exit");
    while (!Console.KeyAvailable)
    {
      try
      {
        arrays.Add(new byte[ArrSize]);
      }
      catch (OutOfMemoryException)
      {
        Console.WriteLine("OutOfMemoryException!");
        arrays.Clear();
      }
    }

    GC.CancelFullGCNotification();
  }

  private static void WaitForGCThread(object arg)
  {
    const int MaxWaitMs = 10000;
    while (true)
    {
      // Существует также перезагружаемая версия WaitForFullGCApproach
      // с бесконечным ожиданием
      GCNotificationStatus status =
                    GC.WaitForFullGCApproach(MaxWaitMs);
```

```
      bool didCollect = false;
      switch (status)
      {
        case GCNotificationStatus.Succeeded:
          Console.WriteLine("GC approaching!");
          Console.WriteLine(
             "-- redirect processing to another machine -- ");
          didCollect = true;
          GC.Collect();
          break;
        case GCNotificationStatus.Canceled:
          Console.WriteLine("GC Notification was canceled");
          break;
        case GCNotificationStatus.Timeout:
          Console.WriteLine("GC notification timed out");
          break;
      }

      if (didCollect)
      {
        do
        {
          status = GC.WaitForFullGCComplete(MaxWaitMs);
          switch (status)
          {
            case GCNotificationStatus.Succeeded:
              Console.WriteLine("GC completed");
              Console.WriteLine(
              "-- accept processing on this machine again --");
              break;
            case GCNotificationStatus.Canceled:
              Console.WriteLine(
                  "GC Notification was canceled");
              break;
            case GCNotificationStatus.Timeout:
              Console.WriteLine(
                  "GC completion notification timed out");
              break;
          }
          // В зацикливании нет необходимости, но оно пригодится,
          // если требуется проверка состояния перед новым ожиданием
        } while (status == GCNotificationStatus.Timeout);
      }
    }
  }
}
```

Еще одной возможной причиной задействования этого механизма может стать уплотнение кучи больших объектов, но его можно инициировать на основе использования памяти, что может оказаться более приемлемым вариантом.

Применяйте для кэширования слабые ссылки

Слабыми называются ссылки на объект, которые все же позволяют сборщику мусора его вычищать. Они являются противоположностью применяемых по умолчанию сильных ссылок, полностью препятствующих сборке мусора в данном объекте. Наиболее полезны они для кэширования затратных в создании объектов, которые хотелось бы иметь под рукой, но при этом иметь возможность удалять в случае появления дефицита памяти. Слабые ссылки — это базовая концепция среды CLR, доступ к ним можно получить, используя два .NET-класса:

- `WeakReference`;
- `WeakReference<T>`.

С появлением в .NET 4.5 дженерик-версии слабых ссылок первый из упомянутых классов можно игнорировать. У его API есть слабые стороны, которые устранены в более новой версии, и здесь речь пойдет только о дженерик-версии.

Рассмотрим пример простого использования:

```
// Внутренний объект Foo может попасть под сборку мусора в любой момент!
WeakReference <Foo> weakRef = new WeakReference(new Foo());
...
// Создание сильной ссылки на объект,
// теперь уже недоступный для очистки со стороны сборщика мусора
Foo myFoo;
if (weakRef.TryGetTarget(out myFoo))
{
      ...
}
```

Обратите внимание на то, что ссылка на сам объект `WeakReference<T>` является сильной, значит, он не будет изъят во время сборки мусора — слабая ссылка относится только к внутреннему объекту. Если вы настолько сознательны, работая с памятью, что задействуете `WeakReference<T>`, то вполне обоснованно можете опасаться постоянного выделения памяти для создания объектов типа `WeakReference<T>`. К счастью, можно повторно использовать эти объекты-оболочки путем применения метода `SetTarget` для замены внутреннего значения по мере надобности.

Вы по-прежнему можете иметь и другие ссылки на тот же самый объект, как сильные, так и слабые. Сборка будет происходить лишь в том случае, если останутся только слабые ссылки на него или их не будет вообще.

Большинству приложений слабые ссылки не требуются, но существует ряд условий, при которых в этом появляется смысл.

- Наложение жестких ограничений на использование памяти (например, на мобильных устройствах).
- Время существования объекта весьма переменчиво. Если это время предсказуемо, можно применять сильные ссылки и непосредственно контролировать время их существования.

- Объект велик, но прост в создании. Слабая ссылка идеально подходит объектам, которые хорошо бы иметь под рукой, но если их нет, нетрудно их создать или вовсе без них обойтись. Следует заметить: это также означает, что размеры объектов должны быть весьма существенными, чтобы в первую очередь оправдать издержки от использования дополнительных объектов `WeakReference<T>`.
- Для объектов потребуются вторичные индексы (см. приведенный далее пример).

В следующих двух примерах показано применение `WeakReference<T>` для эффективного кэширования.

Пример: HybridCache

`WeakReference<T>` находит хорошее применение в качестве части кэша. Объекты начинают удерживаться с помощью сильных ссылок, но по истечении определенного времени их невостребованности (или возникновения иных условий на ваш выбор) они могут быть понижены до уровня слабых ссылок и со временем исчезнуть при сборке мусора.

В следующем примере показан простой кэш, внутри которого происходит управление двумя уровнями кэширования:

```
public class HybridCache <TKey, TValue > where TValue : class
{
    class ValueContainer <T>
    {
        public T value;
        public long additionTime;
        public long demoteTime;
    }

    private readonly TimeSpan maxAgeBeforeDemotion;

    // Здесь значения существуют до предельного возраста
    private readonly ConcurrentDictionary <TKey,
                                           ValueContainer <TValue >>
      strongReferences =
        new ConcurrentDictionary <TKey, ValueContainer <TValue >>();

    // Сюда значения перемещаются по достижении предельного возраста
    private readonly ConcurrentDictionary <
        TKey,
        WeakReference <ValueContainer <TValue >>>
      weakReferences =
        new ConcurrentDictionary <
          TKey,
          WeakReference <ValueContainer <TValue >>>();

    public int Count
    {
```

```
    get
    {
        return this.strongReferences.Count;
    }
}

public int WeakCount
{
    get
    {
        return this.weakReferences.Count;
    }
}

public HybridCache(TimeSpan maxAgeBeforeDemotion)
{
    this.maxAgeBeforeDemotion = maxAgeBeforeDemotion;
}

public void Add(TKey key, TValue value)
{
    RemoveFromWeak(key);
    var container = new ValueContainer <TValue >();
    container.value = value;
    container.additionTime = Stopwatch.GetTimestamp();
    container.demoteTime = 0;
    this.strongReferences.AddOrUpdate(
      key,
      container ,
      (k, existingValue) => container);
}

private void RemoveFromWeak(TKey key)
{
    WeakReference <ValueContainer <TValue >> oldValue;
    weakReferences.TryRemove(key, out oldValue);
}

public bool TryGetValue(TKey key, out TValue value)
{
    value = null;
    ValueContainer <TValue > container;
    if (this.strongReferences.TryGetValue(key, out container))
    {
        AttemptDemotion(key, container);
        value = container.value;
        return true;
    }

    WeakReference <ValueContainer <TValue >> weakRef;
    if (this.weakReferences.TryGetValue(key, out weakRef))
```

```
        {
            if (weakRef.TryGetTarget(out container))
            {
                value = container.value;
                return true;
            }
            else
            {
                RemoveFromWeak(key);
            }
        }
        return false;
    }

    /// <summary >
    /// Этот метод периодически вызывается из другого потока.
    /// </summary >
    public void DemoteOldObjects()
    {
        var demotionList =
          new List<KeyValuePair <TKey,
                                 ValueContainer <TValue >>>();
        long now = Stopwatch.GetTimestamp();

        foreach (var kvp in this.strongReferences)
        {
            var age = CalculateTimeSpan(kvp.Value.additionTime ,
                                        now);
            if (age > this.maxAgeBeforeDemotion)
            {
                demotionList.Add(kvp);
            }
        }

        foreach (var kvp in demotionList)
        {
            Demote(kvp.Key, kvp.Value);
        }
    }
    private void AttemptDemotion(TKey key,
                                 ValueContainer <TValue > container)
    {
        long now = Stopwatch.GetTimestamp();
        var age = CalculateTimeSpan(container.additionTime , now);
        if (age > this.maxAgeBeforeDemotion)
        {
            Demote(key, container);
        }
    }

    private void Demote(TKey key,
                        ValueContainer <TValue > container)
```

```
    {
        ValueContainer <TValue > oldContainer;
        this.strongReferences.TryRemove(key, out oldContainer);
        container.demoteTime = Stopwatch.GetTimestamp();
        var weakRef =
          new WeakReference <ValueContainer <TValue >>(container);
        this.weakReferences.AddOrUpdate(key,
                                        weakRef ,
                                        (k, oldRef) => weakRef);
    }

    private static TimeSpan CalculateTimeSpan(long offsetA ,
                                              long offsetB)
    {
        long diff = offsetB - offsetA;
        double seconds = (double)diff / Stopwatch.Frequency;
        return TimeSpan.FromSeconds(seconds);
    }
}
```

Пример: вторичный индекс

В этом примере слабые ссылки используются для более эффективного обновления простой базы данных за счет избавления от немедленных потенциально затратных обновлений индексов:

```
class Person
{
    public string Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
}

class PersonDatabase
{
    private Dictionary <string , Person > index =
      new Dictionary <string , Person >();
    private Dictionary <DateTime ,
                        List<WeakReference <Person >>>
                          birthdayIndex =
      new Dictionary <DateTime , List<WeakReference <Person >>>();

    public bool NeedsIndexRebuild { get; private set; }

    public void AddPerson(Person person)
    {
        this.index[person.Id] = person;
        List<WeakReference <Person >> birthdayList;
        if (!this.birthdayIndex.TryGetValue(person.Birthday ,
                                            out birthdayList))
```

```
        {
            birthdayIndex[person.Birthday]
             = birthdayList
             = new List<WeakReference <Person >>();
        }

        birthdayList.Add(new WeakReference <Person >(person));
    }

    public void RemovePerson(string id)
    {
        index.Remove(id);
    }

    public bool TryGetById(string id, out Person person)
    {
        return this.index.TryGetValue(id, out person);
    }

    public bool TryGetByBirthday(DateTime birthday ,
                                 out List<Person > people)
    {
        people = null;
        List<WeakReference <Person >> weakPeople;
        if (this.birthdayIndex.TryGetValue(birthday ,
                           out weakPeople))
        {
            var list = new List<Person >(weakPeople.Count);
            foreach(var wp in weakPeople)
            {
                Person person;
                if (wp.TryGetTarget(out person))
                {
                    list.Add(person);
                }
                else
                {
                    //   мы получили нулевой указатель -
                    //   надо перестроить индексы
                    this.NeedsIndexRebuild = true;
                }
            }
            if (list.Count > 0)
            {
                people = list;
                return true;
            }
        }
        return false;
    }
}
```

Воскрешение объектов

Существует перезагружаемый вариант конструктора `WeakReference<T>`, получающий булево значение `trackResurrection` (отслеживать воскрешение):

```
WeakReference <MyObject > weakRef =
    new WeakReference <MyObject >(myObj , trackResurrection: true);
```

Воскрешение происходит при совершении в финализаторе класса действий, подобных следующему:

```
class MyObject
{
    static MyObject myObj;

    ~MyObject()
    {
        myObj = this;
    }
}
```

Иначе говоря, берется объект, больше не имеющий на себя ссылок (именно поэтому и запущен финализатор), и на него снова добавляется ссылка. Этот прием используется в продвинутых сценариях кэширования, но у него есть ряд недостатков.

- Объект был сборщиком мусора повышен до поколения gen 1 и никогда уже не будет понижен до более раннего поколения.
- В отношении объекта нужно вызвать метод `GC.ReRegisterForFinalizer`, иначе для него еще один запуск финализатора не состоится.
- Состояние объекта может быть неопределенным. Объекты с нативными ресурсами будут вынуждены их высвободить и станут нуждаться в повторной инициализации. Работа с точным состоянием объекта может оказаться весьма непростой.
- Любые объекты, на которые ссылается воскрешенный объект, также являются воскрешенными. Если у любого из них имеются финализаторы, то к моменту воскрешения они тоже отработают, еще больше усложняя ваше состояние.

Если нет реального понимания состояния тех объектов, с которыми ведется работа, этот прием следует рассматривать как ошибочный. Для повторного использования объектов есть более удачные способы.

Если же задействуется данный прием, то методу `WeakReference<T>` следует сообщить о разрешении получить более продолжительный доступ к внутреннему объекту. Если у объекта нет финализатора, этот параметр не действует.

Динамически выделяйте память в стеке

Вместо выделения памяти из кучи можно выделять буферы динамических размеров в стеке, используя метод `stackalloc`. Такое выделение происходит быстрее, чем в куче, и не подвергается сборке мусора. Но есть ряд существенных оговорок.

- Это явно небезопасное действие. Вместо управляемого массива вам возвращается указатель на начало буфера, и вы несете ответственность за то, чтобы не пересечь его границы.
- Имеются серьезные ограничения на количество выделяемых данных. Управляемые стеки обычно ограничиваются размером 1 Мбайт (или всего 256 Кбайт в ASP.NET/IIS). Каждый стековый фрейм забирает часть этого пространства, и некоторые среды могут иметь очень глубокие стеки.

Чтобы увидеть, как работает `stackalloc`, рассмотрите программу из примера StackAlloc, содержащую следующий код:

```
private static unsafe void DoStackAlloc(int size)
{
    int* buffer = stackalloc int[size];
    for (int i = 0; i < size; i++)
    {
        buffer[i] = i;
    }
}
```

Остальная часть программы запускает этот код в цикле, запрашивая ввод данных о количестве выделяемой памяти. Пример такой работы имеет следующий вид:

```
Enter size to stackalloc ('q' to exit): 100
Allocated 100-size array
Enter size to stackalloc ('q' to exit): 200
Allocated 200-size array
Enter size to stackalloc ('q' to exit): 100000
Allocated 100000-size array
Enter size to stackalloc ('q' to exit): 1000000

Process is terminated due to StackOverflowException.
```

У исключения `StackOverflowException` имеется характерная особенность: ваша программа не может его перехватить. В коде примера выделение памяти превращается в обработку исключений, но тщетно. При выдаче этого исключения происходит моментальный выход из вашего приложения. Это исключение можно лишь перехватить при запуске под управлением отладчика.

Несмотря на присущие ему риски и ограничения, `stackalloc` считается весьма ценным инструментом, когда в методах нужны небольшие массивы динамически задаваемых размеров без затрат на издержки выделения памяти в куче.

Исследование памяти и сборки мусора

В этом разделе будут рассмотрены многочисленные советы и приемы, относящиеся к исследованию всего происходящего в куче, в которой идет сборка мусора. Во многих случаях одну и ту же информацию можно получить от нескольких инструментальных средств. Я попробую рассказать об использовании некоторых в каждом из применяемых сценариев.

Счетчики производительности

Среда .NET предоставляет несколько счетчиков производительности Windows, относящихся к категории .NET CLR Memory. Все они, за исключением скорости выделения памяти в байтах в секунду (Allocated Bytes/sec), обновляются в конце сборки мусора. Если значения застыли, причина этого, скорее всего, в нечастой сборке мусора.

- # Bytes in all Heaps (Количество байтов во всех кучах) — суммарный объем всех куч, за исключением поколения gen 0 (см. далее описание размера кучи поколения 0 (Gen 0 heap size)).
- # GC Handles (Количество дескрипторов сборщика мусора) — количество задействованных дескрипторов.
- # Gen 0 Collections (Количество сборок в поколении gen 0) — совокупное количество сборок мусора в gen 0 с момента запуска процесса. Следует заметить, что значение этого счетчика увеличивается также для сборок в поколениях 1 и 2, поскольку сборка мусора в более высоких поколениях никогда не обходится без сборки в поколениях более низкого уровня.
- # Gen 1 Collections (Количество сборок в поколении gen 1) — совокупное количество сборок в поколении gen 1 с момента запуска процесса. Следует заметить, что значение этого счетчика увеличивается также для сборок в поколении gen 2, поскольку сборка мусора в этом поколении не обходится без сборки в gen 1.
- # Gen 2 Collections (Количество сборок в поколении gen 2) — совокупное количество сборок в поколении gen 2 с момента запуска процесса.
- # Induced GC (Количество принудительных сборок мусора) — количество вызовов `GC.Collect` для явного запуска сборки мусора.
- # of Pinned Objects (Количество закрепленных объектов) — количество закрепленных объектов, повстречавшихся сборщику мусора в ходе сборки.
- # of Sink Blocks in use (Количество используемых блоков синхронизации) — у каждого объекта есть заголовок, в котором может храниться ограниченный объем информации, например хеш-код или сведения о синхронизации. При возникновении каких-либо конфликтов за использование заголовка создается блок синхронизации. Эти блоки применяются также для метаданных взаимодействия (interop). Высокое значение данного счетчика может показать

наличие конфликта блокировок. И да, в написании названия счетчика допущена ошибка (см. описание в PerfMon).

- # Total committed Bytes (Общее количество выделенных байтов) — количество байтов, выделенных сборщиком мусора, которое в действительности обеспечивается файлом страничного обмена.
- # Total reserved Bytes (Общее количество зарезервированных байтов) — количество байтов, зарезервированных, но еще не выделенных сборщиком мусора.
- % Time in GC (Процент времени, затрачиваемого на сборку мусора) — процент времени, затрачиваемого процессором в потоках сборки мусора по сравнению с другими процессами с момента последней сборки. Этот счетчик не берет в расчет сборку мусора в фоновом режиме.
- Allocated Bytes/sec (Скорость выделения памяти в байтах в секунду) — количество байтов, выделенных в GC-куче в секунду. Этот счетчик обновляется не постоянно, а только при запуске сборки мусора.
- Finalization Survivors (Количество выживших из-за финализации) — количество объектов, подвергаемых финализации, переживших сборку мусора по причине ожидания финализации, которая случается только в поколении gen 1 (см. также Promoted Finalization-Memory from Gen 0).
- Gen 0 heap size (Размер кучи gen 0) — максимальное количество байтов, которое может быть выделено в gen 0, а не фактическое количество выделенных байтов.
- Gen 0 Promoted Bytes/Sec (Скорость продвижения объектов из gen 0 в байтах в секунду) — скорость продвижения из gen 0 в gen 1. Нужно стремиться к сведению этого числа к минимуму, что будет свидетельствовать о коротких сроках задействования памяти.
- Gen 1 heap size (Размер кучи gen 1) — количество байтов в gen 1 с момента последней сборки мусора.
- Gen 1 Promoted Bytes/Sec (Скорость продвижения объектов из gen 1 в байтах в секунду) — скорость продвижения из gen 1 в gen 2. Высокий показатель свидетельствует о весьма длительных сроках задействования памяти и наличии подходящих кандидатов на объединение в пулы.
- Gen 2 heap size (Размер кучи gen 2) — количество байтов в gen 2 с момента последней сборки мусора.
- Large Object Heap Size (Размер кучи больших объектов) — количество байтов кучи больших объектов.
- Promoted Finalization–Memory from Gen 0 (Финализируемая память, продвинутая из gen 0) — общее количество байтов, продвинутое в gen 1, поскольку существует ожидающий финализации объект, находящийся где-то в дереве ссылок объектов, которым принадлежат эти байты. Это количество не только памяти, непосредственно занятой финализируемыми объектами, но и памяти, на которую указывают любые ссылки, удерживаемые такими объектами.

- Promoted Memory from Gen 0 (Объем памяти, продвинутой из gen 0) — количество байтов, продвинутых из gen 0 в gen 1 с момента последней сборки.
- Promoted Memory from Gen 1 (Объем памяти, продвинутой из gen 1) — количество байтов, продвинутых из gen 1 в gen 2 с момента последней сборки.

События ETW

В среде CLR выдается множество событий, касающихся поведения сборщика мусора. В большинстве случаев для их анализа можно воспользоваться собранным для вас инструментарием, но все же будет полезно разобраться в том, как регистрируется данная информация, чтобы вы могли при необходимости отследить источники конкретных событий и связать их с другими событиями, происходящими в приложении. Подробный анализ можно выполнить в PerfView с помощью представления событий (events). Рассмотрим наиболее важные события.

- GCStart_V1 — сборка мусора запущена. Поля:
  - `Count` — количество сборок, случившихся с момента запуска процесса;
  - `Depth` — в каком поколении происходит сборка;
  - `Reason` — что стало причиной запуска сборки;
  - `Type` — блокирующая, фоновая или блокирующая в ходе фонового выполнения.
- GCEnd_V1 — сборка мусора завершена. Поля:
  - `Count`, `Depth` — те же, что и для `GCStart`.
- GCHeapStats_V1 — статистика в конце сборки мусора. Предоставляет множество полей для описания всех характеристик кучи, таких как размеры поколений, количество продвинутых байтов, финализация, дескрипторы и многое другое.
- GCCreateSegment_V1 — создан новый сегмент. Поля:
  - `Address` — адрес сегмента;
  - `Size` — размер сегмента;
  - `Type` — куча малых или больших объектов.
- GCFreeSegment V1 — освобожден сегмент. Всего одно поле:
  - `Address` — адрес сегмента.
- GCAllocationTick_V2 — выдается при каждом выделении около 100 Кбайт памяти (суммарно). Поля:
  - `AllocationSize` — точный размер выделения, инициировавшего событие;
  - `Kind` — выделение в куче малых или больших объектов.
- GCFinalizersBegin_V1 — начало работы финализаторов.

- GCFinalizersEnd_V1 — завершение работы финализаторов:
  - Count — количество выполненных финализаторов.
- GCCreateConcurrentThread_V1 — создание параллельного потока сборки мусора.
- GCTerminateConcurrentThread_V1 — завершение параллельного потока сборки мусора.
- GCSuspendEE_V1 — начало приостановки потоков:
  - Reason — причина инициализации приостановки;
  - Count — показание количества сборок мусора на момент выдачи этого события.
- GCSuspendEEEnd_V1 — потоки приостановлены.
- GCRestartEEBegin_V1 — потоки начинают возобновлять работу.
- GCRestartEEEnd_V1 — потоки завершили возобновление работы.

Порядок получаемых событий играет важную роль. Для обычных сборок мусора любого поколения, выполняемых в фоновом режиме, выдерживается следущий порядок.

1. GCSuspendEE_V1 — начало приостановки потоков.
2. GCSuspendEEEnd_V1 — все потоки приостановлены.
3. GCStart_V1 — начало сборки мусора.
4. GCEnd_V1 — работа сборщика мусора завершена.
5. GCRestartEEBegin_V1 — начало возобновления работы потоков.
6. GCRestartEEEnd_V1 — работа всех потоков возобновлена. Сборка мусора завершена.

Если вам нужно проанализировать эти события в своих приложениях или утилитах, изучите разделы, посвященные TraceEvent и PerfView главы 8, где будет представлена простая в освоении библиотека. Благодаря анализу и разумному использованию ETW-событий вы сможете определить, влияет ли на исполнение вашего приложения сборщик мусора или любой другой источник внешнего воздействия.

Как выглядит куча памяти моего приложения

WinDbg может показать несколько различных представлений кучи. Сначала по сегментам:

```
!eeheap -gc
```

Вывод будет выглядеть примерно так:

```
Number of GC Heaps: 1
generation 0 starts at 0x05824e2c
generation 1 starts at 0x0532100c
```

```
generation 2 starts at 0x05321000
ephemeral segment allocation context: none
 segment begin allocated size
05320000 05321000 05891ff4 0x570ff4(5705716)
Large object heap starts at 0x06321000
 segment begin allocated size
06320000 06321000 07312c80 0xff1c80(16718976)
07900000 07901000 088ee660 0xfed660(16701024)
08a30000 08a31000 09a1e660 0xfed660(16701024)
09c80000 09c81000 0ac6e660 0xfed660(16701024)
0ac80000 0ac81000 0bc6e540 0xfed540(16700736)
...more segments...
Total Size: Size: 0x213b9d94 (557555092) bytes.
------------------------------
GC Heap Size: Size: 0x213b9d94 (557555092) bytes.
```

Если в процессе запущена сборка мусора в режиме сервера, то будет выведено более одной кучи, каждая со своим собственным набором из эфемерного сегмента, сегмента gen 2 и сегмента больших объектов.

Еще одно представление получается с помощью команды `!HeapStat`, агрегирующей все сегменты с разбиением по поколениям для вывода их размеров, включая свободное пространство.

```
0:007> !HeapStat
Heap         Gen0         Gen1      Gen2                 LOH
Heap0      446920      5258784        12           551849376

Free space:                                          Percentage
Heap0          12         1948         0             15936SOH: 0% LOH: 0%
```

В выведенной на экран информации показано, что на кучу gen 2 приходится очень мало памяти и количество свободного пространства в ней незначительное (то есть низкая фрагментация). Аббревиатурой SOH (Small Object Heap) обозначается куча малых объектов — все то, что не относится к сегментам кучи больших объектов (Large Object Heap).

С помощью команды `!VMMap` выводится информация об областях виртуальных адресов и применяемых к ним уровнях защиты:

```
0:000> !VMMap
Start    Stop     Length   AllocProtect Protect State     Type
00000000-00f5ffff 00f60000               NA      Free
00f60000-00f60fff 00001000 ExWrCp       Rd      Commit    Image
00f61000-00f61fff 00001000 ExWrCp               Reserve   Image
00f62000-00f62fff 00001000 ExWrCp       Rd      Commit    Image
00f63000-00f63fff 00001000 ExWrCp               Reserve   Image
00f64000-00f64fff 00001000 ExWrCp       Rd      Commit    Image
00f65000-00f65fff 00001000 ExWrCp               Reserve   Image
00f66000-00f66fff 00001000 ExWrCp       Rd      Commit    Image
00f67000-00f67fff 00001000 ExWrCp               Reserve   Image
...
```

С помощью команды `!VMStat` данная информация будет обобщена по состоянию — `State`:

```
0:000> !VMStat
TYPE       MINIMUM      MAXIMUM      AVERAGE     BLK COUNT          TOTAL
====       =======      =======      =======     =========          =====
Free:
Small           8K          64K          43K            30         1,315K
Medium         84K         996K         332K            10         3,323K
Large       1,152K   2,090,816K     204,209K            17     3,471,563K
Summary         8K   2,090,816K      60,986K            57     3,476,203K

Reserve:
Small           4K          64K          34K            34         1,183K
Medium         68K       1,012K         299K            56        16,779K
Large       1,376K      32,768K      12,073K             7        84,515K
Summary         4K      32,768K       1,056K            97       102,479K

Commit:
Small           4K          64K          12K           204         2,575K
Medium         68K         964K         347K            44        15,307K
Large       1,048K      16,332K      12,716K            47       597,671K
Summary         4K      16,332K       2,086K           295       615,555K

Private:
Small           4K          64K          19K            88         1,716K
Medium         68K       1,012K         285K            57        16,267K
Large       1,376K      32,768K      15,215K            41       623,851K
Summary         4K      32,768K       3,450K           186       641,835K

Mapped:
Small           4K          64K          25K             8           204K
Medium         68K       1,004K         374K             6         2,247K
Large       1,540K      18,320K       5,442K             5        27,211K
Summary         4K      18,320K       1,561K            19        29,663K

Image:
Small           4K          64K          12K           142         1,839K
Medium         68K         964K         366K            37        13,571K
Large       1,048K      15,712K       3,890K             8        31,124K
Summary         4K      15,712K         248K           187        46,535K
```

Входящее в SysInternals средство VMMap также может дать хорошую сводку всех сегментов процесса. После выбора процесса нужно выделить в таблице строчку, относящуюся к управляемой куче, — `Managed Heap`, и будет показан список всех сегментов, задействованных в процессе (рис. 2.8).

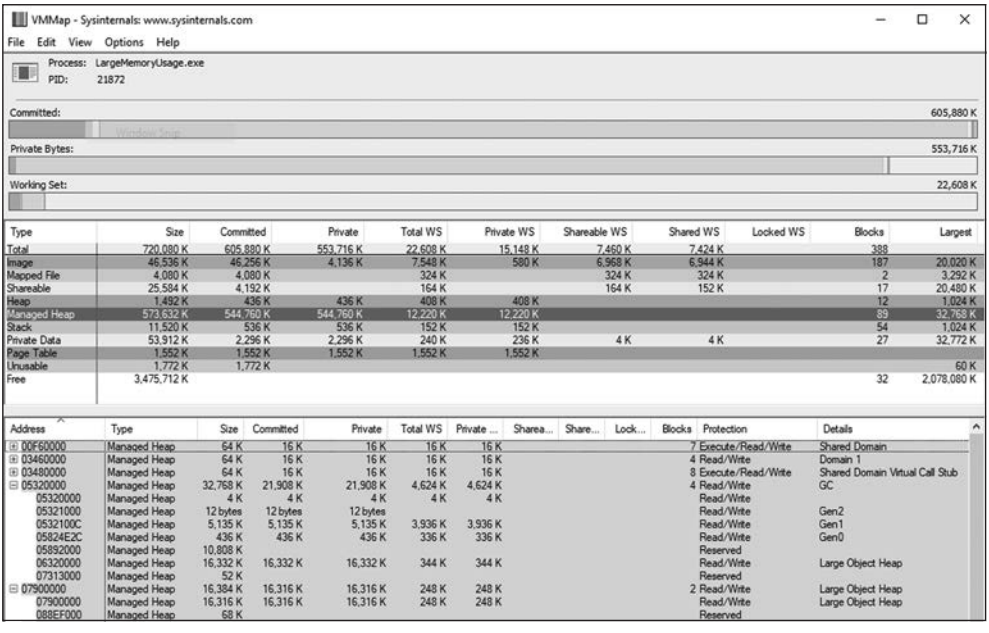

Рис. 2.8. Средство VMMap способно разделить на части все разнообразие областей памяти, использованных в процессе, включая сегменты GC-кучи

Сколько времени занимает сборка мусора

Сборщик мусора записывает множество событий, связанных с его операциями. Для эффективного анализа этих событий можно воспользоваться средством PerfView.

Чтобы увидеть статистику, относящуюся к сборке мусора, запустите программу из примера AllocateAndRelease.

Запустите PerfView и выполните следующие действия.

1. Выберите в меню Collect (Сбор) пункт Collect (Alt+C).
2. Раскройте пункт Advanced Options (Дополнительные параметры). При желании можно выключить все категории событий, кроме сборки мусора (GC Only), но сейчас просто оставьте без изменений исходные установки, поскольку GC-события включены в .NET-события.
3. Установите флажок No V3.X NGEN Symbols (Не использовать символы V3.X NGEN) — это ускорит разрешение символов.
4. Нажмите кнопку Start (Пуск).
5. Подождите несколько минут, в течение которых будут измеряться параметры активности процесса. Рассуждения по поводу количества собираемых выборок

можно найти в главе 1 (если сборка выполняется более нескольких минут, может потребоваться отключить события центрального процессора).

6. Нажмите кнопку Stop Collection (Остановить сбор).
7. Подождите окончания объединения файлов.
8. В получившемся дереве просмотра выполните двойной щелчок на узле `GCStats`, в результате чего откроется новое представление.
9. Найдите раздел для вашего процесса.

Для каждого процесса вы найдете набор данных и таблиц со сводной информацией о поведении сборщика мусора.

В верхней части каждого раздела имеется список элементов, предоставляющих общую информацию.

Элемент, предоставляющий общую информацию о сборке мусора	**Описание**
CommandLine	Точная командная строка, которая запустила процесс
Runtime Version	Выполняемая версия CLR
CLR Startup Flags	Флаги, контролирующие поведение сборщика мусора, например CONCURRENT_GC или SERVER_GC
Total CPU Time	Общее время в миллисекундах, потраченное на выполнение процесса за время профилирования
Total GC CPU Time	Общее время, потраченное на сборку мусора
Total Allocs	Количество случаев выделения памяти
GC CPU MSec/MB Alloc	Время в миллисекундах, которое сборщик потратил на обработку 1 Мбайт памяти
Total GC Pause	Время в миллисекундах, в течение которого процесс был на паузе из-за сборки мусора
% Time paused for Garbage Collection	Время на паузе из-за сборки мусора как процент от общего процессорного времени
% CPU Time spent Garbage Collecting	Процент времени, которое процессор провел, собирая мусор. Может отличаться от предыдущего показателя, если запущен серверный режим сборки мусора
Max GC Heap Size	Максимальный размер кучи, подвергнутой сборке мусора, за время профилирования
Peak Process Working Set	Максимальный размер рабочего множества за время профилирования
Peak Virtual Memory Usage	Максимальный размер зарезервированной виртуальной памяти за время профилирования

Столбец таблицы обобщенной информации о сборке мусора	Описание
Gen	Поколение, включая ALL (Все), где все сборки мусора объединяются в единый статистический набор
Count	Количество сборок
Max Pause	Наибольшее время паузы в сборке мусора в миллисекундах
Max Peak MB	Максимальный размер поколения в куче
Max Alloc MB/sec	Пиковая скорость выделения
Total Pause	Суммарное время всех пауз в миллисекундах
Total Alloc MB	Объем выделенной памяти
Alloc MB/MSec GC	Объем выделенной памяти за миллисекунду времени сборки мусора. Это показатель эффективности сборки мусора. Чем он выше, тем эффективнее сборка мусора (то есть тем меньше она влияет на выполнение программы)
Survived MB/MSec GC	Объем выжившей памяти за миллисекунду времени сборки мусора. Это еще один показатель эффективности сборки мусора. Большее значение свидетельствует о выживании более существенного объема памяти
Mean Pause	Среднее время пауз в миллисекундах
Induced	Количество явных вызовов сборки мусора (GC.Collect)

Столбец таблицы подробной информации о сборке мусора	Описание
GC Index	Порядок, в котором произошла сборка мусора
Pause Start	Отметка времени в миллисекундах от начала профилирования до начала выполнения сборки мусора
Trigger Reason	Причина, по которой выполнялась сборка мусора
Gen	Поколение и буква кода, показывающая тип сборки мусора. Поколение обозначается цифрой в диапазоне 0–2. N обозначает NonConcurrent (Последовательная), B — Background (Фоновая), F — Foreground (Первого плана), I — Induced (Вынужденная), i — Induced, not forced (Вынужденная, но не принудительная)
Suspend Msec	Количество миллисекунд, понадобившееся для приостановки запущенных потоков
Pause Msec	Общее время приостановки процесса для сборки мусора

Продолжение ⇩

(Продолжение)

Столбец таблицы подробной информации о сборке мусора	Описание
% Pause Time	Процент времени на сборку мусора со времени предыдущей сборки
% GC	Процент времени ЦП, затраченного на сборку мусора
Gen0 Alloc MB	Объем выделенной памяти в нулевом поколении со времени предыдущей сборки мусора
Gen0 Alloc Rate MB/sec	Скорость выделения памяти в нулевом поколении со времени предыдущей сборки мусора
Peak MB	Пиковый размер кучи в ходе сборки мусора
After MB	Размер кучи после завершения сборки мусора
Ratio Peak/After	Эффективность. Чем выше показатель, тем лучше
Promoted MB	Объем памяти, пережившей сборку мусора
Gen0 MB	Размер нулевого поколения после завершения данной сборки мусора
Gen0 Survival Rate %	Процент объектов в нулевом поколении, переживших сборку мусора
Gen 0 Frag %	Процент свободной памяти в нулевом поколении
Gen 1 MB	Размер первого поколения после завершения данной сборки мусора
Gen1 Survival Rate %	Процент объектов в первом поколении, переживших сборку мусора
Gen1 Frag %	Процент свободной памяти в первом поколении
Gen2 MB	Размер второго поколения после завершения данной сборки мусора
Gen2 Survival Rate %	Процент объектов во втором поколении, переживших сборку мусора
Gen2 Frag %	Процент свободной памяти во втором поколении
LOH MB	Размер LOH после завершения данной сборки мусора
LOH Survival Rate %	Процент объектов в LOH, переживших сборку мусора
LOH Frag %	Процент свободной памяти в LOH
Finalizable Surv MB	Размер финализируемых объектов, переживших сборку мусора
Pinned Obj	Количество закрепленных объектов, продвинутых данной сборкой мусора. Чем показатель меньше, тем лучше

Под ним будет находиться таблица сводной информации обо всех поколениях сборки мусора (рис. 2.9).

GC Rollup By Generation

All times are in msec.

Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/ MSec GC	Survived MB/ MSec GC	Mean Pause	Induced
ALL	12651	12.3	5.5	291.128	1,495.3	25,648.2	17.2	Infinity	0.1	0
0	11780	0.7	5.4	261.359	716.6	23,816.5	0.0	Infinity	0.1	0
1	0	0.0	0.0	0.000	0.0	0.0	0.0	NaN	NaN	0
2	871	12.3	5.5	291.128	778.7	1,831.7	0.0	Infinity	0.9	0

Рис. 2.9. Таблица GCStats для программ из примера AllocateAndRelease. В ней показаны количество выполненных сборок мусора, а также интересная статистика, включая среднее и максимальное время пауз и скорости выделения памяти

Ниже этой таблицы будут еще более подробные таблицы с перечислением конкретных случаев сборки мусора в различных категориях, например Pauses > 200 MSec (Паузы более 200 мс), LOH Allocation Pause (due to background GC) > 200 MSec (Пауза выделения в LOH, вызванная фоновой сборкой мусора, превышающая 200 мс), Gen 2 (Поколение 2) и All GC Events (Все события, относящиеся к сборке мусора).

Как видите, информации, касающейся каждого GC-события, которой можно воспользоваться для анализа производительности сборки мусора, предостаточно.

Где именно происходит выделение памяти

Среда Visual Studio может отслеживать выделение памяти в .NET через выборки событий ETW. Следует заметить, что это в корне отличается от работы профилировщика Memory Usage. По сути, этот отчет является инструментом для анализа дампа кучи, в котором показаны моментальные состояния объектов в куче и ссылки на объекты, которым они принадлежат, вплоть до корня. Отчет о выделении .NET-памяти в Performance Wizard, в свою очередь, использует ETW-события чтобы понять, какими именно методами было фактически произведено выделение памяти, независимо от того, чьими в конечном итоге стали ссылки (рис. 2.10). Это средство использует ETW-событие `GCAllocationTick_V2`, выдаваемое средой CLR после каждых выделенных 100 Кбайт.

Щелчок на имени метода вновь перенесет вас в уже известное представление Function Details. Только в данном случае будет показано выделение памяти, а не время центрального процессора (рис. 2.11).

У этого отчета имеется множество других представлений, позволяющих углубиться в различные измерения. Особый интерес вызывает представление Allocation. Вот что в нем будет показано после щелчка на названии типа в главном сводном представлении (рис. 2.12).

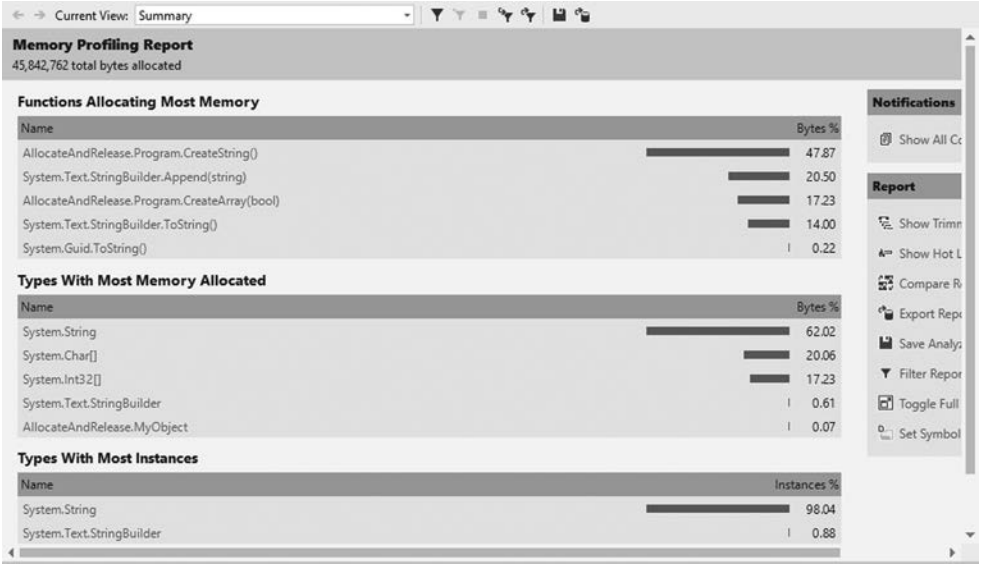

Рис. 2.10. Отчет о профилировании памяти (Memory Profiling Report), показывающий, какими методами выделен основной объем памяти, а также какие типы стали потребителями основных объемов памяти

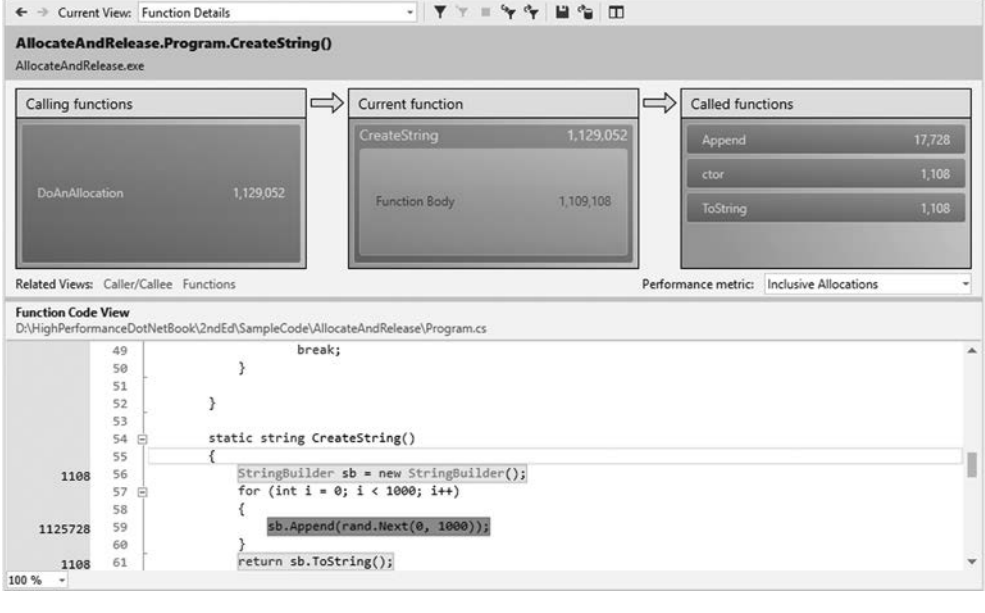

Рис. 2.11. Подробности о функциях, вызывающих выделение памяти. Здесь показано, какие вызываемые методы, а также строки в исходном коде ответственны за выделение памяти

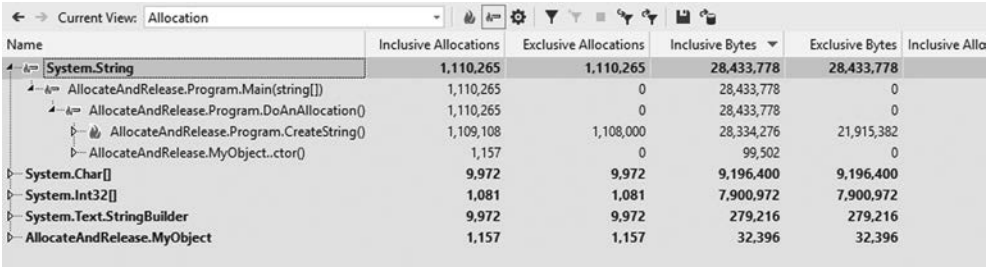

Рис. 2.12. Сводка выделений памяти по типу, а не по стеку методов

Это представление агрегировано типу и показывает, какие методы каждого конкретного типа выделяли память наиболее часто.

Как альтернатива с той же целью может использоваться PerfView, который способен показать ту же самую информацию, что и Visual Studio, и даже больше, но его интерфейс не настолько совершенен.

1. С помощью PerfView можно собрать либо .NET-события, либо события, относящиеся исключительно к сборке мусора (GC Only).
2. По завершении сборки откройте представление GC Heap Alloc Stacks (Стеки распределений в GC-куче) и выберите из списка процессов нужный процесс (для простого примера воспользуйтесь программой AllocateAndRelease).
3. На вкладке By Name (По имени) будут видны типы, отсортированные по общему размеру выделенной памяти. Двойной щелчок на названии типа перенесет вас на вкладку Callers (Вызывающие), где будут показаны стеки, в которых произошло это выделение.

Дополнительная информация о применении интерфейса PerfView для получения более весомой отдачи от представления дана в главе 1.

Воспользовавшись изложенной информацией, вы сможете найти стеки для всех распределений, произошедших в тестовой программе, и относительную частоту их реализации. Например, в моем случае выделение памяти под строки составляет примерно 59,5 % от всех случаев выделения памяти (рис. 2.13).

Кроме того, можно воспользоваться профилировщиком CLR Profiler, который способен найти ту же информацию и отобразить ее несколькими способами.

После сбора информации и открытия окна Summary (Сводка) нажмите кнопку Allocation Graph (Диаграмма выделений), чтобы открыть графическое представление связей между выделением памяти для объектов и методами, ответственными за это (рис. 2.14).

Объекты, которым наиболее часто требуется выделение памяти, — это наиболее вероятные инициаторы запуска сборки мусора. Сокращение числа случаев выделения памяти приведет к снижению частоты сборки мусора.

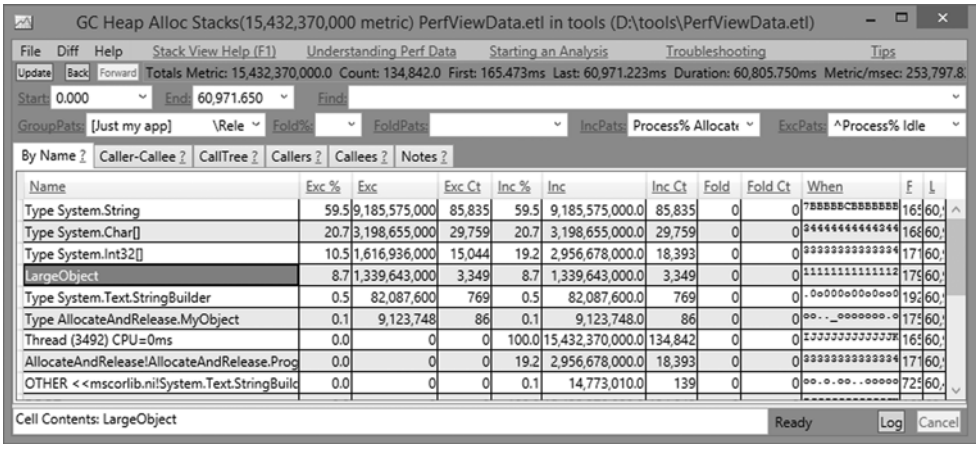

Рис. 2.13. Представление GC Heap Alloc Stacks, показывающее наиболее распространенные случаи выделения памяти в вашем процессе. Запись LargeObject — это псевдоузел, двойной щелчок на нем покажет фактические объекты, распределенные в LOH

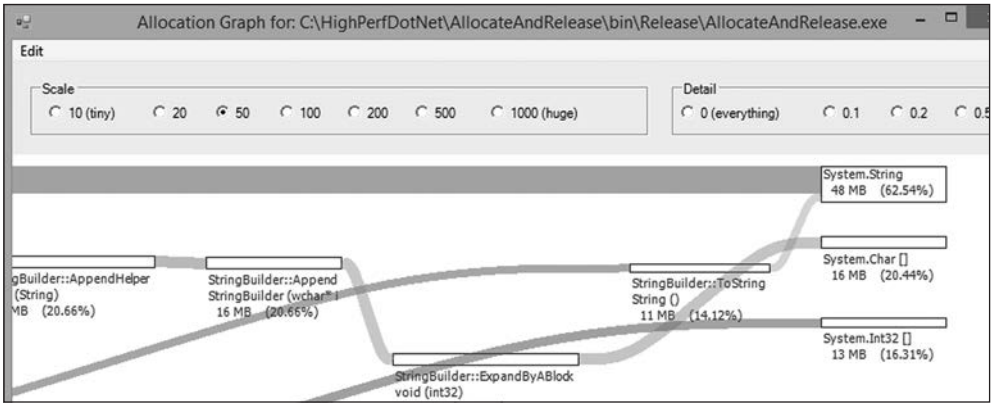

Рис. 2.14. Визуальное отображение стеков выделения объектов, предоставляемое профилировщиком CLR Profiler, быстро указывает вам на объекты, которым нужно уделить наиболее пристальное внимание

Что за объекты находятся в куче

Все инструментальные средства, упоминаемые в данном разделе, будут использовать программу из примера LargeMemoryUsage, код которой показан далее:

```
class Program
{
  const int ArraySize = 1000;
  static object[] staticArray = new object[ArraySize];
```

```
  static void Main(string[] args)
  {
    var localArray = new object[ArraySize];

    var rand = new Random();
    for (int i = 0; i < ArraySize; i++)
    {
      staticArray[i] = GetNewObject(rand.Next(0, 4));
      localArray[i] = GetNewObject(rand.Next(0, 4));
    }

    Console.WriteLine("Examine heap now. Press any key to exit.");
    Console.ReadKey();

    // Это помешает сборщику мусора вычистить из памяти массив localArray,
    // прежде чем вы сделаете снимок его состояния
    Console.WriteLine(staticArray.Length);
    Console.WriteLine(localArray.Length);
}

private static Base GetNewObject(int type)
{
     Base obj = null;
     switch (type)
     {
        case 0: obj = new A(); break;
        case 1: obj = new B(); break;
        case 2: obj = new C(); break;
        case 3: obj = new D(); break;
     }
     return obj;
  }
}

class Base
{
  private byte[] memory;
  protected Base(int size) { this.memory = new byte[size]; }
}

class A : Base { public A() : base(1000) { } }
class B : Base { public B() : base(10000) { } }
class C : Base { public C() : base(100000) { } }
class D : Base { public D() : base(1000000) { } }
```

Эта простая программа всего лишь выделяет память для случайного количества объектов различных классов и ожидает от вас анализа кучи, прежде чем завершить свою работу.

Существует несколько способов анализа данной кучи, начиная с самого низкоуровневого.

С помощью WinDbg можно выполнить команду `!DumpHeap`, чтобы просто получить дамп памяти с перечислением каждого отдельно взятого объекта в куче:

```
0:007> !DumpHeap
Address        MT        Size
02aa1000 00b2ac70          10 Free
02aa100c 00b2ac70          10 Free
02aa1018 00b2ac70          10 Free
02aa1024 71911eac          84
02aa1078 71912000          84
02aa10cc 71912044          84
02aa1120 71912088          84
02aa1174 719120cc          84
02aa11c8 719120cc          84
02aa121c 71912104          12
02aa1228 71911d64          14
```

В столбце MT указывается адрес таблицы методов Method Table, что, по сути, эквивалентно классу.

Для получения информации о конкретном объекте можно вывести его дамп:

```
0:007> !DumpObj /d 02bb8cf4
Name:        LargeMemoryUsage.A
MethodTable: 00de4f6c
EEClass:     00de196c
Size:        12(0xc) bytes
File:        D:\SampleCode\...\LargeMemoryUsage.exe
Fields:
      MT   Field Offset          Type VT     Attr    Value Name
719160e8 4000003      4 System.Byte[]  0 instance 02bb8d00 memory
```

Сбор дампов для каждого объекта в куче обычно выдает громадный объем информации. К счастью, можно слегка отфильтровать вывод, например, по типу:

```
0:007> !DumpHeap -type LargeMemoryUsage.A
Address        MT Size
02aaba98 00de4f6c   12
02ab82cc 00de4f6c   12
02ab86cc 00de4f6c   12
02ab8acc 00de4f6c   12
```

Вывод можно отфильтровать так, чтобы отображались только объекты, находящиеся внутри указанного диапазона в куче (если, конечно, у вас есть адреса начала и конца этого диапазона):

```
!DumpHeap -type LargeMemoryUsage.A 02aaba98 02ab86cc
```

Существуют дополнительные параметры `DumpHeap`, позволяющие выполнять более изощренную фильтрацию результатов.

DumpHeap	Описание параметра
-min	Отображение объектов размером не менее заданного
-max	Отображение объектов размером не более заданного

DumpHeap	Описание параметра
-startatLowerBound	Запуск сканирования кучи с указанного адреса (это должен быть адрес объекта)
-type	Проверка присутствия аргумента как подстроки в названии типа
-mt	Отображение объектов с заданным адресом таблицы методов. Это более точный способ получения вывода конкретного типа объектов по сравнению с -type, где аргумент может соответствовать различным типам
-short	Вывод только адресов объектов
-strings	Отображение сводных данных о строках в куче
-stat	Отображение только статистической сводки

В WinDbg имеется элементарный скриптовый язык, однако расширенный анализ кучи может вызвать затруднения. В этом случае для анализа объектов можно воспользоваться средствами библиотеки CLR MD:

```
private static void PrintGen0Objects(ClrRuntime clr)
{
     var heap = clr.Heap;

     foreach(var obj in heap.EnumerateObjects())
     {
         Console.WriteLine($"0x{obj.Address:x} - {obj.Type.Name}");
     }
}
```

Поскольку тут имеется программный доступ к тем же свойствам объекта, что и в WinDbg, то можно как выполнить фильтрацию по тем же критериям, так и придумать более сложные критерии для поиска и анализа найденных объектов.

До сих пор рассматривались способы анализа каждого отдельно взятого объекта. Конечно же, это пригодится при отладке, но зачастую, анализируя общее поведение, требуется рассматривать все объекты в совокупности.

Начиная с выпуска Visual Studio 2013 Premium Edition (Enterprise Edition начиная с Visual Studio 2015), эта среда имеет анализатор управляемой кучи (рис. 2.15). Доступ к нему можно получить, открыв дамп управляемой памяти и выбрав пункт Debug Managed Memory (Отладка управляемой памяти).

Здесь можно выполнить три действия.

1. Просмотреть все экземпляры данного типа после двойного щелчка на его названии.
2. Просмотреть различные корни объектов (благодаря чему они существуют).
3. Просмотреть, на какие другие типы имеются ссылки из выделенного типа.

У профилировщика производительности (Performance Profiler) имеется также функция получения моментальных образов кучи в ходе выполнения программы. Для доступа к ней нужно выбрать в меню пункты Analyze ▸ Performance Profiler ▸ Memory Usage (Анализ ▸ Профилировщик производительности ▸ Использование памяти).

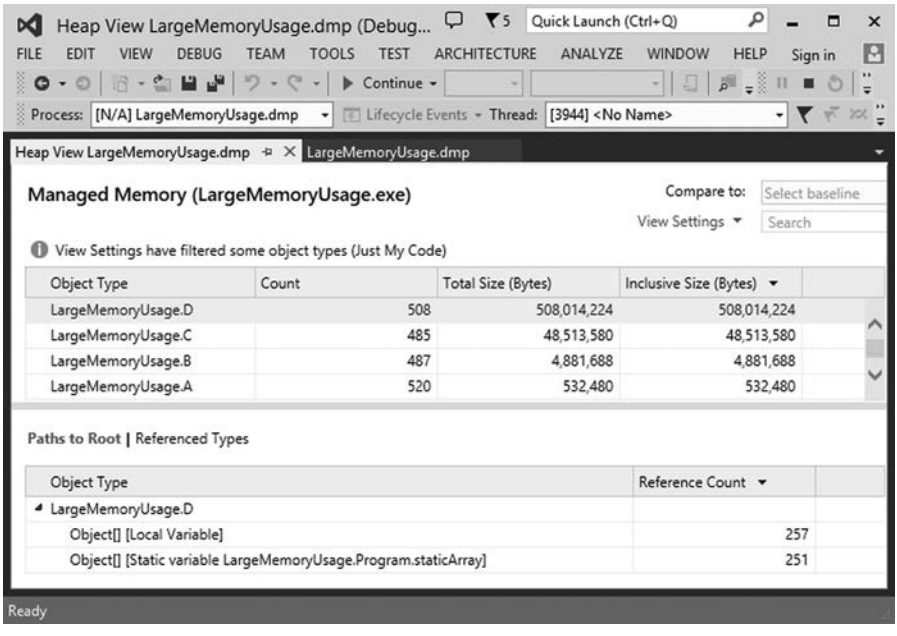

Рис. 2.15. Новые версии Visual Studio включают это отображение анализатора кучи, работающее на основе дампов управляемой памяти

Вывод будет представлять собой график использования памяти в сопоставлении со сборками мусора (рис. 2.16). При запущенном анализе можно получать моментальные образы кучи в любом месте.

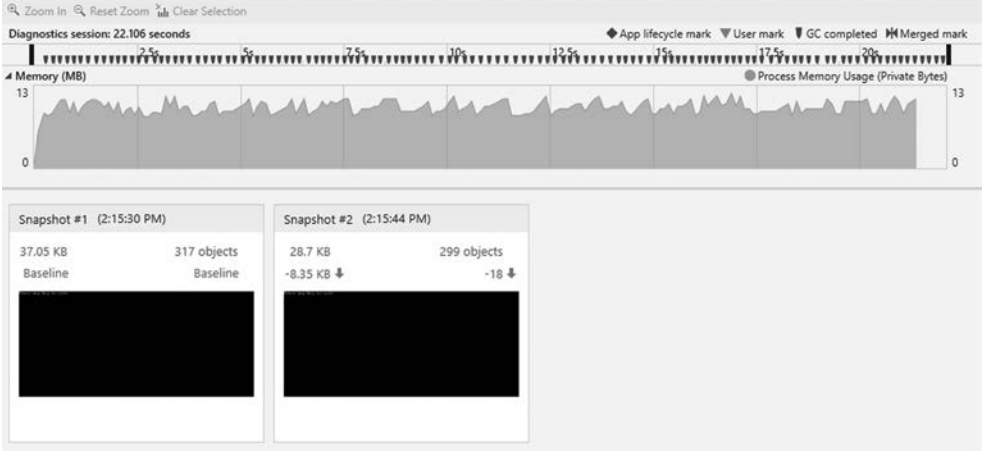

Рис. 2.16. График, показывающий совокупное использование памяти, сборки мусора и моментальные образы кучи

Щелкнув на размере или количестве объектов в моментальном образе, можно перейти к таблице всех объектов, имеющихся в куче, и их путей к корню (благодаря чему они существуют) (рис. 2.17).

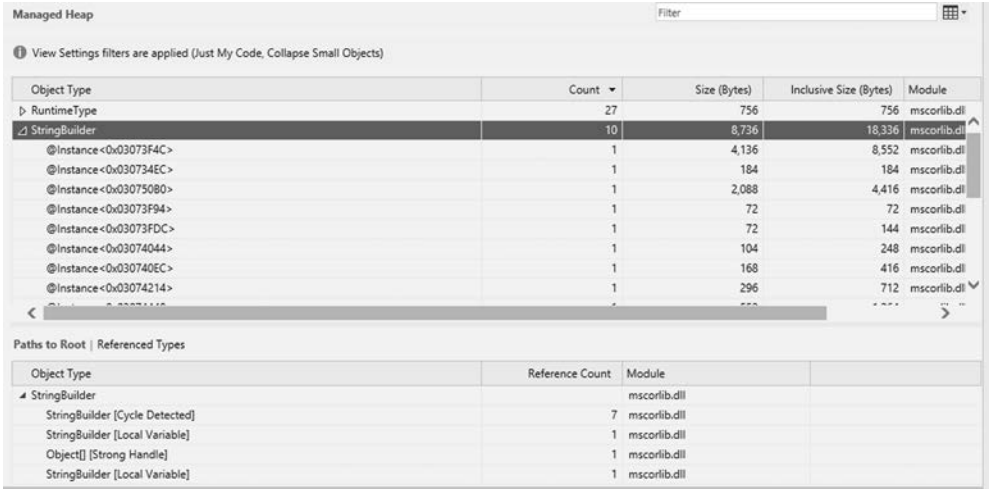

Рис. 2.17. Представление моментального образа кучи в Visual Studio. Здесь показаны различные пути к корню каждого типа объекта в совокупности или с конкретными экземплярами

Каждый моментальный образ также позволяет вам просматривать только те объекты, которые изменились со времени создания предыдущего моментального образа, помогая с анализом выполняемых случаев выделения памяти.

Эти функции предоставляют весьма простой, но полезный обзор кучи. Если требуются более серьезные средства анализа, я рекомендую воспользоваться средством PerfView. Оно не покажет отдельно взятые объекты, но его возможности отображения взаимоотношений объектов не имеют себе равных.

Чтобы воспользоваться этой функцией PerfView, нужно выполнить следующие действия.

1. В меню Memory (Память) выбрать пункт Take Heap Snapshot (Получить моментальный образ кучи). Следует отметить, что при этом процесс не становится на паузу, если только не установлен флажок Freeze (Заморозить), но на производительность процесса оказывается весьма существенное влияние.
2. Выделить нужный процесс в появившемся диалоговом окне.
3. Щелкнуть на пункте Dump GC Heap (Получить дамп GC-кучи).
4. Дождаться завершения сбора, после чего закрыть окно.
5. Открыть файл в дереве файлов PerfView (после закрытия окна сбора он может открыться в автоматическом режиме).

Должна появиться таблица, похожая на изображенную на рис. 2.18.

Здесь сразу видно, что на D приходится 88 % памяти программы — 462 Мбайт в 924 объектах. Можно также увидеть, что на локальные переменные затрачено до 258 Мбайт памяти, а на объект `staticArray` — 263 Мбайт.

Name	Exc %	Exc	Exc Ct	Inc %	Inc
LargeMemoryUsage!LargeMemoryUsage.D	88.5	462,013,000	924	88.5	462,013,000.0
LargeMemoryUsage!LargeMemoryUsage.C	10.4	54,213,010	1,084	10.4	54,213,010.0
LargeMemoryUsage!LargeMemoryUsage.B	1.0	5,242,552	1,046	1.0	5,242,552.0
LargeMemoryUsage!LargeMemoryUsage.A	0.1	484,352	946	0.1	484,352.0
[Pinned handle]	0.0	21,586	153	0.0	22,926.0
[local var]	0.0	4,032	3	49.5	258,117,400.0
[static var LargeMemoryUsage.Program.staticArray]	0.0	4,016	2	50.5	263,847,600.0

Рис. 2.18. Полученный посредством PerfView срез самых больших объектов в куче

Средство PerfView в своем роде уникально тем, что позволяет контролировать вклад подчиненных объектов в размер их родительских объектов. Это достигается конфигурацией свертки. Можно указать процент свертки, ниже которого вся память привязывается к родительскому объекту, или шаблон свертки, чтобы определить, какие конкретно типы объектов всегда свертываются в свои родительские объекты (они фактически исчезают из анализа). Подробности использования PerfView можно найти в главе 1.

Можно также получить графическое представление такой же информации с помощью профилировщика CLR Profiler (рис. 2.19). Для захвата выборки из кучи нужно в ходе выполнения программы нажать кнопку Show Heap Now (Показать кучу прямо сейчас).

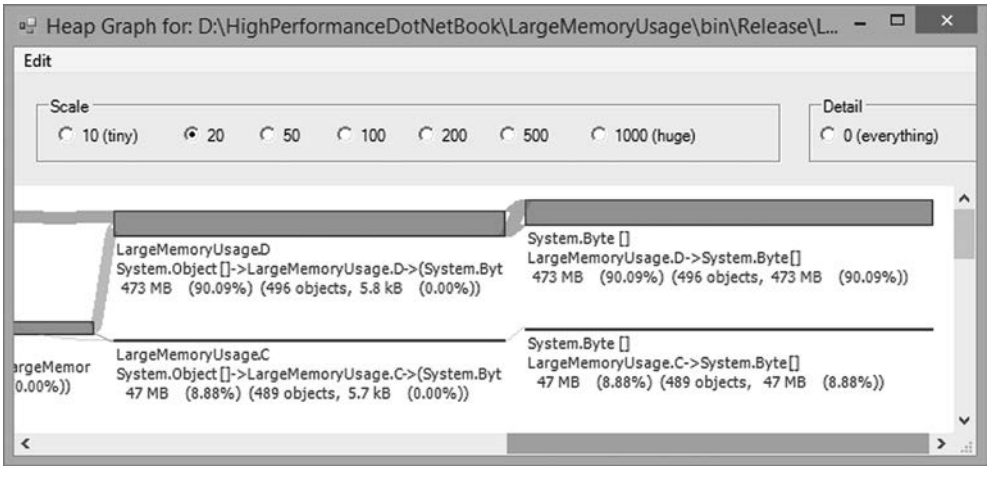

Рис. 2.19. Профилировщик CLR Profiler выдает ту же самую информацию, что и PerfView, но в графическом формате

Где именно допущена утечка памяти

Утечка памяти возможна во множестве случаев. Все подразделы раздела «Исследование памяти и сборки мусора» этой главы могут помочь вам сузить круг соответствующих проблем. Есть несколько общих путей возможной утечки памяти в управляемых приложениях.

- Для объектов сохраняются непредвиденные ссылки, не дающие их вычистить сборщиком мусора.
- Полная сборка мусора осуществляется крайне редко и сохраняет в процессе много памяти, которая не используется иным образом и не имеет корня. Обычно это не проблема и, можно сказать, соответствует изначальному замыслу.
- Высока степень фрагментации, особенно в поколении gen 2 или в куче больших объектов.
- Эффективной сборке мусора мешает большое количество закрепленных объектов. Диагностирование экстремальных ситуаций с закреплением объектов будет рассмотрено в данной главе чуть позже.

В среде Visual Studio (изданий Premium или Enterprise) можно открыть дамп кучи и выполнить отладку управляемой кучи. При щелчке на типе расположенные ниже вкладки позволят увидеть, какие другие типы ссылаются на рассматриваемые объекты.

Для более подробного анализа можно воспользоваться средством PerfView (рис. 2.20).

1. Выберите в меню Memory (Память) пункт Take Heap Snapshot (Получить моментальный образ кучи).
2. Выберите в открывшемся диалоговом окне анализируемый процесс и нажмите кнопку Dump GC Heap (Дамп GC-кучи). В дополнение к этому можно заморозить процесс или запустить принудительную сборку мусора до получения моментального образа.
3. После создания моментального образа нажмите кнопку Close (Закрыть).

Как только моментальный образ будет создан, на левой панели появится файл. Чтобы открыть представление типов в куче, нужно дважды щелкнуть на его названии. Этим представлением можно манипулировать точно так же, как и всеми другими стековыми представлениями в PerfView (например, использовать объединение в группы, свертку и фильтрацию). На записи, обозначающей тип, можно сделать двойной щелчок, что приведет к переключению на представление Referred-From (Имеется ссылка из).

В этом представлении четко показано, что объекты типа `D` принадлежат переменной `staticArray` и локальной переменной (они теряют свои имена в ходе компиляции) (рис. 2.21).

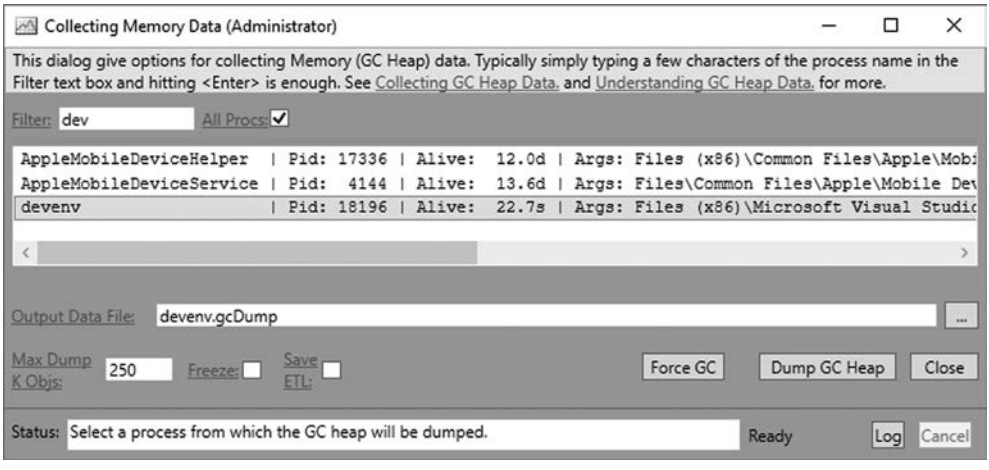

Рис. 2.20. В диалоговом окне PerfView под названием Heap Snapshot отображается управляемая куча для проведения простого анализа

Objects that refer to LargeMemoryUsage!LargeMemoryUsage.D

Name ?	Inc % ?	Inc ?	Inc Ct ?	Exc % ?	Exc ?
☑ LargeMemoryUsage!LargeMemoryUsage.D ?	88.5	462,013,000.0	924	88.5	462,013,000
+ ☐ [static var LargeMemoryUsage.Program.staticArray] ?	44.8	234,006,300.0	468	0.0	0
+ ☑ [local var] ?	43.7	228,006,800.0	456	0.0	0
+ ☑ [.NET Roots] ?	43.7	228,006,800.0	456	0.0	0
+ ☑ ROOT ?	43.7	228,006,800.0	456	0.0	0

Рис. 2.21. В PerfView показана принадлежность объектов кучи, что облегчает анализ причин утечки

Из этого можно составить неплохое общее представление о содержимом кучи. Если взять два дампа, сделанных в разное время, то можно воспользоваться меню Diff (Различия) для вычисления различий между двумя моментальными образами (рис. 2.22). Это может дать представление о накоплении не поддающегося сборке мусора, если таковое имеется.

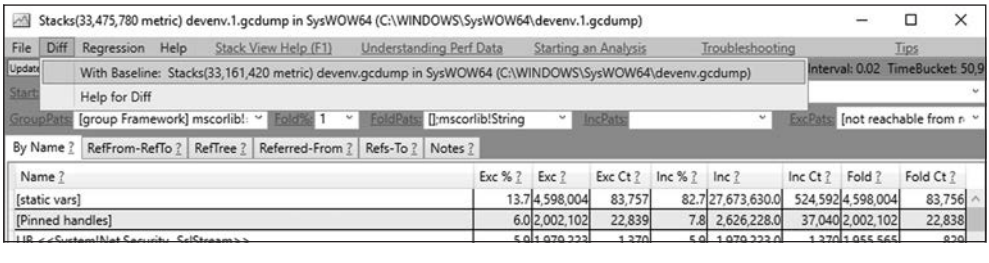

Рис. 2.22. Если открыть два моментальных образа кучи, их можно сравнить, чтобы получить представление, показывающее различия

Наиболее подходящие для обобщенного анализа средства — Visual Studio и PerfView. Средство PerfView — это профилировщик на основе выборок, выполняемых даже при анализе кучи, поэтому иногда он выдает искаженное изображение того, как именно выглядит куча. Если необходимо углубиться в конкретный объект или получить истинную полную картину, следует начать использовать отладчик или средства библиотеки CLR MD.

Для получения в WinDbg быстрой сводки, дающей представление о содержимом кучи, нужно запустить команду `!DumpHeap -stat` (рис. 2.23):

```
0:023> !DumpHeap -stat
...
71f718f8     8752      525120 System.Reflection.RuntimeMethodInfo
139e5424    15138      544968 System.Collections.Immutable.Sort...
71f7ffe4    11294      573796 System.Object[]
1370f7d0     4605      626280 Microsoft.VisualStudio.Compositio...
13707114     6190      990400 Microsoft.VisualStudio.Compositio...
1370f24c     5482     1227968 Microsoft.VisualStudio.Compositio...
71f8419c     4799     4684529 System.Byte[]
71f7fbf0   108732     8303452 System.String
00586810    30707    72014878    Free
```

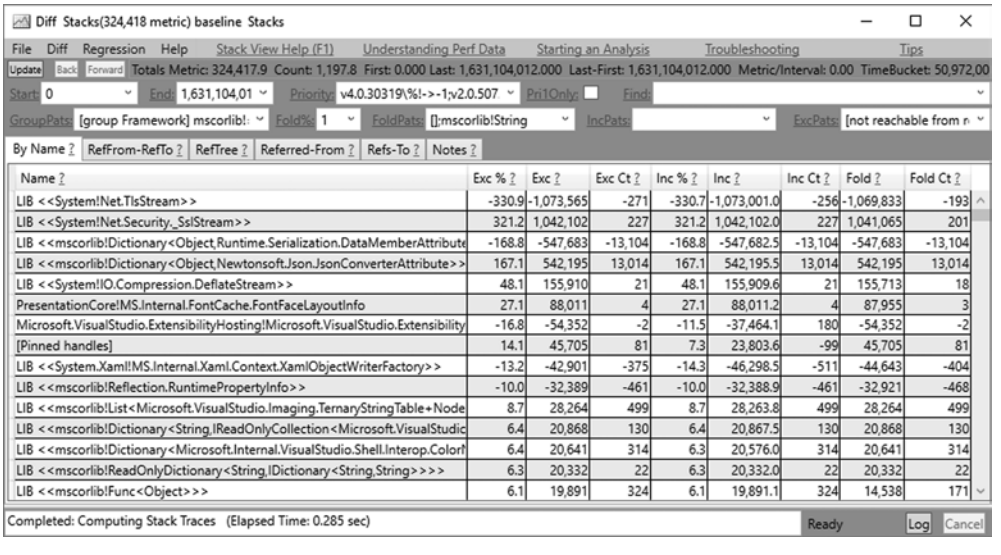

Рис. 2.23. Представление Heap Diff аналогично всем стековым представлениям в PerfView, но числа будут показывать отличия целевого моментального образа от базового

Команда выдаст немало выходной информации. Обычно я прокручиваю ее до конца сводки об объектах, чтобы посмотреть на самых последних потребителей пространства кучи. Заметьте, что после сводки об объектах выводится список объектов, появляющихся после свободных блоков, и вам нужно прокрутить выходную информацию чуть выше этой позиции.

Если проделать это пару раз между периодами свободного исполнения программы, в ходе которых, предположительно, произойдет утечка, можно получить представление о том, какие объекты поглощают память. Если вы увидите увеличение размера `Free`, значит, либо не происходит сборка мусора, либо куча фрагментирована. Порядок диагностирования фрагментации будет рассмотрен чуть позже.

Недостаток WinDbg заключается в сложности получения общей картины принадлежности объектов, особенно для таких весьма часто используемых объектов, как `System.Byte[]` или `System.String`. Для этого, как сказано ранее, лучше воспользоваться средством PerfView.

Если нужно проанализировать отдельно взятый объект, сначала следует получить его адрес. Для получения адресов объектов воспользуйтесь командой `!DumpStackObjects` или задействуйте `!DumpHeap` для поиска в куче интересующих вас объектов, как в следующем примере:

```
0:004> !DumpHeap -type LargeMemoryUsage.C
  Address    MT    Size
021b17f0 007d3954     12
021b664c 007d3954     12
...

Statistics:
    MT   Count  TotalSize Class Name
007d3954   475       5700 LargeMemoryUsage.C
Total 475 objects
```

Получив адреса объектов, можно воспользоваться командой `!gcroot`:

```
0:003> !gcroot 02ed1fc0
HandleTable:
  012113ec (pinned handle)
  -> 03ed33a8 System.Object[]
  -> 02ed1fc0 System.Random

Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

Команда `!gcroot` зачастую ведет себя прилично, но может пропустить некоторые случаи, в частности, если ваш объект уходит корнями в более старое поколение. Для этого нужно будет воспользоваться командой `!findroots`.

Чтобы эта команда сработала, сначала нужно установить точку останова в сборщике мусора непосредственно перед намечающейся сборкой, задав выполнение следующих команд:

```
!findroots -gen 0
g
```

Эти команды устанавливают точку останова непосредственно перед моментом выполнения следующей сборки мусора в поколении gen 0. Затем эффект команд утрачивается, и придется запускать команду еще раз для прерывания выполнения на следующей сборке мусора.

Как только выполнение кода будет прервано, следует найти интересующий вас объект и выполнить с его адресом следующую команду:

```
!findroots 027624fc
```

Если объект уже находится в более старшем поколении, чем поколение текущей сборки мусора, будет выведена примерно следующая информация о том, что данный объект выживет при сборке мусора, поскольку он находится в поколении, которое старше оцениваемого поколения:

```
Object 027624fc will survive this collection:
  gen(0x27624fc) = 1 > 0 = condemned generation.
```

Если сам объект находится в поколении, попадающем в текущий момент под сборку мусора, но имеет корни в старшем поколении, будет выведена примерно следующая информация:

```
older generations::Root: 027624fc (object)->
  023124d4(System.Collections.Generic.List'1
  [[System.Object, mscorlib]])
```

Если это слишком утомительно, можно создать собственную команду `!gcroot` с помощью средств библиотеки CLR MD.

```
const string TargetType = "LargeMemoryUsage.D";

private static void PrintRootsOfObjects(ClrRuntime clr)
{
    PrintHeader("Roots of Object");

    Dictionary <ulong , ClrObject > childToParents =
      new Dictionary <ulong , ClrObject >();
    var heap = clr.Heap;

    // Поиск произвольного объекта в демонстрационных целях
    ClrObject targetObject = FindObjectOfType(clr, TargetType);

    if (targetObject.Address == 0)
    {
        Console.WriteLine(
            $"Could not find any objects of type {TargetType}");
        return;
    }

    // Анализ всех объектов, построение карты ссылок
    foreach (var obj in heap.EnumerateObjects())
    {
        foreach (var objRef in obj.EnumerateObjectReferences())
        {
            childToParents[objRef.Address] = obj;
        }
    }
```

```
    // Подъем по цепочке ссылок
    ClrObject currentObj = targetObject;
    int indentSize = 0;
    while(true)
    {
        Console.Write(new string(' ', indentSize));
        Console.WriteLine(
          $"0x{currentObj.Address:x} - {currentObj.Type.Name}");

        ClrObject parentObject;
        if (!childToParents.TryGetValue(currentObj.Address ,
                                        out parentObject))
        {
            break;
        }
        currentObj = parentObject;
        indentSize += 4;
    }
}

private static ClrObject FindObjectOfType(ClrRuntime clr,
                                          string typeName)
{
    foreach (var obj in clr.Heap.EnumerateObjects())
    {
        if (obj.Type.Name == TargetType)
        {
            return obj;
        }
    }
    return new ClrObject();
}
```

Выполнение кода приведет к выводу примерно такой информации:

```
Roots of Object
===============
0x2e46bfc - LargeMemoryUsage.D
    0x2e43428 - System.Object[]
```

Каков размер моих объектов

Вычисление размера объекта — вопрос неоднозначный. Подразумевается ли при этом размер всех полей в этом объекте? Если в нем имеется ссылка на другой объект, такой как массив, его тоже нужно брать в расчет? Что если два объекта ссылаются друг на друга?

К счастью, большинство инструментальных средств, показывающих размер объекта, придерживаются алгоритма с использованием следующих понятий.

1. Исключающим считается размер объекта и всех его полей. Объекты, на которые имеются ссылки, не включаются, но сами ссылки на такие объекты, четырех- или восьмибайтные, берутся в расчет.

2. Включающим является размер объекта и всех объектов, на которые имеется ссылка в данном объекте.
3. Ссылки на объекты учитываются, пока они не закончатся или пока не встретится уже пройденная ссылка. Тем самым исключается двойной подсчет.

Для получения размеров объектов в Visual Studio воспользуйтесь профилировщиком Memory Usage (Использование памяти) (рис. 2.24).

1. Пройдите по пунктам меню Analyze ▸ Performance profiler (Анализ ▸ Профилировщик производительности) или нажмите сочетание клавиш Alt+F2.
2. Выберите пункт Memory Usage (Использование памяти).
3. Запустите на выполнение целевую программу.
4. В нужный момент получите мгновенный образ кучи.
5. Завершите профилирование или выполнение целевой программы.

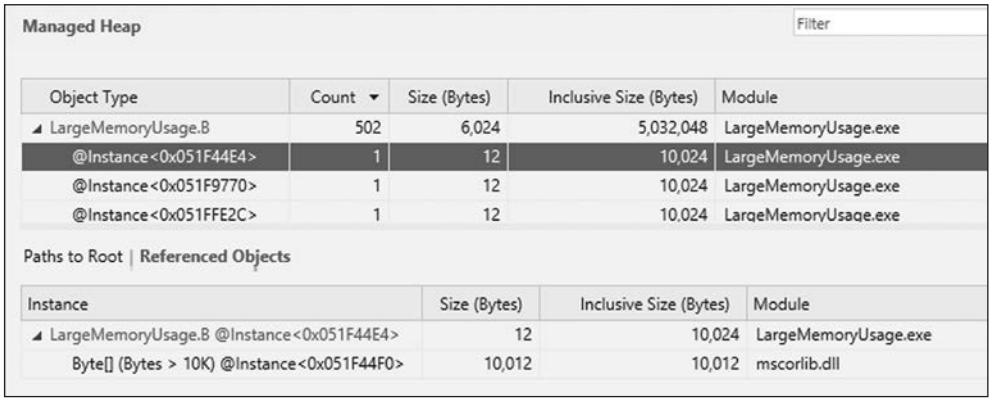

Рис. 2.24. Имеющийся в Visual Studio профилировщик Memory Usage может показать совокупные или индивидуальные размеры объектов, в том числе тех, на которые есть ссылки

Если нужная детализация не достигнута, проверьте, отключены ли настройки табличного представления Collapse Small Objects (Свернуть малые объекты) и Just My Code (Только мой код).

В WinDbg есть две SOS-команды, способные показать такую же информацию.

Команда `!DumpObj` может показать исключающий размер объекта:

```
0:007> !DumpObj /d 058e8230
Name:        LargeMemoryUsage.D
MethodTable: 035d4e74
EEClass:     035d1870
Size:        12(0xc) bytes
File:        D:\HighPerformanceDotNetBook\...\LargeMemoryUsage.exe
Fields:
      MT   Field Offset          Type VT     Attr    Value Name
71b54080 4000003      4 System.Byte[]  0 instance 2a895510 memory
```

Как видите, она не принимает в расчет принадлежащий объекту массив байтов. Чтобы учесть его, нужно воспользоваться командой `!ObjSize`:

```
0:007> !ObjSize 058e8230
sizeof(058e8230) = 1000028 (0xf425c) bytes (LargeMemoryUsage.D)
```

Если запустить команду `!ObjSize` без параметров, она выведет список всех потоков и GC-дескрипторов, собрав воедино размеры объектов, уходящих корнями в каждый из них:

```
0:007> !ObjSize
...
Thread 5580 (LargeMemoryUsage.Program.Main(System.String[])
  [D:\HighPerformanceDotNetBook\...\Program.cs @ 29]):
  ebp+1c: 012ff37c -> <exec cmd="!DumpObj /d 05383448">
  05383448</exec>: 283846000 (0x10eb2570) bytes (System.Object[])
...
Handle (pinned): 035b13ec -> 06383510: 286744176 (0x11175e70) bytes
  (System.Object[])
Handle (pinned): 035b13f0 -> 06382500: 8864 (0x22a0) bytes
  (System.Object[])
Handle (pinned): 035b13f4 -> 063822e0: 640 (0x280) bytes
  (System.Object[])
Handle (pinned): 035b13f8 -> 0538121c: 12 (0xc) bytes
  (System.Object)
Handle (pinned): 035b13fc -> 06381020: 8440 (0x20f8) bytes
  (System.Object[])
```

Можно подсчитать этот размер и с помощью средств библиотеки CLR MD. Для этого придется самостоятельно проделать работу по обходу объектов:

```
private static void PrintObjectSize(ClrRuntime clr)
{
    PrintHeader("Object Size");

    var obj = FindObjectOfType(clr, TargetType);
    Console.WriteLine($"0x{obj.Address:x} - {obj.Type.Name}");
    var heap = clr.Heap;
    // Стек для подсчетов
    Stack <ulong > stack = new Stack <ulong >();

    HashSet <ulong > considered = new HashSet <ulong >();

    int count = 0;
    ulong size = 0;
    stack.Push(obj.Address);

    while (stack.Count > 0)
    {
        var objAddr = stack.Pop();
        if (considered.Contains(objAddr))
```

```
            continue;

        considered.Add(objAddr);

        ClrType type = heap.GetObjectType(objAddr);
        if (type == null)
        {
            continue;
        }

        count++;
        size += type.GetSize(objAddr);

        type.EnumerateRefsOfObject(objAddr ,
                                   delegate (ulong child ,
                                             int offset)
        {
            if (child != 0 && !considered.Contains(child))
                stack.Push(child);
        });
    }
    Console.WriteLine($"Object Size: {obj.Size}");
    Console.WriteLine($"Full size: {size}");
}
```

Выводимая информация будет примерно такой:

```
Object Size
===========
0x4636c24 - LargeMemoryUsage.D
Object Size: 12
Full size: 1000024
```

Если интересны только совокупные размеры объектов, эта информация может быть получена с помощью средства PerfView, позволяющего объединять подчиненные объекты несколькими способами для подробнейшего анализа. Такая работа с этим средством была рассмотрена в предыдущем разделе.

Каким объектам выделена память в LOH

Понимать, каким именно объектам выделена память в куче больших объектов, очень важно для обеспечения эффективной работы системы. Ключевое правило, рассмотренное ранее в данной главе, гласит, что все объекты должны либо вычищаться при сборке мусора в поколении gen 0, либо жить вечно.

Большие объекты попадают под очистку только при весьма затратной сборке мусора в поколении gen 2, что изначально нарушает упомянутое ранее правило.

Чтобы определить, какие объекты находятся в LOH, нужно воспользоваться средством PerfView и придерживаться приведенных ранее инструкций для получения трассировки событий сборки мусора. В полученном представлении GC Heap

Alloc Stacks (Стеки выделения памяти в GC-куче) на вкладке By Name (По имени) вы найдете специально созданный PerfView узел LargeObject. В результате двойного щелчка на этом узле будет выполнен переход в представление Callers (Вызывающие), показывающий, какие у LargeObject имеются вызывающие. В нашем примере к ним относятся все массивы Int32. После двойного щелчка на них будет показано место в коде, вызвавшее выделение памяти (рис. 2.25).

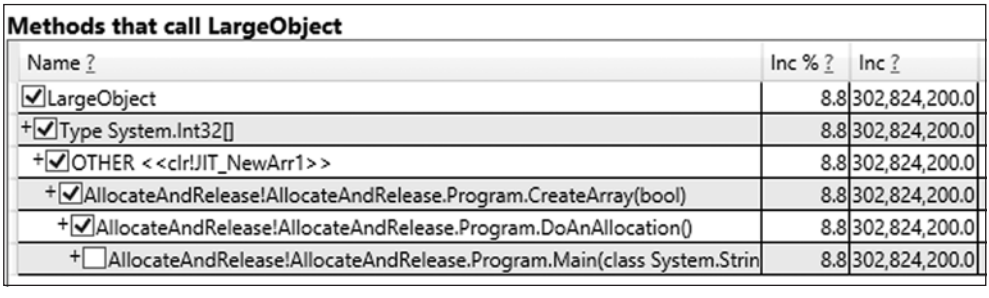

Рис. 2.25. Средство PerfView может показать большие объекты и их типы со стеками, вызвавшими выделение памяти для них

Средства библиотеки CLR MD могут также сообщить о том, какие объекты находятся в куче больших объектов.

```
private static void PrintLOHObjects(ClrRuntime clr)
{
    PrintHeader("LOH Objects (limit:10)");

    int objectCount = 0;
    const int MaxObjectCount = 10;
    if (clr.Heap.CanWalkHeap)
    {
        foreach (var segment in clr.Heap.Segments)
        {
            if (segment.IsLarge)
            {
                for (ulong objAddr = segment.FirstObject;
                     objAddr != 0;
                     objAddr = segment.NextObject(objAddr))
                {
                   var type = clr.Heap.GetObjectType(objAddr);
                   if (type == null)
                   {
                       continue;
                   }
                   var obj = new ClrObject(objAddr , type);
                   if (++objectCount > MaxObjectCount)
                   {
                       break;
                   }
```

```
                Console.WriteLine(
                  $"{obj.Address} {obj.Type.Name}");
            }
        }
    }
}
```

Какие объекты были закреплены

Ранее уже говорилось, что счетчик производительности сообщит о количестве закрепленных объектов, попавшихся сборщику мусора в ходе сборки, но это не поможет вам определить, какие именно объекты были закреплены.

Рассмотрим пример из проекта Pinning, где закрепление производится путем явного использования инструкций `fixed` и вызова ряда Windows API.

Для просмотра закрепленных объектов нужно воспользоваться WinDbg с командой `!gchandles`:

```
0:010> !gchandles
  Handle Type      Object    Size    Data  Type
...
003511f8 Strong    01fa5dbc    52          System.Threading.Thread
003511fc Strong    01fa1330   112          System.AppDomain
003513ec Pinned    02fa33a8  8176          System.Object[]
003513f0 Pinned    02fa2398  4096          System.Object[]
003513f4 Pinned    02fa2178   528          System.Object[]
003513f8 Pinned    01fa121c    12          System.Object
003513fc Pinned    02fa1020  4420          System.Object[]
003514fc AsyncPinned 01fa3d04 64
  System.Threading.OverlappedData
```

Присутствие множества закрепленных объектов `System.Object[]` — обычная ситуация. Эти массивы используются внутри среды CLR для статистики и других закрепленных объектов. В показанном ранее случае можно увидеть один дескриптор `AsyncPinned`. В данном примере этот объект относится к `FileSystemWatcher`.

К сожалению, отладчик не укажет причину закрепления, но зачастую у вас будет возможность изучить закрепленный объект и проследить за тем, какой объект за него отвечает.

В следующем сеансе работы с WinDbg показано отслеживание таких ссылок на объект с целью поиска объектов более высокого уровня, который может дать ключ к истокам закрепленного объекта. Посмотрите, получится ли у вас пройти по цепочке ссылок на объекты, начиная с адреса показанного ранее дескриптора `AsyncPinned`.

```
0:010> !do 01fa3d04
Name: System.Threading.OverlappedData
MethodTable: 64535470
EEClass: 646445e0
Size: 64(0x40) bytes
File: C:\windows\Microsoft.Net\...\mscorlib.dll
Fields:
```

```
      MT    Field Offset Type                VT Attr     Value    Name
64927254 4000700      4 System.IAsyncResult  0 instance 020a7a60 m_asyncResult
64924904 4000701      8 ...ompletionCallback 0 instance 020a7a70 m_iocb
...
0:010> !do 020a7a70
Name: System.Threading.IOCompletionCallback
MethodTable: 64924904
EEClass: 6463d320
Size: 32(0x20) bytes
File: C:\windows\Microsoft.Net\...\mscorlib.dll
Fields:
      MT    Field Offset Type         VT Attr     Value    Name
649326a4 400002d      4 System.Object 0 instance 01fa2bcc _target
...
0:010> !do 01fa2bcc
Name: System.IO.FileSystemWatcher
MethodTable: 6a6b86c8
EEClass: 6a49c340
Size: 92(0x5c) bytes
File: C:\windows\Microsoft.Net\...\System.dll
Fields:
      MT    Field Offset Type                  VT Attr     Value    Name
649326a4 400019a      4 System.Object         0  instance 00000000 __identity
6a699b44 40002d2      8 ...ponentModel.ISite 0  instance 00000000 site
...
```

Отладчик предоставляет вам максимальные возможности, но все же он, мягко выражаясь, несколько трудоемок в применении. Вместо него можно воспользоваться средством PerfView, способным упростить многие рутинные задачи. В трассировке PerfView будет показано представление под названием Pinning at GC Time Stacks (Стеки закрепленных на время сборки мусора), где можно увидеть стеки объектов, закрепленных в ходе всех наблюдаемых сборок (рис. 2.26).

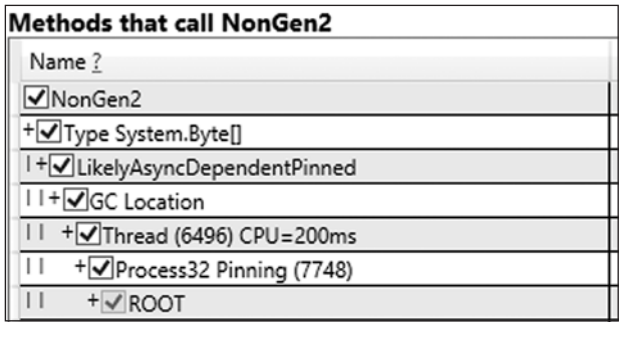

Рис. 2.26. Средство PerfView выводит информацию о типах объектов, закрепленных при сборках мусора, а также их вероятном происхождении

К проблеме изучения закрепленных объектов можно подойти также путем изучения дыр свободного пространства, созданных в различных кучах, о чем пойдет речь в следующем подразделе.

Где происходит фрагментация

Фрагментация возникает при наличии свободных блоков памяти внутри сегментов, содержащих используемые блоки памяти. Она может происходить на нескольких уровнях: внутри сегмента GC-кучи или на уровне виртуальной памяти для всего процесса. Фрагментация становится проблемой при слишком большом количестве небольших свободных блоков памяти, непригодных для последующего выделения.

Фрагментация в поколении gen 0 обычно не вызывает никаких вопросов, если только нет серьезных проблем с закреплением, когда закрепленных объектов слишком много и каждый блок свободной памяти слишком мал для удовлетворения любого нового выделения памяти. Такое положение дел вызовет разрастание кучи малых объектов и увеличение количества сборок мусора.

Обычно фрагментация создает проблему в поколении gen 2 или в куче больших объектов, особенно когда не используется сборка мусора в фоновом режиме. Степень фрагментации может показаться слишком большой, возможно, даже 50 %, но это не обязательно говорит о проблеме. Нужно брать в расчет размер всей кучи, и если он вполне приемлем и со временем не увеличивается, то, вполне вероятно, никаких действий предпринимать не нужно.

В первую очередь требуется узнать, есть ли фрагментация вообще. Если воспользоваться командой `!HeapStat`, средство WinDbg может показать процент свободного пространства кучи, служащий признаком фрагментации:

```
0:023> !HeapStat
Heap     Gen0    Gen1     Gen2     LOH
Heap0 2870384 2423640 93212392 9692760

Free space:                                Percentage
Heap0 177940 21480 65552412 6324464 SOH: 66% LOH: 65%
```

В результате выполнения этой команды выводятся сведения о каждой куче и сообщается процент свободного пространства как в куче малых, так и в куче больших объектов. Что касается фрагментации в куче больших объектов, зачастую вероятных виновников можно найти, посмотрев на то, какие объекты находятся в куче больших объектов, и изучив их размеры и относящийся к ним код. Приемы обнаружения были рассмотрены ранее в этой главе.

Сводку типов и объектов, примыкающих к свободным блокам, можно получить с помощью команды `!DumpHeap -stat`. В самом конце сводных данных о куче будет выведена примерно следующая информация:

```
Fragmented blocks larger than 0.5 MB:
    Addr Size Followed by
16b61000 1.7MB 16d08948 System.Byte[]
16d08d7c 1.7MB 16ec4aa4 System.Byte[]
16f530c4 6.0MB 1755fb10 System.Byte[]
175e978c 0.6MB 17680ae0 System.Byte[]
176b9694 1.8MB 1787fff4 System.Byte[]
1e461000 1.5MB 1e5d7300 System.Byte[]
1e5d7734 1.4MB 1e74660c System.Byte[]
1e746a40 2.4MB 1e9a20d8 System.Byte[]
```

Если нужна подробная информация о фрагментации, включая то, какие именно объекты вызвали появление областей свободного пространства, можно воспользоваться другими командами WinDbg.

Список свободных блоков можно получить с помощью команды `!DumpHeap -type Free`:

```
0:010> !DumpHeap -type Free
 Address    MT    Size
02371000 008209f8   10 Free
0237100c 008209f8   10 Free
02371018 008209f8   10 Free
023a1fe8 008209f8   10 Free
023a3fdc 008209f8   22 Free
023abdb4 008209f8  574 Free
023adfc4 008209f8   46 Free
023bbd38 008209f8  698 Free
023bdfe0 008209f8   18 Free
023d19c0 008209f8 1586 Free
023d3fd8 008209f8   26 Free
023e578c 008209f8 2150 Free
...
```

А с помощью команды `!eeheap -gc` можно выяснить, в каком из сегментов кучи находится каждый блок:

```
0:010> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x02371018
generation 1 starts at 0x0237100c
generation 2 starts at 0x02371000
ephemeral segment allocation context: none
     segment    begin    allocated   size
02370000 02371000 02539ff4 0x1c8ff4(1871860)
Large object heap starts at 0x03371000
     segment  begin      allocated   size
03370000 03371000 03375398 0x4398(17304)
Total Size: Size: 0x1cd38c (1889164) bytes.
------------------------------
GC Heap Size: Size: 0x1cd38c (1889164) bytes.
```

Следующая команда выведет дамп всех объектов, находящихся в данном сегменте или в узком диапазоне вокруг свободного пространства:

```
0:010> !DumpHeap 0x02371000 02539ff4
 Address    MT    Size
02371000 008209f8   10 Free
0237100c 008209f8   10 Free
02371018 008209f8   10 Free
02371024 713622fc   84
02371078 71362450   84
023710cc 71362494   84
```

```
02371120 713624d8   84
02371174 7136251c   84
023711c8 7136251c   84
0237121c 71362554   12
...
```

Это весьма утомительный, выполняемый вручную процесс, но он когда-нибудь может пригодиться, и вы должны понять, как это делается. Для получения выходной информации можно написать сценарий, генерирующий команды WinDbg на основе предыдущего вывода, но профилировщик CLR Profiler может показать вам ту же самую информацию в графическом, сводном виде, которого может быть вполне достаточно для удовлетворения ваших потребностей (рис. 2.27).

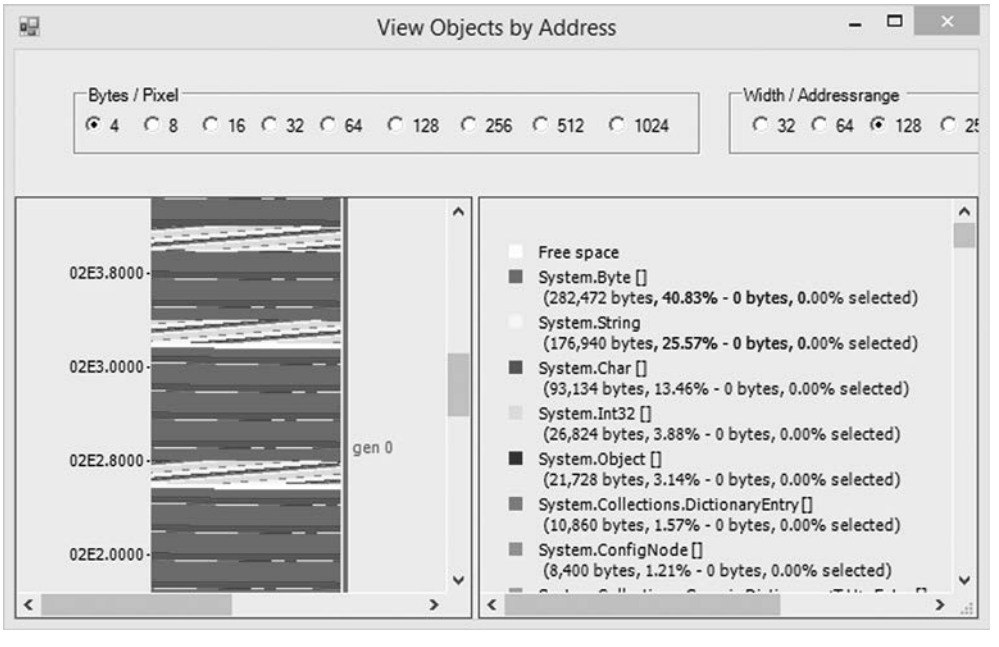

Рис. 2.27. Профилировщик CLR Profiler может показать визуальное представление кучи, позволяющее увидеть, какие типы объектов следуют за свободными блоками памяти. Здесь свободные блоки памяти граничат с блоками SystemByte[] и множеством других типов

Средство PerfView может также в представлении GCStats сообщить, когда произошла фрагментация. Посмотрите на данные столбца Frag % (Процент фрагментации). Разумеется, сообщений о точной причине возникновения фрагментации вы там не найдете.

Библиотека CLR MD дает возможность создавать собственные инструментальные средства, позволяющие получать данные о фрагментации. У каждого объекта `ClrObject` есть свойство `Type`, которое обладает булевым свойством `IsFree`, показывающим, представляет ли этот тип свободное пространство кучи.

Фрагментация виртуальной памяти

Может произойти также фрагментация виртуальной памяти, способная вызвать сбой неуправляемого распределения из-за невозможности найти достаточно большой диапазон, удовлетворяющий запросу. Сюда может быть включено распределение нового сегмента GC-кучи, а это будет означать сбой управляемого выделения памяти.

Чтобы получить визуальное представление вашего процесса, можно воспользоваться средством VMMap, являющимся частью инструментария SysInternals. В нем куча будет поделена на управляемую, неуправляемую и свободную области. При выборе части Free (Свободная) будут показаны свободные сегменты (рис. 2.28). Если максимальный размер не соответствует запрошенному вами размеру памяти, будет выдано исключение `OutOfMemoryException`.

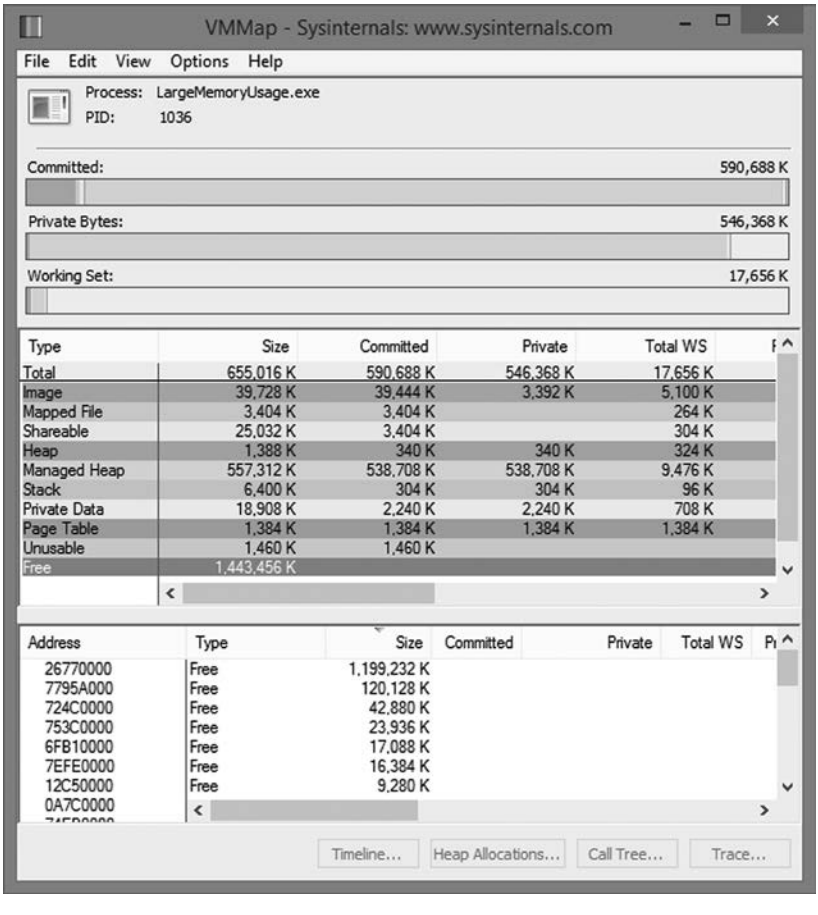

Рис. 2.28. Средство VMMap показывает большой объем информации, относящейся к памяти, включая размер всех свободных блоков в адресном диапазоне. В данном случае размер самого большого блока — более 1,1 Гбайт, что немало!

Средство VMMap имеет также представление о фрагментации, которое в состоянии показать, где именно эти блоки размещаются в общем пространстве процесса (рис. 2.29).

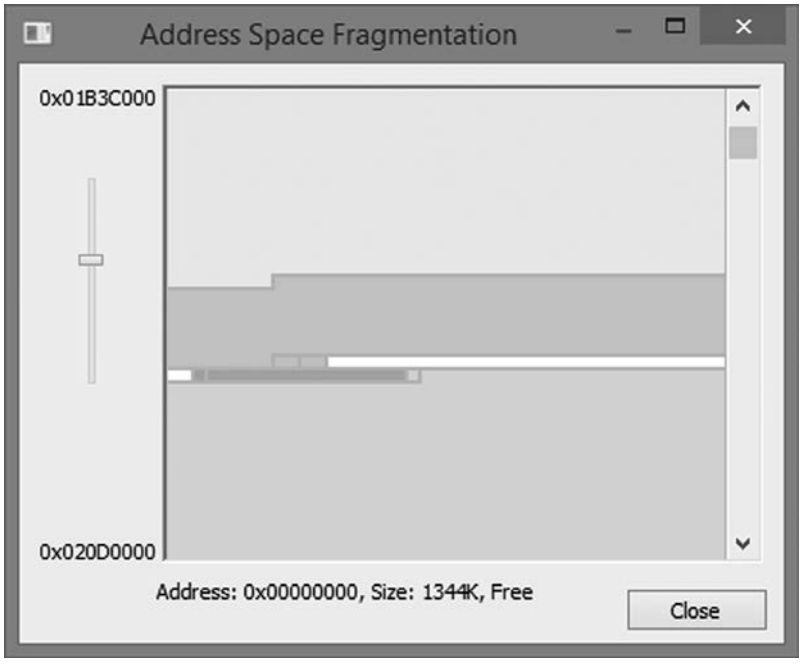

Рис. 2.29. Представление о фрагментации, имеющееся у средства VMMap, где показано свободное пространство в контексте других сегментов

Эту же информацию можно извлечь в WinDbg:

```
!address -summary
```

Эта команда приводит к выводу следующей информации:

```
...
-- Largest Region by Usage -- Base Address -- Region Size --
Free              26770000    49320000 (1.144 Gb)
...
```

С помощью следующей команды можно получить информацию о конкретных блоках:

```
!address -f:Free
```

Ее выполнение даст примерно такой вывод:

```
BaseAddr EndAddr+1 RgnSize Type State Protect Usage
----------------------------------------------------------------
   0    150000    150000   MEM_FREE  PAGE_NOACCESS Free
```

Виртуальной фрагментации сильнее подвержены 32-разрядные процессы, где для вашей программы изначально действует ограничение всего на 2 Гбайт адресного пространства. Самым явным симптомом служит исключение `OutOfMemoryException`. Проще всего исправить ситуацию, преобразовав ваше приложение в 64-разрядный процесс с адресным пространством 129 Тбайт. Если это невозможно, единственным вариантом остается повышение эффективности выделения памяти. Нужно сделать кучи более компактными и, возможно, реализовать крупные пулы.

В каком поколении находится объект

Эту информацию можно извлечь из собственного кода вашего приложения, воспользовавшись методом `GC.GetGeneration` и передав ему интересующий вас объект.

В WinDbg после получения адреса интересующего вас объекта (скажем, из вывода команды `!DumpStackObjects` или `!DumpHeap`) нужно воспользоваться командой `!gcwhere`:

```
0:003> !gcwhere 02ed1fc0
Address  Gen Heap segment  begin   allocated size
02ed1fc0 1   0   02ed0000 02ed1000 02fe5d4c  0x14(20)
```

При применении библиотеки CLR MD можно воспользоваться методом `ClrHeap.GetGeneration`:

```
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
}
```

Какие объекты выжили в поколении gen 0

Проще всего это выяснить, перечислив все объекты, находящиеся в той части кучи, которая относится к gen 1 или gen 2.

С помощью библиотеки CLR MD это можно сделать, использовав минимум кода:

```
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
    if (gen > 0)
    {
        // проведение анализа
    }
}
```

Перебор всех объектов в большой куче был бы крайне неэффективным. Если, к примеру, интерес представляет только куча поколения gen 1, это можно сделать немного рациональнее, пройдя кучу посегментно.

```
private static void PrintGen1ObjectsByHeapSegment(ClrRuntime clr)
{
    PrintHeader("Gen1 Objects by Heap Segment");
    if (clr.Heap.CanWalkHeap)
    {
        foreach(var segment in clr.Heap.Segments)
        {
          // Поколения gen 0 и gen 1 содержатся только в эфемерном сегменте
            if (segment.IsEphemeral)
            {
                // Получение диапазона поколения gen 1
                ulong start = segment.Gen1Start;
                ulong end = start + segment.Gen1Length;
                Console.WriteLine(
                  $"Segment Info: Start: {start}, End {end}");

                for (ulong objAddr = segment.FirstObject;
                     objAddr != 0;
                     objAddr = segment.NextObject(objAddr))
                {
                    if (objAddr >= start && objAddr < end)
                    {
                        var type =
                          clr.Heap.GetObjectType(objAddr);
                        if (type == null)
                        {
                            continue;
                        }
                        var obj = new ClrObject(objAddr , type);
                        Console.WriteLine(
                          $"{obj.Address} {obj.Type.Name}");
                    }
                }

                break;
            }
        }
    }
}
```

С другой стороны, возможно, при отладке вам потребуется узнать, какие объекты пережили конкретную сборку мусора. Например, может быть, вы работаете в среде отладчика, остановили выполнение кода в контрольной точке и хотите узнать, что произойдет после следующей сборки мусора. В WinDbg это сделать можно, но для этого придется немного потрудиться.

Выполните в WinDbg следующие команды:

```
!FindRoots -gen 0
g
```

С их помощью перед началом следующей сборки мусора в поколении gen 0 будет установлена контрольная точка. Как только на ней будет прервано выполнение

кода, можно будет отправить нужные вам команды для получения дампа объектов в куче, например такую:

```
!DumpHeap
```

Будет создан дамп каждого объекта в куче, что может дать чересчур объемный результат. При желании можно добавить параметр `-stat`, чтобы вывести лишь сводную информацию о найденных объектах — их количестве, размерах и типах. Но если нужно ограничить анализ только поколением gen 0, команда `!DumpHeap` позволяет указать диапазон адресов. Вспомните описание сегментов памяти (приведено в начале главы) и то, что поколение gen 0 находится в конце сегмента (рис. 2.30).

Рис. 2.30. Базовая структура сегмента

Для получения перечня куч и сегментов можно воспользоваться командой `eeheap -gc`:

```
0:003> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x02ef0400
generation 1 starts at 0x02ed100c
generation 2 starts at 0x02ed1000
ephemeral segment allocation context: none
     segment  begin  allocated  size
02ed0000 02ed1000 02fe5d4c 0x114d4c(1133900)
Large object heap starts at 0x03ed1000
     segment  begin     allocated  size
03ed0000 03ed1000 041e2898 0x311898(3217560)
Total Size:      Size: 0x4265e4 (4351460) bytes.
------------------------------
GC Heap Size:  Size: 0x4265e4 (4351460) bytes.
```

Эта команда выведет данные о каждом поколении и каждом сегменте. Сегмент, содержащий gen 0 и gen 1, называется эфемерным. Команда `!eeheap` сообщает, где начинается поколение gen 0. Для получения его конечного адреса нужно просто найти сегмент, содержащий адрес начала. Каждый сегмент содержит несколько адресов и длину. В показанном ранее примере эфемерный сегмент начинается с адреса 02ed0000 и заканчивается адресом 02fe5d4c. Следовательно, диапазон поколения gen 0 в этой куче — 02ef0400–02fe5d4c.

После того как вы об этом узнали, команде `!DumpHeap` можно задать ограничения и вывести только объекты, которые относятся к поколению gen 0:

```
!DumpHeap 02ef0400 02fe5d4c
```

После этого нужно сравнить результат своих действий с тем, что получится сразу же по завершении сборки мусора. Сделать это чуть сложнее. Нужно установить контрольную точку на внутренний метод CLR. Этот метод вызывается, когда среда CLR готова возобновить выполнение управляемого кода. Если используется сборка мусора в режиме рабочей станции, следует выполнить вызов:

```
bp clr!WKS::GCHeap::RestartEE
```

А для сборки мусора в режиме сервера выполните следующий вызов:

```
bp clr!SVR::GCHeap::RestartEE
```

После установки контрольных точек нужно продолжить выполнение кода (с помощью клавиши F5 или команды g). Как только сборка мусора завершится, выполнение программы опять прервется и можно будет повторно отправить команды `!eeheap -gc` и `!DumpHeap`.

Теперь у вас будет два набора выходных данных, и их можно сравнить, чтобы увидеть изменения и те объекты, которые остались после сборки мусора. Использовав другие команды и приемы, показанные в этом разделе, можно увидеть, кто удерживает ссылку на конкретный объект.

ПРИМЕЧАНИЕ

Нужно понимать, что при сборке мусора в режиме сервера имеется несколько куч. Для подобного анализа следует повторить команды для каждой из них. Команда `!eeheap` выведет информацию для каждой кучи в процессе.

Откуда был сделан явный вызов метода GC.Collect

То, что код явно вызывает метод `GC.Collect`, называется принудительной сборкой мусора, и есть счетчики и ETW-события, выявляющие соответствующую информацию. Но они ничего не скажут вам о том, откуда именно был вызван метод. Можно без особого труда отследить собственный код в Visual Studio или в любом современном текстовом редакторе, но если ничего не получится, то, чтобы увидеть, как ваша программа добирается до этого метода, придется устанавливать точку останова на самом методе `GC.Collect`.

В WinDbg нужно установить управляемую точку останова на методе `Collect` класса `GC`:

```
!bpmd mscorlib.dll System.GC.Collect
```

Затем продолжить выполнение кода. Как только оно дойдет до точки останова, следует изучить трассировку стека, чтобы увидеть, откуда именно пошел вызов явной сборки мусора:

```
!DumpStack
```

Какие слабые ссылки имеются в моем процессе

Поскольку слабые ссылки — это один из типов GC-дескриптора, для их поиска можно в WinDbg воспользоваться командой `!gchandles`:

```
0:003> !gchandles
  Handle Type      Object   Size Data Type
006b12f4 WeakShort 022a3c8c 100  System.Diagnostics.Tracing...
006b12fc WeakShort 022a3afc 52   System.Threading.Thread
006b10f8 WeakLong  022a3ddc 32   Microsoft.Win32.UnsafeNati...
006b11d0 Strong    022a3460 48   System.Object[]
...

Handles:
  Strong Handles:    11
  Pinned Handles:     5
  Weak Long Handles:  1
  Weak Short Handles: 2
```

Короткие слабые дескрипторы Weak Short являются обычными слабыми ссылками, которыми можно воспользоваться. В длинных слабых дескрипторах Weak Long отслеживается, был ли финализируемый объект восстановлен (для объектов без финализаторов всегда используются короткие дескрипторы). Воскрешение может произойти, когда объект был финализирован и вместо того, чтобы позволить сборщику мусора его вычистить, вы решили его повторно задействовать, прямо в финализаторе присвоив этот объект новой ссылке. Это может быть связано со сценариями объединения в пулы. Конечно, объединение в пулы возможно и без финализации, и, учитывая все сложности работы с воскрешением, его нужно избегать, отдавая предпочтение использованию предсказуемых методов.

Какие финализируемые объекты имеются в куче

WinDbg-команда `!FinalizeQueue` покажет все объекты, зарегистрированные для финализации, а также сводку их типов:

```
0:042> !FinalizeQueue
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
---------------------------------
generation 0 has 13 finalizable objects (288603b4->288603e8)
generation 1 has 6 finalizable objects (2886039c->288603b4)
generation 2 has 57247 finalizable objects (28828520->2886039c)
Ready for finalization 0 objects (288603e8->288603e8)
Statistics for all finalizable objects
  (including all objects ready for finalization):
```

```
      MT Count TotalSize Class Name
72753184     1        12 System.WeakReference'1...
6df6bea8     1        12 System.Windows.Forms.VisualStyles...
6df68c44     1        12 System.Windows.Forms.ImageList...
584582f0     1        12 System.WeakReference'1...
58443158     1        12 Microsoft.Build.BackEnd.Components...
...
```

Если нужно увидеть сводку объектов, готовых к финализации, можно запустить на выполнение следующую команду:

```
!FinalizeQueue -detail
```

Она покажет список имен типов, которые в данный момент доступны для финализации. Если нужно получить конкретные объекты, относящиеся к этой категории, можно воспользоваться диапазоном адресов, полученным в выходных данных для вывода дампа всех объектов в диапазоне готовых к финализации:

```
!DumpHeap 288603e8 288606c4
```

При использовании библиотеки CLR MD можно для перечисления всех финализируемых объектов воспользоваться методом `EnumerateFinalizableObjectAddresses`:

```
private static void PrintFinalizableObjects(ClrRuntime clr)
{
    foreach (var objAddr in
                clr.Heap.EnumerateFinalizableObjectAddresses())
    {
        ClrType type = clr.Heap.GetObjectType(objAddr);
        if (type == null)
        {
            continue;
        }
        ClrObject obj = new ClrObject(objAddr , type);
        // Выполнение с этим объектом каких-либо действий ...
    }
}
```

К сожалению, так вы не сможете получить сведения о готовности этих объектов к финализации.

Резюме

Чтобы действительно оптимизировать свои приложения, нужно как можно тщательнее разобраться со сборкой мусора. Следует выбрать правильные настройки конфигурации для приложения, например сборку мусора в режиме сервера, если приложение является единственным запущенным на машине, но проявить осторожность при использовании расширенных настроек. Необходимо обеспечить

кратковременность существования объектов, низкую интенсивность выделения памяти и объединение в пулы объектов, которые должны существовать дольше средней частоты сборки мусора, или же в противном случае их вечную жизнь в поколении gen 2. Чтобы избежать распределений в кучах, можно применить метод `stackalloc`, но делать это следует осмотрительно.

По возможности откажитесь от закреплений и финализаторов. Чтобы можно было избежать полной сборки мусора, все объекты из LOH должны объединяться в пулы и сохраняться вечно. Сокращайте фрагментацию LOH, сохраняя объекты одинакового размера и по мере необходимости уплотняя кучу. Рассмотрите возможность использования уведомлений о сборке мусора, чтобы запретить полную сборку мусора, которая может повлиять на работу приложения, если начнется в неподходящий момент.

Сборщик мусора — предсказуемый компонент, его операции можно контролировать, более тщательно управляя интенсивностью выделения памяти под объекты и их сроками существования. Согласившись с применением сборщика мусора среды .NET, вы не отказываетесь от контроля, но он требует деликатности.

3 JIT-компиляция

.NET-код распространяется в виде сборок на языке Microsoft Intermediate Language (MSIL, или для краткости просто IL). Этот язык — что-то вроде языка ассемблера, но более простой. Если захотите глубже изучить IL или другие CLR-стандарты, поищите информацию в Интернете, задав в строке поиска `ECMA C# CLI standards`.

При выполнении ваша управляемая программа загружает CLR-среду, которая приступает к выполнению некоего кода-обертки. Весь этот код машинный. При первом вызове из вашей сборки управляемого метода он фактически запускает заглушку, выполняющую код компилятора времени использования (just-in-time (JIT) compiler), который преобразует IL для данного метода в аппаратные инструкции машины. Этот процесс называется компиляцией времени использования (just-in-time compilation, JITting) или JIT-компиляцией. Заглушка заменяется полученным результатом, и при следующем вызове этого же метода инструкции на языке ассемблера вызываются напрямую. Это означает, что при первом вызове любого метода всегда возникает провал производительности. В большинстве случаев он невелик и его можно проигнорировать. После этого каждый раз код выполняется напрямую и не несет никаких издержек (рис. 3.1).

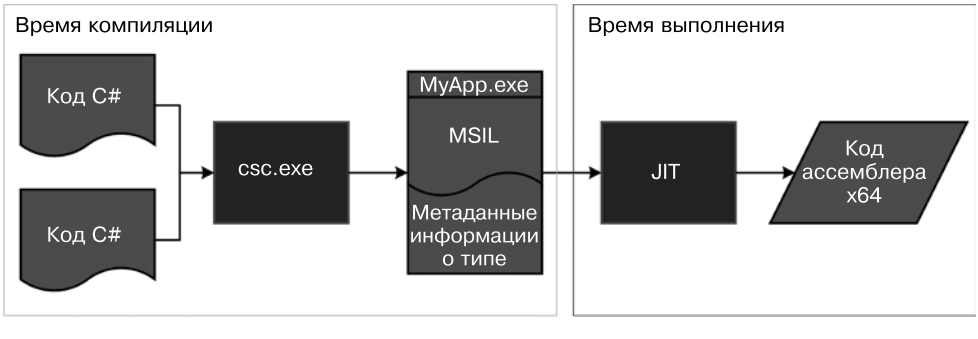

Рис. 3.1. Схема компиляции в IL и JIT-компиляции

Хотя весь код в методе в ходе JIT-компиляции будет преобразован в ассемблерные инструкции, некоторые фрагменты могут быть помещены в «холодные»

секции памяти отдельно от пути обычного выполнения метода. Таким образом, эти редко выполняемые пути не станут вытеснять другой код из «горячих» секций, позволяющих повысить общую производительность, поскольку часто выполняемый код хранится в памяти, тогда как «холодные» страницы могут быть сброшены на диск. Поэтому выполнение редко востребуемых путей, например обработка ошибок и исключений, может обойтись очень дорого.

В большинстве случаев прогон метода через JIT-компиляцию понадобится всего один раз. Исключение — использование в методе аргументов обобщенного (универсального) типа. Если какой-либо из типов-аргументов значим, то новый код должен быть сгенерирован для каждого типа. Если же все типы-аргументы относятся к ссылочным, генерируется только одна копия кода аппаратной платформы, поскольку, несмотря на разные типы, каждая ссылка в машинном коде выглядит как стандартный указатель (4 или 8 байт). Учтите, что это не означает, что сами типы при этом одинаковы, — система типов сохраняет целостность, и, к примеру, `List<string>` по-прежнему отличается от `List<Regex>`. Это просто означает, что реализация машинного кода методов не должна заботиться об этих различиях и поэтому дублирующий код не создается.

Затраты на JIT-компиляцию следует принять в расчет, если затраты на первоначальное создание машинного кода важны для вашего приложения или его пользователей. В большинстве приложений уделяется внимание только производительности в уже установившемся режиме работы, но если нужна экстремально высокая доступность, JIT-компиляция должна стать предметом оптимизации. В таком случае правильным решением может стать применение генератора образов в машинном коде (NGEN). В этой главе речь пойдет об использовании генератора и обоснованности этого действия.

Преимущества JIT-компиляции

Код, подвергнутый компиляции времени использования, имеет ряд существенных преимуществ над откомпилированным неуправляемым кодом.

1. Оптимальная локальность ссылок. Совместно используемый код зачастую будет находиться на одной и той же странице памяти или линии кэширования процессора, исключая весьма затратную обработку ошибок отсутствия страницы и относительно затратные обращения к основной памяти.
2. Потенциально уменьшенное использование памяти. В среде CLR возникают издержки на задействование управляемых DLL-библиотек и их метаданных, но в ней компилируются только фактически применяемые методы.
3. Замещения вызовов между сборками. Коды методов из других DLL-библиотек, включая .NET Framework, могут быть вставлены вместо их вызовов в ваше приложение, что способно существенно сэкономить время выполнения.

Оптимизация под конкретное оборудование также дает преимущество, но на практике оно выражается в виде лишь нескольких фактических оптимизаций под конкретные платформы. Тем не менее возможность ориентироваться с одним и тем же кодом на несколько платформ встречается все чаще, и, по всей вероятности, в будущем мы увидим более агрессивную оптимизацию под конкретные платформы.

Основной объем оптимизации в среде .NET выполняется не в компиляторе языка (преобразование кода C#/VB.NET в код IL), а на лету в JIT-компиляторе.

JIT в действии

Увидеть на практике преобразование кода из IL в ассемблер довольно легко. Рассмотрим программу из примера JitCall, в которой показана закулисно проводимая JIT-адаптация кода под платформу:

```
static void Main(string[] args)
{
  int val = A();
  int val2 = A();
  Console.WriteLine(val + val2);
}

[MethodImpl(MethodImplOptions.NoInlining)]
static int A()
{
  return 42;
}
```

У этого метода есть атрибут `MethodImplOptions.NoInlining`. Он предназначен для того, чтобы принудительно оставить вызов метода даже в оптимизированном коде. Если его не будет, вызов подвергнется полной оптимизации так же, как и сложение, после чего останется только константа, помещенная в аргумент метода `WriteLine`:

```
029D0450   mov         ecx ,54h
029D0455   call        72B2CE9C
029D045A   ret
```

Нам нужно посмотреть на поведение JIT в отношении простого метода, поэтому применение атрибута `MethodImplOptions.NoInlining` — это простой способ сохранить этот метод доступным для анализа.

Чтобы посмотреть, что произойдет, сначала получим дизассемблированный код метода `Main`. Приступить к этому действию не так-то просто. Сначала нужно запустить программу и прервать ее выполнение до выполнения `Main`.

1. Запустите WinDbg.
2. Выполните переход File ▸ Open Executable (Файл ▸ Открыть исполняемый) (Ctrl+E).

3. Перейдите к двоичному коду JitCall. Убедитесь, что выбрана версия двоичного кода Release, в противном случае ассемблерный код будет выглядеть совершенно иначе, чем напечатано здесь.
4. Отладчик немедленно прервет выполнение.
5. Запустите команду `sxe ld clrjit`. Это приведет к остановке выполнения кода в отладчике после загрузки `clrjit.dll`. Нам это нужно, так как после загрузки можно будет установить точку останова на методе `Main` до его выполнения.
6. Запустите команду `g`.
7. Программа станет выполняться, пока не будет загружен файл библиотеки `clrjit.dll`, и появится вывод, похожий на следующий:

```
ModLoad: 6fe50000 6fecd000
  C:\Windows\Microsoft.NET\Framework\v4.0.30319\clrjit.dll
```

Затем перейдите к методу `Main`.

1. Запустите команду `.loadby sos clr`.
2. Запустите команду `!bpmd JitCall Program.Main`. Она установит точку останова в начале функции `Main`.
3. Запустите команду `g`.
4. WinDbg прервет выполнение кода точно внутри метода `Main`. Появится вывод, похожий на следующий:

```
(11b4.10f4): CLR notification exception
  - code e0444143 (first chance)
JITTED JitCall!JitCall.Program.Main(System.String[])
Setting breakpoint: bp 007A0050
  [JitCall.Program.Main(System.String[])]
Breakpoint 0 hit
```

Наконец-то мы оказались в нужном месте. Откройте окно дизассемблера Disassembly (Alt+7). Может быть интересно посмотреть и на содержимое окна регистров Registers (Alt+4). Дизассемблированный код `Main` выглядит следующим образом:

```
push ebp
mov  ebp,esp
push edi
push esi

; Вызов A
call dword ptr ds:[0E537B0h] ds:002b:00e537b0=00e5c015
mov  edi,eax
call dword ptr ds:[0E537B0h]
mov  esi,eax

call mscorlib_ni+0x340258 (712c0258)
mov  ecx,eax
add  edi,esi
mov  edx,edi
```

```
mov  eax,dword ptr [ecx]
mov  eax,dword ptr [eax+38h]

; Вызов Console.WriteLine
call dword ptr [eax+14h]
pop  esi
pop  edi
pop  ebp
ret
```

Имеется два вызова на один и тот же указатель (конкретные значения, которые вы увидите, будут отличаться). Это вызов функции `A`. Установите точки останова на обе строки и запустите пошаговое выполнение кода, добираясь при этом до вызовов. После первого вызова указатель `0E537B0h` будет обновлен.

Пройдя при пошаговом режиме выполнения в первый вызов `A`, можно увидеть, что там немного больше кода, чем содержит обычный переход `jmp` к CLR-методу `ThePreStub`. Здесь нет возвращения из этого метода, поскольку возвращение выполнит `ThePreStub`.

```
mov al,3
jmp 00e5c01d
mov al,6
jmp 00e5c01d
(00e5c01d) movzx eax,al
shl eax,2
add eax,0E5379Ch
jmp clr!ThePreStub (72102af6)
```

Во втором вызове `A` будет видно, что адрес функции в исходном указателе был обновлен и код в новом месте уже больше похож на настоящий метод. Обратите внимание на то, что `2Ah` (наше постоянное десятичное значение `42` из исходного кода) было присвоено и возвращено через регистр `eax`:

```
012e0090 55 push ebp
012e0091 8bec mov ebp,esp
012e0093 b82a000000 mov eax,2Ah
012e0098 5d pop ebp
012e0099 c3 ret
```

Для большинства приложений эти затраты на предварительную подготовку не столь существенны, но есть определенные типы кода, рассматриваемые в нескольких следующих разделах, на которые требуется много времени при JIT-компиляции.

JIT-оптимизации

JIT-компилятор будет выполнять ряд стандартных оптимизаций, например встраивание методов и исключение проверки диапазона массивов, но теперь вам следует узнать, что есть нечто способное помешать ему оптимизировать ваш код. Некоторым из этих методов посвящены части главы 5. Следует заметить: JIT-компилятор

работает в ходе выполнения приложения, поэтому время, затрачиваемое им на оптимизацию, ограничено. Несмотря на это, он способен выполнить множество важных улучшений.

Один из самых больших классов оптимизации — встраивание методов, при котором код из тела метода помещается в то место, откуда он вызывается, исключая в первую очередь вызов метода. Встраивание очень важно для небольших часто вызываемых методов, когда издержки на вызов функции превышают издержки на выполнение самого кода функции.

Встраиванию препятствует все перечисленное далее:

- виртуальные методы;
- интерфейсы с различными реализациями в одном месте вызова (диспетчеризация интерфейсов будет рассматриваться в главе 5);
- циклы;
- обработка исключений;
- рекурсия;
- тела методов, превышающие 32 байта кода на языке IL. Чтобы узнать размеры методов, можно воспользоваться инструментами анализа IL, рассмотренными в главе 1.

С момента выхода .Net 4.6 была выпущена новая версия JIT-компилятора. Ранее известная как RyuJIT, она показала существенно повышенную производительность генерации кода, а также повышенное качество сгенерированного кода, особенно 64-разрядного.

Остерегайтесь вызовов свойств или методов внутри циклов. В большинстве случаев оптимизировать такие вызовы JIT-компилятор не способен. Нужно позаботиться о минимально возможной затратности выполнения кода тел циклов и провести эту оптимизацию самостоятельно. Если метод или свойство могут быть вызваны за пределами цикла, то так обычно и нужно делать, сохраняя результат в локальной переменной.

Сокращение времени JIT-компиляции и запуска

Другой важный фактор — количество времени, затрачиваемое JIT-компилятором на генерацию кода. Здесь все сводится в основном к объему кода, который нужно обработать JIT-компилятором.

Например, большие методы будут проходить JIT-компиляцию дольше, чем небольшие. Если есть один большой метод с большим количеством ветвлений, он потребует полноценных затрат по JIT-компиляции, даже если основная часть его

кода никогда не будет выполнена. Разбиение такого метода на несколько более мелких методов может снизить предварительные расходы на JIT-компиляцию.

На необходимость генерации большого объема кода могут повлиять также свойства языка и API. В частности, обратите внимание на следующие ситуации:

- интегрированные в язык запросы — LINQ;
- ключевое слово `dynamic`;
- `async` и `await`;
- регулярные выражения;
- генерация кода;
- многие типы сериализаторов.

Для всех них характерно то, что от вас, возможно, скрыто и фактически выполняется гораздо больше кода, чем кажется при взгляде на исходный код. Для JIT-компиляции скрытого кода может потребоваться значительное время. В частности, что касается регулярных выражений и сгенерированного кода, весьма вероятно возникновение типовой ситуации, при которой появятся большие повторяющиеся блоки кода.

Чаще всего вы будете явно использовать генерацию кода для достижения своих целей, но есть некоторые области платформы .NET Framework, где это происходит без вашего вмешательства, — в первую очередь регулярные выражения и XML-сериализация. Перед выполнением регулярные выражения опционально могут быть преобразованы в конечный автомат на IL в динамической сборке, а затем пройти JIT-компиляцию. Сначала это займет много времени, но затем позволит сэкономить его при повторных выполнениях (при условии, что вы сделали регулярное выражение статическим). Обычно этой опцией стоит воспользоваться, возможно отложив до того момента, когда в этом появится реальная необходимость, чтобы дополнительная компиляция не повлияла на время запуска приложения. Регулярные выражения могут также запустить в JIT-компиляции сложные алгоритмы, занимающие больше времени, чем обычно (многие из них были улучшены в версии JIT-компилятора, которая была поставлена в .NET 4.6). Здесь, как и во всех других случаях, рассмотренных в этой книге, единственный способ достоверно узнать что-то — измерение. Более подробно регулярные выражения рассматриваются в главе 6.

Несмотря на то что генерация кода имеет прямое отношение к потенциальному обострению неприятностей, связанных с JIT-компиляцией, в главе 5 будет показано, что генерация кода в другом контексте способна избавить вас от некоторых других проблем низкой производительности.

Синтаксическая простота технологии LINQ может дать неверное представление о количестве кода, фактически запускаемого для каждого запроса. За ней могут скрываться также создание делегатов, выделение памяти и многое другое. Простые

LINQ-запросы могут не создать особых проблем, но, как и в большинстве случаев, все требует конкретных измерений.

Основная проблема, связанная с динамическим кодом, опять-таки заключается в том объеме кода, в который он преобразуется. Чтобы увидеть, на что похож динамический код, если сбросить все покровы, перейдите к главе 5.

Помимо JIT, есть и другие факторы, способные повысить затраты на «первичный разогрев», например ввод-вывод. Поэтому, прежде чем предположить, что JIT-компиляция — это единственная проблема, нужно провести тщательные исследования. Каждая сборка влечет за собой затраты, связанные с обращениями к диску для чтения файла, внутренними издержками в структурах данных среды CLR и загрузкой типов.

Вы можете сократить затраты на ввод-вывод, объединяя множество мелких сборок в одну большую, но на загрузку типов, наверное, потребуется столько же времени, сколько и на JIT-компиляцию.

При больших объемах JIT-компиляции нужно посмотреть на стеки, содержащие вызовы метода `PreStubWorker` и других методов, которые в итоге оказываются внутри библиотеки `clrjit.dll` (рис. 3.2).

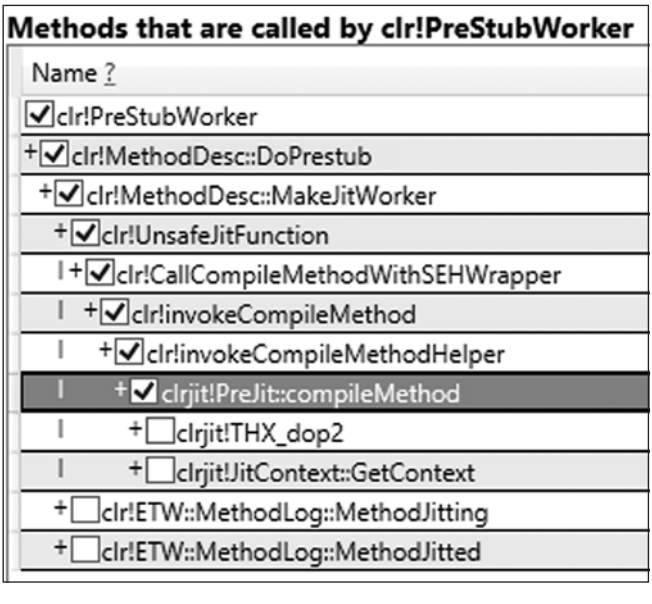

Рис. 3.2. Выполняемое в PerfView профилирование центрального процессора покажет любые вызванные JIT-заглушки

Чуть позже в этой главе будет продемонстрировано, как PerfView может показать, какие именно методы попали под JIT-компиляцию и сколько времени она заняла для каждого из них.

Оптимизация JIT-компиляции с помощью профилирования (Multicore JIT)

В .NET 4.5 был добавлен API, предписывающий среде .NET создать профиль запуска вашего приложения и сохранить результаты на диске для применения в будущем. При последующих запусках этот профиль используется для начала генерации кода сборки еще до того, как он будет выполнен. Это происходит в специально выделенном потоке независимо от потоков выполнения вашего собственного кода, поэтому данная функция называется многоядерной JIT-компиляцией (Multicore JIT). Сохраненные профили позволяют сгенерированному коду получать те же самые преимущества локальности, что и при JIT-компиляции. Профили обновляются автоматически при каждом выполнении вашей программы.

Для использования этой опции нужно просто вызвать в начале программы следующий код:

```
ProfileOptimization.SetProfileRoot(@"C:\MyAppProfile");
ProfileOptimization.StartProfile("default");
```

Заметьте, что корневая папка профиля уже должна существовать и вы должны дать своим профилям имя. Это пригодится, если у вашего приложения имеются разные режимы с совершенно разными профилями выполнения.

При использовании этой функции совместно с собственным профилированием производительности при запуске имейте в виду следующее: примененная оптимизация внесет изменения в результаты измерений. В зависимости от направления поисков может потребоваться временное отключение многоядерной JIT-компиляции.

Когда следует применять NGEN

Если затраты на запуск или предварительную подготовку приложения к работе слишком высоки и упомянутая ранее оптимизация за счет профилирования не отвечает вашим требованиям к уровню производительности, то вполне резонным может стать применение NGEN.

Аббревиатура NGEN означает Native Image Generator — генератор образов в машинном коде. По сути, его работа заключается в преобразовании вашей сборки на языке IL в образ в машинном коде путем запуска JIT-компилятора и сохранения результатов в кэше образов сборок в машинном коде. Это сокращает время запуска и уменьшает общий объем JIT-компиляции. Этот образ в машинном коде не нужно путать с машинным кодом в понятии неуправляемого кода. Несмотря на то что теперь образ представляет собой по большей части код на языке ассемблера, он по-прежнему остается управляемой сборкой, поскольку должен запускаться под управлением среды CLR.

Если исходная сборка была названа `foo.dll`, NGEN создаст файл `foo.ni.dll` и поместит его в кэш образа в машинном языке. Как только поступит запрос на загрузку `foo.dll`, среда CLR проверит кэш на наличие соответствующего файла с элементом расширения `.ni`, а этот файл — на полное соответствие файлу на языке IL. При этом используется комбинация отметок времени, имен и GUID-идентификаторов, чтобы быть совершенно уверенными в том, что будет загружен правильный файл.

При всей ценности генератора NGEN у него имеется ряд недостатков. Во-первых, теряется локальность ссылок, поскольку весь код в сборке размещается последовательно независимо от того, как он фактически выполняется. Вдобавок могут быть утрачены определенные оптимизационные решения, например кросс-ассемблерные замещения вызовов. Большую часть этих оптимизационных решений можно восстановить, если у NGEN есть одновременный доступ ко всем сборкам. Кроме того, образы в машинном коде должны обновляться при каждом изменении, что не требует больших усилий, но добавляет еще один этап к развертыванию. NGEN может работать очень медленно, а образы в машинном коде могут быть существенно больше своих управляемых двойников. Иногда JIT будет выдавать лучше оптимизированный код, особенно для наиболее часто выполняемых путей. Принимая решение об использовании NGEN, следует помнить основополагающее правило производительности: измерение, измерение, измерение! Советы по измерению затрат на JIT-компиляцию в вашем приложении даны в конце главы.

Большинство фрагментов кода с обобщенными типами может быть успешно обработано NGEN-генератором, но случается, что компилятор не способен заранее определить правильные обобщенные типы. Этот код в ходе выполнения программы все равно попадет под JIT-компиляцию. И конечно, предварительно обработать NGEN-генератором те фрагменты кода, которые полагаются на динамическую загрузку или генерацию типов, просто невозможно.

Чтобы обработать сборку NGEN-генератором в режиме работы с командной строкой, нужно выполнить следующую команду:

```
D:\>ngen install ReflectionExe.exe

1> Compiling assembly D:\...\ReflectionExe.exe (CLR v4.0.30319) ...
2> Compiling assembly ReflectionInterface, Version=1.0.0.0,
     Culture=neutral, PublicKeyToken=null (CLR v4.0.30319) ...
```

Из выходной информации следует, что фактически обработаны два файла. NGEN-генератор будет автоматически обращаться к каталогу целевого файла и выполнять NGEN-обработку всех найденных зависимостей. Он делает это по умолчанию, позволяя коду эффективно совершать вызовы между сборками, например вставлять небольшие методы. Такое поведение можно подавить с помощью флага `/NoDependencies`, но в ходе выполнения программы это может серьезно ударить по производительности.

Чтобы удалить принадлежащий сборке образ в машинном коде из имеющегося в компьютере кэша образов в машинном коде, можно запустить следующую команду:

```
D:\>ngen uninstall ReflectionExe.exe

Uninstalling assembly D:\...\ReflectionExe.exe
```

Проверить создание образа в машинном коде можно путем вывода на экран кэша образов в машинном коде:

```
D:\>ngen display ReflectionExe
NGEN Roots:
D:\Book\ReflectionExe\bin\Release\ReflectionExe.exe
NGEN Roots that depend on "ReflectionExe":
D:\Book\ReflectionExe\bin\Release\ReflectionExe.exe
Native Images:
ReflectionExe, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=null
```

Можно также вывести на экран все образы в машинном коде из кэша, запустив следующую команду:

```
ngen display.
```

Следует заметить, что файл образа в машинном коде всегда больше по размеру, чем версия того же файла на чистом управляемом IL-языке. IL-версия полностью содержится внутри образа в машинном коде, кроме этого, код x86/x64 может быть более пространным, чем код на языке IL. К примеру, размер использованного ранее файла `ReflectionExe.exe` — 5632 байта, а образ в машинном коде в GAC-кэше занимает 11 264 байта. Мне приходилось наблюдать четырехкратное увеличение размера файла в зависимости от объема метаданных и типа кода, присутствующего в управляемых сборках.

Оптимизация NGEN-образов. Ранее уже говорилось, что одной из потерь от использования NGEN является локальность ссылок. С появлением .NET 4.5 появилась возможность в значительной степени устранить эту проблему, задействуя инструментальное средство Managed Profile Guided Optimization (MPGO). Как и при оптимизации JIT с помощью профилирования, это средство запускается вручную для профилирования пусковой стадии вашего приложения или нужного сценария. Затем профиль будет использоваться NGEN-генератором для создания образа в машинном коде с лучшей оптимизацией для наиболее часто применяемых цепочек функций.

Средство MPGO включено в среду Visual Studio 2012 и выше. Чтобы им воспользоваться, нужно запустить следующую команду:

```
Mpgo.exe -scenario MyApp.exe -assemblyList *.* -OutDir c:\Optimized
```

Она заставит MPGO работать с некоторыми сборками среды, после чего будет выполнен код файла `MyApp.exe`. Теперь приложение находится в режиме обучения.

Нужно поработать с приложением в приближенном к реальному режиме, а затем закрыть его. В результате в каталоге `C:\Optimized` должна быть создана оптимизированная сборка.

Чтобы получить преимущества от использования оптимизированной сборки, для нее нужно запустить NGEN:

```
Ngen.exe install C:\Optimized\MyApp.exe
```

В результате в кэше образов в машинном коде будут созданы оптимизированные образы. Они будут применены при следующем запуске приложения.

Для эффективного использования средства MPGO вам потребуется включить его в свою систему сборки, чтобы получаемое на его выходе было включено в поставку вашего приложения.

.NET Native

При создании приложений универсальной платформы Windows (Universal Windows Platform) можно воспользоваться средством .NET Native, представляющим собой компилятор, преобразующий ваше скомпилированное управляемое приложение в машинный код, подобно NGEN, но со следующими преимуществами.

- Использование более нового компилятора, основанного на собственном компиляторе Visual C++.
- Получение автономных приложений. После компиляции устраняется зависимость от среды CLR. Эта среда сокращается до простой DLL-библиотеки, поставляемой с вашим приложением. Любой фактически применяемый код среды статически скомпонован с внутренним кодом исполняемого файла.

Компилятор создает CLR в DLL, запуская в вашем коде механизм сокращения зависимостей. Неформально этот процесс называется тряской дерева. Им анализируются ваш код, конфигурация, XAML-файлы, типовые аргументы и многое другое, чтобы определить все, что может быть запущено. В результате получаются быстродействующие компактные приложения, запускаемые с незначительной задержкой.

Тем не менее у этой технологии имеется ряд недостатков, обусловленных в основном требованием запрета JIT-компиляции:

- отсутствие отражения;
- отсутствие динамической загрузки сборок или вызова кода;
- отсутствие сериализации и десериализации;
- отсутствие COM-взаимодействия (обычный P/Invoke работает);
- в настоящий момент работает только для универсальной платформы Windows.

Чтобы воспользоваться .NET Native, нужно создать новое приложение универсальной платформы Windows в Visual Studio. Сборки конечных версий будут автоматически компоноваться с помощью .NET Native (рис. 3.3).

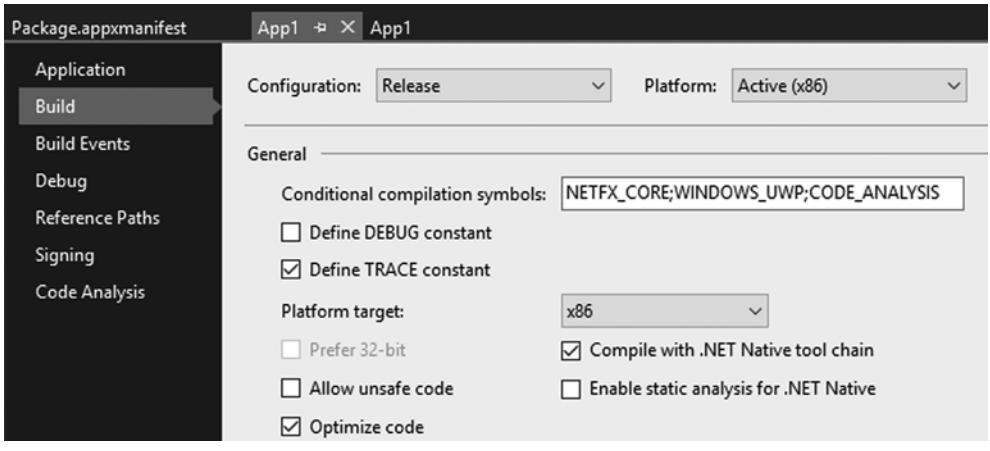

Рис. 3.3. Приложения универсальной платформы Windows позволяют указывать на необходимость использования .NET Native. Изначально для сборок конечных версий это средство применимо

Настраиваемая предварительная подготовка

Если другие упомянутые в этой главе приемы не срабатывают, можно взять дело в свои руки и создать систему для предварительной подготовки вашего кода путем его выполнения перед выполнением приложения в производственном сценарии. Выполнение каждого метода станет причиной проведения JIT-компиляции. Это весьма популярный метод для онлайн-сценариев, где жизненно важная роль отводится физическому времени. Могут быть внутренние периоды ожидания в несколько миллисекунд или же клиент, отказывающийся от услуг через несколько секунд. При наличии большой кодовой базы первый запрос (или первые множество запросов) может потерпеть полный провал из-за JIT-компиляции.

Если JIT-компилятору предстоит большой объем работы, профильной оптимизации может оказаться недостаточно. Перед тем как она сможет справиться с реальной рабочей нагрузкой, возможно, понадобится просто выполнить код в тестовом или автономном режиме.

Прежде чем решать, насколько данный подход будет правильным, было бы неплохо ответить самому себе на ряд наводящих вопросов.

1. Спроектирован ли ваш код таким образом, чтобы его можно было с легкостью вызвать в ходе выполнения сценария предварительной подготовки?
2. Сколько времени уходит на предварительную подготовку? Позволительно ли вашему приложению тратить больше времени на запуск?
3. Если предстоит предварительно подготовить большой объем кода, можно ли выполнить JIT-компиляцию в параллельном режиме?
4. Если для предварительной подготовки требуются данные, нельзя ли их сгенерировать автоматически?

5. Безопасны ли данные, используемые при предварительной подготовке (то есть не относятся ни к производственной среде, ни к потребителям и могут быть повторно использованы)?
6. Воздействует ли ваш код предварительной подготовки на другие системы? Даже если приложение способно занять 32 ядра сплошной JIT-компиляцией, не сможет ли это негативно повлиять на работу внешних систем?
7. Не повлияет ли код предварительной подготовки на измерения? Нужно свести это влияние к минимуму, разделив показатели или по-иному настроив их, чтобы исключить данные, относящиеся к предварительной подготовке.

Чтобы ответить на некоторые из этих вопросов, может понадобиться прототипирование. Предварительная подготовка не должна быть чем-то особо выдающимся, но, скорее всего, отнести ее к весьма тривиальной задаче тоже не получится.

Вряд ли при предварительной подготовке для JIT-компиляции будет вызван каждый отдельный фрагмент необходимого кода, но если такой подготовке подвергнется значительный процент кода (достаточный, чтобы позже избежать ошибок или задержек), результат можно будет считать вполне приемлемым.

Когда JIT-компиляция не может составить конкуренцию

JIT-компиляция подходит для большинства приложений. Когда возникают вопросы к производительности, то по сравнению с качеством или скоростью непосредственно генерации кода обычно имеются проблемы более высокого порядка. Но есть ряд областей, где имеется пространство для улучшения качества JIT-компиляции.

Одна из основных ситуаций, при которой JIT-компилятор не будет так же хорош, как компилятор, создающий машинный код, относится к прямому доступу к собственной памяти по сравнению с доступом к управляемому массиву. Прежде всего непосредственный доступ к собственной памяти обычно означает возможность избежать копирования содержимого памяти, сопровождающего ее маршалинг в управляемый код. И хоть существуют способы скрыть это с применением, к примеру, класса `UnmanagedMemoryStream`, оборачивающего буфер собственной памяти в `Stream`, фактически вы просто создаете небезопасный доступ к памяти.

При передаче байтов в управляемый буфер код, обращающийся к буферу, будет проходить контроль границ. Во многих случаях такой контроль может быть оптимизирован, но это не гарантируется. Управляемый буфер можно обернуть в указатель и выполнять небезопасный доступ для обхода некоторых из проверок.

Если окажется, что в данном случае неуправляемый код действительно более эффективен, можно попробовать провести маршалинг всего набора данных в не-

управляемую функцию посредством вызова платформы (P/Invoke), выполнить вычисления с помощью DLL-библиотеки C++, имеющей высокую степень оптимизации, а затем возвратить результаты управляемому коду. При этом необходимо профилирование, чтобы понять, стоит ли тратиться на передачу данных.

Зрелые компиляторы C++ могут проявить себя с наилучшей стороны и при других типах оптимизации, например при встраивании кода или оптимальном использовании регистров, но, скорее всего, это изменится при выходе последующих версий JIT-компилятора.

Задействуя приложения с очень большим объемом манипуляций с массивами или матрицами, вам придется искать компромисс между производительностью и безопасностью, но, честно говоря, работая с большинством приложений, вам не придется беспокоиться о проверке границ, так как она не приведет к существенным издержкам. Кроме того, если выполняется большой объем математических операций, одним из возможных вариантов станет применение инструкций типа «одна инструкция — множество данных» (Single Instruction, Multiple Data, SIMD), которые стали доступны JIT-компилятору в среде .NET 4.6. Примеры их использования найдете в главе 6.

Исследование поведения JIT-компилятора

Счетчики производительности

Среда CLR публикует несколько счетчиков в категории .NET CLR Jit. В их числе:

- # of IL Bytes Jitted (Количество байтов кода IL, подвергшихся JIT-компиляции);
- # of Methods Jitted (Количество методов, подвергшихся JIT-компиляции);
- % Time in Jit (Процент времени, затраченного на JIT-компиляцию);
- IL Bytes Jitted /sec (Количество байтов кода IL, подвергшихся JIT-компиляции за 1 с);
- Standard Jit Failures (Стандартные сбои JIT-компиляции);
- Total # of IL Bytes Jitted (Общее количество байтов кода IL, подвергшихся JIT-компиляции).

Названия всех этих счетчиков, за исключением Standard Jit Failures, говорят сами за себя. Сбои могут случаться, только если код на языке IL не прошел проверку или возникла внутренняя ошибка JIT-компилятора.

Существует также категория для загрузки, тесно связанная с JIT-компиляцией, которая называется .NET CLR Loading. В число счетчиков этой категории входят:

- % Time Loading (Процент времени, затраченного на загрузку);
- Bytes in Loader Heap (Количество байтов в куче загрузчика);
- Total Assemblies (Общее количество сборок);
- Total Classes Loaded (Общее количество загруженных классов).

ETW-события

Используя ETW-события, можно получить детализированную информацию о производительности каждого метода, проходящего JIT-компиляцию в вашем процессе, включая размер IL-кода, размер машинного кода и количество времени, потраченного на JIT-компиляцию.

- `MethodJittingStarted`. Проходит JIT-компиляция метода. Поля включают:
  - `MethodID` — уникальный идентификатор метода;
  - `ModuleID` — уникальный идентификатор модуля, которому принадлежит метод;
  - `MethodILSize` — размер IL-кода метода;
  - `MethodNameSpace` — полное имя класса, которому принадлежит метод;
  - `MethodName` — имя метода;
  - `MethodSignature` — разделенный запятыми список имен типов из сигнатуры метода.
- `MethodLoad V1`. Метод прошел JIT-компиляцию и был загружен. Для обобщенных и динамических методов эта версия не используется. Поля включают:
  - `MethodID` — уникальный идентификатор для этого метода;
  - `ModuleID` — уникальный идентификатор для модуля, которому принадлежит этот метод;
  - `MethodSize` — размер скомпилированного ассемблерного кода после JIT-компиляции;
  - `MethodStartAddress` — стартовый адрес метода;
  - `MethodFlags` — флаги метода:
    - `0x1` — динамический метод;
    - `0x2` — обобщенный метод;
    - `0x4` — прошедший JIT-компиляцию (если отсутствует, значит, метод был сгенерирован с помощью NGEN);
    - `0x8` — вспомогательный метод.
- `MethodLoadVerbose V1`. Обобщенный или динамический метод прошел JIT-компиляцию и был загружен. Обладает в основном такими же полями, как и `MethodLoad V1` и `MethodJittingStarted`.

Какой код подвергся JIT-компиляции

Если нужно провести ревизию кода в вашем процессе, например, чтобы посмотреть, какая сборка использует больше всего памяти после JIT-компиляции, следует изучить размеры IL и машинного кода всех методов процесса.

Задействуя библиотеку CLR MD, можно проанализировать каждый имеющийся в процессе метод, увидев размер IL-кода, а также размер машинного кода, полученного в результате JIT-компиляции из IL-кода. Рассмотрим метод, выводящий на экран десять самых больших методов процесса:

```
class MethodSize
{
    public string Module { get; set; }
    public string TypeName { get; set; }
    public string Name { get; set; }
    public ulong ILSize { get; set; }
    public ulong NativeSize { get; set; }
}

const string TargetProcessName = "LargeMemoryUsage.exe";

private static void PrintTop10BiggestMethods(ClrRuntime clr)
{
    PrintHeader("Top 10 Methods");
    List<MethodSize > methods = new List<MethodSize >();

    for (int i = 0; i < clr.Modules.Count; i++)
    {
        // Рассматриваются только наши собственные методы

        var module = clr.Modules[i];

        if (!module.FileName.EndsWith(TargetProcessName))
        {
            continue;
        }
        string filename = Path.GetFileName(module.FileName);

        foreach (var type in module.EnumerateTypes())
        {
            for (var iMethod = 0;
                 iMethod < type.Methods.Count;
                 iMethod++)
            {
                ulong ilSize = 0;
                ulong nativeSize = 0;

                var method = type.Methods[iMethod];

                if (method.IL != null)
                {
                    ilSize += (ulong)method.IL.Length;

                    if (method.ILOffsetMap != null)
                    {
                        for (var iOffset = 0;
```

```
                            iOffset < method.ILOffsetMap.Length;
                            iOffset++)
                        {
                            var entry = method.ILOffsetMap[iOffset];
                            var size = entry.EndAddress -
                                       entry.StartAddress;
                            nativeSize += size;
                        }
                    }
                }
                var methodSize = new MethodSize()
                {
                    Module = filename ,
                    TypeName = type.Name,
                    Name = method.Name,
                    ILSize = ilSize ,
                    NativeSize = nativeSize
                };
                methods.Add(methodSize);
            }
        }
    }

    methods.Sort((a, b) =>
    {
        return -a.NativeSize.CompareTo(b.NativeSize);
    });

    Console.WriteLine(
      "Module , Type, Method , IL Size, Native Size");
    Console.WriteLine(
      "-----------------------------------------");
    for (int i=0;i<Math.Min(10, methods.Count);i++)
    {
        var method = methods[i];
        Console.WriteLine(
          $"{method.Module}, {method.TypeName}, {method.Name}, " +
           "{method.ILSize}, {method.NativeSize}");
    }
}
```

Этот код выведет информацию, похожую на следующую:

```
Top 10 Methods
==============
Module, Type, Method, IL Size, Native Size
-----------------------------------------
LargeMemoryUsage.exe,LargeMemoryUsage.Program,Main,116,348
LargeMemoryUsage.exe,LargeMemoryUsage.Program,GetNewObject,67,250
LargeMemoryUsage.exe,LargeMemoryUsage.Base,.ctor,21,113
LargeMemoryUsage.exe,LargeMemoryUsage.Program,.cctor,16,88
```

```
LargeMemoryUsage.exe,LargeMemoryUsage.C,.ctor,14,70
LargeMemoryUsage.exe,LargeMemoryUsage.D,.ctor,14,69
LargeMemoryUsage.exe,LargeMemoryUsage.B,.ctor,14,69
LargeMemoryUsage.exe,LargeMemoryUsage.A,.ctor,14,69
LargeMemoryUsage.exe,LargeMemoryUsage.D,ToString,12,51
LargeMemoryUsage.exe,LargeMemoryUsage.C,ToString,12,51
```

На какие методы и модули затрачивается больше всего времени при JIT-компиляции

В целом время, требующееся на JIT-компиляцию, прямо пропорционально количеству IL-инструкций в методе, но все усложняется тем, что время загрузки типов также может быть включено в это время, особенно при первом применении модуля. В некоторых типичных ситуациях в JIT-компилятор могут быть включены сложные алгоритмы, на выполнение которых нужно больше времени. Для получения подробной информации об активности JIT-компилятора в вашем процессе можно воспользоваться средством PerfView. Если собрать стандартные .NET-события, будет получено особое представление, названное JITStats. Далее приводится часть той информации, которая выводится при использовании данного приема для проекта из примера PerfCountersTypingSpeed.

Name	JitTime msec	Num Methods	IL Size	Native Size
PerfCountersTypingSpeed.exe	12,9	8	1,756	3,156

JitTime msec	IL Size	Native Size	Method Name
9,7	22	45	PerfCountersTypingSpeed.Program.Main()
0,3	176	313	PerfCountersTypingSpeed.Form1..ctor()
1,4	1,236	2,178	PerfCountersTypingSpeed.Form1.Initialize-Component()
0,8	107	257	PerfCountersTypingSpeed.Form1.CreateCus-tomCategories()
0,3	143	257	PerfCountersTypingSpeed.Form1.timerTick(class System.Object,class System.EventArgs)
0,1	23	27	PerfCountersTypingSpeed.Form1. OnKeyPress(class System.Object,class System.Windows.Forms.KeyPressEventArgs)
0,2	19	36	PerfCountersTypingSpeed.Form1.OnClo-sing(classSystem.ComponentModel.Cancel-EventArgs)
0,1	30	43	PerfCountersTypingSpeed.Form1.Dispose(bool)

Единственным методом, на который при JIT-компиляции уходит больше времени, чем можно было бы предположить, исходя из размера его IL-кода, является `Main`. В этом нет ничего удивительного, поскольку именно на него приходятся самые высокие затраты на загрузку.

Исследование кода, полученного после JIT-компиляции

Воспользовавшись WinDbg или Visual Studio, можно без особого труда увидеть дизассемблированный код вокруг местоположения текущей инструкции и с этого места перейти в какое-либо другое место. Находясь в Visual Studio на точке останова, щелкните правой кнопкой мыши где-нибудь на исходном коде и выберите в контекстном меню пункт Go to disassembly (Перейти к дизассемблированному коду).

Легко получить аннотированный дамп дизассемблированного кода непосредственно в конкретном методе в WinDbg, воспользовавшись командой `!U`. Для этого нужно получить указатель структуры `MethodDesc`:

```
0:000> !DumpStack
OS Thread Id: 0x5580 (0)
Current frame: ntdll!NtDeviceIoControlFile+0xc
ChildEBP RetAddr Caller, Callee
...
012ff2e4 7217c50a (MethodDesc 716f0d54 +0xe6
  System.Console.ReadKey(Boolean)), calling 71995a48
012ff374 039b0514 (MethodDesc 035d4d64 +0x9c
  LargeMemoryUsage.Program.Main(System.String[])),
  calling (MethodDesc 716f0d54 +0 System.Console.ReadKey(Boolean))
012ff398 72cceb16 clr!CallDescrWorkerInternal+0x34
...
```

Отсюда мы воспользуемся значением `MethodDesc` для метода `Main`, `035d4d64`:

```
0:000> !U 035d4d64
Normal JIT generated code
LargeMemoryUsage.Program.Main(System.String[])
Begin 039b0478, size bc

D:\...\LargeMemoryUsage\Program.cs @ 16:
039b0478 55         push   ebp
039b0479 8bec       mov    ebp,esp
039b047b 57         push   edi
039b047c 56         push   esi
039b047d 53         push   ebx
039b047e 83ec10     sub    esp,10h
039b0481 b9926a6371 mov    ecx,offset
  mscorlib_ni!System.Collections.IStructuralEquatable.Equals+0x99
  (71636a92)
039b0486 bae8030000 mov    edx,3E8h
```

```
039b048b e8202dc1ff call   035c31b0
  (JitHelp: CORINFO_HELP_NEWARR_1_OBJ)
039b0490 8945e4     mov    dword ptr [ebp-1Ch],eax

D:\...\LargeMemoryUsage\Program.cs @ 18:
039b0493 b9dc7eb471 mov ecx,offset
  mscorlib_ni+0x517edc (71b47edc) (MT: System.Random)
039b0498 e82b2cc1ff call 035c30c8
  (JitHelp: CORINFO_HELP_NEWSFAST)
039b049d 8bf0 mov esi,eax
039b049f e8dc70326f call clr!SystemNative::GetTickCount
  (72cd7580)
039b04a4 8bd0 mov edx,eax
039b04a6 8bce mov ecx,esi
039b04a8 e817d40c6e call mscorlib_ni+0x44d8c4 (71a7d8c4)
  (System.Random..ctor(Int32), mdToken: 060010dc)
...
```

Резюме

Чтобы свести влияние JIT-компиляции к минимуму, следует тщательно изучить любые области объемного сгенерированного кода, возникшие при использовании регулярных выражений, генерации кода, динамических переменных или любого другого источника. Задействуйте профильную оптимизацию для сокращения времени запуска приложения путем предварительной JIT-компиляции наиболее востребованного кода в параллельном режиме. Убедитесь, что применяется последняя версия .NET, чтобы воспользоваться преимуществами улучшений, внесенных в JIT-компилятор.

Для стимуляции встраивания функций следует избегать виртуальных методов, циклов, обработки исключений, рекурсий или больших тел методов. Но не стоит жертвовать целостностью приложения ради излишней оптимизации в данной области.

Продумайте, как использовать NGEN-генератор для больших приложений или в ситуациях, когда затраты на JIT-компиляцию при запуске приложения недопустимы. Для оптимизации образов в машинном коде предваряйте применение NGEN запуском средства MPGO.

Чтобы создать приложения Windows Store, воспользуйтесь средством .NET Native.

Если ничто не сработает, создайте для своего приложения особую стратегию предварительной подготовки, которая бы задействовала самые востребованные пути, прежде чем они реально понадобятся.

4 Асинхронное программирование

Многопоточные программы повсеместно применяются в современных компьютерах, включая небольшие устройства вроде сотовых телефонов и многоядерных процессоров, поэтому навык их эффективной разработки критически важен для любого программиста.

Есть три причины использования нескольких потоков.

1. Нежелание блокировать основной поток пользовательского интерфейса фоновой работой.
2. Объем работы настолько велик, что нельзя позволить себе тратить попусту время центрального процессора в ожидании завершения ввода-вывода.
3. Желание воспользоваться всеми имеющимися в вашем распоряжении процессорами.

Первая причина связана не столько с производительностью, сколько с желанием не раздражать конечного пользователя. Вторая и третья целиком относятся к эффективности использования вычислительных ресурсов.

Компьютерные процессоры фактически достигли максимума с точки зрения чистой тактовой частоты. Основной метод, который в обозримом будущем станет применяться для достижения более высокой пропускной способности при вычислениях, — распараллеливание. Работа сразу нескольких процессоров имеет решающее значение для создания высокопроизводительного приложения, особенно для серверов, обрабатывающих множество одновременно поступающих запросов.

Есть несколько способов параллельного выполнения кода в среде .NET. Например, можно вручную запустить поток и передать ему метод на выполнение. Это неплохо подходит для довольно продолжительных методов, но для многого другого непосредственная работа с потоками весьма неэффективна. Если, к примеру, нужно выполнить множество коротких задач, издержки на диспетчеризацию отдельно взятых потоков легко смогут превысить затраты на фактическое выполнение кода. Чтобы разобраться в причинах этого, следует узнать порядок диспетчеризации потоков в Windows.

Каждый процессор способен одновременно выполнять только один поток. Когда настает момент диспетчеризации потока для процессора, системе Windows требуется выполнить переключение контекста. В ходе этого ядро Windows сохраняет текущее состояние потока процессора во внутреннем объекте потока операционной системы, выбирает нужный поток из готовых потоков с наивысшей степенью приоритета, переносит контекстную информацию потока из объекта потока в процессор и в завершение запускает поток на выполнение. Если Windows

переключается на поток из другого процесса, затраты увеличиваются, поскольку выгружается все адресное пространство.

Затем поток выполнит код для выделенного ему кванта времени, составляющего несколько тактовых интервалов (на современных многопроцессорных системах тактовый интервал длится примерно 15 мс и не может быть просто изменен, что, собственно, и не рекомендуется делать). Когда происходит возврат с вершины стека, или поток входит в состояние ожидания, или истекает квант времени, диспетчер выбирает для выполнения другой готовый поток. Это может быть тот же самый поток или другой поток, в зависимости от конкуренции за процессорное время. Поток может войти в состояние ожидания, если он блокируется на любой разновидности ввода-вывода, или же он добровольно входит в это состояние путем вызова метода `Thread.Sleep`.

ПРИМЕЧАНИЕ

У системы Windows Server более высокий показатель кванта времени, чем у версии Windows для настольных систем, следовательно, потоки запускаются на более длительный период времени, прежде чем будет выполнено переключение контекста. Этот параметр отчасти контролируется в параметрах производительности (Performance Options) расширенных настроек системы (Advanced System Settings). Установка для этой настройки значения Background services (Фоновые службы) (рис. 4.1) приведет к увеличению для системы исходного кванта времени потока, возможно, за счет времени реагирования программы на внешние события. Эти настройки сохраняются в реестре, но непосредственно манипулировать ими не стоит.

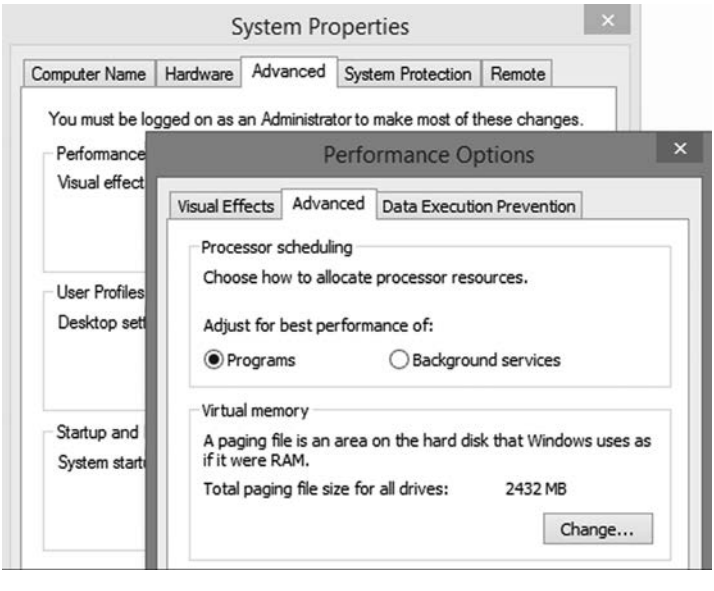

Рис. 4.1. Решение о том, чему на машине отдать приоритет: фоновым задачам или задачам переднего плана

Пул потоков

Создание новых потоков — весьма затратная процедура. Она требует размещения в ядре, передачи памяти для пространства стека, роста количества переключений контекста и, возможно, запуска кода инициализации в нескольких DLL-библиотеках для каждого потока. Так как эти затраты неизбежны, не следует, к примеру, создавать поток для обработки каждого отдельно взятого запроса. К потокам нужно относиться как к объединенному в пул общему ресурсу. Как только потоки становятся доступными, они могут повторно использоваться для следующей части работы. К счастью, среда .NET облегчает эту задачу и предоставляет управляемый пул потоков для каждого управляемого процесса. Этот пул содержит список потоков, сохраняя их в работоспособном состоянии на период невостребованности. Чтобы воспользоваться пулом, нужно поставить метод в очередь на выполнение:

```
static void Main()
{
          ThreadPool.QueueUserWorkItem(new WaitCallback(MyFunc),
                                       "my data");
}

private static void MyFunc(Object obj)
{
    var data = obj as string;
    // выполнение реальной работы
}
```

Эти потоки создаются по мере необходимости и сохраняются для того, чтобы удовлетворить возникающие впоследствии потребности и избежать затрат на их повторное создание. Программа экономит на создании и удалении, и почти всегда у нее есть поток, готовый к обработке возникающих асинхронных задач.

В системе имеется два пула.

1. Пул рабочих потоков занимается задачами, связанными с работой центрального процессора и назначением потоков различным ядрам.
2. Пул потоков ввода-вывода выполняет асинхронный ввод-вывод с оборудования, например, когда происходит возвращение данных с устройств хранения или из сети.

Пул потоков проделывает сложную работу. Ему нужно обеспечить наличие доступного потока для обработки любых возникающих задач, а также умеренное количество запущенных потоков, чтобы оно не оказывало катастрофического влияния на производительность. Пул стремится оптимизироваться под пропускную способность, соответствующую количеству задач, выполняемых за единицу времени. Этот показатель может балансировать между слишком малым количеством потоков (недостаточным для выполняемой работы) и слишком большим их числом (вызывающим чрезмерную конкуренцию за ресурсы).

В мире без конкуренции идеально было бы, чтобы один процессор выполнял один поток. Но большинство задач, выполняемых потоками, далеки от совершенства и подвержены блокировкам либо для выполнения ввода-вывода, либо при потреблении некоторых других ресурсов. Поэтому имеет смысл для пула потоков выполнять диспетчеризацию большего количества потоков на одно ядро, обеспечивая тем самым его достаточную занятость. Но если ядро окажется перегруженным, увеличившееся количество потоков повлечет за собой повышение издержек пула на работу с ними и процессор может начать тратить больше времени на переключение, чем на какие-либо другие дела (точнее, это может произойти независимо от источника потоков, но самым распространенным источником является пул). Это весьма тонкий баланс.

Основной способ соблюдения такого баланса пулом потоков — использование стандартного алгоритма Hill Climbing (восхождение к вершине), основанного на саморегуляции для подстройки с течением времени. Пул занимается диспетчеризацией потоков на выполнение и через определенные интервалы времени отслеживает свою пропускную способность (количество завершенных задач за установленный интервал). Он продолжит добавлять потоки, пока будет отмечать падение пропускной способности, после чего приступит к удалению потоков. Поскольку пропускная способность может изменяться из-за типологии рабочей нагрузки, этот подход приводит к слишком частым малопредсказуемым спорадическим изменениям числа потоков. Поэтому пул потоков рассматривает пропускную способность как непрерывный сигнал и применяет к нему преобразование Фурье, которое уменьшает влияние небольших колебаний. Даже когда пул потоков прекращает диспетчеризацию такого большого количества потоков, он может сохранить некоторые из них в памяти, чтобы позднее избежать нового выделения памяти для потоков (именно поэтому и делает его пулом).

Пул потоков должен выбираться для диспетчеризации потоков в первую очередь, но бывают случаи, когда это решение неприемлемо. Например, оно не будет оптимальным для потоков, требующих конкретных уровней приоритета или выполняющих долговременные задачи. При большом количестве задач, на выполнение которых уходит несколько сотен миллисекунд, пул потоков будет неэффективен. Могут также возникать ситуации, когда требуется выделенный поток для выполнения конкретной задачи, как долговременной, так и нет. Для таких случаев следует создавать собственный поток и управлять им.

Библиотека распараллеливания задач

Если ваша программа состоит из чистых задач центрального процессора, время выполнения которых существенно превышает продолжительность кванта потока, то вполне приемлемым, хотя, как выяснится в дальнейшем, и не необходимым, будет непосредственное создание и применение потоков. Но если ваш код состоит из множества небольших задач, на которые вряд ли уйдет столько времени, непосредственное использование потоков будет неэффективным, поскольку программа

будет тратить существенно большее количество времени на переключение контекстов, чем на выполнение собственного кода. Хотя, как сказано в предыдущем разделе, вы можете непосредственно воспользоваться пулом потоков, так делать больше не рекомендуется. Вместо этого как для продолжительных, так и для непродолжительных действий можно применить объект `Task`.

В .NET 4.0 была введена абстракция потоков под названием `Task`, которая служит частью библиотеки распараллеливания задач (Task Parallel Library, TPL), являющейся в большинстве случаев наиболее предпочтительным способом достижения параллелизма в .NET. TPL предоставляет широкие возможности для управления запуском вашего кода, позволяя определить, что случилось при возникновении ошибки, давая возможность выстроить условную последовательность из нескольких методов и открывая многие другие возможности.

В своей внутренней структуре TPL задействует имеющийся в .NET пул потоков, но делает это более эффективно, последовательно выполняя несколько `Task`-делегатов в одном и том же потоке, прежде чем возвратить поток в пул. Такая возможность возникает благодаря рациональному использованию делегатов. Тем самым оказывается эффективно обойденной проблема расточительного расхода кванта потока на одну небольшую задачу, который является причиной слишком большого количества переключений контекста.

TPL представляет собой большой всеобъемлющий набор API, но освоить его легко. Основной принцип заключается в том, что вы передаете делегат имеющемуся в классе `Task` методу `Start`. Вы также можете дополнительно вызвать в отношении объекта `Task` метод `ContinueWith` и передать второй делегат, выполняемый, как только первоначальная задача завершится. Освоить различные варианты выполнения и продолжения важно для достижения наивысшей производительности с наименьшими издержками. В этом разделе для расстановки ориентиров будут кратко рассмотрены наиболее распространенные способы управления задачами.

Выполнять объекты `Task` можно как с вычислениями, связанными с процессором, так и с задачами ввода-вывода. Все приводимые здесь примеры покажут чистую обработку данных центральным процессором. А чуть позже в этой главе вы увидите раздел, посвященный эффективному выполнению задач ввода-вывода.

Следующий пример кода можно найти в проекте Tasks, он демонстрирует создание объекта `Task` для каждого процессора. Когда каждый объект `Task` завершает свое выполнение, он ставит в очередь на выполнение `Task`-продолжение, в котором есть ссылка на метод обратного вызова.

```
class Program
{
    static Stopwatch watch = new Stopwatch();
    static int pendingTasks;
    static void Main(string[] args)
    {
        const int MaxValue = 1000000000;

        watch.Restart();
```

```
        int numTasks = Environment.ProcessorCount;
        pendingTasks = numTasks;
        int perThreadCount = MaxValue / numTasks;
        int perThreadLeftover = MaxValue % numTasks;

        var tasks = new Task<long >[numTasks];

        for (int i = 0; i < numTasks; i++)
        {
            int start = i * perThreadCount;
            int end = (i + 1) * perThreadCount;
            if (i == numTasks - 1)
            {
                end += perThreadLeftover;
            }
            tasks[i] = Task<long >.Run(() =>
            {
                long threadSum = 0;
                for (int j = start; j <= end; j++)
                {
                    threadSum += (long)Math.Sqrt(j);
                }
                return threadSum;
            });
            tasks[i].ContinueWith(OnTaskEnd);
        }
    }

    private static void OnTaskEnd(Task<long> task)
    {
        Console.WriteLine("Thread sum: {0}", task.Result);
        if (Interlocked.Decrement(ref pendingTasks) == 0)
        {
            watch.Stop();
            Console.WriteLine("Tasks: {0}", watch.Elapsed);
        }
    }
}
```

Если `Task`-продолжение является быстро выполняемым коротким фрагментом кода, следует указать, что оно запускается в том же самом потоке, что и владеющая им задача. В экстремально многопоточной системе это жизненно важный момент, поскольку на выстраивание в очередь объектов `Task` для выполнения в отдельном потоке, использование которого может повлечь за собой переключение контекста, может уйти впустую слишком много времени.

```
task.ContinueWith(OnTaskEnd ,
TaskContinuationOptions.ExecuteSynchronously);
```

Если `Task`-продолжение вызывается из потока ввода-вывода, использовать `TaskContinuationOptions.ExecuteSynchronously` не стоит, поскольку можно занять этой задачей поток ввода-вывода, который вам нужен для извлечения данных из

сети. И как всегда, вам придется поэкспериментировать и тщательно измерить результаты. Зачастую более эффективными для потока ввода-вывода оказываются быстрое продолжение работы и отказ от дополнительной диспетчеризации.

Если нужно выполнить долговременную задачу, следует передать методу `Task.Factory.StartNew` флаг `TaskCreationOptions.LongRunning`. Существует также версия этого флага для продолжений:

```
var task = Task.Factory.StartNew(action ,
            TaskCreationOptions.LongRunning);
task.ContinueWith(OnTaskEnd , TaskContinuationOptions.LongRunning);
```

Продолжения — это неоспоримое достоинство TPL. Можно выполнять всевозможные сложные действия, выходящие за рамки вопросов повышения производительности, и некоторые из них я здесь вкратце упомяну.

В рамках одной задачи `Task` можно выполнить несколько продолжений:

```
Task task = ...
task.ContinueWith(OnTaskEnd);
task.ContinueWith(OnTaskEnd2);
```

`OnTaskEnd` и `OnTaskEnd2` никак не связаны между собой и выполняются независимо друг от друга и, насколько это возможно, параллельно.

В то же время можно выстроить продолжения в цепочку:

```
Task task = ...
task.ContinueWith(OnTaskEnd).ContinueWith(OnTaskEnd2);
```

Продолжения, выстроенные в цепочку, последовательно зависят друг от друга. Когда выполнение задачи завершается, запускается `OnTaskEnd`. Когда завершается выполнение и этой задачи, выполняется `OnTaskEnd2`.

Продолжениям может предписываться выполнение только после успешного завершения предыдущей задачи (или ее сбоя, или прекращения и т. д.):

```
Task task = ...
task.ContinueWith(OnTaskEnd ,
                  TaskcontinuationOptions.OnlyOnRanToCompletion);
task.ContinueWith(OnTaskEnd ,
                  TaskContinuationOptions.NotOnFaulted);
```

Продолжение можно вызвать, только когда завершится выполнение нескольких `Task`-объектов или любого из них:

```
Task[] tasks = ...
Task.Factory.ContinueWhenAll(tasks , OnAllTaskEnded);
Task.Factory.ContinueWhenAny(tasks , OnAnyTaskEnded);
```

Использование этих API существенно упрощает обеспечение того, что большие участки вашей программы останутся на 100 % асинхронными, без вызова блокировок или ненужных точек синхронизации.

Отмена задачи

Отмена уже запущенной задачи `Task` требует организации взаимодействия. Принудительно завершать выполнение потока не рекомендуется, и Task Parallel Library не дает вам доступа к исходному потоку, не говоря уже о прерывании его работы. Надо отметить, что при непосредственном применении объекта `Thread` может быть вызван метод `Abort`, но это небезопасно и так делать не рекомендуется. Просто действуйте так, будто этого API не существует. Прерывание выполнения потока может оставить объекты синхронизации в неизвестных состояниях, совместно используемое состояние может быть повреждено, и асинхронные операции могут никогда не завершиться.

Чтобы отменить `Task`, делегату следует передать объект `CancellationToken`, и затем он сможет опросить токен, чтобы определить, нужно ли ему завершать работу. В следующем примере показано также применение лямбда-выражения в качестве `Task`-делегата.

Этот код можно найти в проекте TaskCancellation:

```
static void Main(string[] args)
{
    var tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;

    Task task = Task.Run(() =>
    {
        while (true)
        {
            // выполнение реальной работы ...
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Cancellation requested");
                return;
            }
            Thread.Sleep(100);
        }
    }, token);

    Console.WriteLine("Press any key to exit");

    Console.ReadKey();

    tokenSource.Cancel();

    task.Wait();

    Console.WriteLine("Task completed");
}
```

Обработка исключений

Если в ходе выполнения `Task` выдает исключение, то происходящее в дальнейшем зависит от того, как вы управляете самой задачей и результатом ее выполнения. Примитивный способ заключается в простом получении доступа к свойству `Result`. Если результата нет по причине выдачи задачей `Task` исключения, то поток, обращающийся к свойству `Result` или явным образом вызывающий метод `Wait`, получит исключение `AggregateException`.

```
var task = Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
});
task.Wait(); // исключение здесь!
int result = task.Result;
```

Если явное ожидание не задается, но при этом имеется продолжение, возникает аналогичная проблема:

```
Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
}).ContinueWith(task =>
{
    int val = task.Result; // исключение здесь!
});
```

Существует несколько решений этой проблемы. Так, доступ к `Result` можно заключить в обработчик исключений:

```
Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
}).ContinueWith(task =>
{
    try
    {
        // безопасная обработка результата
        int val = task.Result;
    }
    catch(AggregateException ex)
    {
        LogException(ex);
    }
});
```

Но чуть позже мы узнаем, что выдача исключений крайне нежелательна. Класс `Task` предоставляет несколько свойств для определения состояния задачи:

- `IsCanceled` — задача отменена;
- `IsFaulted` — выдано исключение, доступ к которому можно получить через свойство `Exception`;
- `IsCompleted` — соответствует истине, если задача считается завершенной из-за ошибки, отмены или успешного окончания;
- `IsCompletedSuccessfully` — соответствует истине, если задача считается завершенной без ошибки или отмены.

Свойством `IsFaulted` можно воспользоваться, чтобы выяснить, есть ли необработанные исключения:

```
Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
}).ContinueWith(task =>
{
    if (task.IsFaulted)
    {
        LogException(task.Exception);
    }
    else
    {
        // безопасная обработка результата
        int val = task.Result;
    }
});
```

Незамеченные исключения задачи Task. А что произойдет, если вы не обработаете исключение в задаче `Task`? До выпуска .NET 4.5 то, что исключение задачи `Task` было оставлено без обработки, приводило к сбою процесса. Когда исключение остается незамеченным, объект, который знает о нем, будет помещен в очередь финализации. При следующем запуске финализаторов сборщик мусора выдаст исключение. Это вызовет трудности с определением фактического стека вызовов, при котором произошло исключение, поскольку исключение на этот момент будет иметь стек, берущий начало в потоке финализации. Вам нужно будет докопаться до исключений, включенных в свойство `InnerExceptions`, принадлежащее объекту `AggregateException`, чтобы увидеть реальные исключения и извлечь из них стеки.

Если нужно просто увидеть стандартный строковый дамп дерева исключений, можно в отношении `AggregateException` вызвать метод `ToString`. В дампе будут содержаться и все вложенные исключения.

Если требуется более тонкое управление логированием исключений, можно воспользоваться методом, похожим на следующий, который правильно выполняет обработку `AggregateException`:

```
public static class ExceptionUtils
{
    public static void LogException(System.Exception exception)
    {
        LogExceptionRecursive(exception , 0, null);
    }

    private static void LogExceptionRecursive(System.Exception ex,
                                              int recursionLevel)
    {
        if (ex == null)
        {
            return;
        }
        if (recursionLevel >= 10)
        {
            // Обрезание рекурсии в целях безопасности
            return;
        }

        Console.WriteLine(
            $"Type: {ex.GetType()}, Message: {ex.Message}," +
            $" Stack: {ex.StackTrace}, Level: {recursionLevel}");
        var aggEx = ex as AggregateException;
        if (aggEx != null && aggEx.InnerExceptions != null)
        {
            foreach (var inner in aggEx.InnerExceptions)
            {
                LogExceptionRecursive(inner , recursionLevel + 1);
            }
        }
        else if (ex.InnerException != null)
        {
            LogExceptionRecursive(ex.InnerException ,
                                  recursionLevel + 1);
        }
    }
}
```

В каких случаях можно утверждать, что наблюдение за исключениями ведется?

- При вызове метода `Wait` в отношении задачи `Task`.
- При обращении к свойству `Result`.
- При обращении к свойству `Exception`.

С выпуском .NET 4.5 исходное поведение незамеченных исключений в задачах изменилось. По умолчанию они больше не вызывают выдачи исключения из потока финализатора. Это решение было принято, так как в платформе .NET Framework все

более широко стала использоваться библиотека Task Parallel и требовалось избежать сбоев в множестве программных продуктов после обновления .NET.

Несмотря на это, как правило, нежелательно позволять исключениям быть «проглоченными» таким образом. Данная ситуация свидетельствует о серьезных проблемах, которые ваше приложение игнорирует, — неопределенном состоянии, способном привести к возникновению впоследствии трудно обнаруживаемых ошибок или даже к повреждению данных. К счастью, прежнее поведение можно восстановить простым переключением конфигурации:

```
<configuration>
    <runtime>
        <ThrowUnobservedTaskExceptions enabled="true"/>
    </runtime>
</configuration>
```

Эти настройки нужно включить во все новые проекты, чтобы гарантированно избежать выполнения своей программы после выдачи необработанного исключения. Следующий вопрос заключается в действенном способе перехвата и регистрации этих исключений. Если ничего не делать, операционная система выполнит захват мини-дампа, из которого можно будет исследовать исключения, но если ваши потребности этим не ограничиваются, нужно установить обработчик необработанных исключений.

```
static void Main()
{
    AppDomain.CurrentDomain.UnhandledException
      += OnUnhandledException;
}

private static void OnUnhandledException(
    object sender ,
    UnhandledExceptionEventArgs e)
{

        var ex = e.ExceptionObject as Exception;
        if (ex != null)
        {
            ExceptionUtils.LogException(ex,
                                        Events.Log.UnhandledException,
                                        true);
        }
        else if (e.ExceptionObject != null)
        {
            var type = e.ExceptionObject.GetType().ToString();
            Console.WriteLine(
              $"Non-exception object: {type} - {e.ExceptionObject}");
        }
        else
        {
            Console.WriteLine("Unknown object");
    }
}
```

Дочерние задачи

Когда задачи создаются из другой задачи, то они по умолчанию независимы друг от друга. В следующем примере создается дочерняя задача, выполняющая произвольный код. Как родительская, так и дочерняя задачи выполняют вывод в консоль, но они совершенно не связаны друг с другом и любая из них может вернуть управление и завершиться первой.

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    });

}).ContinueWith(task => Console.WriteLine("Parent done"));
```

Обычно это то, что вам нужно, поскольку при этом обеспечивается строгая асинхронность — весьма полезная привычка в подобных ситуациях, но есть возможность создать более крепкие взаимоотношения между родительской и дочерней задачами. Один из способов заключается в том, чтобы заставить родительскую задачу ждать завершения дочерней (или полагаться на значение ее свойства `Result`):

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    });

    // Явно заданное ожидание!
    childTask.Wait();
}).ContinueWith(task => Console.WriteLine("Parent done"));
```

Это далеко не идеальное решение, поскольку оно вводит блокирующий вызов в процесс, бывший до этого асинхронным, что нарушает главное правило эффективности асинхронных программ. Альтернативное решение — передача флага `TaskCreationOptions.AttachedToParent`, сообщающего родительской задаче, что она не может завершиться до окончания работы всех ее дочерних задач:

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    }, TaskCreationOptions.AttachedToParent);
}).ContinueWith(task => Console.WriteLine("Parent done"));
```

Этот вариант подойдет, если необходимо обеспечить согласованность между задачами, но все же он вносит новые сложности, от которых нужно избавляться.

Прежде всего, родительские задачи способны не дать дочерним прикрепляться к себе:

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        // выполнение реальной работы
    }, TaskCreationOptions.AttachedToParent);
}, TaskCreationOptions.DenyChildAttach);
```

В данном случае предпочтения родительской задачи соблюдаются, а запрос дочерней на прикрепление игнорируется. Поведение будет таким же, как и в первом примере данного раздела. При запуске задачи через вызов `Task.Run` по умолчанию устанавливается флаг `TaskCreationOptions.DenyChildAttach` вместо используемого здесь вызова метода `Task.Factory.StartNew`.

Нужно также обрабатывать исключения. Исключение в прикрепленной дочерней задаче автоматически передается в родительскую задачу, чтобы обработкой занимался метод продолжения родительской задачи или любой поток, ожидающий результата выполнения задачи. Открепленная дочерняя задача, выдающая исключение, должна обрабатываться точно так же, как и любой другой код внутри родительской задачи, — с проверкой ее свойства `Exception` или с заключением проверки свойства `Result` в обработчик исключений.

Отмена также немного усложняется. Чтобы упростить ситуацию, нужно воспользоваться одним и тем же объектом `CancellationToken` для всех задач в иерархии родителей и потомков. Когда поступает сигнал об отмене, все происходящее далее определяется тем, где и когда случилась отмена.

- Родительская задача отменена до запуска дочерней. Дочерняя вообще не запускается.
- Родительская задача отменена после запуска дочерней. Дочерняя задача выполняется, пока в ней не произойдет проверка на отмену.
- Отменяется прикрепленная дочерняя задача. Исключение `TaskCanceledException` передается родительской задаче и будет включено в принадлежащее ей исключение `AggregateException`.

Среда TPL Dataflow

Многие приложения задействуют конвейер обработки данных, где части данных проходят через различные стадии преобразования и анализа. Для повышения эффективности, производительности и максимального использования ресурсов было бы идеально сделать эти стадии асинхронными, что позволило бы одновременно обрабатывать сразу несколько частей данных в конвейере. Такую систему можно было бы спроектировать с самостоятельным применением Task Parallel Library, но

это повлечет за собой образование сложной системы продолжений, синхронизации и координации. Есть другой вариант — воспользоваться средой TPL Dataflow.

ПРИМЕЧАНИЕ

Среда TPL Dataflow распространяется через NuGet-пакет Microsoft.Tpl.Dataflow. С платформой .NET Framework она не поставляется.

С помощью TPL Dataflow создаются блоки обработки, соединяемые в конвейер или сеть. У вас есть возможность управлять распараллеливанием вычислений каждого блока, и каждый блок может быть выполнен в асинхронном режиме. Блоки передают друг другу сообщения, которые являются простыми объектами, объявляемыми с помощью спецификаторов типа. Блок может быть связан с несколькими источниками или несколькими целевыми объектами. Вся обработка может происходить в асинхронном режиме.

Важно отметить, что применение TPL Dataflow не делает автоматически вашу программу быстрее. Данное средство скорее упрощает достижение высокой производительности для данного алгоритмического решения, используя преимущества TPL и принимая на себя задачу создания большей части рутинного кода, что позволяет вам сконцентрироваться на важной логике приложения.

Прежде чем рассмотреть простой пример, исследуем типы блоков, которые вы можете создать. Существует несколько заранее созданных блоков, предоставляемых вам средой TPL Dataflow.

Тип блока TPL Dataflow	**Описание**
BufferBlock<T>	FIFO-очередь сообщений
BroadcastBlock<T>	Отправляет последнее сообщение всем целевым объектам
WriteOnceBlock<T>	Аналогичен BroadcastBlock, но может установить значение только один раз
ActionBlock<T>	Выполняет делегат для входных данных, выходных данных не производит
TransformBlock<TInput, TOutput>	Выполняет делегат, который может возвращать тип, отличный от принятого
TransformManyBlock<TInput,TOutput>	Аналогичен TransformBlock, но может выдать несколько выходных данных для каждого входа
BatchBlock<T>	Преобразует несколько входных данных в один массив на выходе

Собственные типы блоков можно создавать путем реализации интерфейсов `ISourceBlock<TOutput>` и/или `ITargetBlock<TInput>`.

Пример использования TPL Dataflow. Чтобы увидеть, как блоки TPL Dataflow работают вместе, создадим простой пример конвейера обработки текста. Это приложение обрабатывает каталог текстовых файлов и анализирует частоту появления всех слов в каждом файле. После обработки делаются определенные шаги, объединяющие данные анализа и отсеивающие некоторые общие, не представляющие интереса слова.

Вот как выглядит создание конвейера, возвращающее ссылку на первый блок:

```
private static readonly HashSet <string > IgnoreWords =
    new HashSet <string >() { "a", "an", "the", "and", "of", "to" };
private static readonly Regex WordRegex =
    new Regex("[a-zA-Z]+", RegexOptions.Compiled);

private static ITargetBlock <string > CreateTextProcessingPipeline(
    string inputPath ,
    out Task completionTask)
{
    int fileCount = Directory.GetFiles(inputPath , "*.txt").Length;

    var getFilenames = new TransformManyBlock <string , string >(
        path =>
    {
        return Directory.GetFiles(path, "*.txt");
    });

    var getFileContents = new TransformBlock <string , string >(
        async (filename) =>
    {
        Console.WriteLine("Begin: getFileContents");
        using (var streamReader = new StreamReader(filename))
        {
            return await streamReader.ReadToEndAsync();
        }
        Console.WriteLine("End: getFileContents");
    }, new ExecutionDataflowBlockOptions
        {
        MaxDegreeOfParallelism = Environment.ProcessorCount
        }
    );
    var analyzeContents =
        new TransformBlock <string , Dictionary <string , ulong >>(
    contents =>
    {
        Console.WriteLine("Begin: analyzeContents");
        var frequencies =
            new Dictionary <string , ulong >(
                10000, StringComparer.OrdinalIgnoreCase);

        var matches = WordRegex.Matches(contents);
        foreach (Match match in matches)
        {
            ulong currentValue;
```

```
            if (!frequencies.TryGetValue(match.Value ,
                                         out currentValue))
            {
                currentValue = 0;
            }
            frequencies[match.Value] = currentValue + 1;
        }
        Console.WriteLine("End: analyzeContents");
        return frequencies;
    }, new ExecutionDataflowBlockOptions
        {
            MaxDegreeOfParallelism = Environment.ProcessorCount
        }
    );

    var eliminateIgnoredWords =
        new TransformBlock <Dictionary <string , ulong >,
                            Dictionary <string , ulong >>(
    input =>
    {
        foreach(var word in IgnoreWords)
        {
            input.Remove(word);
        }
        return input;
    });

    var batch =
        new BatchBlock <Dictionary <string , ulong >>(fileCount);
    // Это единая точка синхронизации —
    // вся обработка сходится на ней
    var combineFrequencies =
        new TransformBlock <Dictionary <string , ulong >[],
                            List<KeyValuePair <string , ulong >>>(
    inputs =>
    {
        Console.WriteLine("Begin: combiningFrequencies");
        var sortedList = new List<KeyValuePair <string , ulong >>();
        var combinedFrequencies = new Dictionary <string , ulong >(
         10000, StringComparer.OrdinalIgnoreCase);

        foreach (var input in inputs)
        {
            foreach (var kvp in input)
            {
                ulong currentFrequency;
                if (!combinedFrequencies.TryGetValue(
                        kvp.Key,
                        out currentFrequency))
                {
                    currentFrequency = 0;
                }
                combinedFrequencies[kvp.Key] = currentFrequency +
```

```
                                                    kvp.Value;
            }
        }
        foreach(var kvp in combinedFrequencies)
        {
            sortedList.Add(kvp);
        }
        sortedList.Sort((a, b) =>
        {
            return -a.Value.CompareTo(b.Value);
        });
        Console.WriteLine("End: combineFrequencies");

        return sortedList;
    }, new ExecutionDataflowBlockOptions()
        {
            MaxDegreeOfParallelism = 1
        }
    );
    var printTopTen = new ActionBlock <List<KeyValuePair <string ,
                                                          ulong >>>(
        input =>
    {
        for (int i=0;i<10;i++)
        {
            Console.WriteLine(
            $"{input[i].Key} - {input[i].Value}");
        }
        getFilenames.Complete();
    });

    // Присоединение блоков
    getFilenames.LinkTo(getFileContents);
    getFileContents.LinkTo(analyzeContents);
    analyzeContents.LinkTo(eliminateIgnoredWords);
    eliminateIgnoredWords.LinkTo(batch);
    batch.LinkTo(combineFrequencies);
    combineFrequencies.LinkTo(printTopTen);

    completionTask = getFilenames.Completion;

    return getFilenames;
}
```

Чтобы воспользоваться конвейером (рис. 4.2), нужно просто извлечь ссылку на блок и отправить в него сообщение:

```
Task completionTask;
ITargetBlock <string > startBlock =
    CreateTextProcessingPipeline(args[0], out completionTask);

startBlock.Post(args[0]);

completionTask.Wait();
```

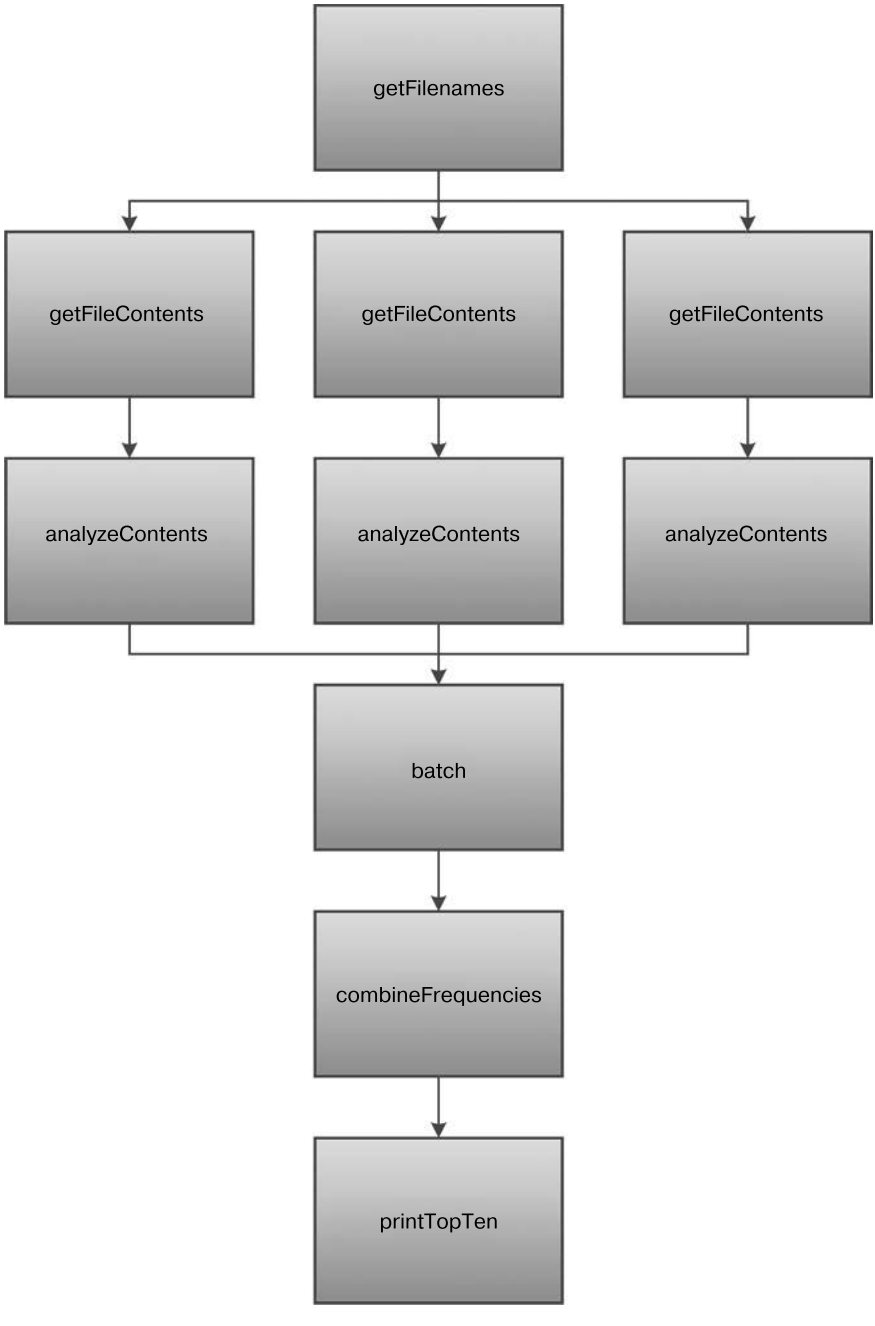

Рис. 4.2. Графическое представление конвейера

Данные, выведенные в консоль, показывают, что часть блоков выполняется одновременно, в то время как другие блоки выполняются последовательно после завершения выполнения предыдущих блоков, и что разные типы блоков могут чередоваться:

```
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: analyzeContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: analyzeContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: getFileContents
Begin: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: analyzeContents
End: analyzeContents
Begin: combiningFrequencies
End: combineFrequencies
I - 7693
you - 5188
in - 4266
My - 4237
that - 4158
is - 3790
NOT - 3209
FOR - 3201
d - 3117
IT - 2947
```

Параллельно выполняемые циклы

В одном из примеров в разделе «Библиотека распараллеливания задач» показана схема, используемая так часто, что для параллельного выполнения циклов существует специальный API:

```
const int MaxValue = 1000;
long sum = 0;
Parallel.For(0, MaxValue , (i) =>
    {
        Interlocked.Add(ref sum, (long)Math.Sqrt(i));
    });
```

Кроме того, имеется версия `foreach` для обработки обобщенных коллекций `IEnumerable<T>`:

```
var urls = new List<string >
{
    @"http://www.microsoft.com",
```

```
    @"http://www.bing.com",
    @"http://msdn.microsoft.com"
};
var results = new ConcurrentDictionary <string ,string >();
var client = new System.Net.WebClient();

Parallel.ForEach(urls, url => results[url] =
        client.DownloadString(url));
```

Если нужно прервать обработку цикла, делегату цикла можно передать объект `ParallelLoopState`. Есть два варианта остановки цикла:

- `Break` — циклу предписывается не выполнять никаких итераций, которые будут последовательно больше текущей итерации. Если для цикла `Parallel.For` вызвать `ParallelLoopState.Break` на `i`-й итерации, то любым итерациям меньше `i` запуск будет по-прежнему разрешен, но любым итерациям больше `i` — запрещен. Этот же механизм работает в отношении цикла `Parallel.ForEach`, но каждому элементу присваивается индекс, который с точки зрения программы может иметь произвольное значение. Обратите внимание на то, что, если требует логика кода цикла, вызов `Break` возможно осуществить на нескольких итерациях;
- `Stop` — циклу предписывается больше не выполнять никаких итераций.

В следующем примере `Break` применяется для остановки цикла в произвольном месте:

```
Parallel.ForEach(urls, (url, loopState) =>
{
    if (url.Contains("bing"))
    {
        loopState.Break();
    }
    results[url] = client.DownloadString(url);
});
```

Применяя параллельно выполняемые циклы, следует добиться того, чтобы каждая итерация цикла работала по возможности только с локальным состоянием. Если цикл тратит все свое время, будучи заблокированным на синхронном доступе к переменной, совместно используемой всеми потоками, то лишиться всех преимуществ параллельных вычислений очень легко. Если необходимо применить совместно используемое состояние, следует гарантировать, что время работы, выполняемой в ходе итерации, будет существенно больше периода ожидания при синхронизации доступа к этому состоянию.

Еще одной проблемой параллельно выполняемых циклов является вызов делегата для каждой итерации, на что может впустую затрачиваться время, если производимая работа не превышает затрат на вызов делегата или метода (см. главу 5).

Обе проблемы могут быть решены за счет использования класса `Partitioner` (разбивающий на части), преобразующего диапазон в набор объектов `Tuple` (кор-

теж), в каждом из которых содержится описание поддиапазона, через который проходит итерация в исходной коллекции.

В следующем примере показано, насколько сильным может оказаться негативное влияние синхронизации на эффективность параллельных вычислений:

```
static void Main(string[] args)
{
    Stopwatch watch = new Stopwatch();
    const int MaxValue = 1000000000;

    long sum = 0;

    // Простой цикл For
    watch.Restart();
    sum = 0;
    Parallel.For(0, MaxValue , (i) =>
    {
        Interlocked.Add(ref sum, (long)Math.Sqrt(i));
    });
    watch.Stop();
    Console.WriteLine("Parallel.For: {0}", watch.Elapsed);
    // Разбитый цикл For
    var partitioner = Partitioner.Create(0, MaxValue);
    watch.Restart();
    sum = 0;
    Parallel.ForEach(partitioner ,
         (range) =>
        {
            long partialSum = 0;
            for (int i = range.Item1; i < range.Item2; i++)
            {
                partialSum += (long)Math.Sqrt(i);
            }
            Interlocked.Add(ref sum, partialSum);
        });
    watch.Stop();
    Console.WriteLine("Partitioned Parallel.For: {0}",
                        watch.Elapsed);
}
```

Этот код можно найти в учебном проекте ParallelLoops. При запуске на моей машине он выдал следующую информацию:

```
Parallel.For: 00:01:47.5650016
Partitioned Parallel.For: 00:00:00.8942916
```

Показанная ранее схема разбиения статическая в том смысле, что, как только разделы определены, делегат выполняется для каждого диапазона и, если выполнение одного из них завершится раньше, не будет делаться никаких попыток нового разбиения, чтобы снабдить работой другой процессор. Статические разделы можно создавать для любых коллекций `IEnumerable<T>` и без указания диапазона, но тогда делегат будет вызываться для каждого элемента, а не для поддиапазона.

Эту проблему можно обойти, создав свой вариант `Partitioner`, который может оказаться весьма сложным. Дополнительные сведения и очень большие примеры можно найти в статье Стивена Туба (Stephen Toub) *Custom Parallel Partitioning With .NET 4*.

Советы по повышению производительности

Избегайте использования блокировок

Для достижения наивысшей производительности нужно, чтобы ваша программа никогда не тратила впустую один ресурс, дожидаясь доступности другого. Чаще всего такие обстоятельства принимают форму блокирования текущего потока в ожидании завершения некой операции ввода-вывода. В данной ситуации возможен один из двух результатов.

1. Поток окажется заблокирован в состоянии ожидания, не будет планироваться на выполнение и вызовет запуск другого потока. Это может привести к созданию нового потока для выполнения ожидающих рабочих элементов или задач, если все текущие потоки уже задействованы или заблокированы.
2. Поток дойдет до объекта синхронизации, который в ожидании сигнала может войти в пустой цикл на несколько миллисекунд. Если он не будет получен вовремя, поток войдет в состояние, описанное в п. 1.

В обоих случаях происходит ненужное увеличение пула потоков и, возможно, ресурс центрального процессора впустую затрачивается на циклы в блокировках. Попадать в такую ситуацию нежелательно.

Блокировка и другие виды непосредственной синхронизации потоков относятся к легко обнаруживаемым явным блокирующим вызовам. Но иногда бывает непонятно, что к блокировке могут приводить и другие вызовы методов. С ними часто связаны различного рода операции ввода-вывода, поэтому нужно убедиться в том, что любые взаимодействия с сетью, файловой системой, базами данных или любыми другими сервисами с высокой степенью задержек выполняются в асинхронном режиме. К счастью, среда .NET существенно облегчает эту задачу, позволяя задействовать объекты `Task`.

При использовании любого API ввода-вывода — для сети, файловой системы, базы данных или чего-то еще — следует убедиться, что он возвращает `Task`, в противном случае он будет весьма подозрителен и, вполне вероятно, станет выполнять блокировку при вводе-выводе. Следует заметить, что устаревшие API будут возвращать `IASyncResult` и обычно начинаются с префикса `Begin-`. Как бы то ни было, найдите API, возвращающий вместо этого `Task`, или воспользуйтесь методом `Task.FactoryFromAsync`, чтобы поместить эти методы внутрь объектов `Task`, чтобы добиться согласования с собственным интерфейсом программирования.

Метод `Task.Wait` следует применять лишь изредка или вовсе отказаться от него. Вместо этого воспользуйтесь продолжениями. Ожидание в отношении `Task` является разновидностью более масштабной проблемы блокирующих вызовов.

Избегайте конвоев при блокировке и диспетчеризации

Общим ограничением для приложений пользовательских интерфейсов является возможность изменения состояния пользовательского интерфейса только из одного потока, который зачастую называют потоком диспетчера. Это означает, что даже если нужно, чтобы некую фоновую работу выполняли сразу несколько потоков, то, чтобы сообщить о любых видимых изменениях, они должны переводить свою работу обратно в поток пользовательского интерфейса.

В платформе WPF это имеет примерно следующий вид:

```
// В фоновом рабочем потоке
Application.Current.Dispatcher.BeginInvoke(
  DispatcherPriority.Background ,
  () => this.statusBar.Value = "Complete");
```

Если в приложении пользовательского интерфейса есть постоянно занимающееся этим множество потоков, можно в итоге начать блокировать пользовательский интерфейс из-за множественных обновлений. В таких случаях понадобится либо переосмыслить сами изменения пользовательского интерфейса, либо ввести уровень обобщения, где можно было бы собрать в пакет несколько обновлений и разом обновить все части пользовательского интерфейса.

Это похоже на проблему конвоя при блокировке, где соперничество за один заблокированный ресурс настолько высоко, что на пустые циклы или ожидание освобождения блокировки затрачивается больше времени, чем на выполнение работы. В обоих случаях нужно пересмотреть стратегию асинхронной работы, чтобы снизить число конфликтов на отдельно взятом ресурсе.

Использование объектов Tasks для неблокирующего ввода-вывода

В .NET 4.5 к классу `Stream` были добавлены `–Async`-методы, поэтому весь обмен данными на `Stream`-основе может довольно легко стать полностью асинхронным. Рассмотрим простой пример:

```
int chunkSize = 4096;
var buffer = new byte[chunkSize];

var fileStream = new FileStream(filename , FileMode.Open,
     FileAccess.Read, FileShare.Read, chunkSize , useAsync: true);
var task = fileStream.ReadAsync(buffer , 0, buffer.Length);
task.ContinueWith((readTask) =>
  {
    int amountRead = readTask.Result;
    fileStream.Dispose();
    Console.WriteLine("Async(Simple) read {0} bytes", amountRead);
});
```

Больше уже не получится воспользоваться синтаксисом `using` для очистки `IDisposable`-объектов, таких как `Stream`-объекты. Вместо этого нужно передать такие объекты методу продолжения, чтобы обеспечить их очистку в каждом из путей исполнения.

На самом деле этот пример требует дополнения. В реальном сценарии для получения полноценного содержимого зачастую приходится выполнять множественные считывания потока. Подобное может произойти, если файлы больше предоставленного вами буфера или если работа ведется с сетевыми потоками, а не с файлами. В таком случае байты еще даже не дошли до вашей машины. Чтобы справиться с подобной ситуацией в асинхронном режиме, следует продолжать считывание потока, пока от него не поступит сообщение о том, что данных больше не осталось.

Еще одной шероховатостью является то, что теперь вам нужны два уровня `Task`-объектов. Верхний уровень предназначен для общего чтения — той части, которой интересуется вызывающая программа. Нижестоящий уровень представляет собой набор `Task`-объектов для каждой отдельной порции фрагментированного считывания.

Подумаем, для чего это нужно. Первое асинхронное считывание возвратит `Task`. Если это вернуть вызывающему коду для ожидания или продолжения, то он продолжит выполнение после первого чтения. А на самом деле нужно было бы, чтобы он продолжил выполнение после завершения считываний. Это означает, что эту первую задачу `Task` вызывающему коду возвращать нельзя. Нужна фиктивная задача `Task`, которая завершится, как только будут выполнены все считывания.

Чтобы все это проделать, следует воспользоваться классом `TaskCompletionSource<T>`, способным создать фиктивную задачу `Task`, предназначенную для возвращения. Когда серия из асинхронных считываний завершится, вызовите в отношении `TaskCompletionSource` метод `TrySetResult`, что заставит объект этого класса запустить связанный с ним ожидающий код или код продолжения.

В следующем примере показано расширение предыдущего примера и продемонстрировано использование `TaskCompletionSource`:

```
private static Task<int> AsynchronousRead(string filename)
{
    int chunkSize = 4096;
    var buffer = new byte[chunkSize];
    var tcs = new TaskCompletionSource <int >();

    var fileContents = new MemoryStream();
    var fileStream = new FileStream(filename , FileMode.Open,
        FileAccess.Read, FileShare.Read, chunkSize , useAsync: true);
    fileContents.Capacity += chunkSize;

    var task = fileStream.ReadAsync(buffer , 0, buffer.Length);
    task.ContinueWith(
        readTask =>
        ContinueRead(readTask , fileStream ,
                     fileContents , buffer , tcs));
```

```
    return tcs.Task;
}

private static void ContinueRead(Task<int> task,
                FileStream stream ,
                MemoryStream fileContents ,
                byte[] buffer ,
                TaskCompletionSource <int> tcs)
{
    if (task.IsCompleted)
    {
        int bytesRead = task.Result;
        fileContents.Write(buffer , 0, bytesRead);
        if (bytesRead > 0)
        {
            // Не все байты считаны, поэтому
            // выполнение еще одного асинхронного вызова
            var newTask = stream.ReadAsync(buffer , 0, buffer.Length);
            newTask.ContinueWith(
                readTask =>
                ContinueRead(readTask , stream ,
                            fileContents , buffer , tcs));
        }
        else
        {
            // Все сделано, освобождение ресурсов
            // и завершение задачи высшего уровня.
            tcs.TrySetResult((int)fileContents.Length);
            stream.Dispose();
            fileContents.Dispose();
        }
    }
}
```

Заметьте, что ни от кого не требуется дожидаться установки результата для объекта `Task` — отсутствие какой-либо упорядоченной зависимости является весьма важным требованием асинхронного программирования.

Адаптируйте модель асинхронного программирования к задачам

В асинхронных методах старого стиля, присутствующих в среде .NET Framework, есть методы с префиксами `Begin-` и `End-`. Они вполне работоспособны, и их можно без особого труда заключить в `Task`, чтобы получить согласованный интерфейс, как в следующем примере, взятом из учебного проекта TaskFromAsync:

```
const int TotalLength = 1024;
const int ReadSize = TotalLength / 4;

static Task<string > GetStringFromFileBetter(string path)
{
    var buffer = new byte[TotalLength];
```

```
    var stream = new FileStream(
        path,
        FileMode.Open,
        FileAccess.Read,
        FileShare.None,
        buffer.Length ,
        FileOptions.DeleteOnClose | FileOptions.Asynchronous);

    var task = Task<int>.Factory.FromAsync(
        stream.BeginRead ,
        stream.EndRead ,
        buffer ,
        0,
        ReadSize , null);

    var tcs = new TaskCompletionSource <string >();

    task.ContinueWith(readTask => OnReadBuffer(readTask ,
                    stream , buffer , 0, tcs));
    return tcs.Task;
}

static void OnReadBuffer(Task<int> readTask ,
            Stream stream ,
            byte[] buffer ,
            int offset ,
            TaskCompletionSource <string > tcs)
{
    int bytesRead = readTask.Result;
    if (bytesRead > 0)
    {
        var task = Task<int>.Factory.FromAsync(
            stream.BeginRead ,
            stream.EndRead ,
            buffer ,
            offset + bytesRead ,
            Math.Min(buffer.Length - (offset + bytesRead), ReadSize),
            null);

        task.ContinueWith(
            callbackTask => OnReadBuffer(
                callbackTask ,
                stream ,
                buffer ,
                offset + bytesRead ,
                tcs));
    }
    else
    {
        tcs.TrySetResult(Encoding.UTF8.GetString(buffer , 0, offset));
    }
}
```

Метод `FromAsync` принимает два аргумента в виде принадлежащих потоку методов `BeginRead` и `EndRead`, а также целевой буфер для хранения данных. Он выполнит методы и после завершения работы метода `EndRead` вызовет продолжение, возвращая управление вашему коду, который в данном примере закрывает поток и возвращает преобразованное содержимое файла.

Используйте эффективный ввод-вывод

То, что ко всем вызовам ввода-вывода применяются приемы асинхронного программирования, еще не означает, что от выполняемого ввода-вывода вы получаете максимально возможную отдачу. Устройства ввода-вывода обладают разными возможностями, скоростями и особенностями, в силу чего приемы программирования зачастую приходится приспосабливать под них.

В показанном ранее примере для чтения и записи на диск был выбран буфер 16 Кбайт. Подходящее ли это значение? Если принять во внимание размеры буферов на жестких дисках и скорость работы твердотельных накопительных устройств, возможно, и нет. Чтобы определить меру эффективного деления данных ввода-вывода на части, нужно поэкспериментировать. Чем меньше буферы, тем больше издержки. Чем буферы больше, тем дольше придется дожидаться результатов, позволяющих приступить к работе. Правила, применяемые к дискам, не подойдут для сетевых устройств, и наоборот.

Но важнее всего то, что для наиболее эффективного использования ввода-вывода программа нуждается в особой структуре. Если часть вашей программы когда-либо оказывается заблокированной в ожидании завершения ввода-вывода, то время ожидания не расходуется на обработку полезных данных с помощью ЦП или как минимум приводит к пустым тратам пула потоков. В ожидании завершения операций ввода-вывода нужно проделывать как можно больше другой работы.

Кроме того, следует заметить, что асинхронный ввод-вывод и выполнение синхронного ввода-вывода на другом потоке в корне отличаются друг от друга. В первом случае управление фактически передается операционной системе и оборудованию и нигде никакой код в системе не блокируется, пока управление не будет возвращено. Если же выполняется синхронный ввод-вывод в другом потоке, происходит просто блокировка потока, который мог бы проделывать другую работу, по-прежнему ожидая возвращения от операционной системы. Это может быть допустимо в ситуации, не требующей высокой производительности (например, при выполнении фонового ввода-вывода в программе пользовательского интерфейса вместо основного потока), но не рекомендуется.

Иными словами, код следующего примера нерационален и противоречит целям асинхронного ввода-вывода:

```
Task.Run( ()=>
{
    using (var inputStream = File.OpenRead(filename))
    {
        byte[] buffer = new byte[16384];
```

```
        // Вызов метода синхронного ввода-вывода –
        // это приведет к блокировке потока
        var input = inputStream.Read(buffer , 0, buffer.Length);
        ...
    }
});
```

Дополнительные советы по эффективному вводу-выводу, осуществляемому с помощью API среды .NET Framework применительно к доступу к дискам или сетям, даны в главе 6.

async и await

В .NET 4.5 появилось два новых ключевых слова, позволяющих во многих ситуациях упростить код: `async` и `await`. При совместном использовании они превращают ваш TPL-код в нечто похожее на простой линейный синхронный код. Но «под капотом» фактически применяются `Task`-объекты и продолжения.

Следующий пример взят из учебного проекта AsyncAwait:

```
static Regex regex = new Regex("<title >(.*)</title >",
                               RegexOptions.Compiled);

private static async Task<string > GetWebPageTitle(string url)
{
    System.Net.Http.HttpClient client =
        new System.Net.Http.HttpClient();
    Task<string > task = client.GetStringAsync(url);

    // теперь нам нужен результат, поэтому используем await
    string contents = await task;
    Match match = regex.Match(contents);
    if (match.Success)
    {
        return match.Groups[1].Captures[0].Value;
    }
    return string.Empty;
}
```

Чтобы увидеть, в чем заключается эффективность этого синтаксиса, рассмотрим пример посложнее, где выполняются одновременное считывание из одного файла и запись в другой файл с попутным сжатием данных на выходе. Для выполнения этой задачи было бы не слишком трудно воспользоваться `Task`-объектами напрямую, но посмотрим, как просто все будет выглядеть, когда применяется синтаксис `async-await`. Следующий код взят из учебного проекта CompressFiles.

Сначала для сравнения посмотрим на синхронную версию:

```
private static void SyncCompress(IEnumerable <string > fileList)
{
    byte[] buffer = new byte[16384];
```

```
    foreach (var file in fileList)
    {
        using (var inputStream = File.OpenRead(file))
        using (var outputStream = File.OpenWrite(file+".compressed"))
        using (var compressStream = new GZipStream(outputStream ,
                            CompressionMode.Compress))
        {
            int read = 0;
            while ((read = inputStream.Read(buffer ,
                                            0,
                                            buffer.Length)) > 0)
            {
                compressStream.Write(buffer , 0, read);
            }
        }
    }
}
```

Чтобы превратить этот код в асинхронный, нужно лишь добавить ключевые слова `async` и `await` и изменить методы `Read` и `Write` на `ReadAsync` и `WriteAsync` соответственно:

```
private static async Task AsyncCompress(
    IEnumerable <string > fileList)
{
    byte[] buffer = new byte[16384];
    foreach (var file in fileList)
    {
        using (var inputStream = File.OpenRead(file))
        using (var outputStream = File.OpenWrite(file+".compressed"))
        using (var compressStream =
                new GZipStream(outputStream,
                               CompressionMode.Compress))
        {
            int read = 0;
            while ((read = await inputStream.ReadAsync(buffer, 0,
                                    buffer.Length)) > 0)
            {
                await compressStream.WriteAsync(buffer, 0, read);
            }
        }
    }
}
```

Похоже на то, что этот код будет заблокирован в ожидании результата чтения из файла, но не стоит путать `await` со словом wait («ожидать») — при всей схожести они заведомо разные. Все, что предшествует ключевому слову `await`, происходит в вызывающем потоке. А все, что следует за этим словом, является продолжением. В вашем коде `await` может применяться в любом методе, возвращающем `Task<T>`, при условии, что этот метод помечен ключевым словом `async`. При использовании данных ключевых слов компилятор займется нелегкой работой

по преобразованию вашего кода в структуру, подобную той, что задействовалась в прежних примерах с TPL.

Важно отметить, что ключевое слово `await` использовано в инструкции `using`. Шаблон проектирования Dispose корректно работает с `async` и `await`, что существенно упрощает код.

Пара ключевых слов `async-await` может заметно упростить код, но в некоторых ситуациях, основанных на использовании `Task`, применить ее невозможно. Например, если завершение `Task` носит недетерминированный характер или у вас должно быть несколько уровней `Task`-объектов и задействуются объекты `TaskCompletionSource<T>`, то пара `async-await` может не вписаться в эту схему.

ИСТОРИЯ

С проблемой детерминированности мне пришлось столкнуться при реализации функции повторного запуска в качестве надстройки над имеющейся в .NET функцией HTTP-клиента, которая целиком поддерживает использование Task. Работа началась с простого класса-оболочки HTTP-клиента, и я изначально воспользовался async-await, поскольку при этом код упрощался. Но, когда пришло время реализовать функцию повторного запуска, я сразу понял, что застрял, поскольку упустил контроль над моментом завершения задач. В своей реализации я хотел отправлять повторный запрос, пока не вышло время ожидания ответа на первый запрос. Я хотел возвратить вызывающему коду результат того запроса, который завершится первым. К сожалению, пара async-await не справляется с недетерминированной ситуацией произвольного выбора из нескольких дочерних ожидаемых Task-объектов. Один из способов разрешить данную ситуацию — использовать ранее рассмотренный метод ContinueOnAny. В качестве альтернативы можно применить метод TaskCompletionSource, чтобы самому проконтролировать завершение задачи Task верхнего уровня. В рассматриваемой ситуации я выбрал последний вариант.

О структуре программы

Вспомните важную интересную деталь, упомянутую в предыдущем разделе: все инструкции `await` должны находиться в методах с пометкой `async`, следовательно, такие методы должны возвращать `Task`-объекты. Технически при условии непосредственного использования `Task`-объектов ограничений такого рода не существует, но сохраняется тот же принцип написания кода. Применение асинхронного программирования похоже на полезный вирус, заражающий вашу программу на всех уровнях. Начавшись в одной части, он будет идти вверх по уровням вызовов функций как можно выше.

Разумеется, используя `Task`-объекты напрямую, можно создать следующий (весьма неудачный) пример:

```
Task<string > task = Task<string >.Run(()=> { ... });
task.Wait();
```

Но если выполняется чуть менее тривиальная многопоточная работа, такой подход полностью разрушает масштабируемость вашего приложения. Да, конечно, можно вставить какую-то работу между созданием задачи и вызовом ожидания `Wait`, но это нивелирует смысл применения данного механизма. Отказываться от ожидания для объектов `Task` нужно практически всегда, так как это приводит к пустому использованию потока, который при иных обстоятельствах мог бы выполнять полезную работу. Пустое выполнение потока может вызвать увеличение количества переключений контекста и увеличить издержки от использования пула потоков, поскольку для того, чтобы справиться с доступными рабочими элементами, потребуется больше потоков.

Если довести идею полного отказа от ожиданий до логического завершения, можно прийти к пониманию: весьма возможно и даже вполне вероятно, что практически весь код вашей программы окажется в каком-либо продолжении. Если задуматься, в этом есть интуитивно понятный смысл. Программа пользовательского интерфейса практически ничего не делает, пока пользователь не щелкнет кнопкой мыши или не наберет что-нибудь, воспользовавшись клавиатурой — она реагирует на ввод посредством механизма событий. Серверная программа действует аналогично, но вместо мыши или клавиатуры ввод-вывод осуществляется посредством сети или файловой системы.

В результате высокопроизводительное приложение может представляться разрозненным, поскольку логика программы будет разбита вами на основе границ ввода-вывода. Чем раньше вы это спланируете, тем лучше будет. Очень важно установить небольшое число стандартных шаблонов, применяемых в большинстве или даже во всех ваших программах, например таких.

- Определите, где находятся задачи `Task` и методы продолжения. Используется ли отдельный метод или лямбда-выражение? Зависит ли это от его размера?
- Если для продолжений задействуются методы, установите стандартный префикс, например, `OnMyTaskEnd`.
- Стандартизируйте обработку ошибок. Есть ли у вас одно продолжение, обрабатывающее все ошибки, отмены и обычные завершения? Или же используются отдельные методы для обработки каждого из этих исходов и `TaskContinuationOptions` — для их выборочного выполнения?
- Решите, чем именно нужно воспользоваться — парой ключевых слов `async-await` или непосредственно `Task`-объектами.
- Если приходится вызывать старомодные асинхронные методы `Begin.../End...`, заключите их в `Task`-объекты, чтобы, как сказано ранее, добиться стандартизации обработки.
- Не чувствуйте себя обязанными использовать все функции TPL. Некоторые из них в большинстве ситуаций применять не рекомендуется (например, задачи `AttachedToParent`). Примите за стандарт минимальный набор функций, с которым вы в состоянии справиться.

Асинхронное программирование, в отличие от стандартного, синхронного, с линейными процедурами и вызовами, — это совершенно иная разновидность программирования. Оно требует развития нового мышления, непримиримо относящегося к ожиданиям и блокирующим вызовам и рассматривающего структуру программы в понятиях независимых частей, лишь временами объединяемых для синхронизации. Выработка мышления такого типа может занять некоторое время. Начните с малого и постепенно сформируйте у себя такое мышление.

Правильно используйте таймеры

Чтобы спланировать выполнение метода через определенный промежуток времени и, возможно, через равномерные промежутки времени после этого, воспользуйтесь классом `System.Threading.Timer`. Не следует применять механизмы вроде `Thread.Sleep`, блокирующие поток на определенное время, хотя, как будет показано далее, в некоторых ситуациях этот прием может пригодиться.

В следующем примере показано, как задействовать таймер путем указания метода обратного вызова и передачи ему двух значений времени ожидания. Первое значение представляет собой время до первого запуска таймера, а второе — частоту повторений после запуска таймера. Для обоих значений можно указать бесконечность `Timeout.Infinite`, выражающуюся числом –1. Здесь задается однократный запуск таймера после 15 мс:

```
private System.Threading.Timer timer;

public void Start()
{
    this.timer = new Timer(TimerCallback,
                           null,
                           15,
                           Timeout.Infinite);
}

private void TimerCallback(object state)
{
    // Выполнение полезной работы
}
```

Не создавайте слишком много таймеров. Все объекты `Timer` обслуживаются одним потоком из пула потоков. Слишком большое количество экземпляров `Timer` вызовет задержки при выполнении их обратных вызовов. Когда придет время, поток таймера спланирует рабочий элемент в пуле потоков, и он будет принят следующим доступным потоком. Если имеется большое количество задач, выстроенных в очередь рабочих элементов, или высока загруженность центрального процессора, то обратные вызовы объектов `Timer` будут неточны. Фактически гарантируется, что они никогда не будут точнее счетчика тиков таймера операционной системы, установленного по умолчанию на 15,625 мс. Это то же самое значение, которое определяет продолжительность кванта потока. Установка меньшего времени ожи-

дания не приведет к желаемому результату. Есть несколько вариантов установки точности выше 15 мс.

1. Уменьшить разрешение таймера операционной системы. Этот шаг может увеличить загруженность центрального процессора и сильно повлиять на срок службы батареи, но в ряде ситуаций оказывается вполне оправданным. Следует заметить, что подобное изменение может иметь далеко идущие последствия наподобие возрастания частоты переключений контекста, повышения издержек системы и ухудшения производительности в других областях.
2. Использовать цикл с таймером с высоким разрешением (см. главу 6) для измерения прошедшего времени. Это также увеличит затраты времени центрального процессора и расход энергии, но будет носить более локализованный характер.
3. Вызвать `Thread.Sleep`. Поток приостановится, и не будет гарантировано возобновление его работы в желаемое время. В системах с высокой степенью загруженности вполне возможно, что поток будет переключен из контекста и вы получите его назад значительно позже истечения желаемого интервала.

Когда используется объект `Timer`, нужно помнить о классическом состоянии гонки (race condition). Рассмотрим следующий код:

```
private System.Threading.Timer timer;

public void Start()
{
    this.timer = new Timer(TimerCallback,
                           null,
                           15,
                           Timeout.Infinite);
}

private void TimerCallback(object state)
{
   // Выполнение полезной работы
   this.timer.Dispose();
}
```

Этот код настраивает объект `Timer` на выполнение функции обратного вызова за 15 мс. Обратный вызов просто удаляет объект таймера, завершив с ним работу. Также код вполне может выдать в `TimerCallback` исключение `NullReferenceException`. Причина в том, что с высокой степенью вероятности функция обратного вызова будет выполнена еще до того, как в методе `Start` полю `this.timer` будет присвоен объект `Timer`. К счастью, это легко исправить:

```
this.timer = new Timer(TimerCallback,
                       null,
                       Timeout.Infinite ,
                       Timeout.Infinite);
this.timer.Change(15, Timeout.Infinite);
```

ПРИМЕЧАНИЕ

В проекте, над которым я однажды работал, для отслеживания задержек для каждого узла в диаграмме зависимостей назначался объект System.Threading.Timer. Пока диаграммы были небольшими, все шло хорошо. Как только они стали достигать размеров в несколько тысяч узлов, сотни которых были запущены одновременно, стали наблюдаться весьма существенные конфликты блокировок и нестабильное поведение в отношении задержек. Вся конструкция стала неприемлемой и весьма затратной. Вместо нее мы разработали собственную систему отслеживания задержек, основанную на опросе через определенные интервалы времени, то есть, по сути, свели конструкцию к использованию одного таймера, проверяющего все запущенные узлы с приемлемой частотой.

Если целевой метод нужно запустить только один раз, то есть без повторяющихся сценариев, можно воспользоваться `Task.Delay`, чтобы спланировать однократный запуск делегата в будущем:

```
var task = Task.Delay(1000).ContinueWith(_ =>
    {
        Console.WriteLine("After delay");
    });
```

Подберите подходящий размер пула потоков

Со временем пул потоков настраивается самостоятельно, но в самом начале у него нет истории и запускаться он будет в исходном состоянии. Если ваш программный продукт исключительно асинхронный и значительно задействует центральный процессор, он может пострадать от непомерно высоких затрат на начальный запуск в ожидании создания и доступности еще большего количества потоков. Быстрее достичь устойчивого состояния поможет подстройка пусковых параметров так, чтобы с момента запуска приложения в вашем распоряжении имелось определенное число готовых потоков:

```
const int MinWorkerThreads = 25;
const int MinIoThreads = 25;
ThreadPool.SetMinThreads(MinWorkerThreads, MinIoThreads);
```

Здесь следует действовать осторожно. При использовании `Task`-объектов их диспетчеризация будет осуществляться на основе числа доступных для этого потоков. При слишком большом их количестве `Task`-объекты могут подвергнуться излишней диспетчеризации, что как минимум приведет к снижению эффективности применения центрального процессора из-за более частого переключения контекста. Если же рабочая нагрузка будет не столь высока, пул потоков сможет перейти к использованию алгоритма, способного уменьшить количество потоков, доведя его до числа ниже заданного.

Можно также установить максимальное их количество, воспользовавшись методом `SetMaxThreads`, но данный прием подвержен аналогичным рискам.

Чтобы выяснить нужное количество потоков, оставьте этот параметр в покое и проанализируйте свое приложение в устойчивом состоянии, воспользовавшись методами `ThreadPool.GetMaxThreads` и `ThreadPool.GetMinThreads` или счетчиками производительности, которые покажут количество потоков, задействованных в процессе.

Не прерывайте потоки

Прерывание работы потоков без согласования с работой других потоков — довольно опасная процедура. Потоки должны очищать себя сами, и вызов для них метода `Abort` не позволяет закрыть их без негативных последствий. При уничтожении потока части приложения оказываются в неопределенном состоянии. Было бы лучше выполнить аварийный выход из программы, но в идеале нужен чистый перезапуск.

Для безопасного завершения работы потока нужно воспользоваться каким-то совместно используемым состоянием, и сама функция потока должна проверить это состояние, чтобы определить, когда должна завершиться его работа. Безопасность должна достигаться за счет согласованности.

Вообще, стоит всегда задействовать `Task`-объекты — API для прерывания задачи `Task` не предоставляется. Чтобы получить возможность согласованно завершить работу потока, нужно, как отмечалось ранее, воспользоваться маркером отмены `CancellationToken`.

Не меняйте приоритет потоков

В общем, изменение приоритета потоков — затея крайне неудачная. В Windows диспетчеризация потоков выполняется в соответствии с уровнями их приоритетов. Если высокоприоритетные потоки всегда готовы к запуску, то низкоприоритетные будут обойдены вниманием и довольно редко станут получать шанс на запуск. Повышая приоритет потока, вы говорите, что его работа должна иметь приоритет над всей остальной работой, включая другие процессы. Это небезопасно для стабильной системы.

Лучше понизить приоритет потока, если в нем выполняется что-то, что может подождать завершения выполнения задач обычной приоритетности. Одной из веских причин понижения приоритета потока может быть обнаружение вышедшего из-под контроля потока, выполняющего бесконечный цикл. Безопасно прервать работу потока невозможно, поэтому единственный способ вернуть данный поток и ресурсы процессора — перезапуск процесса. До тех пор пока не появится возможность закрыть поток и сделать это чисто, понижение приоритета вышедшего из-под контроля потока будет вполне разумным средством минимизации последствий. Следует заметить, что даже потокам с пониженным приоритетом все же со временем гарантируется запуск: чем дольше они будут обделены запусками, тем

выше будет устанавливаемый системой Windows их динамический приоритет. Исключение составляет приоритет простоя `THREAD_-PRIORITY_IDLE`, при котором операционная система спланирует выполнение потока только в том случае, когда ей в буквальном смысле будет больше нечего запускать.

Могут найтись и вполне оправданные причины для повышения приоритета потока, например необходимость быстро реагировать на редкие ситуации. Но пользоваться такими приемами следует весьма осмотрительно. Диспетчеризация потоков в Windows осуществляется независимо от процессов, которым они принадлежат, поэтому высокоприоритетный поток из вашего процесса будет запускаться в ущерб не только другим вашим потокам, но и всем потокам из других приложений, запущенных в вашей системе.

Если применяется пул потоков, то любые изменения приоритетов сбрасываются при каждом возвращении потока в пул. Если при использовании библиотеки Task Parallel продолжить управлять базовыми потоками, следует иметь в виду, что в одном и том же потоке до его возвращения в пул могут запускаться несколько задач.

Синхронизация потоков и блокировки

Как только разговор заходит о нескольких потоках, возникает необходимость их синхронизации. Синхронизация заключается в обеспечении доступа только одного потока к совместно используемому состоянию, например к полю класса. Обычно синхронизация потоков выполняется с помощью таких объектов синхронизации, как `Monitor`, `Semaphore`, `ManualResetEvent` и т. д. Иногда их неформально называют блокировками, а процесс синхронизации в конкретном потоке — блокировкой.

Одна из основополагающих истин, касающихся блокировок, заключается в следующем: они никогда не повышают производительность. В лучшем случае — при наличии хорошо реализованного примитива синхронизации и отсутствии конкуренции — блокировка может быть нейтральной. Она приводит к остановке выполнения полезной работы другими потоками и к тому, что время центрального процессора расходуется впустую, увеличивает время контекстного переключения и вызывает другие негативные последствия. С этим приходится мириться из-за того, что правильность намного важнее простой производительности. Быстро ли вычислен неверный результат, не играет никакой роли!

Прежде чем приступить к решению проблемы использования аппарата блокировки, рассмотрим наиболее фундаментальные принципы.

Нужно ли вообще заботиться о производительности?

Сперва обоснуйте необходимость повышения производительности. Это возвращает нас к принципам, рассмотренным в главе 1. Не для всего кода вашего приложения производительность одинаково важна. Не весь код должен подвергаться оптимизации *n*-й степени. Как правило, все начинается с «внутреннего цикла» — кода,

выполняемого наиболее часто или наиболее критического для производительности, — и распространяется во все стороны, пока затраты не превысят получаемую выгоду. В коде есть множество областей, гораздо менее важных с точки зрения производительности. В такой ситуации, если нужна блокировка, спокойно применяйте ее.

А теперь следует проявить осмотрительность. Если ваш некритический фрагмент кода выполняется в потоке из пула потоков и вы надолго его блокируете, пул потоков может начать вставлять большее количество потоков, чтобы справиться с другими запросами. Если один-два потока делают это время от времени, ничего страшного. Но если подобные вещи проделывает множество потоков, может возникнуть проблема, поскольку из-за этого без пользы расходуются ресурсы, которые должны выполнять реальную работу. Неосмотрительность при запуске программы со значительной постоянной нагрузкой может вызвать негативное воздействие на систему даже из тех ее частей, для которых неважна высокая производительность, из-за лишних переключений контекста или необоснованного задействования пула потоков. Как и во всех других случаях, для оценки ситуации нужно выполнять измерения.

А нужна ли вообще блокировка?

Самый эффективный механизм блокировки — тот, которого нет. Если можно вообще избавиться от необходимости синхронизации потоков, это будет наилучшим способом получить высокую производительность. Это идеал, достичь которого не так-то просто. Обычно это означает, что нужно обеспечить отсутствие изменяемого совместно используемого состояния, — каждый запрос, проходящий через ваше приложение, может быть обработан независимо от другого запроса или каких-то централизованных изменяемых (посредством чтения-записи) данных. Такая возможность будет оптимальным сценарием достижения высокой производительности.

И все же будьте осторожны. С реструктуризацией легко переборщить и превратить код в беспорядочную мешанину, в которой никто, включая вас самих, не сможет разобраться. Не стоит заходить слишком далеко, если только высокая производительность не окажется действительно критическим фактором и ее нельзя будет добиться иным образом. Превратите код в асинхронный и независимый, но так, чтобы он оставался понятным.

Если несколько потоков просто выполняют чтение из переменной (и нет никаких намеков на запись в нее со стороны какого-либо потока), синхронизация не нужна. Все потоки могут иметь неограниченный доступ. Это автоматически распространяется на такие неизменяемые объекты, как строки или значения неизменяемых типов, но может относиться к любому типу объектов, если гарантировать неизменяемость его значения в ходе чтения несколькими потоками.

Если есть несколько потоков, ведущих запись в какую-нибудь совместно используемую переменную, посмотрите, нельзя ли устранить потребность в синхронизированном доступе путем перехода к применению локальной переменной. Если можно создать для работы временную копию, необходимость в синхронизации

отпадет. Это особенно важно для повторяющегося синхронизированного доступа. От повторного доступа к совместно используемой переменной нужно перейти к повторному доступу к локальной переменной, следующему за однократным доступом к совместно используемой переменной, как в следующем простом примере добавления элементов к совместно используемой несколькими потоками коллекции.

```
object syncObj = new object();
var masterList = new List<long >();
const int NumTasks = 8;
Task[] tasks = new Task[NumTasks];

for (int i = 0; i < NumTasks; i++)
{
    tasks[i] = Task.Run(()=>
    {
        for (int j = 0; j < 5000000; j++)
        {
            lock (syncObj)
            {
                masterList.Add(j);
            }
        }
    });
}
Task.WaitAll(tasks);
```

Этот код можно преобразовать следующим образом:

```
object syncObj = new object();
var masterList = new List<long >();
const int NumTasks = 8;
Task[] tasks = new Task[NumTasks];

for (int i = 0; i < NumTasks; i++)
{
    tasks[i] = Task.Run(()=>
    {
        var localList = new List<long >();
        for (int j = 0; j < 5000000; j++)
        {
            localList.Add(j);
        }
        lock (syncObj)
        {
            masterList.AddRange(localList);
        }
    });
}
Task.WaitAll(tasks);
```

На моей машине второй вариант кода выполняется более чем в два раза быстрее первого.

В конечном счете изменяемое совместно используемое состояние — принципиальный враг производительности. Оно требует синхронизации для безопасности данных, что ухудшает производительность. Если в вашей конструкции есть хоть малейшая возможность избежать блокировки, то вы близки к реализации идеальной многопоточной системы.

Порядок предпочтения синхронизации

Принимая решение о необходимости какой-либо разновидности синхронизации, следует понимать, что не все они имеют одинаковые характеристики производительности или поведения. В большинстве ситуаций требуется просто использовать блокировку, и обычно это и должно быть исходным вариантом. Применение чего-то иного, чем блокировка, для обоснования дополнительной сложности требует интенсивных измерений. В целом рассмотрим механизмы синхронизации в следующем порядке.

1. `lock`/класс `Monitor` — сохраняет простоту, доступность кода для понимания и обеспечивает хороший баланс производительности.
2. Полное отсутствие синхронизации. Избавьтесь от совместно используемых изменяемых состояний, проведите реструктуризацию и оптимизацию. Это труднее, но если получится, то в основном будет работать лучше, чем применение блокировки (кроме случаев, когда допущены ошибки или ухудшена архитектура).
3. Простые методы взаимной блокировки `Interlocked` — в некоторых сценариях могут оказаться более подходящими, но, как только ситуация начнет усложняться, перейдите к использованию блокировки `lock`.

И наконец, если действительно можно будет доказать пользу от их применения, задействуйте более замысловатые, сложные блокировки (имейте в виду: они редко оказываются настолько полезными, как вы ожидаете):

- асинхронные блокировки (будут рассмотрены далее в этой главе);
- все остальные.

Конкретные обстоятельства могут диктовать применение некоторых из этих технологий или же препятствовать этому. Например, объединение нескольких методов `Interlocked` вряд ли превзойдет по эффективности одну инструкцию `lock`.

Модели памяти

Прежде чем рассматривать детали синхронизации потоков, нужно провести краткий экскурс в мир моделей памяти. Под моделью памяти понимается действующий в системе (оборудовании или программном обеспечении) набор правил, которым определяется порядок возможного переупорядочения компилятором или процессором операций чтения и записи между несколькими потоками. Вносить в модель

какие-либо изменения вы не можете, но разбираться в ней, чтобы получить правильный код в любых ситуациях, крайне важно.

Модель памяти считается жесткой, если в ней имеются абсолютные ограничения на переупорядочение, не позволяющие компилятору или оборудованию выполнять масштабную оптимизацию. Мягкая модель дает компилятору и процессору намного больше свободы по переупорядочению инструкций чтения и записи с целью получения лучшей производительности. Большинство платформ — это что-то среднее между абсолютно жесткими и совершенно мягкими.

Стандартом ECMA определяется минимальный набор правил, которым должна следовать среда CLR. Им задается весьма мягкая модель памяти, но конкретно взятая реализация среды CLR фактически может иметь в виде надстройки более жесткую модель.

Более жесткая модель памяти может диктоваться также конкретной архитектурой процессора. Например, архитектура x86/x64 обладает относительно жесткой моделью памяти, которая будет автоматически препятствовать одним видам переупорядочения и допускать другие. В то же время архитектура ARM имеет относительно мягкую модель памяти. JIT-компилятор отвечает за обеспечение правильного порядка выдачи не только машинных инструкций, но и специальных инструкций, гарантирующих, что процессор не станет переупорядочивать инструкции таким способом, который нарушит модель памяти, используемую в среде CLR. Выполнение этих инструкций может быть недешевым, и это еще один аргумент в пользу полного уклонения от синхронизации.

Разница между архитектурами x86/x64 и ARM существенно влияет на ваш код, тем более когда в нем есть ошибки в синхронизации потоков. Поскольку JIT-компилятор, запущенный на ARM, более свободен при переупорядочении операций чтения и записи, чем он же, но запущенный на x86/x64, конкретные классы ошибок синхронизации могут остаться незамеченными на x86/x64 и становятся очевидными только при портировании на ARM.

В некоторых случаях среда CLR будет помогать скрывать эти различия из соображений совместимости, но на практике лучше не допускать ошибок в коде, чтобы его можно было задействовать в более мягкой модели памяти. Самое важное требование в данном случае — правильное применение совместно используемого потоками изменчивого (`volatile`) состояния. Ключевое слово `volatile` служит для JIT-компилятора сигналом, что для этой переменной важен порядок. На платформе x86/x64 оно приведет к соблюдению правил упорядочения инструкций, а на платформе ARM — также к появлению дополнительных инструкций, выдаваемых для соблюдения правильной семантики на уровне оборудования. Если пренебречь ключевым словом `volatile`, стандарт ECMA допустит полное переупорядочение этих инструкций, что может вызвать появление ошибок.

Еще один способ обеспечения правильного порядка обращения к совместно используемому состоянию заключается в применении класса `Interlocked` или в содержании всех обращений внутри полной блокировки `lock`.

Всеми этими методами синхронизации создается так называемый барьер памяти. Любые операции чтения, происходящие до этого места в коде, не могут быть переупорядочены после барьера, а любые операции записи не могут быть переупорядочены, чтобы оказаться перед барьером. В этом случае обновления переменных происходят правильно и видны всем центральным процессорам.

Использование volatile при необходимости

Рассмотрим классический пример неправильной реализации блокировки с двойной проверкой, пытающейся создать эффективную защиту от вызова несколькими потоками метода `DoCompletionWork`. Она старается предотвратить имеющие потенциально высокие издержки конкурирующие вызовы `lock` в общем случае, где нет необходимости вызова `DoCompletionWork`.

```
private bool isComplete = false;
private object syncObj= new object();
// Неверная реализация!
private void Complete()
{
    if (!isComplete)
    {
        lock (syncObj)
        {
            if (!isComplete)
            {
                DoCompletionWork();
                isComplete = true;
            }
        }
    }
}
```

Хоть инструкция `lock` и создаст эффективную защиту блока, который размещен внутри нее, но одновременно с этим находящаяся снаружи проверка `isComplete`, не имеющая никакой защиты, будет доступна сразу нескольким потокам. К сожалению, обновления этой переменной могут не работать, поскольку оптимизация, позволенная компилятору моделью памяти, приводит к тому, что сразу несколько потоков видят значение `false` даже после того, как другой поток установил для нее значение `true`. Но фактически все еще хуже: вполне возможно, значение `isComplete` будет установлено в `true` до завершения `DoCompletionWork()`, а это значит, что состояние программы может оказаться неправильным, если другие потоки выполняют проверку и совершают действия на основе значения переменной `isComplete`. А почему бы в любом случае не использовать блокировку вокруг доступа к `isComplete`? Можно, конечно, но такое решение приведет к более высокой конкуренции и повлечет за собой неоправданно высокие издержки.

Чтобы исправить ситуацию, нужно проинструктировать компилятор таким образом, чтобы гарантировать правильный порядок доступа к этой переменной. Это делается с помощью ключевого слова `volatile`. При этом единственным необходимым изменением будет следующее:

```
private volatile bool isComplete = false;
```

Разъясним получившийся результат: `volatile` используется для обеспечения правильного хода выполнения программы, а не для повышения производительности. В большинстве случаев это не оказывает заметного положительного или отрицательного влияния на производительность. Если есть возможность, примените данное ключевое слово: в любом сценарии с высоким уровнем конфликтов доступа это лучше, чем задействовать `lock`, и именно поэтому полезно будет использовать шаблон блокировки с двойной проверкой.

Блокировка с двойной проверкой часто задействуется при реализации шаблона «Одиночка» (Singleton), когда первый поток, использующий значение, должен его инициализировать. Этот шаблон инкапсулируется в .NET с помощью класса `Lazy<T>`, внутри которого применяется шаблон блокировки с двойной проверкой. Вместо собственной реализации шаблона лучше отдать предпочтение `Lazy<T>`. Более подробно работа с `Lazy<T>` рассмотрена в главе 6.

Использование Monitor (lock)

Проще всего защитить любую область кода с помощью объекта `Monitor`, у которого в C# есть эквивалент в виде ключевого слова.

Код:

```
object obj = new object();
bool taken = false;
try
{
    Monitor.Enter(obj, ref taken);
}
finally
{
    if (taken)
    {
        Monitor.Exit(obj);
    }
}
```

является эквивалентом следующего:

```
object obj = new object();

lock(obj)
{
    ...
}
```

Значение параметра `taken` устанавливается в `true`, если не выдается исключение. Это гарантировано и позволяет правильно вызывать метод `Exit`.

Если не будет веских аргументов в пользу использования других, более сложных механизмов блокировки, то при прочих равных условиях следует отдавать предпочтение `Monitor`/`lock`. `Monitor` представляет собой гибридную блокировку, при которой сначала предпринимается попытка прокрутки цикла в течение некоторого времени, прежде чем код войдет в состояние ожидания и отдаст поток. Это делает его идеальным для тех мест, где предвидится небольшой или быстро проходящий конфликт.

У `Monitor` имеется также более гибкий API, который может пригодиться, если есть необязательная работа, которую необходимо выполнить, когда немедленное получение блокировки невозможно:

```
object obj = new object();
bool taken = false;
try
{
    Monitor.TryEnter(obj, ref taken);
    if (taken) // выполнение работы, требующей блокировки
    {...}
    else // выполнение чего-нибудь еще
    {...}
}
finally
{
    if (taken)
    {
        Monitor.Exit(obj);
    }
}
```

В данном случае возврат из `TryEnter` происходит сразу, вне зависимости от того, получена блокировка или нет. Чтобы узнать, что делать дальше, можно проверить значение переменной `taken`. Существуют также переопределения, принимающие значение времени ожидания.

Какой объект блокировать

В качестве аргумента класс `Monitor` получает объект синхронизации[1]. Этому методу можно передать любой объект ссылочного типа. Ссылочные типы требуются по той причине, что, в отличие от типов значений, каждый такой объект содержит в качестве части своей структуры памяти блок синхронизации. Дополнительные сведения о структуре объекта приводятся в главе 5. При выборе используемого объекта нужно проявлять осмотрительность. Если будет передан явно видимый объект, для другой части кода появится возможность также им воспользоваться

[1] Неточность у автора: классы не могут получать аргументы, могут только методы. Не указано, о каком методе речь. Вероятнее всего, это Enter. — *Примеч. науч. ред.*

как объектом синхронизации, даже если в синхронизации между двумя этими разделами кода нет необходимости. Если будет передан сложный объект, то, беря на себя блокировку, вы рискуете функциональностью в этом классе. Обе эти ситуации могут ухудшить производительность или, того хуже, вызвать взаимные блокировки.

Чтобы избежать этого, стоит почти всегда, как показано в ранее приведенных примерах, выделять конкретно под блокировку простой закрытый объект.

В то же время в некоторых ситуациях явные объекты синхронизации могут создавать проблемы, особенно при очень большом количестве объектов и весьма обременительных издержках, связанных с дополнительным полем в классе. В таком случае можно найти какой-нибудь другой безопасный объект, способный послужить объектом синхронизации, или, что еще лучше, переструктурировать код, чтобы в первую очередь отпала необходимость в блокировке.

Есть такие классы объектов, которые ни в коем случае не следует применять в качестве объекта синхронизации для `Monitor`. К их числу относятся любой `MarshalByRefObject` (прокси-объект, который не станет защищать базовый ресурс), строковые объекты (они изолируются и совместно используются непредсказуемым образом), `System.Type`, или значимый тип (он будет упаковываться всякий раз при выполнении на нем блокировки, не допуская какой-либо синхронизации вообще).

Убедитесь, что вы не злоупотребляете совместным использованием объектов синхронизации между различными сценариями блокировки. Требуется, чтобы один и тот же объект синхронизации применяли только те сценарии, которые вступают в конфликт. Если есть несколько независимых сценариев, не мешающих друг другу, не задействуйте повторно один и тот же объект, а воспользуйтесь двумя разными объектами блокировки, гарантируя тем самым лучшие возможности для масштабирования.

Область действия блокировки

При реализации блокировки вокруг какого-нибудь кода следует решить, сколько кода в нее заключить. В общем виде ответ должен быть таким: как можно меньше (как правило). Войти, проделать работу, выйти.

Чем меньше кода охватывает блокировка, тем меньше потенциальная возможность конфликта блокировки. Однако не стоит впадать в крайности. Если вы обнаружите, что множество раз в одном и том же потоке устанавливаете и снимаете блокировку, то издержки на используемый механизм блокировки могут превысить расходы на выполнение работы внутри блокировки. Такое запросто может случиться в двух типах весьма распространенных сценариев: при обращении к коллекциям и в циклах.

Что касается коллекций, примите во внимание то, как производится доступ к элементам. Будет ли эффективнее ставить блокировку на каждый отдельно взятый доступ? Или лучше ставить блокировку на более высоком уровне, выполнять все обращения, а затем снимать ее?

Те же соображения применимы и для циклов. Если ставите блокировку на каждую итерацию цикла, выполните измерения и посмотрите, не лучше ли будет просто поставить блокировку на весь цикл.

Использование методов Interlocked

Рассмотрим следующий код с блокировкой, гарантирующий, что метод `Complete` может выполняться только одним потоком:

```
private bool isComplete = false;
private object completeLock = new object();

private void Complete()
{
    lock(completeLock)
    {
        if (isComplete)
        {
            return;
        }

        DoCompletionWorkHere();

        isComplete = true;
    }
}
```

Для того чтобы определить, войти ли в метод, вам нужны два поля и несколько строк кода — это выглядит нерационально. Вместо этого вызовите метод `Interlocked.Increment`:

```
private int isComplete = 0;

private void Complete()
{
    if (Interlocked.Increment(ref isComplete) == 1)
    {
        DoCompletionWorkHere();
    }
}
```

Или же рассмотрим немного иную ситуацию, где `Complete` может быть вызван несколько раз, но вам требуется войти в него только один раз на основании некоего внутреннего состояния.

```
enum State { Executing , Done };
private int state = (int)State.Executing;

private void Complete()
{
    if (Interlocked.CompareExchange (ref state , (int)State.Done,
```

```
                    (int)State.Executing) == (int)State.Executing)
    {
        DoCompletionWorkHere();
    }
}
```

При первом выполнении `Complete` будет сравнивать значение переменной `state` со значением `State.Executing` и, если они окажутся равны, заменит значение `state` значением `State.Done`. Выполняющий этот код следующий поток станет сравнивать значение `state` со значением `State.Executing`, при этом результат не будет истинным и `CompareExchange` возвратит `State.Done`, что не позволит выполнить тело инструкции `if`.

Методы `Interlocked` преобразуются в одну инструкцию процессора и являются атомарными. Они хорошо подходят для подобного рода простой синхронизации. Существует несколько методов `Interlocked`, которые можно использовать для простой синхронизации, и все они выполняют операцию атомарно.

- `Add` — складывает два целочисленных значения, заменяя первое из них суммой и возвращая сумму.
- `CompareExchange` — принимает значения A, B и C. Сравнивает значения A и C и, если они равны, заменяет значение A значением B, затем возвращает исходное значение. Смотрите далее код примера `LockFreeStack`.
- `Increment` — прибавляет единицу к значению и возвращает новое значение.
- `Decrement` — вычитает единицу из значения и возвращает новое значение.
- `Exchange` — устанавливает для переменной указанное значение и возвращает исходное значение.

У всех этих методов есть несколько переопределений для различных типов данных.

Операции `Interlocked` — это барьеры памяти, поэтому они выполняются не так быстро, как простая запись в бесконфликтном сценарии. Вообще-то не стоит рассчитывать, что они значительно быстрее простого `lock`-сценария, но простое эталонное тестирование в учебном проекте `InterlockedVsLock` покажет все же некоторое весьма незначительное преимущество.

При всей своей простоте методы `Interlocked` позволяют реализовывать более эффективные подходы, такие как структуры данных без блокировок (`lock-free`). Однако следует серьезно вас предостеречь: при реализации подобных собственных структур данных будьте максимально осмотрительны. Привлекательные слова «без блокировок» могут ввести в заблуждение. На самом деле они являются синонимом выражения «с повторением операции до достижения правильного результата». Как только начнут происходить многочисленные вызовы методов `Interlocked`, весьма вероятно, что в первую очередь по эффективности они уступят простому `lock`, даже если предположить, что ваш код на 100 % корректен (скорее всего, это не так). Превратить его в правильный или высокоэффективный может быть очень трудно. Реализация подобной структуры данных хорошо подходит для обучения,

но в реальном коде в первую очередь стоит выбирать использование встроенных .NET-коллекций. При реализации собственной коллекции, ориентированной на безопасное выполнение в нескольких потоках, следует приложить максимум усилий, чтобы убедиться, что она не только на 100 % правильная функционально, но и работает лучше тех вариантов, которые имеются в .NET. Задействуя методы `Interlocked`, убедитесь, что они предоставляют больше преимуществ, чем простая блокировка `lock`. Ситуации, где такой подход оправдан, складываются крайне редко.

В качестве примера (см. исходный код проекта LockFreeStack) приведу простую реализацию стека, использующую методы `Interlocked` для поддержки безопасности потоков без применения более строгих механизмов блокировки:

```
class LockFreeStack <T>
{
    private class Node
    {
        public T Value;
        public Node Next;
    }

    private Node head;

    public void Push(T value)
    {
        var newNode = new Node() { Value = value };

        while (true)
        {
            newNode.Next = this.head;
            if (Interlocked.CompareExchange(ref this.head,
                                            newNode ,
                                            newNode.Next)
                == newNode.Next)
            {
                return;
            }
        }
    }

    public T Pop()
    {
        while (true)
        {
            Node node = this.head;
            if (node == null)
            {
                return default(T);
            }
            if (Interlocked.CompareExchange(ref this.head,
                                            node.Next, node)
```

```
                == node)
            {
                return node.Value;
            }
        }
    }
}
```

В этом фрагменте кода показана общая схема с реализацией структур данных или более сложной логики с использованием методов `Interlocked` — зацикливание. Код с циклом постоянно тестирует результаты операции до ее успешного завершения. В большинстве сценариев будет выполнено весьма незначительное количество итераций.

При всей простоте и относительной быстроте выполнения методов `Interlocked` вам зачастую придется ставить под защиту все более крупные области кода, что не принесет желаемого результата или как минимум сделает код громоздким и сложным.

Асинхронные блокировки

С выходом .NET 4.5 появился ряд интересных функциональных возможностей, которые в будущем могут распространиться на другие типы. В классе `SemaphoreSlim` имеется метод `WaitAsync`, возвращающий `Task`. Вместо блокировки на ожидании можно спланировать для задачи `Task` метод продолжения, который запустится, как только семафор позволит ей это сделать. Здесь нет входа в режим ядра и нет блокировки. Когда пропадут конфликты из-за блокировки, метод продолжения будет спланирован как обычная задача `Task` в пуле потоков. Его использование ничем не будет отличаться от применения любой другой задачи `Task`.

Чтобы вы посмотрели, как это работает, приведу пример, использующий стандартные блокирующие механизмы ожидания. Этот учебный код из проекта WaitAsync, включенного в сборник учебного кода, не отличается особой сложностью, но показывает, как потоки передают управление через семафор:

```
class Program
{
    static SemaphoreSlim semaphore = new SemaphoreSlim(1);
    const int WaitTimeMs = 1000;

    static void Main(string[] args)
    {
        Task.Run((Action)Func1);
        Task.Run((Action)Func2);
        Console.ReadKey();
    }

    static void Func1()
    {
```

```
        while (true)
        {
            semaphore.Wait();
            Console.WriteLine("Func1");
            semaphore.Release();
            Thread.Sleep(WaitTimeMs);
        }
    }

    static void Func2()
    {
        while (true)
        {
            semaphore.Wait();
            Console.WriteLine("Func2");
            semaphore.Release();
            Thread.Sleep(WaitTimeMs);
        }
    }
}
```

Каждый поток входит в бесконечный цикл, ожидая завершения другим потоком своей операции цикла. Когда вызывается метод `Wait`, этот поток блокируется до высвобождения семафора. В высоконагруженной программе блокировка станет крайне нерационально использовать ресурсы и способна сократить возможности обработки данных и увеличит пул потоков. Если блокировка продлится довольно долго, может произойти переход в режим ядра, что приведет к еще большей пустой трате времени.

Чтобы воспользоваться `WaitAsync`, замените методы следующими реализациями:

```
static void AsyncFunc1()
{
    semaphore.WaitAsync().ContinueWith(_ =>
    {
        Console.WriteLine("AsyncFunc1");
        semaphore.Release();
        Thread.Sleep(WaitTimeMs);
    }).ContinueWith(_ => AsyncFunc1());
}

static void AsyncFunc2()
{
    semaphore.WaitAsync().ContinueWith(_ =>
    {
        Console.WriteLine("AsyncFunc2");
        semaphore.Release();
        Thread.Sleep(WaitTimeMs);
    }).ContinueWith(_ => AsyncFunc2());
}
```

Заметьте, что вместо цикла появилась цепочка продолжений, которые будут осуществлять обратный вызов в эти методы. Логически это носит рекурсивный характер, но фактически так не получается, поскольку каждое продолжение запускает новый стек с новой задачей `Task`.

Если применяется `WaitAsync`, никаких блокировок не происходит, но даром это не обходится. Издержки возникают за счет повышения объема диспетчеризации задач и вызовов функций. Если программа выполняет диспетчеризацию слишком большого количества `Task`-объектов, возрастающие нагрузки планирования могут свести на нет выигрыш, полученный в результате отказа от вызова блокировок. Если ваши блокировки крайне непродолжительны и заключаются просто в холостом ходу без входа в режим ядра или выполняются крайне редко, то, может быть, лучше просто провести блокировку на несколько микросекунд, использовав стандартный метод `lock`. Но если под блокировкой требуется выполнить довольно продолжительную задачу, рассматриваемый вариант может оказаться предпочтительным, поскольку издержки от новой задачи `Task` окажутся меньше времени блокировки процессора в случае выбора иного варианта. Здесь понадобятся тщательные измерения и проведение различных экспериментов.

Если эта схема вас заинтересовала, обратите внимание на серию статей Стивена Туба (Stephen Toub), посвященных примитивам асинхронной координации, под общим названием *Building Async Coordination Primitives (Parts 1–7)*, в которых рассматривается больше типов и схем.

Другие механизмы блокировки

Существует множество механизмов блокировки, которыми можно воспользоваться, но предпочтение стоит отдавать как можно меньшему количеству. Как и во многих других областях, особенно в вычислениях с использованием нескольких потоков, чем проще такие механизмы, тем лучше.

Самый простой способ гарантировать вызов метода только из одного потока — его оформление параметром `MethodImplOptions.Synchronized`. Его можно применять к статическим методам или методам экземпляров:

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void DoSomething(int arg)
{
    ...
}
```

Однако это очень грубый инструмент, потребность в использовании подобных слабоориентированных средств возникает крайне редко. Как правило, лучше поставить блокировку на более узкую область внутри метода.

Если известно, что блокировка крайне непродолжительна (десятки циклов) и требуется обеспечить невозможность ее вхождения в состояние ожидания, можно воспользоваться `SpinLock`. Он просто будет прокручивать пустой цикл, пока не разрешится конфликтная ситуация. В большинстве случаев наилучшим исходным

выбором станет Monitor, который входит сначала в пустой цикл, а затем, если необходимо, в ожидание:

```
private SpinLock spinLock = new SpinLock();

private void DoWork()
{
    bool taken = false;
    try
    {
        spinLock.Enter(ref taken);
        // Выполнение полезной работы
    }
    finally
    {
        if (taken)
        {
            spinLock.Exit();
        }
    }
}
```

Вообще-то следует по возможности избегать использования других блокирующих механизмов. Обычно они не настолько производительны, как простой Monitor. Объекты, подобные ReaderWriterLockSlim, Semaphore, Mutex, или другие пользовательские объекты синхронизации применяются в определенных местах, зачастую они сложны и подвержены ошибкам.

Нужно полностью исключить использование ReaderWriterLock — этот метод не рекомендован к применению.

Если существует –Slim-версия объекта синхронизации, ей нужно отдавать предпочтение перед не-Slim-версией. Все –Slim-версии являются гибридными блокировками, а это означает, что в них реализованы некие формы пустых циклов, предшествующих входу в режим ядра, который существенно замедляет работу. -Slim-блокировки намного лучше подходят при низком уровне конфликтов небольшой продолжительности.

Например, имеются оба класса, ManualResetEvent и ManualResetEventSlim, и есть класс AutoResetEvent, но нет класса AutoResetEventSlim. Для получения точно такого же эффекта можно воспользоваться SemaphoreSlim со значением 1 параметра initialCount.

Конкурентность и коллекции

Существует несколько коллекций, предоставляемых .NET, которые допускают одновременное обращение из нескольких потоков. Все они находятся в пространстве имен System.Collections.Concurrent, и к их числу относятся:

- ConcurrentBag<T> — неупорядоченная коллекция;
- ConcurrentDictionary<TKey, TValue> — пары «ключ — значение»;

- `ConcurrentQueue<T>` — очередь по принципу «первым пришел — первым ушел»;
- `ConcurrentStack<T>` — стек («последним пришел — первым ушел»).

Большинство из них реализованы внутренним образом с использованием примитивов синхронизации `Interlocked` или `Monitor`, и я рекомендую проэкзаменовать их реализации с помощью инструмента отражения IL.

Несмотря на удобство коллекций, следует проявлять осмотрительность — каждое отдельно взятое обращение к одной из них не обходится без синхронизации. Зачастую при повышенной конфликтности обращений ресурсы расходуются чрезмерно, что может навредить производительности. Если не обойтись без «слишком болтливых» схем обращения к коллекциям для чтения-записи, эта низкоуровневая синхронизация может быть целесообразной.

В этом разделе рассмотрим несколько альтернатив для коллекций с параллельным доступом, которые могут упростить ваши требования к блокировкам. Коллекции в целом, включая перечисленные здесь, в частности для описания уникальных для этих коллекций API, правильное освоение которых может вызвать затруднения, рассматриваются в главе 6.

Блокировка на более высоком уровне

Если приходится иметь дело с параллельным обновлением или чтением многих значений, то, по всей видимости, нужно будет воспользоваться коллекцией, не рассчитанной на параллельное обращение, и управлять блокировкой самостоятельно на более высоком уровне (или же найти способ, при котором синхронизация вообще не понадобится, — в следующем разделе есть одна идея на этот счет).

Уровень раздробленности в вашем механизме синхронизации оказывает огромное влияние на общую эффективность. Во многих случаях выполнение пакетных обновлений под одной блокировкой окажется лучше установки блокировки для каждого отдельного небольшого обновления. Выполняя неформальное тестирование, я понял, что элемент вставляется в коллекцию `ConcurrentDictionary<TKey,TValue>` примерно в два раза медленнее, чем в стандартную коллекцию `Dictionary<TKey, TValue>`.

В своем приложении вам придется проводить измерения и взвешивать все за и против.

Также следует заметить, что иногда придется ставить блокировку на более высоком уровне, чтобы гарантировать соблюдение ограничений, присущих контексту. Рассмотрим классический пример банковского перевода между двумя счетами. Баланс обоих счетов должен, разумеется, изменяться в индивидуальном порядке, но транзакция имеет смысл только в том случае, если эти действия рассматриваются как единое целое. Транзакции базы данных являются взаимосвязанными. Сама по себе вставка одной строки может быть атомарной операцией, но чтобы гарантировать целостность операций, может понадобиться использовать транзакции, чтобы обеспечить атомарность на более высоком уровне.

Замена всей коллекции

Если ваши данные предназначены в основном для чтения, тогда, получая доступ к ним, смело можно воспользоваться коллекцией, не предназначенной для обеспечения безопасности при параллельных обращениях. Когда наступает время обновления коллекции, можно создать абсолютно новую коллекцию и по окончании ее загрузки однократно заменить исходную ссылку, как в следующем примере:

```
private volatile Dictionary <string , MyComplexObject > data = new
  Dictionary <string , MyComplexObject >();

public Dictionary <string , MyComplexObject > Data
{
    get
    {
        return data;
    }
}

private void UpdateData()
{
    var newData = new Dictionary <string , MyComplexObject >();
    newData["Foo"] = new MyComplexObject();
    ...
    data = newData;
}
```

Обратите внимание на использование ключевого слова `volatile`, которое гарантирует, что данные не обновятся, пока не будет полностью построена коллекция `newData`, а после обновления она станет действительной во всех потоках.

Если потребителю этого класса несколько раз необходим доступ к свойству `Data` и нужно, чтобы этот объект не был заменен новым, он может создать локальную копию ссылки и задействовать ее вместо исходного свойства:

```
private void CreateReport(DataSource source)
{
    Dictionary <string , MyComplexObject > data = source.Data;

    foreach(var kvp in data)
    {
       ...
    }
}
```

Это ненадежная стратегия. Она немного усложняет ваш код и рассеивает ответственность за правильное применение структуры данных. Может потребоваться предупредить остальную часть программы с помощью специальной семантики об использовании этой структуры данных. Следует также учитывать баланс между такой заменой и затратами на возможную полную сборку мусора при выделении

памяти для новой коллекции. Эта схема может пригодиться для многих сценариев при соблюдении условия, что перезаписывать коллекцию придется нечасто и что вы справитесь со случающейся время от времени полной сборкой мусора. О том, как избежать полной сборки мусора, говорилось в главе 2.

Копирование ресурса для каждого потока

Если имеется непотокобезопасный ресурс, активно используемый несколькими потоками, можно рассмотреть вариант его оборачивания в объект `ThreadLocal<T>`.

Вот классический пример, использующий непотокобезопасный класс `Random`:

```
private static readonly ThreadLocal <Random > threadLocalRand
    = new ThreadLocal <Random >(()=>new Random());

static void Main(string[] args)
{
    int[] results = new int[100];

    Parallel.For(0, 5000,
        i =>
        {
            var randomNumber = threadLocalRand.Value.Next(100);
            Interlocked.Increment(ref results[randomNumber]);
        });
}
```

Конструктор `ThreadLocal<T>` получает объект `Func<T>`, позволяющий указать фабричный метод. Он будет вызван при первом использовании переменной в потоке, перед тем как объект вам возвращается.

`ThreadLocal<T>` доступен в .Net 4 и последующих версиях. Если по каким-то причинам необходимо воспользоваться более ранней версией .Net, существует альтернатива — атрибут `[ThreadStatic]`. Его можно применять только к статическим переменным.

Вот тот же самый пример, реализованный с использованием `[ThreadStatic]` (он взят из учебного проекта MultiThreadRand):

```
[ThreadStatic]
static Random threadStaticRand;

static void Main(string[] args)
{
    int[] results = new int[100];

    Parallel.For(0, 5000,
        i =>
        {
            // статическое поле данного потока не инициализировано
            if (threadStaticRand == null)
            {
```

```
                threadStaticRand = new Random();
            }
            var randomNumber = threadStaticRand.Next(100);
            Interlocked.Increment(ref results[randomNumber]);
        });
}
```

Всегда нужно исходить из того, что статические объекты с пометкой `[ThreadStatic]` во время первого использования не инициализированы — .NET инициализирует только первый из них. У остальных будут исходные значения (обычно `null`). Из-за таких ограничений для применения `[ThreadStatic]` мало оснований, и в большинстве случаев следует отдавать предпочтение `ThreadLocal<T>`.

Важно отметить, что при использовании `[ThreadStatic]` и `ThreadLocal<T>` скорость выполнения кода ниже, чем когда применяются переменные, не учитывающие работу с несколькими потоками. Везде, где только можно, следует отдавать предпочтение стандартным переменным, но эти две возможности следует рассматривать в качестве простого способа превращения совместно используемого ресурса, нуждающегося в блокировке, в необщий ресурс, которому не требуется синхронизация.

Исследование потоков и конфликтов

Одним из наиболее сложных типов отладки является поиск проблем, связанных с многопоточностью. Тот, кто в первую очередь прикладывает усилия к получению правильного кода, впоследствии получает огромные дивиденды в виде времени, сэкономленного на отладке.

В то же время источники конфликтов в .NET можно обнаружить без особых затруднений, и для этого есть ряд хороших современных инструментальных средств, которые могут помочь в проведении общего анализа в многопоточных сценариях.

Счетчики производительности

В категории Process (Процесс) имеется счетчик Thread Count (Количество потоков).

В категории Synchronization (Синхронизация) можно найти следующие счетчики:

- Spinlock Acquires/sec — количество захватов спинлоков (циклических блокировок) в секунду;
- Spinlock Contentions/sec — количество конфликтов за спинлок в секунду;
- Spinlock Spins/sec — количество итераций циклов ожидания освобождения спинлока в секунду.

В пункте System (Система) есть счетчик Context Switches/sec (Количество приключений контекста в секунду). Каким должно быть его идеальное значение, понять довольно трудно — можно встретить множество противоречивых мнений (мне в качестве нормы обычно попадалось значение 300, а значение 1000 считалось слишком

высоким). Поэтому я предполагаю, что вы будете рассматривать данный счетчик в основном как относительный и отслеживать динамику изменения его значения: резкое увеличение будет свидетельствовать о наличии проблем в приложении.

В .NET в категории .NET CLR LocksAndThreads предоставляется несколько счетчиков, включая:

- # of current logical Threads — показывает количество управляемых потоков в процессе;
- # of current physical Threads — показывает количество потоков операционной системы, выделенных процессу для выполнения управляемых потоков, исключая те, которые используются только средой CLR;
- Contention Rate/sec — показывает уровень конкуренции в секунду. Важен для обнаружения «горячих» блокировок, требующих реструктуризации или удаления;
- Current Queue Length — подсчитывает количество потоков, заблокированных рассматриваемой блокировкой.

ETW-события

Среди следующих событий наиболее полезны `ContentionStart V1` и `ContentionStop`. Остальные могут представлять интерес, когда уровень параллельности выполнения кода неожиданно изменяется и нужно узнать, как ведет себя пул потоков.

- `ContentionStart V1` — конфликт начался. Для гибридных блокировок фаза пустого цикла в расчет не принимается, учитывается только момент вхождения в реальное заблокированное состояние. В число полей входит:
  - `Flags` — `0` для управляемого, `1` для обычного.
- `ContentionStop` — конфликт закончился.
- `ThreadPoolWorkerThreadStart` — поток пула потоков запущен. В число полей входят:
  - `ActiveWorkerThreadCount` — количество доступных рабочих потоков, как выполняющих работу, так и ожидающих ее;
  - `RetiredWorkerThreadCount` — количество рабочих потоков, удерживаемых в резерве на случай востребованности в дальнейшем большего количества потоков.
- `ThreadPoolWorkerThreadStop` — завершение выполнения потока пула потоков. Поля те же, что и для `ThreadPoolWorkerThreadStart`.
- `IOThreadCreate V1` — был создан поток ввода-вывода. В число полей входит:
  - `Count` — количество потоков ввода-вывода в пуле.

Дополнительные сведения о ETW-событиях, имеющих отношение к потокам, можно получить по адресу https://docs.microsoft.com/dotnet/framework/performance/thread-pool-etw-events.

Получение информации о потоках

Показать запущенные потоки, позволить выборочно поставить их на паузу и продолжить выполнение (или, в терминологии Visual Studio, заморозить и растопить) могут большинство отладчиков.

В WinDbg получить подробную информацию об управляемых потоках можно с помощью команд `!Threads` и `!ThreadState`:

```
0:007> !Threads
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
DeadThread: 0
Hosted Runtime: no

  ID OSID ThreadOBJ State GC Mode GC Alloc Context Domain ...
0 1 5580 034578a0 2a020 Preemptive 058ECED4:00000000 0344d1e0...
5 2 4bb8 034675c8 2b220 Preemptive 00000000:00000000 0344d1e0...
6 3 42b0 03486658 21220 Preemptive 00000000:00000000 0344d1e0...

0:007> !ThreadState 2a020
    Legal to Join
    CoInitialized
    In Multi Threaded Apartment
    Fully initialized
```

Команда `!ThreadState -special` покажет потоки, принадлежащие среде CLR, и то, чем они занимаются:

```
0:000> !Threads -special
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
DeadThread: 0
Hosted Runtime: no

 ID OSID ThreadOBJ State GC Mode GC Alloc Context Domain ...
0 1 5580 034578a0 2a020 Preemptive 058ECED4:00000000 0344d1e0...
5 2 4bb8 034675c8 2b220 Preemptive 00000000:00000000 0344d1e0...
6 3 42b0 03486658 21220 Preemptive 00000000:00000000 0344d1e0...

          OSID Special thread type
        4 4c30 DbgHelper
        5 4bb8 Finalizer
        6 42b0 GC
```

В данном случае можно увидеть поток финализатора и поток сборщика мусора (предположительно фоновый поток сборки мусора, поскольку в данном приложении используется сборка мусора в режиме рабочей станции).

Визуализация задач и потоков с помощью Visual Studio

В Visual Studio имеется дополнительное инструментальное средство Concurrency Visualizer (визуализатор параллелизма). Его работа основана на использовании ETW-событий с их показом в графической форме, чтобы у вас была точная картина того, чем в приложении занимаются все `Task`-объекты и потоки. Оно сообщает вам о том, когда `Task` делегирует «старт» и «стоп», о том, что поток заблокирован, о выполнении работы центральным процессором, об ожидании в ходе операций ввода-вывода и о многом другом (рис. 4.3).

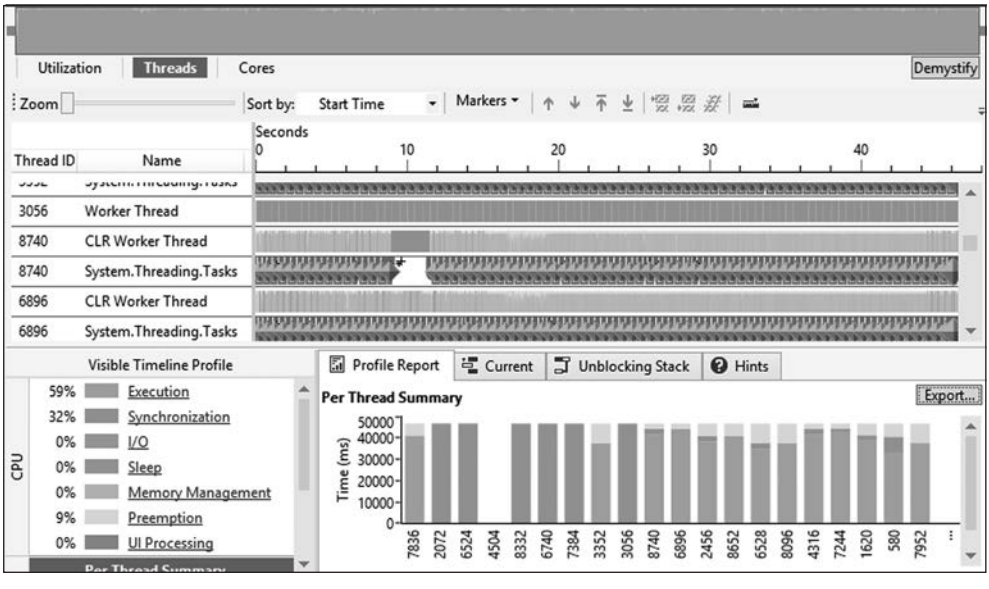

Рис. 4.3. В имеющемся в Visual Studio представлении Thread Times потоки, задачи, время центрального процессора, время блокировки, прерывания и многое другое объединяются в единую кореллированную шкалу времени

Важно отметить, что захват информации такого рода может существенно снизить производительность приложения, ведь событие будет записываться при каждом изменении состояния потока, что происходит очень часто.

На момент написания книги средство Concurrency Analyzer было недоступно[1] для Visual Studio 2017, но если нужно, то для сбора этих данных можно воспользоваться предоставляющим лишь базовые возможности Profiling Wizard. Запустите Profiling Wizard (из меню Analyze ▸ Performance Profiler (Анализ ▸ Профилировщик производительности)) и выберите параллельные вычисления (рис. 4.4).

[1] Выпущен не так давно как опциональное расширение (плагин). Можно найти по адресу https://marketplace.visualstudio.com/items?itemName=Diagnostics.ConcurrencyVisualizer2017#overview. — *Примеч. науч. ред.*

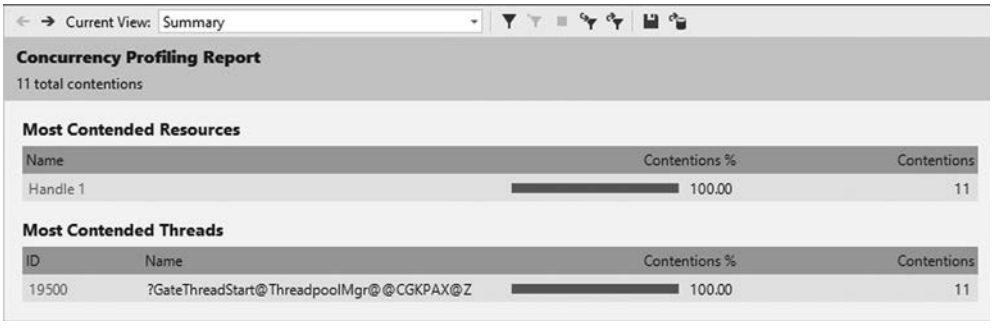

Рис. 4.4. Базовый анализатор параллелизма. Здесь показано, какие обработчики и потоки наиболее конфликтны

Использование PerfView для обнаружения конфликта блокировок

Отличной альтернативой Visual Studio, особенно при анализе производственных нагрузок, является PerfView. Чтобы увидеть это средство в действии, можно воспользоваться учебной программой HighContention. Запустите программу и соберите .NET-события, применив PerfView. Как только будет готов для просмотра файл с расширением `.etl`, откройте представление Any Stacks (Любые стеки), найдите запись Event Microsoft-Windows-DotNETRuntime/Contention/Start и сделайте на ней двойной щелчок кнопкой мыши. В результате откроется представление стека вашего процесса и будут показаны стеки, содействующие конкуренции потоков (рис. 4.5).

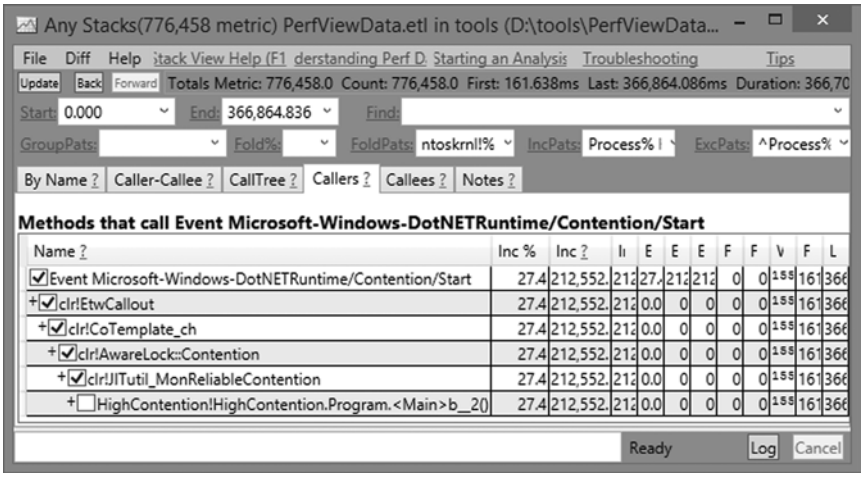

Рис. 4.5. PerfView показывает, какие стеки приводят к повышенной конкуренции для всех управляемых объектов синхронизации. В данном случае можно увидеть, что конкуренция исходит из безымянного метода внутри Main

Где потоки блокируются на вводе-выводе

Средство PerfView можно применять для сбора информации о потоках, например о том, когда они входят в различные состояния. Тем самым предоставляется более точная информация о затраченном вашей программой реальном времени, если она не использует центральный процессор. Но следует помнить, что включение этого параметра существенно замедляет работу программы.

В имеющемся в PerfView диалоговом окне Collection (Сбор) установите флажок Thread Times (Замеры времени для потоков), а также другие флажки, применяемые по умолчанию (Kernel, .NET и CPU-события).

После завершения сбора данных в дереве результатов будет показан узел Thread Times (Время потока). Раскройте его и посмотрите на вкладку By Name (По имени). В верхней части увидите две основные категории: BLOCKED_TIME и CPU_TIME (рис. 4.6). Дважды щелкните на BLOCKED_TIME, чтобы увидеть вызывающие объекты для этой группы.

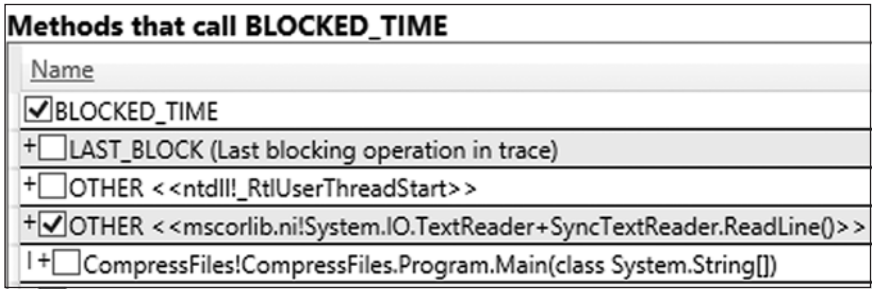

Рис. 4.6. Стек в PerfView для времени блокировки, установленной вызовом метода TextReader.ReadLine

Резюме

Когда нужно избежать блокировки потока пользовательского интерфейса, распараллелить работу по нескольким центральным процессорам или не допустить потерь мощности центрального процессора из-за блокировки потоков в ожидании завершения операций ввода-вывода, воспользуйтесь асинхронным кодом, например `async` и `await` (или непосредственно `Task`-объектами). Задействуйте `Task`-объекты вместо чистых потоков. Не создавайте ожиданий на `Task`-объектах, а планируйте продолжение, выполняемое по завершении задачи. Упрощайте синтаксис `Task` с помощью `async` и `await`. Рассматривайте возможность использования TPL Dataflow для соответствующих бизнес-задач.

Никогда не устанавливайте блокировку на операции ввода-вывода. Всегда применяйте асинхронные API при чтении из `Stream`-объектов и записи в них. Для дис-

петчеризации функций обратного вызова по завершении операций ввода-вывода используйте продолжения.

Старайтесь избегать блокировок, даже если для этого придется существенно реструктурировать код. При необходимости задействуйте как можно более простую блокировку. Для небольших или редко конфликтующих разделов применяйте `lock`/`Monitor` и воспользуйтесь в качестве объекта синхронизации закрытым полем. Для простых изменений состояния берите методы `Interlocked`. При наличии критически важного раздела с высоким уровнем конкуренции, имеющей место более чем несколько миллисекунд, рассмотрите возможность применения схем асинхронной блокировки, подобных `SemaphoreSlim`.

5 Общие подходы к написанию кода и классов

В этой главе речь пойдет об общих принципах, используемых при написании кода и типов, не рассматриваемых больше нигде в данной книге. В .NET содержатся функции для большого числа сценариев, и хотя многие из них как минимум нейтральны по отношению к производительности, некоторые, несомненно, не позволяют достичь ее высокого уровня. И мы должны решить, каким будет правильный подход в конкретной ситуации.

Если свести все, что будет рассмотрено в этой и следующей главах, к одному-единственному принципу, то он звучит так: **глубокая оптимизация кода с целью повышения производительности часто будет противоречить абстракциям кода**.

Это означает, что, стараясь достичь очень высокой производительности, вам нужно усвоить подробности реализации на всех уровнях и, возможно, положиться на них. О многих из них поговорим в данной главе.

Классы и структуры

Экземпляры класса всегда размещаются в куче, и обращение к ним происходит посредством разыменования указателя. Затраты на их передачу между методами невысоки, поскольку это просто копия указателя (4 или 8 байт). Однако у объекта также имеются фиксированные издержки: 8 байт для 32-разрядных процессов и 16 байт для 64-разрядных. Эти издержки включают указатель на таблицу методов и поле блока синхронизации, используемое для достижения нескольких целей. Однако если посмотреть на объект, не имеющий полей, в отладчике, можно обнаружить, что его длина показана как 12 байт (32 разряда) или 24 байта (64 разряда). Почему так происходит? .NET выровняет все объекты в памяти, и это эффективные минимальные размеры объектов.

Структура, также известная как тип значения, вообще не имеет издержек, и объем используемой ею памяти — это сумма размеров всех ее полей. Если структура объявлена в качестве локальной переменной в методе, она располагается в стеке. Если структура объявлена в качестве части класса, память структуры станет частью схемы памяти класса и, следовательно, будет существовать в куче. При передаче структуры методу происходит ее побайтовое копирование. Поскольку она не находится в куче, память, выделенная под структуру, никогда не будет подвергаться

сборке мусора. При этом, если безостановочно выделять память под большие структуры, при наличии очень глубоких стеков можно столкнуться с ограничениями на размер стека (что весьма возможно при использовании некоторых сред).

Таким образом, требуется найти компромисс. Вы можете встретить различные суждения по поводу максимально рекомендуемого размера структуры, но я не стал бы ориентироваться на конкретное число. В большинстве случаев стоит придерживаться весьма скромных размеров структур, особенно если они куда-нибудь передаются. С другой стороны, вы также можете передавать структуры по ссылке, поэтому их размер не обязательно для вас актуален. Единственным способом узнать, приносит ли это вам выгоду, является изучение схемы использования памяти и собственное профилирование.

В некоторых случаях получается огромная разница в эффективности. Хотя издержки, связанные с объектом, могут не показаться слишком большими, рассмотрим массив объектов и сравним его с массивом структур. Предположим, что в структуре содержится 16 байт данных, длина массива составляет 1 000 000 и все это в 32-разрядной системе.

Для массива объектов общее использование памяти составит:

8 байт (издержки, связанные с массивом) + (4 байта (размер указателя) · · 1 000 000) + (8 байт (издержки) + 16 байт (данные)) · 1 000 000 = 28 Мбайт.

Для массива структур результат будет совершенно другим:

8 байт (издержки, связанные с массивом) + (16 байт (данные) · 1 000 000) = = 16 Мбайт.

При задействовании 64-разрядного процесса массив объектов занимает более 40 Мбайт, а массив структур по-прежнему требует только 16 Мбайт.

Очевидно, что в массиве структур сопоставимый объем данных потребляет меньше памяти. Вдобавок к издержкам ссылочных типов провоцируется и более частая сборка мусора — просто по причине более интенсивного использования памяти.

Кроме использования пространства памяти возникает также вопрос об эффективности работы центрального процессора. У процессоров имеется несколько уровней кэширования. Те кэши, что ближе всех к процессору, весьма невелики, но работают исключительно быстро и оптимизированы под последовательный доступ.

В памяти массива структур имеется множество последовательно идущих значений. Обращение к элементу в массиве структур осуществляется очень просто. Найденная нужная запись уже содержит требуемое значение. Это может означать существенную разницу во временах доступа при последовательном переборе большого массива. Если при этом значение уже находится в кэше центрального процессора, к нему можно получить доступ на порядок быстрее, чем при его нахождении в оперативной памяти.

Чтобы обратиться к записи в массиве объектов, нужны доступ к памяти массива, а затем разыменование указателя на запись, находящуюся где-то в куче.

Последовательный перебор массивов объектов приводит к разыменовыванию дополнительного указателя, прыжкам по всей куче и более частому вытеснению данных из кэша центрального процессора, что ведет к потенциальной потере возможности использовать кэш для более ценных данных.

Во многих случаях в пользу структур в первую очередь говорит отсутствие издержек как для центрального процессора, так и для памяти. При рациональном использовании они могут дать вам существенный прирост производительности за счет более совершенного размещения в памяти, отсутствия накладных расходов на сборку мусора, и в силу того, что структуры вполне естественно находятся в стеке, побудить вас к применению модели программирования, в которой нет совместно используемого изменяемого состояния. Из-за этих естественных ограничений следует всерьез рассмотреть возможность сделать все ваши структуры неизменяемыми. Однако если потребуется изменить поля внутри структуры, которая является свойством другого класса, обратите внимание на функциональную возможность возврата по ссылке (`ref`), рассматриваемую далее в этой главе. Задействуя эту новую функцию в C#7, можно избежать копирования структур, снижающего производительность.

Исключение из правил: изменяемая структура для хранения иерархии полей

Ранее уже говорилось, что не следует создавать большие структуры, чтобы не тратить много времени на их копирование. Но все же большие, изменяемые структуры иногда используются для хранения иерархии полей. Рассмотрим объект, отслеживающий множество подробностей некоего коммерческого процесса, например множество меток времени:

```
class Order
{
  public DateTime ReceivedTime {get;set;}
  public DateTime AcknowledgeTime {get;set;}
  public DateTime ProcessBeginTime {get;set;}
  public DateTime WarehouseReceiveTime {get;set;}
  public DateTime WarehouseRunnerReceiveTime {get;set;}
  public DateTime WarehouseRunnerCompletionTime {get;set;}
  public DateTime PackingBeginTime {get;set;}
  public DateTime PackingEndTime {get;set;}
  public DateTime LabelPrintTime {get;set;}
  public DateTime CarrierNotifyTime {get;set;}
  public DateTime ProcessEndTime {get;set;}
  public DateTime EmailSentToCustomerTime {get;set;}
  public DateTime CarrerPickupTime {get;set;}

  // множество других данных...
}
```

Чтобы упростить код, было бы неплохо выделить все эти показатели времени в их собственные подструктуры, по-прежнему доступные через класс `Order` посредством кода, похожего на следующий:

```
Order order = new Order();
Order.Times.ReceivedTime = DateTime.UtcNow;
```

Все они могут быть помещены в собственный класс:

```
class OrderTimes
{
  public DateTime ReceivedTime {get;set;}
  public DateTime AcknowledgeTime {get;set;}
  public DateTime ProcessBeginTime {get;set;}
  public DateTime WarehouseReceiveTime {get;set;}
  public DateTime WarehouseRunnerReceiveTime {get;set;}
  public DateTime WarehouseRunnerCompletionTime {get;set;}
  public DateTime PackingBeginTime {get;set;}
  public DateTime PackingEndTime {get;set;}
  public DateTime LabelPrintTime {get;set;}
  public DateTime CarrierNotifyTime {get;set;}
  public DateTime ProcessEndTime {get;set;}
  public DateTime EmailSentToCustomerTime {get;set;}
  public DateTime CarrerPickupTime {get;set;}
}

class Order
{
  public OrderTimes Times;
}
```

Но при этом возникают дополнительные издержки — 12 байт или 24 байта — для каждого объекта `Order`. Если нужно передать объект `OrderTimes` различным методам целиком, возможно, в этом и есть смысл, но почему просто не передать ссылку на весь объект `Order` как таковой? Если параллельно обрабатываются тысячи объектов `Order`, это может вызвать более частые сборки мусора. Кроме того, добавятся дополнительные разыменования ссылок.

Вместо этого превратите `OrderTimes` в структуру. Обращение к отдельным свойствам структуры `OrderTimes` через свойство класса `Order` (`order.Times.ReceivedTime`) не приводит к копированию структуры (в .NET выполняет оптимизацию этого вполне осмысленного сценария). Таким образом, структура `OrderTimes` становится частью схемы памяти для класса `Order` почти так же, как было без подструктуры, и вы получаете более понятный код.

Суть этого приема заключается в том, чтобы рассматривать поля структуры `OrderTimes`, как если бы они были полями объекта `Order`. Вам не нужно передавать структуру `OrderTimes` как некую сущность туда и сюда саму по себе — это всего лишь организационный механизм.

Виртуальные методы и запечатанные классы

Не помечайте в самом начале методы ключевым словом `virtual` на всякий случай. Но если в вашей программе `virtual`-методы необходимы для целостности архитектуры, вам не следует отступать от намеченного пути, удаляя их.

Указание для методов ключевого слова `virtual` препятствует проведению JIT-компилятором определенных оптимизаций, в частности встраиванию их кода. Методы могут встраиваться только в том случае, если компилятор точно знает, какой метод будет вызван. Обозначение метода ключевым словом `virtual` устраняет эту определенность, хотя есть и иные факторы, приводящие к невозможности данной оптимизации, которые проявятся с большей вероятностью (они рассмотрены в главе 3).

`virtual`-методы тесно связаны с запечатыванием классов:

```
public sealed class MyClass {}
```

Класс с пометкой `sealed` заявляет, что никакие другие классы не могут от него наследоваться. Теоретически JIT может воспользоваться этой информацией для более активного встраивания, но пока не пользуется этой возможностью. Несмотря на это, следует изначально помечать классы ключевым словом `sealed` и не применять без необходимости ключевое слово `virtual`. Таким образом, код получит возможность извлечь преимущества из любых имеющихся на данный момент и вероятных будущих усовершенствований JIT-компилятора.

Создавая библиотеку классов для использования во множестве различных ситуаций, особенно вне своей организации, нужно быть более осмотрительными. В таком случае наличие виртуальных API может оказаться важнее производительности, так как это позволяет гарантировать достаточную степень переиспользуемости и настраиваемости. Но, разрабатывая часто изменяемый код, который применяется только для решения задач внутри организации, следуйте дорогой большей производительности.

Свойства

При обращении к свойствам следует проявлять осторожность. Синтаксически свойства похожи на поля, но на самом деле это фактически вызовы функций. Считается хорошим тоном реализовать свойства по возможности в более легкой манере, но если бы все было так просто и без особых затрат, как и при доступе к полям, то свойств бы не было. Главным образом их существование обусловлено тем, что нужно дать людям возможность добавлять проверочные и иные дополнительные функции для доступа к значению поля или его изменения.

Если доступ к свойству находится в цикле, то JIT, возможно, встроит вызываемый код, но это не гарантируется.

Если сомневаетесь, проверьте код свойств, к которым происходит обращение в критических для производительности областях, и примите соответствующее решение.

Переопределение Equals и GetHashCode для структур

Важная часть использования структур — переопределение методов `Equals` и `GetHashCode`. Если этого не сделать, в ход пойдут исходные версии, которые не слишком хороши для достижения высокой производительности. Понять, что они малопригодны, можно, посмотрев на код для метода `ValueType.Equals` с помощью просмотрщика IL. Он включает рефлексию для всех полей структуры. Есть разве что оптимизация для допускающих поблочный маршалинг типов (непреобразуемые типы, blittable). У таких типов одинаковое отображение в памяти как в управляемом, так и в неуправляемом коде. Количество таких типов ограничено примитивными числовыми типами (например, `Int32`, `UInt64`, но не `Decimal`, который не является примитивом) и `IntPtr`/`UIntPtr`. Если структура состоит исключительно из непреобразуемых типов, то реализация `Equals` может выполнять эквивалент побайтового сравнения памяти по всей структуре. В противном случае нужно всегда собственноручно реализовывать `Equals`.

Если просто переопределить `Equals(object other)`, то производительность все равно получится ниже необходимой, потому что этот метод включает в себя приведение и упаковку типов значений. Вместо этого нужно реализовать метод `Equals(T other)`, где `T` — тип вашей структуры. Именно для этого предназначен интерфейс `IEquatable<T>`, и все структуры должны его реализовывать. В ходе компиляции компилятор по возможности отдаст предпочтение более строго типизированной версии. Пример показан в следующем фрагменте кода:

```
struct Vector : IEquatable <Vector >
{
    public int X { get; }
    public int Y { get; }
    public int Z { get; }

    public int Magnitude { get; }

    public Vector(int x, int y, int z, int magnitude)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
        this.Magnitude = magnitude;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
        {
            return false;
        }
        if (obj.GetType() != this.GetType())
```

```
        {
            return false;
        }
        return this.Equals((Vector)obj);
    }

    public bool Equals(Vector other)
    {
        return this.X == other.X
            && this.Y == other.Y
            && this.Z == other.Z
            && this.Magnitude == other.Magnitude;
    }

    public override int GetHashCode()
    {
        return X ^ Y ^ Z ^ Magnitude;
    }
}
```

Если тип реализует метод `IEquatable<T>`, обобщенные коллекции среды .NET обнаружат его присутствие и воспользуются им для выполнения более эффективных поиска и сортировки.

Стоит также реализовать для значимых типов операторы `==` и `!=` и сделать так, чтобы они вызывали уже существующий метод `Equals<T>`.

Все эти методы следует реализовывать оптимально. В них должно быть минимальное количество операций и не должно быть повторений и выделений памяти. Они будут вызываться во многих непредвиденных ситуациях. Для работы с большими коллекциями их нужно приспособить под вызовы миллионы раз в секунду. Кроме того, во многих коллекциях, чтобы очень быстро сузить диапазон записей, необходимых для проверки равенства, используется метод `GetHashCode`. Если вычисление хеш-кода порождает слишком большое количество коллизий, то потенциально более затратный метод `Equals` будет вызываться слишком часто.

Если ваш тип пригоден для сортировки, нужно также реализовать интерфейс `IComparable<T>`, чтобы позволить методу `Sort` некоторых типов коллекций использовать его автоматически.

Даже если вам никогда не приходится сравнивать структуры или помещать их в коллекции, я все же предлагаю реализовать эти методы. Ведь нельзя знать наперед, как они будут использоваться, а на создание этих методов уйдет всего несколько минут времени и немного байтов IL, которые даже никогда не будут подвергнуты JIT-компиляции.

Переопределение `Equals` и `GetHashCode` для классов не настолько важно, поскольку по умолчанию они определяют равенство лишь на основе их ссылок на объекты. Пока это предположение вполне резонно для ваших объектов, можно оставить методы в исходной реализации.

Потоковая безопасность

О потоковой безопасности классов вспоминают довольно редко — если только для этого есть веские основания. Соответствующие средства редко встречаются за пределами классов коллекций, и, как будет показано при их рассмотрении, даже здесь этот вопрос требует тщательной проработки.

В большинстве случаев синхронизация должна происходить на более высоком уровне, а сам класс может ничего не знать о ней. Это придает наибольшую гибкость повторному использованию класса.

Исключение — статические классы. Они имеют только глобальное состояние, поэтому нужно с самого начала позаботиться об их потоковой безопасности, если не будет причин для обратного.

Более подробно синхронизация потоков рассматривалась в главе 4.

Кортежи

Обобщенный класс `System.Tuple` можно использовать для создания простых структур данных, обходясь без явных именованных классов. Кортеж представляет собой ссылочный тип, следовательно, на него распространяются все издержки, связанные с классами. С выходом .NET 4.7 и C# 7 появилась версия кортежей значимого типа `System.ValueTuple`. Именно ей в большинстве случаев следует отдавать предпочтение, при этом выбирать между конструкциями на основе ссылочного или значимого типа нужно, руководствуясь теми же соображениями, которые были изложены ранее.

```
var tuple = new ValueTuple <int, string >(1, "Ben");
int id = tuple.Item1;
```

Для объявления кортежей можно наряду с новым типом воспользоваться новыми элементами синтаксиса языка:

```
(int, string) tuple = (1, "Ben");
int id = tuple.Item1;
```

Вместо использования свойств с именами `Item` теперь им можно давать названия:

```
(int id, string name) tuple = (1, name: "Ben");
int id = tuple.id;
```

Этот синтаксис можно применять в качестве возвращаемого методом значения или в качестве типа параметра — все это эквивалентно использованию `ValueTuple`. Если посмотреть на эти значения в отладчике, то нельзя будет увидеть имена свойств, которые, возможно, были задействованы, а только `Item1`, `Item2` и т. д.

Диспетчеризация интерфейсов

При первом вызове метода через интерфейс среда .NET должна определить, какой метод какого типа вызывать. Сначала будет вызвана заглушка, которая находит нужный метод объекта, реализующего этот интерфейс. Когда такое случится несколько раз, среда CLR распознает, что вызывался один и тот же конкретный тип, и этот непрямой вызов через заглушку будет сокращен до заглушки, состоящей всего из нескольких ассемблерных инструкций, совершающих прямой вызов нужного метода. Эта группа инструкций называется мономорфной заглушкой, поскольку в ее ведении только вызов метода для одного типа. Это идеальный вариант для ситуаций, когда при вызове метода интерфейса неизменно вызываются конкретные методы одного и того же типа.

Мономорфная заглушка способна определить, когда она перестала быть верной. Если вдруг место вызова станет использовать объект другого типа, то в конечном итоге среда CLR заменит заглушку другой мономорфной заглушкой для нового типа.

Если ситуация осложнится еще больше и придется иметь дело с несколькими менее предсказуемыми типами (например, при наличии массива интерфейсного типа, но с несколькими конкретными типами в этом массиве), заглушка будет заменена полиморфной заглушкой, использующей для выбора вызываемого метода хеш-таблицу. Поиск в таблице выполняется довольно быстро, но все же работа идет медленнее, чем при задействовании мономорфной заглушки. К тому же размер хеш-таблицы очень ограничен, и при наличии слишком большого числа типов, возможно, придется вернуться к общему механизму поиска типа. Это может обойтись весьма дорого.

Заглушки создаются для каждого места вызова, то есть непосредственно там, откуда вызываются методы. Каждое место вызова при необходимости обновляется, причем независимо от других мест вызова.

В случае, когда производительность этого механизма является поводом для беспокойства, есть два варианта действий.

- Постарайтесь не вызывать эти объекты через общий интерфейс.
- Выберите общий базовый интерфейс и замените его абстрактным базовым классом.

Такая проблема возникает нечасто, но с ней можно столкнуться при наличии слишком большой иерархии типов с повсеместной реализацией общего интерфейса и вызове методов через этот корневой интерфейс. Для вас все это проявилось бы в виде высокого и необъяснимого объема задействованности центрального процессора на месте вызова этих методов.

ИСТОРИЯ

В ходе разработки крупной системы мы знали, что у нас потенциально будут тысячи типов, которые, скорее всего, стали бы производными общего типа. Мы знали, что будет пара мест, где потребуется обращение к ним из базового типа. Поскольку в команде был специалист, понимающий потенциальные проблемы диспетчеризации интерфейсов на подобных масштабах, мы решили использовать общий базовый абстрактный класс вместо интерфейса.

Для более глубокого изучения диспетчеризации интерфейсов обратитесь к записи блога Вэнса Моррисона (Vance Morrison) на эту тему под названием *Digging into interface calls in the .NET Framework: Stub-based dispatch*.

Избегайте упаковки

Упаковка представляет собой процесс заключения значимого типа, например примитива или структуры, в объект, который находится в куче, что позволяет передавать его методам, требующим ссылок на объект. В результате распаковки получается исходное значение в первозданном виде.

Упаковка тратит процессорное время на связанное с объектом выделение памяти, копирование и приведение типа, но куда существеннее то, что это приводит к увеличению нагрузки на сборку мусора в куче. Небрежность по отношению к упаковке может привести к слишком большому количеству выделений памяти, каждое из которых придется обрабатывать сборщику мусора.

Явная упаковка происходит при каждом подобном действии:

```
int x = 32;
object o = x;
```

На языке IL это выглядит так:

```
IL_0001: ldc.i4.s 32
IL_0003: stloc.0
IL_0004: ldloc.0
IL_0005: box [mscorlib]System.Int32
IL_000a: stloc.1
```

Это означает, что найти большинство источников упаковки в вашем коде довольно легко — нужно просто воспользоваться ILDASM для преобразования всего IL-кода в текст и выполнить поиск.

Зачастую случайная упаковка получается при использовании API, получающих в качестве параметра `object` или `object[]`. Наиболее известными примерами могут послужить `String.Format` или устаревшие коллекции, сохраняющие только ссылки на объекты, и от них нужно полностью избавиться по этой и ряду других причин (см. главу 6).

Упаковка может произойти также при присваивании структуре ссылки на интерфейс, например:

```
interface INameable
{
    string Name { get; set; }
}

struct Foo : INameable
{
    public string Name { get; set; }
}

void TestBoxing()
```

```
{
    Foo foo = new Foo() { Name = "Bar" };
    // Это подвергается упаковке!
    INameable nameable = foo;
    ...
}
```

Если вы сами захотите протестировать этот код, учтите, что в случае, когда упакованная переменная фактически не используется, компилятор в целях оптимизации удалит инструкцию упаковки, поскольку она никогда не задействуется. Как только будет вызван какой-нибудь метод или же значение будет использовано иным образом, инструкция упаковки окажется в IL.

Нужно знать еще об одном эффекте применения упаковки, который становится виден в результате выполнения следующего кода:

```
int val = 13;
object boxedVal = val;
val = 14;
```

Каким будет значение `boxedVal` в результате?

Упаковка похожа на использование ссылок-псевдонимов, но на самом деле происходит копирование значения и между двумя значениями больше нет никакой связи. В данном примере переменная `val` меняет значение на `14`, но в `boxedVal` сохраняется ее исходное значение `13`.

Иногда выполнение упаковки можно отловить в профиле центрального процессора, но вызовы упаковки часто встраиваются в код, так что это весьма ненадежный метод ее поиска. Признаками чрезмерной упаковки в профиле центрального процессора могут послужить интенсивные выделения памяти через ключевое слово `new`.

При большом количестве упакованных структур и невозможности избавиться от них нужно, вероятно, просто преобразовать структуру в класс, что может в целом потребовать меньших затрат.

И наконец, следует заметить, что передача значения по ссылке — это не упаковка. Изучите IL и увидите, что упаковки не произошло. Методу передан адрес значимого типа.

Возвращения по ссылке (ref) и локальные значения

В C# 7 появились новые элементы синтаксиса языка, позволяющие гораздо проще получать прямой доступ к памяти в безопасном коде. Этими же преимуществами можно было пользоваться и ранее с помощью доступа через указатель к закрытым полям в небезопасном коде, но стандартный способ программирования, как будет показано далее в этом разделе, обычно приводил бы к копированию значений.

Используя возвращение по указателю, можно получить преимущества от применения совершенно безопасного кода, использования правильных абстракций, а также повысить производительность в результате прямого доступа к памяти.

В качестве примера рассмотрим локальную ссылку `ref` на существующее значение:

```
int value = 13;
ref int refValue = value;

refValue = 14;
```

Что окажется в переменной `value` после выполнения последней строки кода? В ней будет `14`, поскольку `refValue` фактически ссылается на место в памяти значения переменной `value`.

Этой функциональной возможностью можно воспользоваться для получения ссылки на закрытые данные класса:

```
class Vector
{
    private int magnitude;
    public ref int Magnitude {
        get { ref return this.magnitude; } }
}

class Program
{
    void TestMagnitude()
    {
        Vector v = new Vector;
        ref int mag = ref v.Magnitude;
        mag = 3;

        int nonRefMag = v.Magnitude;
        mag = 4;

        Console.WriteLine($"mag: {mag}");
        Console.WriteLine($"nonRefMag: {nonRefMag}");
    }
}
```

Каким будет вывод этой программы?

```
4
3
```

Первым присваиванием изменяется значение под указателем. Нам интересно присваивание, касающееся `nonRefMag`. Несмотря на то что `Magnitude` является свойством с возвращением по ссылке, поскольку оно не вызывалось через `ref`, переменная `nonMagRef` просто получит копию значения, как будто `Magnitude` было

обычным свойством. Таким образом nonRefMag сохраняет первоначально полученное значение, несмотря на то что память базового класса была изменена. Запомните, что способ вызова метода не менее важен, чем способ его объявления.

Можно также использовать ref для ссылки на конкретное место в массиве. Следующий пример является методом, обнуляющим среднюю позицию в массиве. Добиться этого без ref будет можно примерно так:

```
private static void ZeroMiddleValue(int[] arr)
{
    int midIndex = GetMidIndex(arr);
    arr[midIndex] = 0;
}

private static int GetMidIndex(int[] arr)
{
    return arr.Length / 2;
}
```

Версия с использованием ref очень похожа:

```
private static void RefZeroMiddleValue(int[] arr)
{
    ref int middle = ref GetRefToMiddle(arr);
    middle = 0;
}

private static ref int GetRefToMiddle(int[] arr)
{
    return ref arr[arr.Length / 2];
}
```

Функциональные возможности возвращения по ссылке позволяют применять ранее недопустимые операции наподобие помещения метода в левую часть присваивания:

```
GetRefToMiddle(arr) = 0
```

Поскольку GetRefToMiddle возвращает ссылку, а не значение, допустимо что-нибудь ей присваивать.

Глядя на эти простые примеры, сложно понять, где же здесь большой прирост производительности. Для небольших фрагментов с одноразовым применением его и нет. Прирост дадут повторяющиеся ссылки на одно и то же место в памяти, позволяющие избегать математических вычислений смещения в массиве или не копировать значения.

Более эффектным примером послужит использование возвращения по ссылке для того, чтобы избежать копирования значений структуры, когда не получается использовать неизменяемую структуру. Рассмотрим следующие определения:

```
struct Point3d
{
    public double x;
```

```
    public double y;
    public double z;

    public string Name { get; set; }
}

class Vector
{
    private Point3d location;
    public Point3d Location { get; set; }
    public ref Point3d RefLocation
        { get { return ref this.location; } }
    public int Magnitude { get; set; }
}
```

Предположим, что нужно изменить расположение `location` на начало координат (0, 0, 0). Без возвращения по ссылке пришлось бы скопировать структуру, используя свойство `Location`, установить ее поля в `0`, после чего вызвать `set`-метод (сеттер), чтобы значение попало в нужное место:

```
private static void SetVectorToOrigin(Vector vector)
{
    Point3d location = vector.Location;
    pt.x = 0;
    pt.y = 0;
    pt.z = 0;
    vector.Location = pt;
}
```

Используя возвращение по ссылке, копирования можно избежать:

```
private static void RefSetVectorToOrigin(Vector vector)
{
    ref Point3d location = ref vector.RefLocation;
    location.x = 0;
    location.y = 0;
    location.z = 0;
}
```

Разница в эффективности будет зависеть от размера структуры — чем она больше, тем медленнее будет выполняться не-`ref`-версия этого метода.

Проект RefReturn, включенный в исходный код книги, содержит простой эталонный тест показанного ранее кода, дающий на выходе следующую информацию:

```
Benchmarks:
SetVectorToOrigin: 40ms
RefSetVectorToOrigin: 20ms
```

Если к структуре добавить всего несколько полей, разница станет заметнее:

```
Benchmarks:
SetVectorToOrigin: 470ms
RefSetVectorToOrigin: 20ms
```

Углубившись в ассемблерный код, можно увидеть, что в неэффективной версии имеются инструкции для копирования, а также вызов метода:

```
02E005A8  push    esi
02E005A9  cmp     al,byte ptr [ecx+24h]
02E005AC  lea     esi,[ecx+24h]
02E005AF  mov     eax,dword ptr [esi+18h]
02E005B2  fldz
02E005B4  fldz
02E005B6  fldz
02E005B8  lea     esi,[ecx+24h]
02E005BB  fxch    st(2)
02E005BD  fstp    qword ptr [esi]
02E005BF  fstp    qword ptr [esi+8]
02E005C2  fstp    qword ptr [esi+10h]
02E005C5  lea     edx,[esi+18h]
02E005C8  call    72BDDCB8
02E005CD  pop     esi
02E005CE  ret
```

А вот в версии с использованием возвращения по ссылке содержится всего лишь установка значений, а в качестве бонуса приведу все во встроенном виде:

```
02E005E0  cmp     byte ptr [ecx],al
02E005E2  lea     eax,[ecx+8]
02E005E5  fldz
02E005E7  fstp    qword ptr [eax]
02E005E9  fldz
02E005EB  fstp    qword ptr [eax+8]
02E005EE  fldz
02E005F0  fstp    qword ptr [eax+10h]
02E005F3  ret
```

Возможность использовать возвращение по ссылке регламентируют строгие правила.

- Результат обычного метода (то есть метода без возвращения по ссылке) не может быть присвоен локальной ссылочной переменной. Но значения, возвращенные по ссылке, могут быть неявно скопированы в нессылочные переменные.
- Нельзя возвратить ссылку на локальную переменную. Чтобы избежать недопустимого обращения к памяти, последняя должна оставаться за пределами локальной области видимости.
- После инициализации ссылочной переменной нельзя присваивать новое место в памяти.
- Методы структуры не могут возвращать по ссылке поля экземпляров.
- Эту функциональную возможность нельзя использовать с методами `async`.

Вряд ли вы будете часто пользоваться этой функцией, но иногда в ней возникает потребность, особенно в перечисленных далее ситуациях:

- при изменении полей в структуре, выставленной через свойство;
- при прямом доступе к расположению массива;
- при повторяющемся доступе к одному и тому же месту в памяти.

for или foreach

Инструкция `foreach` обеспечивает весьма удобный способ последовательного перебора любого перечисляемого типа коллекции, от массивов до словарей.

Разницу в последовательном переборе коллекций с использованием циклов `for` и `foreach` можно увидеть, применив инструментальное средство MeasureIt, которое упоминалось в главе 1. Стандартные циклы `for` во всех случаях работают существенно быстрее. Но при выполнении собственного простого теста можно, в зависимости от сценария, увидеть одинаковую производительность. В некоторых случаях среда .NET преобразует простые инструкции `foreach` в стандартные циклы `for`.

Посмотрите на учебный проект ForEachVsFor, в котором есть такой код:

```
int[] arr = new int[100];
for (int i = 0; i < arr.Length; i++)
{
    arr[i] = i;
}

int sum = 0;
foreach (int val in arr)
{
    sum += val;
}

sum = 0;
IEnumerable <int> arrEnum = arr;
foreach (int val in arrEnum)
{
    sum += val;
}
```

После его сборки и последующей декомпиляции с использованием инструмента отражения IL вы увидите, что первая инструкция `foreach` скомпилировалась в виде цикла `for`. Код IL имеет следующий вид:

```
// loop start (head: IL_0034)
IL_0024: ldloc.s CS$6$0000
IL_0026: ldloc.s CS$7$0001
IL_0028: ldelem.i4
```

```
IL_0029: stloc.3
IL_002a: ldloc.2
IL_002b: ldloc.3
IL_002c: add
IL_002d: stloc.2
IL_002e: ldloc.s CS$7$0001
IL_0030: ldc.i4.1
IL_0031: add
IL_0032: stloc.s CS$7$0001
IL_0034: ldloc.s CS$7$0001
IL_0036: ldloc.s CS$6$0000
IL_0038: ldlen
IL_0039: conv.i4
IL_003a: blt.s IL_0024
// end loop
```

Здесь присутствует довольно много сохранений, загрузок, добавлений, а также ветвление — все довольно просто. Но как только мы приведем массив к типу `IEnumerable<int>` и проделаем с ним то же самое, затраты существенно возрастут:

```
IL_0043: callvirt instance class
    [mscorlib]System.Collections.Generic.IEnumerator '1<!0>
    class [mscorlib]System.Collections.Generic.IEnumerable '1<int32 >
        ::GetEnumerator()
IL_0048: stloc.s CS$5$0002
.try
{
    IL_004a: br.s IL_005a
    // loop start (head: IL_005a)
        IL_004c: ldloc.s CS$5$0002
        IL_004e: callvirt instance !0 class [mscorlib]
            System.Collections.Generic.IEnumerator '1<int32 >
                ::get_Current()
        IL_0053: stloc.s val
        IL_0055: ldloc.2
        IL_0056: ldloc.s val
        IL_0058: add
        IL_0059: stloc.2
        IL_005a: ldloc.s CS$5$0002
        IL_005c: callvirt instance bool
             [mscorlib]System.Collections.IEnumerator::MoveNext()
        IL_0061: brtrue.s IL_004c
    // end loop

    IL_0063: leave.s IL_0071
} // end .try
finally
{
    IL_0065: ldloc.s CS$5$0002
    IL_0067: brfalse.s IL_0070

    IL_0069: ldloc.s CS$5$0002
```

```
    IL_006b: callvirt instance void
         [mscorlib]System.IDisposable::Dispose()

    IL_0070: endfinally
} // end handler
```

У нас появилось четыре вызова виртуального метода, пара `try-finally` и не показанное здесь выделение памяти для переменной локального нумератора, отслеживающего состояние перебора. На это затрачивается намного больше ресурсов, чем на простой цикл `for`, больше времени центрального процессора и больше памяти!

Следует помнить, что исходная структура данных по-прежнему является массивом (что допускает использование цикла `for`), но мы все запутали приведением к типу `IEnumerable`. Отсюда можно вынести важный урок — один из тех, что упоминался в начале главы, — всесторонняя оптимизация производительности зачастую сводит на нет попытки использовать программные абстракции. Инструкция `foreach` является абстракцией цикла, а `IEnumerable` является абстракцией коллекции. В сочетании они диктуют поведение, которое не дает произвести простую оптимизацию, использующую цикл `for` для перебора массива.

Приведение типов

В большинстве случаев приведения типов следует избегать. Зачастую оно свидетельствует о недостаточной проработке архитектуры классов, но временами без него не обойтись. Например, довольно часто потребность в приведении типов возникает при выполнении преобразований между беззнаковыми и знаковыми целыми числами из-за специфичных требований некоторых API. Приведение типов объектов должно происходить гораздо реже.

Приведение типов объектов без затрат не обходится, но они сильно разнятся в зависимости от связи объектов. Приведение типа объекта к типу его родительского объекта относительно дешево. Приведение типа родительского объекта к допустимому типу дочернего объекта обходится намного дороже, и чем глубже иерархия, тем выше затраты. Приведение типа к интерфейсу обходится дороже, чем приведение к конкретному типу.

А вот недопустимое приведение следует исключить. Иначе будет выдано исключение типа `InvalidCastException`, которое на несколько порядков больше затрат на фактическое приведение типов.

Посмотрите на код учебного проекта CastingPerf (находится в сопроводительном исходном коде), где выполняется эталонное тестирование нескольких разновидностей приведения типов. После запуска одного теста на моем компьютере на выходе была получена следующая информация:

```
No cast: 1.00x
Up cast (1 gen): 1.00x
Up cast (2 gens): 1.00x
```

```
Up cast (3 gens): 1.00x
Down cast (1 gen): 1.25x
Down cast (2 gens): 1.37x
Down cast (3 gens): 1.37x
Interface: 2.73x
Invalid Cast: 14934.51x
as (success): 1.01x
as (failure): 2.60x
is (success): 2.00x
is (failure): 1.98x
```

Оператор `is` является приведением типа, тестирующим результат и возвращающим булево значение. Оператор `as` похож на стандартное приведение типа, но возвращает `null`, если выполнить приведение не удалось. Показанные ранее результаты говорят о том, что приведение выполняется значительно быстрее выдачи исключения.

Никогда не используйте следующую схему, в которой выполняются два приведения типа:

```
if (a is Foo)
{
    Foo f = (Foo)a;
}
```

Вместо нее примените для приведения типов `as` и кэшируйте результат, после чего проверяйте возвращаемое значение:

```
Foo f = a as Foo;
if (f != null)
{
    ...
}
```

Если придется выполнять проверку сразу для нескольких типов, наиболее часто встречающийся тип нужно поставить первым.

ПРИМЕЧАНИЕ

Мне регулярно приходится сталкиваться с одним весьма неприятным приведением типов, связанным с использованием MemoryStream.Length, тип которого — long. Большинство задействующих его API используют ссылку на исходный буфер (извлекаемый методом MemoryStream.GetBuffer), смещение и длину, тип которой зачастую int, то есть возникает необходимость приведения long к нижестоящему типу. Приведение типов подобного рода весьма распространено, и его не избежать.

Следует заметить, что не все приведения типов являются явными. В зависимости от реализации классов могут встречаться и неявные приведения типов, в результате которых могут происходить выделения памяти.

P/Invoke

Вызов платформы P/Invoke используется для совершения вызовов из управляемого кода в неуправляемые платформенно-ориентированные методы. Это влечет за собой ряд фиксированных издержек плюс затраты на маршализацию аргументов. Под маршализацией понимается преобразование типов из одного формата в другой.

Внутренняя реализация P/Invoke довольно изощрена, что необходимо для его корректной работы. Примерный план его работы выглядит следующим образом.

1. Подстроить переменные фрейма стека.
2. Установить текущий фрейм стека.
3. Отключить сборку мусора для текущего потока.
4. Выполнить целевой код.
5. Снова включить сборку мусора.
6. Проверить, не запущена ли текущая сборка мусора, и при необходимости остановить поток.
7. Перенастроить переменные фрейма стека на их предыдущие значения.

Используя программу MeasureIt, упомянутую в главе 1, можно посмотреть результаты сравнительного эталонного тестирования затрат на вызов P/Invoke и вызов обычной управляемой функции. На моем компьютере вызов P/Invoke занимал в 6–10 раз больше времени, чем вызов пустого статического метода. Не стоит вызывать через P/Invoke метод в коротком цикле, если есть его управляемый эквивалент, и определенно следует избегать совершения множественных переходов между неуправляемым и управляемым кодом. Но одиночные вызовы P/Invoke не настолько затратны, чтобы запрещать их во всех случаях.

Есть несколько способов минимизации затрат на совершение вызовов P/Invoke.

1. Избегайте использования слишком «болтливых» интерфейсов. Совершайте один вызов, способный работать с большим количеством данных, в ходе которого время, затрачиваемое на обработку данных, существенно превышает фиксированные издержки на P/Invoke-вызов.
2. Как можно чаще задействуйте непреобразуемые (blittable) типы. Вспомните обсуждение структур: к ним относятся типы, имеющие одинаковые двоичные значения как в управляемом, так и в неуправляемом коде, — в основном это числовые типы и типы указателей. Они являются наиболее эффективными аргументами для передачи, поскольку процесс маршализации по своей сути сводится к копированию памяти.
3. Избегайте вызовов ANSI-версий API Windows. Например, `CreateProcess` фактически является макрокомандой, разрешаемой в одну из двух настоящих функций — `CreateProcessA` для ANSI-строк и `CreateProcessW` для Unicode-строк. Какая из версий будет получена, определяется настройками компиляции кода соответствующей платформы. Вам нужно гарантировать, что всегда будут вызваны Unicode-версии API, поскольку все .NET-строки уже имеют формат

Unicode и в случае возникновения здесь несоответствия не обойдется без затратного и, возможно, не обходящегося без потерь преобразования.

4. Избавьтесь от ненужных закреплений. Примитивы вообще никогда не закрепляются, а уровень маршализации будет автоматически закреплять строки и массивы примитивов. Если не требуется закреплять что-либо еще, избегайте продолжительных закреплений объекта, сводя их к минимуму. Сведения о негативном влиянии закреплений на сборку мусора можно найти в главе 2. При использовании закреплений вам придется выдержать баланс потребности в непродолжительности закрепления и уклонении от излишне «болтливых» интерфейсов. Во всех случаях возврат из неуправляемого кода должен происходить как можно быстрее.
5. Если нужно переместить большой объем данных в неуправляемый код, рассмотрите возможность закрепления буфера и непосредственной работы с ним из нативного кода. Буфер будет закреплен в памяти, но если функция работает довольно быстро, эффективность может оказаться выше, чем при копировании большого объема данных. Если есть возможность гарантировать присутствие буфера в поколении gen 2 или в куче больших объектов, тогда закрепление будет намного меньшей проблемой, поскольку сборке мусора вряд ли вообще понадобится куда-либо перемещать объект.
6. Снабдите параметры импортированного метода атрибутами `In` и `Out`. Это подскажет среде CLR, в каком направлении маршализировать каждый аргумент. Для многих типов, например целочисленных, это может определяться неявно, то есть вам не придется давать каких-либо конкретных указаний. Но для строк и массивов нужно давать явные установки, чтобы избежать ненужной маршализации в ненужном направлении.

Отключение проверок безопасности для надежного кода. Для кода, пользующегося абсолютным доверием, можно сократить расходы на P/Invoke, отключив ряд проверок безопасности в объявлениях методов P/Invoke:

```
[DllImport("kernel32.dll", SetLastError=true)]
[System.Security.SuppressUnmanagedCodeSecurity]
static extern bool GetThreadTimes(IntPtr hThread,
                                  out long lpCreationTime,
                                  out long lpExitTime,
                                  out long lpKernelTime,
                                  out long lpUserTime);
```

Атрибут `SuppressUnmanagedCodeSecurity` объявляет, что, запуская метод, можно ему полностью доверять. Это может вызвать получение предупреждений Code Analysis (FxCop), поскольку тем самым отключается довольно значительная часть модели безопасности, имеющейся в среде .NET. Отключать ее можно только при соблюдении следующих условий.

1. Ваше приложение запускает только код, пользующийся доверием.
2. Входные данные тщательно очищены или запускаются в доверенной среде.
3. Вы обеспечили, чтобы публичные API не делали вызовов P/Invoke.

Таким образом, если вы действительно можете отключить проверки безопасности, можно несколько увеличить производительность, как показано в выходных данных MeasureIt.

Имя	Значение
PInvoke: 10 FullTrustCall() (10 call average) [count=1000 scale=10.0]	6945
PInvoke: PartialTrustCall() (10 call average) [count=1000 scale=10.0]	17778

Запуск метода, пользующегося полным доверием, может протекать в 2,5 раза быстрее, чем прочих.

Делегаты

С использованием делегатов связаны два вида затрат: на их создание и вызов. К счастью, практически при любых обстоятельствах их вызов сравним с вызовом метода. Но делегаты — это объекты, и их создание может быть весьма затратным. Хотелось бы понести эти затраты только один раз, а результат кэшировать. Рассмотрим следующий код:

```
private delegate int MathOp(int x, int y);
private static int Add(int x, int y) { return x + y; }
private static int DoOperation(MathOp op, int x, int y)
  { return op(x, y); }
```

Какой из следующих циклов быстрее?

Вариант 1:

```
for (int i = 0; i < 10; i++)
{
    DoOperation(Add, 1, 2);
}
```

Вариант 2:

```
MathOp op = Add;
for (int i = 0; i < 10; i++)
{
    DoOperation(op, 1, 2);
}
```

Похоже, что во втором варианте создается лишь псевдоним функции `Add` с использованием локальной переменной-делегата, но фактически это приводит к едва заметным изменениям в поведении при выделении памяти! Это станет понятно, если рассмотреть IL-код для соответствующих циклов.

Вариант 1:

```
// loop start (head: IL_0020)
IL_0004: ldnull
IL_0005: ldftn int32 DelegateConstruction.Program
    ::Add(int32 , int32)
```

```
IL_000b: newobj instance void DelegateConstruction.Program/MathOp
    ::.ctor(object , native int)
IL_0010: ldc.i4.1
IL_0011: ldc.i4.2
IL_0012: call int32 DelegateConstruction.Program
           ::DoOperation(
              class DelegateConstruction.Program/MathOp ,
              int32 , int32)
...
```

И хотя в варианте 2 производится такое же выделение памяти, оно происходит за пределами цикла:

```
L_0025: ldnull
IL_0026: ldftn int32 DelegateConstruction.Program
  ::Add(int32 , int32)
IL_002c: newobj instance void DelegateConstruction.Program/MathOp
  ::.ctor(object , native int)
...
// loop start (head: IL_0047)
IL_0036: ldloc.1
IL_0037: ldc.i4.1
IL_0038: ldc.i4.2
IL_0039: call int32 DelegateConstruction.Program
    ::DoOperation(class DelegateConstruction.Program/MathOp ,
                  int32 , int32)
...
```

Обратите внимание на то, что команда `newobj` сместилась вверх, теперь она располагается выше запуска цикла. Ключевым в данном вопросе является то, что делегаты построены на объектах, подобных другим объектам .Net. Это же относится и к встроенному классу `Func`. Следовательно, если вы хотите избежать повторного выделения памяти под объекты-делегаты, ссылка на них должна делаться из такого места, которое, как показано в приведенном ранее примере, вызывается только один раз.

Есть способ обойти все это простым приемом — использованием лямбда-выражений.

Рассмотрим, что произойдет в следующем примере:

```
for (int i = 0; i < 10; i++)
{
    DoOperation((x,y) => Add(x,y), 1, 2);
}
```

Вот полученный код IL:

```
IL_004c: ldc.i4.0
IL_004d: stloc.3
IL_004e: br.s IL_007f
// loop start (head: IL_007f)
    IL_0050: ldsfld class DelegateConstruction.Program/MathOp
        DelegateConstruction.Program/'<>c'::'<>9__3_0'
```

```
    IL_0055: dup
    IL_0056: brtrue.s IL_006f

    IL_0058: pop
    IL_0059: ldsfld class DelegateConstruction.Program/'<>c'
        DelegateConstruction.Program/'<>c'::'<>9'
    IL_005e: ldftn instance int32
        DelegateConstruction.Program/'<>c'
        ::'<Main>b__3_0'(int32 , int32)
    IL_0064: newobj instance void
        DelegateConstruction.Program/MathOp
        ::.ctor(object , native int)
    IL_0069: dup
    IL_006a: stsfld class DelegateConstruction.Program/MathOp
        DelegateConstruction.Program/'<>c'::'<>9__3_0'

    IL_006f: ldc.i4.1
    IL_0070: ldc.i4.2
    IL_0071: call int32 DelegateConstruction.Program
        ::DoOperation(class DelegateConstruction.Program/MathOp ,
                      int32 , int32)
    ...
// end loop
```

Заметьте, что выделение памяти для делегата вернулось в цикл. Но посмотрите на строку `IL 0056` — вы обнаружите там инструкцию `brtrue`. В этой строке выполняется проверка на существование кэшированного делегата. Если он существует, выделение памяти будет пропущено и осуществлен непосредственный переход к выполнению операции. В цикле по-прежнему есть ряд дополнительных инструкций, но это все же лучше, чем выполнять выделение памяти при каждом проходе цикла.

Эквивалентом предыдущего примера является следующий синтаксис:

```
for (int i = 0; i < 10; i++)
{
DoOperation((x,y) => { return Add(x, y); }, 1, 2);
}
```

Эти примеры можно найти в учебном проекте DelegateConstruction.

Исключения

Заключение кода в блок `try` не приводит в среде .NET к особым затратам, но выдача исключений обходится очень дорого. Причина главным образом в весьма подробном состоянии, содержащемся в .NET-исключениях, куда входит и полный обзор стека. Исключения должны быть зарезервированы для действительно исключительных ситуаций, где производительность сама по себе не особо важна.

Никогда не следует полагаться на обработку исключений для отлавливания простых ошибок, которые более эффективно могут быть обработаны кодом без использования исключений. Намного лучше иметь проверочный код, способный

на простые проверки и возвращение ошибок вместо выдачи исключений. Это означает, что следует тщательно прорабатывать архитектуру API, чтобы обеспечить эффективность обработки ошибок.

Чтобы увидеть разрушительное влияние на производительность, которое может оказать выдача исключений, обратитесь к учебному проекту ExceptionCost. Его выходные данные должны быть похожими на следующие:

```
Empty Method: 1x
Exception (depth = 1): 8525.1x
Exception (depth = 2): 8889.1x
Exception (depth = 3): 8953.2x
Exception (depth = 4): 9261.9x
Exception (depth = 5): 11025.2x
Exception (depth = 6): 12732.8x
Exception (depth = 7): 10853.4x
Exception (depth = 8): 10337.8x
Exception (depth = 9): 11216.2x
Exception (depth = 10): 10983.8x
Exception (catchlist, depth = 1): 9021.9x
Exception (catchlist, depth = 2): 9475.9x
Exception (catchlist, depth = 3): 9406.7x
Exception (catchlist, depth = 4): 9680.5x
Exception (catchlist, depth = 5): 9884.9x
Exception (catchlist, depth = 6): 10114.6x
Exception (catchlist, depth = 7): 10530.2x
Exception (catchlist, depth = 8): 10557.0x
Exception (catchlist, depth = 9): 11444.0x
Exception (catchlist, depth = 10): 11256.9x
```

Эти данные свидетельствуют о трех простых фактах.

1. Метод, который выдает исключение, работает в тысячи раз медленнее простого пустого метода.
2. Чем глубже стек для выданного исключения, тем медленнее он становится (хотя все и так работает настолько медленно, что это уже неважно).
3. Влияние того, что инструкций `catch` несколько, небольшое, но все же ощутимое, поскольку приходится искать единственную нужную инструкцию.

Хотя перехват исключений способен выполняться без особых затрат, обращение к свойству `StackTrace` в объекте `Exception` может обойтись очень дорого, так как в результате этого происходит восстановление стека из указателей и перевод его в читаемый текст. В высокопроизводительных приложениях может оказаться полезным сделать логирование стеков исключений опциональным через параметр конфигурации и включать его только при необходимости. Заметьте, что повторная выдача существующего исключения из обработчика исключений обходится так же дорого, как и выдача нового исключения.

Повторю еще раз: выдача исключений должна обусловливаться по-настоящему исключительными обстоятельствами. Использование исключений как повседневная практика может иметь разрушительные последствия для производительности.

dynamic

Конечно, это должно быть понятно и без особых напоминаний, но все же проясню ситуацию: любой код, в котором используется ключевое слово `dynamic` или среда Dynamic Language Runtime (DLR), изначально не может быть оптимален. Настройка производительности зачастую сводится к избавлению от абстракций, в то время как использование DLR является добавлением еще одного очень большого уровня абстракций. Конечно, у нее есть своя область применения, но быстродействующие системы к ней не относятся.

При использовании ключевого слова `dynamic` все, что похоже на простой и понятный код, таковым уж точно не окажется. Рассмотрим простой искусственный пример:

```
static void Main(string[] args)
{
    int a = 13;
    int b = 14;

    int c = a + b;

    Console.WriteLine(c);
}
```

Код IL для него также выглядит довольно просто:

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2050
    // Code size 17 (0x11)
    .maxstack 2
    .entrypoint
    .locals init (
         [0] int32 a,
         [1] int32 b,
         [2] int32 c
    )

    IL_0000: ldc.i4.s 13
    IL_0002: stloc.0
    IL_0003: ldc.i4.s 14
    IL_0005: stloc.1
    IL_0006: ldloc.0
    IL_0007: ldloc.1
    IL_0008: add
    IL_0009: stloc.2
    IL_000a: ldloc.2
    IL_000b: call void [mscorlib]System.Console::WriteLine(int32)
    IL_0010: ret
} // end of method Program::Main
```

А теперь просто сделаем эти int-значения динамичными:

```
static void Main(string[] args)
{
    dynamic a = 13;
    dynamic b = 14;

    dynamic c = a + b;

    Console.WriteLine(c);
}
```

В целях экономии пространства я не стану здесь показывать код IL, но вот как выглядит его обратная конвертация в код C#:

```
private static void Main(string[] args)
{
object a = 13;
object b = 14;
if (Program.<Main>o__SiteContainer0.<>p__Site1 == null)
{
    Program.<Main>o__SiteContainer0.<>p__Site1 =
        CallSite <Func<CallSite , object , object , object >>.
        Create(Binder.BinaryOperation(CSharpBinderFlags.None,
                        ExpressionType.Add,
                        typeof(Program),
                        new CSharpArgumentInfo[]
    {
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                  null),
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                  null)
    }));
}
object c = Program.<Main>o__SiteContainer0.
    <>p__Site1.Target(Program.<Main>o__SiteContainer0.<>p__Site1 ,
                        a, b);
if (Program.<Main>o__SiteContainer0.<>p__Site2 == null)
{
    Program.<Main>o__SiteContainer0.<>p__Site2 =
        CallSite <Action <CallSite , Type, object >>.
        Create(Binder.InvokeMember(
                        CSharpBinderFlags.ResultDiscarded ,
                        "WriteLine",
                        null,
                        typeof(Program),
                        new CSharpArgumentInfo[]
    {
        CSharpArgumentInfo.Create(
            CSharpArgumentInfoFlags.UseCompileTimeType |
            CSharpArgumentInfoFlags.IsStaticType ,
            null),
```

```
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                  null)
    }));
}
Program.<Main>o__SiteContainer0.<>p__Site2.Target(
    Program.<Main>o__SiteContainer0.<>p__Site2 ,
    typeof(Console), c);
}
```

Даже вызов `WriteLine` сложен. Простой и понятный код превратился в мешанину из выделений памяти, делегатов, динамического вызова методов и этих странных объектов `CallSite`. Применение этих объектов — это прием DLR по замене стандартных вызовов методов динамически типизированными вызовами. Он создает оболочку для сложного кэша, чтобы исключить необходимость весьма тяжелого отражения при каждом вызове метода. Но это по-прежнему обходится весьма дорого.

JIT-статистика вполне предсказуема.

Версия	Время JIT-компиляции, мс	Размер кода IL, байт	Размер кода платформы, байт
Int	0,5	17	25
Dynamic	10,9	209	389

Я не хочу напрасно грешить на DLR. Это прекрасная среда для быстрой разработки и создания сценариев. Она открывает широкие возможности для взаимодействия динамических языков и среды .NET, но при этом не отличается быстротой работы.

Отражение

Отражением называется процесс программного сквозного перебора загруженных типов и изучения их метаданных. Оно может осуществляться также в отношении динамически загруженных .NET-сборок в ходе выполнения программы и выполнения методов найденных типов. При любых обстоятельствах этот процесс не отличается быстротой. Метаданные .NET-сборок организованы в основном не для повышения эффективности выполнения программы, а для загрузки, отладки и обеспечения доступа со стороны автономно работающих средств.

Получение информации обо всех типах в сборке в целом весьма эффективно — в любом случае это просто статичные метаданные о вашем процессе. К примеру, далее показан код, реализующий последовательный перебор всех типов в исполняемой сборке и выводящий имена методов этих типов:

```
foreach(var type in Assembly.GetExecutingAssembly().GetTypes())
{
    Console.WriteLine(type.Name);
```

```
    foreach(var method in type.GetMethods())
    {
        Console.WriteLine("\t" + method.Name);
    }
}
```

Эффективность уменьшается, как только начинаются динамическое выделение памяти и выполнение кода на основе этих метаданных. Чтобы показать, как в целом отражение работает в этом сценарии, рассмотрим простой код из учебного проекта ReflectionExe, загружающий в динамическом режиме сборку-«расширение»:

```
var assembly = Assembly.Load(extensionFile);

var types = assembly.GetTypes();
Type extensionType = null;
foreach (var type in types)
{
    var interfaceType = type.GetInterface("IExtension");
    if (interfaceType != null)
    {
        extensionType = type;
        break;
    }
}

object extensionObject = null;
if (extensionType != null)
{
    extensionObject = Activator.CreateInstance(extensionType);
}
```

В данный момент для выполнения кода в нашем расширении есть два варианта дальнейших действий. Чтобы остановиться на использовании чистого отражения, можно извлечь объект `MethodInfo` для метода, требующего выполнения, а затем вызвать этот метод:

```
MethodInfo executeMethod = extensionType.GetMethod("Execute");
executeMethod.Invoke(extensionObject , new object[] { 1, 2 });
```

Это очень медленный вариант, работающий примерно в 100 раз медленнее приведения типа объекта к интерфейсу и его непосредственного выполнения:

```
IExtension extensionViaInterface = extensionObject as IExtension;
extensionViaInterface.Execute(1, 2);
```

По возможности следует выполнять код именно таким образом, а не полагаться исключительно на `MethodInfo.Invoke`. Если же нет возможности воспользоваться общим интерфейсом, внимательно прочтите следующий раздел, посвященный генерации кода для выполнения загруженных в динамическом режиме сборок, позволяющей существенно ускорить работу по сравнению с применением отражения.

Генерация кода

Если придется иметь дело с динамически загружаемыми типами, например с моделью расширений или подключаемых модулей, нужно тщательно измерить производительность при взаимодействии с этими типами. В идеале взаимодействие с ними можно организовать через общий интерфейс и обойти стороной большинство проблем, связанных с динамически загружаемым кодом. Если воспользоваться этим подходом невозможно, то для обхода проблем с производительностью при вызове динамически загружаемого кода используйте методику, о которой пойдет речь в данном разделе.

В среде .NET Framework поддерживаются выделение памяти под динамический тип и вызов метода такого типа путем применения методов `Activator.CreateInstance` и `MethodInfo.Invoke` соответственно. Вот пример:

```
Assembly assembly = Assembly.Load("Extension.dll");
Type type = assembly.GetType("DynamicLoadExtension.Extension");
object instance = Activator.CreateInstance(type);

MethodInfo methodInfo = type.GetMethod("DoWork");
bool result = (bool)methodInfo.Invoke(instance , new object[]
      { argument });
```

Если этот прием используется от случая к случаю, волноваться не о чем, но если требуется выделить память под большой объем динамически загружаемых объектов или совершить множество динамических вызовов функций, он может стать по-настоящему узким местом вашей программы. Метод `Activator.CreateInstance` не только отнимает много времени у центрального процессора, но и может вызвать ненужные выделения памяти, добавляющие работы сборщику мусора. При задействовании значимых типов в качестве либо параметров функции, либо возвращаемого значения, как в приведенном ранее примере, возникает также вероятность упаковки.

Старайтесь скрывать эти обращения за интерфейсом, известным как расширению, так и исполняющей программе, о чем говорилось в предыдущем разделе. Если этот прием не сработает, может подойти вариант с генерацией кода. К счастью, сгенерировать код для выполнения этой задачи не составляет особого труда.

Создание шаблонов

Чтобы выяснить, какой именно код нужно сгенерировать, воспользуйтесь в качестве примера шаблоном, чтобы сгенерировать IL, который вам впоследствии надо будет воспроизвести. Пример такого подхода приведен в учебных проектах DynamicLoadExtension и DynamicLoadExecutor. В проекте DynamicLoadExecutor выполняется динамическая загрузка расширения, а затем выполняется метод `DoWork`. В проекте DynamicLoadExecutor обеспечивается подходящее размещение `DynamicLoadExtension.dll`. Это достигается заданием шагов после сборки (post-build

steps) и конфигурированием зависимостей при сборке решения вместо зависимостей на уровне проекта, чтобы гарантировать реальную динамическую загрузку и выполнение кода.

Начните с создания объекта расширения. Чтобы разработать шаблон, сначала разберитесь с тем, чего именно следует добиться. Вам требуется метод без параметров, возвращающий экземпляр нужного типа. Программа не будет знать о типе `Extension`, поэтому данный метод будет просто возвращать его в виде объекта. Этот метод похож на следующий:

```
object CreateNewExtensionTemplate()
{
    return new DynamicLoadExtension.Extension();
}
```

Взгляните на код IL, который будет выглядеть следующим образом:

```
IL_0000: newobj instance void
        [DynamicLoadExtension]DynamicLoadExtension.Extension
          ::.ctor()
IL_0005: ret
```

Создание делегата

Теперь можно создать экземпляр типа `System.Reflection.Emit.DynamicMethod`, программным способом добавив к нему несколько IL-инструкций, и присвоить его делегату, которым в дальнейшем можно повторно воспользоваться для создания новых объектов `Extension`.

```
private static T GenerateNewObjDelegate <T>(Type type)
    where T:class
{
    // Создание нового, не имеющего параметров
    // (что задается использованием Type.EmptyTypes)
    // динамического метода.
    var dynamicMethod = new DynamicMethod("Ctor_" + type.FullName ,
                                          type,
                                          Type.EmptyTypes ,
                                          true);
    var ilGenerator = dynamicMethod.GetILGenerator();

    // Поиск конструктора для создаваемого типа
    var ctorInfo = type.GetConstructor(Type.EmptyTypes);
    if (ctorInfo != null)
    {
        ilGenerator.Emit(OpCodes.Newobj , ctorInfo);
        ilGenerator.Emit(OpCodes.Ret);
        object del = dynamicMethod.CreateDelegate(typeof(T));
        return (T)del;
    }
    return null;
}
```

Можно заметить, что производимый код IL точно соответствует нашему шаблонному методу.

Чтобы воспользоваться данным кодом, нужно загрузить сборку расширения, извлечь подходящий тип и передать его методу-генератору:

```
Type type = assembly.GetType("DynamicLoadExtension.Extension");
Func<object > creationDel =
    GenerateNewObjDelegate <Func<object >>(type);
object extensionObj = creationDel();
```

Создав делегат, его можно поместить в кэш для повторного использования (возможно, снабдив ключом, в качестве которого можно выбрать объект `Type`, или применив какую-нибудь схему, подходящую вашему приложению).

Аргументы метода

Сгенерировать вызов метода `DoWork` можно с помощью этого же приема. Он лишь немного усложнится из-за необходимости приведения типа и аргументов метода. IL — стековый язык, поэтому аргументы функции перед ее вызовом должны быть помещены в стек в нужном порядке. Первым аргументом для вызова метода экземпляра должен быть скрытый параметр метода `this`, представляющий собой объект, в контексте которого работает данный метод. Примечательно, что из-за исключительного использования стека в языке IL этот язык не имеет никакого отношения к тому, как JIT-компилятор преобразует вызовы функций в ассемблерный код, где зачастую аргументы функции хранятся в регистрах процессора.

Как и при создании объекта, сначала следует создать шаблонный метод для использования в качестве основы для IL. Поскольку этот метод придется вызывать просто с параметром `object` (это все, чем мы будем располагать в программе), в параметрах функции расширение указывается как `object`. Это означает, что перед вызовом `DoWork` его придется приводить к нужному типу. В шаблоне информация о типе задана жестко, в генераторе же придется получать ее программным способом.

```
static bool CallMethodTemplate(object extensionObj ,
                               string argument)
{
    var extension = (DynamicLoadExtension.Extension)extensionObj;
    return extension.DoWork(argument);
}
```

Получившийся IL-код для этого шаблона выглядит следующим образом:

```
.locals init (
  [0] class [DynamicLoadExtension]DynamicLoadExtension.Extension
    extension
)
IL_0000: ldarg.0
IL_0001: castclass
  [DynamicLoadExtension]DynamicLoadExtension.Extension
IL_0006: stloc.0
IL_0007: ldloc.0
```

```
IL_0008: ldarg.1
IL_0009: callvirt instance bool
  [DynamicLoadExtension]DynamicLoadExtension.Extension
    ::DoWork(string)
IL_000e: ret
```

Обратите внимание на объявление локальной переменной. В ней сохраняется результат приведения типа. Позже будет показано, что ее можно убрать для оптимизации. Этот IL-код можно напрямую преобразовать в `DynamicMethod`:

```
private static T GenerateMethodCallDelegate <T>(
    MethodInfo methodInfo ,
    Type extensionType ,
    Type returnType ,
    Type[] parameterTypes) where T : class
{
    var dynamicMethod = new DynamicMethod(
                  "Invoke_" + methodInfo.Name,
                  returnType ,
                  parameterTypes ,
                  true);
    var ilGenerator = dynamicMethod.GetILGenerator();

    ilGenerator.DeclareLocal(extensionType);
    // параметр this, принадлежащий объекту
    ilGenerator.Emit(OpCodes.Ldarg_0);
    // приведение к нужному типу
    ilGenerator.Emit(OpCodes.Castclass , extensionType);
    // фактический аргумент метода
    ilGenerator.Emit(OpCodes.Stloc_0);
    ilGenerator.Emit(OpCodes.Ldloc_0);
    ilGenerator.Emit(OpCodes.Ldarg_1);
    ilGenerator.EmitCall(OpCodes.Callvirt , methodInfo , null);
    ilGenerator.Emit(OpCodes.Ret);

    object del = dynamicMethod.CreateDelegate(typeof(T));
    return (T)del;
}
```

Чтобы сгенерировать динамический метод, требуется `MethodInfo`, который можно найти в объекте `Type` для расширения. Кроме того, нужны `Type` возвращаемого объекта и `Type`-объекты всех параметров метода, включая подразумеваемый параметр `this` (совпадающий с `extensionType`).

Чтобы воспользоваться нашим делегатом, его нужно просто вызвать:

```
Func<object , string , bool> doWorkDel =
  GenerateMethodCallDelegate <
    Func<object , string , bool >>(
    methodInfo , type, typeof(bool),
    new Type[]
     { typeof(object), typeof(string) });

bool result = doWorkDel(extension , argument);
```

Оптимизация

Со своей задачей этот метод вполне справляется, но присмотритесь к тому, что он делает, и вспомните природу IL-инструкций, основанную на использовании стека. Этот метод работает следующим образом.

1. Объявляет локальную переменную.
2. Помещает `arg0` (`this`-указатель) в стек (`Ldarg_0`).
3. Приводит `arg0` к нужному типу и помещает результат в стек (`Castclass`).
4. Извлекает верхнее содержимое стека и сохраняет его в локальной переменной (`Stloc_0`).
5. Помещает локальную переменную в стек (`Ldloc_0`).
6. Помещает `arg1` (аргумент `string`) в стек (`Ldarg_1`).
7. Вызывает метод `DoWork` (`Callvirt`).
8. Возвращает управление.

Здесь бросается в глаза явная избыточность, в частности, с локальной переменной. В стеке имеется приведенный к нужному типу объект, мы его извлекаем оттуда, после чего снова помещаем в стек. Этот код IL можно оптимизировать, просто удалив все, что имеет отношение к локальной переменной. Вполне возможно, что все это в любом случае будет оптимизировано JIT-компилятором самостоятельно, но оптимизация все же не повредит, а даже поможет, если динамических методов сотни или тысячи и все они должны пройти JIT-компиляцию.

Еще одну оптимизацию можно обнаружить, осознав, что код операции `callvirt` может быть заменен простым кодом операции `call`, поскольку нам известно, что здесь нет виртуальных методов. Теперь код IL приобрел следующий вид:

```
var ilGenerator = dynamicMethod.GetILGenerator();

// параметр this, принадлежащий объекту
ilGenerator.Emit(OpCodes.Ldarg_0);
// приведение к нужному типу
ilGenerator.Emit(OpCodes.Castclass , extensionType);
// фактический аргумент метода
ilGenerator.Emit(OpCodes.Ldarg_1);
ilGenerator.EmitCall(OpCodes.Call, methodInfo , null);
ilGenerator.Emit(OpCodes.Ret);
```

Подведение итогов

А как у сгенерированного кода обстоят дела с производительностью? Вот как выглядит один из тестовых прогонов:

```
==CREATE INSTANCE==
Direct ctor: 1.0x
Activator.CreateInstance: 14.6x
```

```
Codegen: 3.0x

==METHOD INVOKE==
Direct method: 1.0x
MethodInfo.Invoke: 17.5x
Codegen: 1.3x
```

Сравнивая с непосредственными вызовами методов, можно заметить, что применение методов отражения дает худшие результаты. Сгенерированный код не выправляет ситуацию полностью, но очень близок к этому. Эти числовые показатели относятся к вызову функции, которая фактически ничего не делает, следовательно, они представляют собой чистые издержки на вызов функции. Назвать такую ситуацию вполне реалистичной нельзя. Если добавить какую-то минимальную работу (анализ строк и вычисление квадратного корня), числовые показатели немного изменятся:

```
==CREATE INSTANCE==
Direct ctor: 1.0x
Activator.CreateInstance: 9.3x
Codegen: 2.0x

==METHOD INVOKE==
Direct method: 1.0x
MethodInfo.Invoke: 3.0x
Codegen: 1.0x
```

Эти результаты наглядно демонстрируют, что если вы сейчас полагаетесь на использование `Activator.CreateInstance` или `MethodInfo.Invoke`, то вы можете получить существенные преимущества, перейдя на подход с генерацией кода.

ИСТОРИЯ

Мне пришлось работать над проектом, где эти приемы сократили издержки использования центрального процессора на вызов динамически загружаемого кода с более чем 10 % до примерно 0,1 %.

Генерацию кода можно задействовать и для решения других задач. Если в вашем приложении выполняется большая работа по преобразованию строк или используется какая-либо машина состояний, то этот функционал будет вполне подходящим кандидатом на применение генерации кода. Средой .NET этот прием используется в ходе работы с регулярными выражениями и XML-сериализацией.

Предварительная обработка

Если какая-то часть вашего приложения занимается чем-то исключительно важным с точки зрения производительности, нужно убедиться в том, что в ней не делается ничего постороннего или не тратится впустую время на обработку, которую можно было бы выполнить заранее. Если данные требуется преобразовать до того, как они

будут задействованы в ходе выполнения программы, убедитесь в том, что преобразования по максимуму делаются заранее и даже, если это возможно, в автономном процессе.

Иными словами, если что-то можно обработать предварительно, значит, так и нужно сделать. При этом для определения той доли обработки, которую реально сделать автономной, могут понадобиться творческий подход и нетривиальное мышление, но дело того стоит. С точки зрения повышения производительности это станет формой 100%-ной оптимизации за счет полного удаления кода.

Исследование проблем производительности

Каждая из тем, затрагиваемых в данной главе, требует особого подхода к решению проблем производительности. Можно воспользоваться инструментами, известными из предыдущих глав. Профилирование центрального процессора выявит затратные методы `Equals`, неудачное итерирование в цикле, низкую производительность маршализации в рамках организации взаимодействия компонентов и другие области неэффективной работы.

Трассировка памяти укажет на упаковку при выделении памяти под объекты, а общая трассировка .NET-событий покажет, где были выданы исключения, даже если они уже отловлены и обработаны.

Счетчики производительности

В категорию .NET CLR Interop включены следующие счетчики.

- # of CCWs — количество COM-вызываемых оболочек, или количество управляемых объектов, на которые имеется ссылка из неуправляемых COM-объектов.
- # of marshalling — количество случаев, когда производилась маршализация аргументов и возвращаемых значений заглушкой P/Invoke. Если эта заглушка оказывается встроенной (для малозатратных вызовов), значение не инкрементируется. Данной метрикой удобно пользоваться для того, чтобы отследить, насколько серьезно нагружены ваши вызовы P/Invoke.
- # of Stubs — количество заглушек, созданных JIT-компилятором, для маршалинга аргументов в P/Invoke или COM.

ETW-события

- `ExceptionThrown V1` — выданное исключение. Неважно, было это исключение обработано или нет. Поля включают:
  - `Exception Type` — тип исключения;
  - `Exception Message` — свойство `Message` из объекта исключения;
  - `EIPCodeThrow` — указатель на инструкцию в том месте, откуда выдано исключение;

- `ExceptionHR` — значение `HRESULT` исключения;
- `ExceptionFlags` (флаги исключения):
  - `0x01` — имеет внутреннее исключение;
  - `0x02` — вложенное исключение;
  - `0x04` — повторно выданное исключение;
  - `0x08` — исключение поврежденного состояния;
  - `0x10` — CLS-совместимое исключение.

Поиск инструкций упаковки

Просканировать код на наличие упаковки совсем не трудно, поскольку в IL есть конкретная инструкция под названием `box`. Чтобы найти ее в отдельно взятом методе или классе, воспользуйтесь одним из многих доступных IL-декомпиляторов и выберите просмотр IL-кода.

Обнаружить упаковку во всей сборке проще с помощью средства ILDASM, которое поставляется вместе с Windows SDK и располагает весьма богатым набором параметров командной строки.

Следующий пример используется для анализа файла `Boxing.exe` и выводит код IL в файл `output.txt`:

```
ildasm.exe /out=output.txt Boxing.exe
```

Обратите внимание на учебный проект Boxing, в котором показываются разные способы провоцирования упаковки. Если запустить ILDASM в отношении `Boxing.exe`, на выходе получится следующий код:

```
.method private hidebysig static void Main(string[] args)
    cil managed
{
.entrypoint
// Code size 98 (0x62)
.maxstack 3
.locals init ([0] int32 val,
         [1] object boxedVal ,
         [2] valuetype Boxing.Program/Foo foo,
         [3] class Boxing.Program/INameable nameable ,
         [4] int32 result ,
         [5] valuetype Boxing.Program/Foo '<>g__initLocal0')
IL_0000: ldc.i4.s 13
IL_0002: stloc.0
IL_0003: ldloc.0
IL_0004: box [mscorlib]System.Int32
IL_0009: stloc.1
IL_000a: ldc.i4.s 14
IL_000c: stloc.0
IL_000d: ldstr "val: {0}, boxedVal:{1}"
IL_0012: ldloc.0
IL_0013: box [mscorlib]System.Int32
IL_0018: ldloc.1
```

```
IL_0019: call string [mscorlib]System.String::Format(string ,
                        object ,
                        object)
IL_001e: pop
IL_001f: ldstr "Number of processes on machine: {0}"
IL_0024: call class [System]System.Diagnostics.Process[]
     [System]System.Diagnostics.Process::GetProcesses()
IL_0029: ldlen
IL_002a: conv.i4
IL_002b: box [mscorlib]System.Int32
IL_0030: call string [mscorlib]System.String::Format(string ,
                        object)
IL_0035: pop
IL_0036: ldloca.s '<>g__initLocal0'
IL_0038: initobj Boxing.Program/Foo
IL_003e: ldloca.s '<>g__initLocal0'
IL_0040: ldstr "Bar"
IL_0045: call instance void Boxing.Program/Foo
               ::set_Name(string)
IL_004a: ldloc.s '<>g__initLocal0'
IL_004c: stloc.2
IL_004d: ldloc.2
IL_004e: box Boxing.Program/Foo
IL_0053: stloc.3
IL_0054: ldloc.3
IL_0055: call void Boxing.Program::UseItem(
               class Boxing.Program/INameable)
IL_005a: ldloca.s result
IL_005c: call void Boxing.Program::GetIntByRef(int32&)
IL_0061: ret
} // end of method Program::Main
```

Также обнаружить упаковку можно опосредованно — с помощью PerfView. При трассировке центрального процессора обнаруживаются интенсивные вызовы функции `JIT_new` (рис. 5.1).

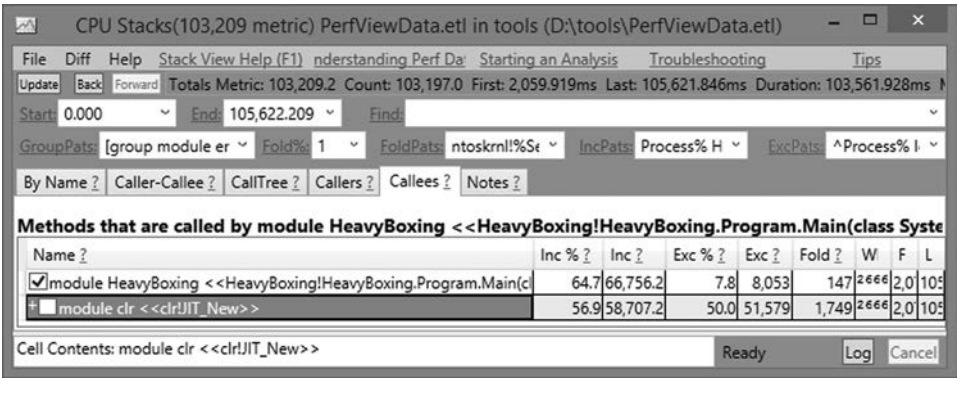

Рис. 5.1. Упаковка проявляется при трассировке ЦП в использовании метода JIT_New — стандартного метода выделения памяти

Чуть менее очевидным способом упаковку можно обнаружить и при изучении трассировки выделений памяти, поскольку вам известно, что значимые типы и примитивы вообще не потребуют таких выделений (рис. 5.2).

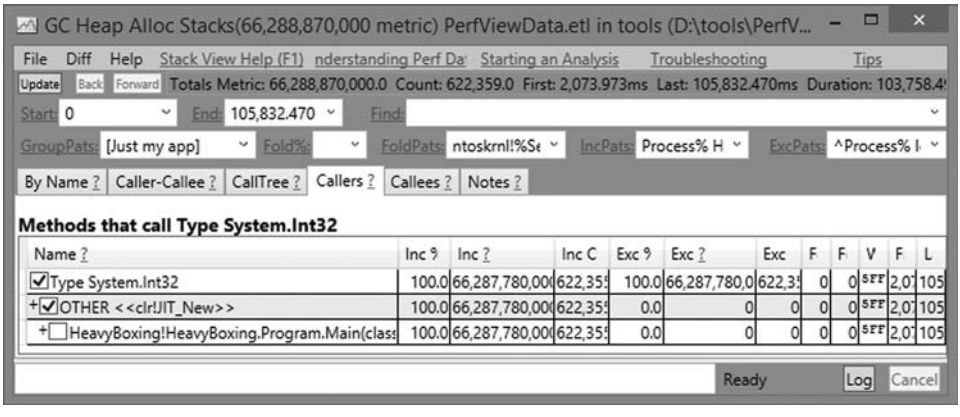

Рис. 5.2. В данной трассировке можно увидеть, что память для объекта типа Int32 была выделена с помощью метода new, что не может восприниматься как должное

Если действовать более прямолинейно, то любой упакованный объект можно найти в самой куче, воспользовавшись CLR MD:

```
private static void PrintBoxedObjects(ClrRuntime clr)
{
    foreach (var obj in clr.Heap.EnumerateObjects())
    {
        if (obj.IsBoxed)
        {
            Console.WriteLine(
                $"0x{obj.Address:x} - {obj.Type.Name}");
        }
    }
}
```

Обнаружение исключений первого шанса

Исключениями первого шанса (first-chance exceptions) на языке отладчика называются исключения, которые выявились до вызова или обнаружения любых возможных обработчиков исключений. Исключениями второго шанса считаются те, что обнаружены после безуспешного поиска обработчиков. Исключение второго шанса, скорее всего, приведет к сбою процесса.

WinDbg прервет работу на исключении второго шанса по умолчанию, а управлять прерыванием его работы на исключениях первого шанса можно с помощью

команды sx. Обработка исключений первого шанса в отношении CLR-исключений отключается командой:

```
sxd clr
```

А для повторного включения используется команда:

```
sxe clr
```

Средство PerfView может легко показать, какие исключения были выданы, независимо от того, были они отловлены или нет (рис. 5.3).

1. Соберите в PerfView .NET-события. Подойдут установки по умолчанию, но центральный процессор исследовать не понадобится, поэтому снимите его флажок, если нужно, чтобы профилирование заняло всего несколько минут.
2. По завершении сбора дважды щелкните кнопкой мыши на узле Exception Stacks (Стеки исключений).
3. Выберите из списка нужный процесс.
4. Представление By Name (По имени) покажет список главных исключений. Представление CallTree (Дерево вызовов) покажет стек для выбранного в данный момент исключения.

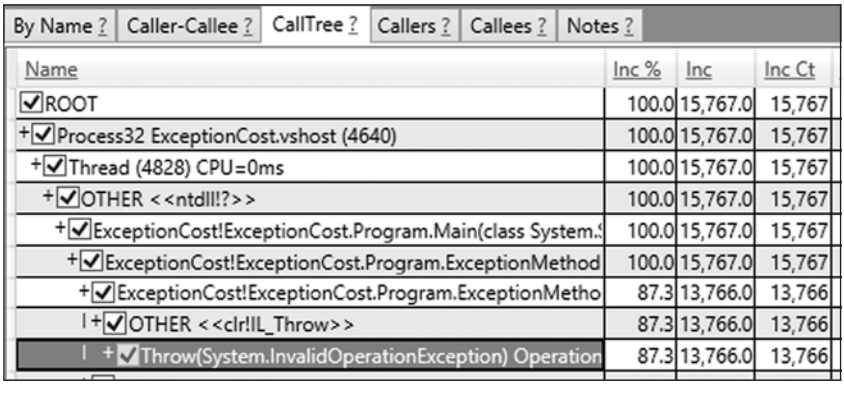

Рис. 5.3. PerfView позволяет легко отыскать, откуда пришли исключения

Резюме

Следует помнить, что углубленная оптимизация с целью повышения производительности противоречит программным абстракциям. Нужно понять, как ваш код будет транслирован в код IL, код ассемблера и машинные операции. Потратьте время, чтобы разобраться с каждым из этих уровней.

Когда объем данных относительно невелик, воспользуйтесь вместо классов структурами, если хотите свести к минимуму издержки или же использовать

массивы данных и получить оптимальную локальность в памяти. Рассмотрите возможность сделать структуры неизменяемыми и всегда реализовывайте для них `Equals`, `GetHashCode` и `IEquatable<T>`. Избегайте упаковки значимых типов и примитивов, защищая их от присваивания ссылкам на объекты.

Для безопасного непосредственного доступа к памяти при обращении к полям используйте возвращение по ссылке.

Сохраняйте высокую скорость последовательного перебора путем отказа от приведения типов коллекций к `IEnumerable`. И вообще по возможности избегайте приведения типов, особенно тех экземпляров, для которых при этом могут выдаваться исключения.

Сводите к минимуму вызовы P/Invoke, отправляя за один вызов как можно больше данных. Стремитесь сделать закрепления в памяти максимально скоротечными.

Если возникнет потребность в интенсивном применении `Activator.CreateInstance` или `MethodInfo.Invoke`, попробуйте задействовать вместо них генерацию кода.

6 Использование среды .NET Framework

В предыдущей главе рассматривались общие приемы и трудности программирования на .NET, в особенности те, которые имеют отношение к специфике языка. В этой главе речь пойдет о том, на что стоит обратить внимание при использовании обширной библиотеки кода, которая поставляется со средой .NET. Рассмотреть все многообразие подсистем и классов, являющихся частью .NET Framework, не представляется возможным, но целью этой главы является обеспечение вас инструментарием, необходимым для исследования приемов повышения производительности, и предоставление сведений о самых распространенных шаблонах, применения которых следует избегать.

Среда .NET Framework создавалась с прицелом на самую широкую аудиторию (фактически под всех разработчиков из всех областей) и задумывалась как универсальная среда, предоставляющая стабильный, правильный, надежный код, который способен справиться с множеством ситуаций. В ней как таковой не делается упор на исключительную производительность, и во внутренних циклах вашего кода найдется немало моментов, требующих доработки. Среда .NET должна работать для всех и везде, выстраивая предположения о вызывающем коде. Зачастую особое значение придается правильности и надежности, а не скорости и эффективности. Но в своем коде нередко можно добиться более впечатляющих успехов, осмыслив собственные ограничения и допущения и соответствующим образом адаптировав код. Это утверждение не дает вам права на переписывание класса `string` в новом проекте, но осведомляет вас об ограничениях среды в сочетании с критическими областями производительности вашего собственного кода.

Чтобы обойти недостатки среды .NET Framework или любой библиотеки стороннего производителя, может понадобиться проявить смекалку. Вот некоторые из возможных подходов.

- Воспользуйтесь альтернативным API с меньшими издержками.
- Переконструируйте свое приложение, чтобы реже вызывать API.
- Заново реализуйте некоторые API, добившись от них более высокой производительности.
- Для выполнения тех же задач перейдите к взаимодействию с API конкретной системы, предположив, что издержки на маршализацию будут ниже.

Разберитесь с каждым вызываемым API

Ведущий принцип этой главы: **необходимо понимать код, выполняющийся при каждом вызове API**.

Говорить о контроле производительности равнозначно тому, чтобы утверждать, что вы знаете код, который выполняется на каждом критическом пути программы. Непонятной библиотеки сторонних разработчиков во внутреннем цикле вашей программы быть не должно — это будет означать потерю контроля.

Доступ к исходному коду каждого вызываемого метода у вас будет не всегда (хотя всегда будет доступ к коду на уровне ассемблера!), но обычно все Windows API снабжаются качественной документацией. В среде .NET можно воспользоваться одним из многочисленных инструментов просмотра IL-кода, позволяющих увидеть, что делает эта среда (такая простота изучения не распространяется на саму среду CLR, которая, несмотря на свою доступность в качестве части ядра .NET Core, написана в основном на компактном, изобилующем макросами машинном коде).

Нужно привыкать к исследованию кода среды на предмет обнаружения чего-либо вам незнакомого. Чем более производительность важна для вас, тем больше вопросов вы должны задавать по поводу реализации сторонних API. Не забывайте, что ваша привередливость должна быть прямо пропорциональна требующейся скорости работы приложения.

Далее в главе рассматриваются несколько общих областей, к которым следует отнестись внимательно, а также некоторые конкретные общие классы, используемые каждой программой.

Множество API для решения одних и тех же задач

Временами встречаются ситуации, в которых можно выбирать, какой из множества API для решения одних и тех же задач использовать. Наглядным примером может послужить разбор XML. Конечно, разбирать XML — это никогда не быстрая работа, но, в зависимости от вашего сценария, некоторые из имеющихся вариантов решения этой задачи могут вам подойти лучше других. В среде .NET есть по крайней мере девять различных средств разбора XML:

- `XmlTextReader`;
- `XmlValidatingReader`;
- `XDocument`;
- `XmlDocument`;
- `XPathNavigator`;
- `XPathDocument`;
- `LINQ-to-XML`;
- `DataContractSerializer`;
- `XmlSerializer`.

Какой из них взять, зависит от таких факторов, как простота использования, продуктивность, целесообразность применения для данной задачи и производительность. Парсер `XmlTextReader` работает очень быстро, но он однонаправленный и не выполняет проверку. `XmlDocument` очень удобен, поскольку обладает полностью загруженной моделью объекта, но относится к самым медленным.

Этот подход приемлем как к разбору XML, так и к другим ситуациям с выбором API: не все варианты будут равноценны и рациональны с точки зрения достижения высокой производительности. Какие-то будут работать быстрее, но потреблять больше памяти. Какие-то обойдутся весьма скромным объемом памяти, но не позволят выполнять определенные операции. Придется определить набор нужных вам качеств и измерить производительность, чтобы выявить тот API, который обеспечивает разумный баланс функциональности и производительности. Необходимо создать прототипы для всех вариантов и провести их профилирование путем запуска на тестовых данных.

Коллекции

В .NET предоставляются свыше 20 встроенных типов коллекций, включая обобщенные версии многих популярных структур данных и версии, предназначенные для параллельной работы. Многим программам понадобится только лишь использование комбинации из существующих коллекций, а потребность в создании своих собственных будет возникать крайне редко.

Выбор коллекций зависит от множества факторов, включая семантическое значение предоставляемого API (помещение и извлечение, постановка в очередь и удаление из нее, добавление и удаление и т. д.), лежащий в основе механизм хранения и локальность кэша, скорость проведения различных операций с коллекцией, таких как `Add` и `Remove`, и необходимость синхронизации доступа к коллекции (или ее отсутствие). Все эти факторы могут существенно повлиять на производительность вашей программы.

Какие коллекции лучше не использовать

Некоторые коллекции все еще присутствуют в среде .NET Framework, но только из соображений обратной совместимости. Их никогда не следует использовать в новом коде. К их числу относятся:

- `ArrayList`;
- `Hashtable`;
- `Queue`;
- `SortedList`;
- `Stack`;
- `ListDictionary`;
- `HybridDictionary`.

Причинами, по которым нужно избегать их применения, являются преобразования типов и упаковка. В этих коллекциях хранятся ссылки на экземпляры `Object`, поэтому вам обязательно потребуется приведение к фактическому типу объекта.

Еще более вредна упаковка. Предположим, что нужно воспользоваться коллекцией `ArrayList`, состоящей из значимых типов Int32. Каждое значение будет в индивидуальном порядке упаковано и сохранено в куче. Вместо перебора последовательного массива памяти для доступа к каждому целочисленному значению каждая ссылка в массиве потребует разыменования указателя, обращения к куче (возможно, подрывая локальность), а затем операции распаковки для получения внутреннего значения. Это ужасно. Вместо этого лучше воспользоваться массивом, не меняющим свой размер, или одним из классов обобщенной коллекции.

В ранних версиях .NET присутствовало несколько коллекций, ориентированных на строковые значения, которые теперь в силу эффективности обобщенных коллекций устарели. В их числе `StringCollection`, `StringDictionary`, `NameValueCollection` и `OrderedDictionary`. Их использование не обязательно приведет к возникновению проблем с производительностью как таковой, но нет никакой необходимости даже брать их в расчет, пока не придется воспользоваться существующим API, требующим их применения.

Массивы

Самой простой и, наверное, наиболее часто используемой коллекцией является старый добрый массив `Array`. Массивы являются идеальной коллекцией в силу своей компактности, задействования одного последовательного блока памяти, улучшающего локальность кэша процессора при обращении к нескольким элементам (при условии использования значимых типов, ведь ссылочные типы будут по-прежнему приводить к переходам к другим местам кучи).

Обращение к ним занимает константное время, а их копирование выполняется быстро. Но изменение их размера будет означать выделение памяти под новый массив и копирование старых значений в новый объект. В качестве надстройки над массивами создаются многие более сложные структуры данных.

Общее требование — возможность передачи сегмента из состава массива. Одним из способов является копирование требуемых значений в новый массив нужного размера, но это влечет за собой дополнительные расходы ресурсов центрального процессора и памяти. Лучше заиметь структуру, которая просто ссылается на соответствующую часть массива. И хорошо, что в среде .NET уже есть такие структуры — `ArraySegment<T>` и `Span<T>`:

```
byte[] fileContents = File.ReadAllBytes("foo.bin");
ArraySegment <byte> header = new ArraySegment <byte >(fileContents,
                                                      1,
                                                      24);
ProcessHeader(header);
```

Многие API .NET, способные работать с массивами, зачастую будут иметь версию, непосредственно принимающую либо `ArraySegment<T>`, либо ссылку на

массив, смещение и количество элементов. При создании собственных API, работающих с массивами, разумно будет предусмотреть разработку версии, совместимой с `ArraySegment<T>`.

Похожими свойствами обладает и структура `Span<T>`, но она может представлять подсегменты, собранные из нескольких типов непрерывной памяти (она подробно рассмотрена в главе 2).

Сравнение ступенчатых и многомерных массивов

В среде .NET есть два способа выделения памяти под многомерные массивы.

- Многомерные массивы — один объект с несколькими индексами:

```
const int Size = 50;

private int[,,] multiArray;

void Init()
{
    this.multiArray = new int[Size, Size, Size];
}
```

- Ступенчатые массивы — массивы массивов (то есть множество объектов), у каждого из которых один индекс:

```
private const int Size = 50;

private int[][][] jaggedArray;

public void GlobalSetup()
{
    this.jaggedArray = new int[Size][][];
    for (int i = 0; i < Size; i++)
    {
        this.jaggedArray[i] = new int[Size][];
        for (int j = 0; j < Size; j++)
        {
            this.jaggedArray[i][j] = new int[Size];
        }
    }
}
```

Выглядят они в целом равнозначно, но устроено все по-разному. Показанный ранее код инициализации не дает реальной картины этого различия, поэтому рассмотрим код, выполняющий обход всех значений и изменяющий каждую запись.

```
public void Negate_JaggedArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
```

```
        {
            for (int k = 0; k < Size; k++)
            {
                this.jaggedArray[i][j][k] *= -1;
            }
        }
    }
}

public void Negate_MultiArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            for (int k = 0; k < Size; k++)
            {
                this.multiArray[i, j, k] *= -1;
            }
        }
    }
}
```

Код выглядит очень похожим, но посмотрим на показанный далее код IL, предназначенный только для внутреннего цикла. Как ступенчатый массив он вполне оправдывает наши ожидания:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[][][] ArrayPerf.ArrayPerfTest::jaggedArray
IL_0012: ldloc.0
IL_0013: ldelem.ref
IL_0014: ldloc.1
IL_0015: ldelem.ref
IL_0016: ldloc.2
IL_0017: ldelema [mscorlib]System.Int32
IL_001c: dup
IL_001d: ldind.i4
IL_001e: ldc.i4.m1
IL_001f: mul
IL_0020: stind.i4
IL_0021: ldloc.2
IL_0022: ldc.i4.1
IL_0023: add
IL_0024: stloc.2
```

Это просто серия обращений к памяти, немного математических вычислений и сохранения данных. Посмотрим для контраста на код для выполнения тех же изменений в многомерном массиве:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[0..., 0..., 0...]
    ArrayPerf.ArrayPerfTest::multiArray
```

```
IL_0012: ldloc.0
IL_0013: ldloc.1
IL_0014: ldloc.2
IL_0015: call instance int32& int32[0..., 0..., 0...]::
    Address(int32 , int32 , int32)
IL_001a: dup
IL_001b: ldind.i4
IL_001c: ldc.i4.m1
IL_001d: mul
IL_001e: stind.i4
IL_001f: ldloc.2
IL_0020: ldc.i4.1
IL_0021: add
IL_0022: stloc.2
```

В основном он такой же, за исключением бросающегося в глаза вызова метода, расположенного в центральной части. Различия, которые могут быть вызваны этим обстоятельством, продемонстрированы в проекте ArrayPerf, взятом из сопровождающего книгу исходного кода.

Метод	Значение, мкс	Ошибка, мкс	Стандартное отклонение (StdDev), мкс
Negate_JaggedArray	281,0	1,7812	1,6661
Negate_MultiArray	439,6	0,2280	0,1780

Из-за этого вызова метода на обход элементов многомерного массива затрачивается намного больше времени. Теоретически характер размещения в памяти многомерных массивов и возможность последовательного размещения в ней всех значений должны дать некоторые преимущества, но из этого обстоятельства не извлекается никакой практической выгоды и, по правде говоря, оснований для применения многомерных массивов при любых обстоятельствах слишком мало.

Обобщенные коллекции

К обобщенным коллекциям относятся следующие классы:

- `Dictionary<TKey, TValue>`;
- `HashSet<T>`;
- `LinkedList<T>`;
- `List<T>`;
- `Queue<T>`;
- `SortedDictionary<TKey, TValue>`;
- `SortedList<TKey, TValue>`;
- `SortedSet<T>`;
- `Stack<T>`.

С появлением этих коллекций все необобщенные версии становятся непригодными к использованию. Предпочтение всегда стоит отдавать именно обобщенным версиям. Их использование влечет за собой избавление от издержек на упаковку или приведение типов и обеспечит более приемлемую локальность в памяти значимых типов (особенно для `List`-подобных структур, реализуемых с помощью массивов).

Сами эти коллекции, однако, могут весьма существенно различаться по производительности. Например, отношения «ключ — значение» хранятся во всех коллекциях `Dictionary`, `SortedDictionary` и `SortedList`, но характеристики вставки и поиска в них сильно разнятся.

Решая вопросы производительности в ходе работы с коллекциями (если конкретнее, то рассматривая алгоритмы работы с коллекциями), полезно было бы иметь под рукой абстрактный способ сравнения. Для этого в компьютерных технологиях используется нотация «“O” большое». Если вкратце, «O» большое дает описание производительности в понятиях размерности задачи. Например, $O(n)$ означает линейное время — нужное время напрямую коррелируется с размерностью задачи. Линейный поиск в неупорядоченном списке будет обозначен $O(n)$. $O(1)$ означает постоянное время — размерность задачи не имеет значения, на операцию, например поиск в хеш-таблице, всегда затрачивается одно и то же время. Более подробные сведения и дополнительные примеры использования «O» большого можно найти в приложении B.

- `Dictionary` реализована в виде хеш-таблицы, время вставки и извлечения — $O(1)$.
- `SortedDictionary` реализована в виде двоичного дерева поиска, и время вставки и извлечения характеризуется описанием $O(\log n)$.
- `SortedList` реализована в виде отсортированного массива. Время извлечения — $O(\log n)$, а время вставки в худшем случае может быть $O(n)$. Если вставлять произвольные элементы, может потребоваться часто изменять размеры и перемещать существующие элементы. Идеальным случаем будет вставка всех элементов по порядку, а затем использование этой коллекции для быстрого поиска.

Среди трех коллекций самые скромные потребности в памяти у `SortedList`. У других двух намного более существенное время произвольного доступа к памяти, но в среднем они смогут гарантировать лучшее время вставки. Какой из коллекций воспользоваться, во многом зависит от требований вашего приложения.

Разница между `HashSet` и `SortedSet` аналогична разнице между `Dictionary` и `SortedDictionary`.

- `HashSet` использует хеш-таблицу, время ее операций вставки и удаления — $O(1)$.
- `SortedSet` применяет двоичное дерево поиска, время ее операций вставки и удаления — $O(\log n)$.

У коллекции `List` вставка — $O(1)$, удаление и поиск — $O(n)$. В коллекциях `Stack` и `Queue` доступно только добавление или удаление с одного из их концов, поэтому все операции имеют время $O(1)$.

У LinkedList вставка и удаление имеют время O(1), но обычно его не стоит применять для больших чисел и элементарных типов, поскольку это повлечет за собой для каждой новой добавляемой записи выделение памяти под новый объект LinkedListNode<T>,что может привести к большим издержкам.

Чтобы дать представление о различиях требуемых размеров хранилища для каждой упомянутой коллекции, я запустил тестовую программу, добавляющую в каждую структуру 1000 четырехбайтовых целочисленных значений, и воспользовался командой !ObjSize в WinDbg, чтобы увидеть разницу в используемом пространстве:

- List — 4036 байт;
- Stack — 4036 байт;
- Queue — 4044 байта;
- SortedList — 8076 байт;
- Dictionary — 22 144 байта;
- LinkedList — 24 028 байт;
- SortedSet — 24 044 байта;
- SortedDictionary — 28 076 байт;
- HashSet — 30 972 байта.

Как видите, коллекции List, Stack и Queue практически идеальны с точки зрения требуемой памяти — 4000 байт чистых данных плюс минимальные издержки, а у остальных — различные показатели накладных расходов.

При выборе структуры данных сначала выберите ту из них, которая имеет наиболее подходящий с точки зрения функциональности логический смысл. Если она демонстрирует приемлемую производительность, значит, выбор оказался удачным. В противном случае проанализируйте, обладают ли другие коллекции более выгодными характеристиками времени вставки или получения элементов для вашего сценария, а также приемлемы ли их требования к размеру хранилища.

Коллекции для многопоточной среды

Классы коллекций для многопоточной среды могут безопасно использоваться множеством потоков без дополнительных средств их синхронизации. Все эти классы находятся в пространстве имен System.Collections.Concurrent, и все они определены как обобщенные типы:

- ConcurrentBag<T> (Bag — это мультимножество, оно аналогично множеству (Set), но допускает дубликаты);
- ConcurrentDictionary<TKey, TValue>;
- ConccurentQueue<T>;
- ConcurrentStack<T>.

Большинство из них внутри реализовано с использованием примитивов синхронизации `Interlocked` или `Monitor`. Разным методам требуются разные уровни синхронизации, поэтому нужно проявлять осмотрительность. Например, при вызове `Count`, `IsEmpty`, `ToArray()` и `TryUpdate()` в отношении `ConcurrentDictionary` всегда применяется блокировка. Их реализацию можно и нужно изучать, задействуя средство для отражения IL. Поскольку каждый API защищен механизмом синхронизации, часто обращаться к нему может быть накладно. Как упоминалось в главе 4, возможно, рациональнее отказаться от использования коллекции для многопоточной среды и выполнить синхронизацию на более высоком уровне.

Обратите внимание на API для вставки записей в эти коллекции и их удаления оттуда. У всех интерфейсов имеются `Try`-методы, которые могут не выполнить операцию, если другой поток выполнит аналогичную операцию раньше, и из-за этого возникнет конфликт. Например, у `ConcurrentStack` есть метод `TryPop`, возвращающий булево значение, показывающее, была ли возможность получить значение их стека. Если другой поток получит последнее значение, метод `TryPop` текущего потока возвратит значение `false`.

В `ConcurrentDictionary` есть несколько методов, заслуживающих особого внимания. Можно вызвать `TryAdd` для добавления к словарю ключа и значения или `TryUpdate` для обновления существующего значения. Нередко в таких сценариях оказывается в принципе безразлично, имеется ли в коллекции этот ключ или нет, так как это никак не влияет на логику приложения. Для таких случаев существует метод `AddOrUpdate`, выполняющий либо добавление, либо обновление, но вместо непосредственной передачи нового значения требуется передать ему два делегата: один для добавления и один — для обновления. Если ключа не существует, будет вызван первый делегат с ключом в качестве параметра и он должен будет вернуть значение. Если ключ существует, то с ним и с существующим значением в качестве параметров вызывается второй делегат и он должен вернуть новое значение — это может совпадать с существующим значением. Следует обратить внимание на кэширование делегатов, рассмотренное в главе 5.

В любом случае метод `AddOrUpdate` возвратит новое значение, но важно понять, что это не обязательно значение, установленное вызовом `AddOrUpdate` из текущего потока! Эти методы безопасны при использовании в многопоточной среде, но не атомарны. Вполне возможно, что с тем же ключом данный метод вызывался из другого потока и в первый поток будет возвращено значение из второго потока.

Существует также переопределение метода, в котором нет делегата для добавления записи — вы просто передаете ему значение.

Полезно изучить простой пример:

```
dict.AddOrUpdate(
    // Ключ, попытка добавления которого предпринимается
    0,
    // Делегат для вызова при добавлении –
    // возвращение строкового значения на основе ключа
    key => key.ToString(),
    // Делегат для вызова, когда ключ уже присутствует, –
    // обновление существующего значения
```

```
     (key, existingValue) => existingValue);

dict.AddOrUpdate(
    // Ключ, попытка добавления которого предпринимается
    0,
    // Значение для добавления, если оно новое
    "0",
    // Делегат для вызова, когда ключ уже присутствует, –
    // обновление существующего значения
     (key, existingValue) => existingValue);
```

Наличие этих делегатов вместо простой передачи значения объясняется тем, что во многих случаях создание значения для заданного ключа — слишком затратная операция и не хотелось бы, чтобы этим одновременно занимались сразу два потока. Делегат дает вам возможность просто использовать существующее значение вместо пересоздания новой копии. Тем не менее следует заметить, что не существует гарантии однократного вызова делегатов. К тому же, если требуется предоставить синхронизацию при создании или обновлении значения, ее нужно добавлять в сами делегаты — коллекция этим заниматься не будет.

Родственным для `AddOrUpdate` является метод `GetOrAdd`, который ведет себя почти так же:

```
string val1 = dict.GetOrAdd(
    // Ключ для извлечения
    0,
    // Делегат для создания значения, если его нет
    k => k.ToString());

string val2 = dict.GetOrAdd(
    // Ключ для извлечения
    0,
    // Значение для добавления, если такого значения нет
    "0");
```

Из этого следует извлечь урок: обращаться с коллекциями для многопоточной среды нужно бережно. У них имеются особые требования и своеобразное поведение для предоставления гарантий безопасности и эффективности, и, чтобы правильно и успешно использовать их, нужно хорошо понимать, как они используются в контексте вашей программы. Из-за семантики, связанной с применением делегатов и потребностью в обслуживании согласованной структуры данных, вполне может оказаться, что при работе с этими структурами данных выполняется куда больше кода, чем кажется.

Коллекции для работы с битами

Есть и еще небольшая группа специализированных коллекций, поставляемых со средой .NET, но большинство из них ориентированы на работу со строками или хранение экземпляров класса `Object`, так что их смело можно игнорировать. Заметные исключения — коллекции `BitArray` и `BitVector32`.

`BitArray` представляет собой массив двоичных значений. Она позволяет устанавливать отдельные биты и выполнять в отношении всего массива булевы операции. Если требуются только 32-разрядные данные, следует воспользоваться коллекцией `BitVector32`, работа с которой выполняется быстрее и с меньшими издержками, поскольку это структура (представляющая собой чуть более чем простую надстройку над Int32).

Исходный объем

Большинство коллекций для хранения данных используют более простую структуру. Зачастую это массив или набор массивов. По мере наполнения коллекции этим массивам могут понадобиться перевыделение памяти и копирование данных. Можно запросто попасть в нелепую ситуацию, когда основное время будет расходоваться просто на изменение размера структур данных.

Благо большинство коллекций, подверженных данной проблеме, предоставляют параметр конструктора, задающий объем предварительно выделяемой памяти. Я рекомендую почаще задействовать его, даже в тех случаях, когда вы не можете точно знать объем, к которому придете в итоге. И, как правило, пока не окажутся существенно превышены требования к используемому объему, это позволит сэкономить ресурсы, так как будет предотвращено повторяющееся перевыделение памяти. В любом случае нужно все измерить с помощью профилировщика.

Коллекция	**Исходный объем**	**Возможность перевыделения памяти**
ConcurrentDictionary<TKey, TValue>	31	Да
Dictionary<TKey, TValue>	0	Да
HashSet<T>	0	Да
List<T>	0	Да
Queue<T>	0	Да
SortedDictionary<TKey, TValue>*	0	Нет
SortedList<T>	0	Да
SortedSet<T>	0	Нет
Stack<T>	0	Да

Для `HashSet` при первой вставке будет выделена область памяти, чей размер будет простым числом. `SortedSet` и `SortedDictionary` реализованы как двоичные деревья, инкапсулирующие отдельно взятые узлы, память под которые выделяется по мере необходимости.

Сравнение ключей

Структуры данных с ключом (например, `Dictionary<TKey, TValue>`) обычно также принимают в качестве параметра «сравнитель» (comparer), позволяющий вам определить, как сравнивать ключи.

В разделе «Строки» будет показано, что почти всегда для сравнения ключей стоит использовать наиболее подходящую по смыслу стратегию сравнения, даже если на вычисления затрачивается больше вычислительных ресурсов. Типичным примером может послужить работа со строками и чувствительность к регистрам символов.

Мне приходилось сталкиваться с кодом, подобным следующему, когда автор не знал, как правильно проверять ключи:

```
var keytoLookup = "myKey";
var dict = new Dictionary <string , object >();
...
foreach(var kvp in dict)
{
    if (kvp.Key.ToUpper() == keyToLookup.ToUpper())
    {
        ...
    }
}
```

Это приводит к очень частой сборке мусора, нерациональному использованию памяти и неоправданному расходованию ресурсов центрального процессора. Нужно также избегать применения методов LINQ, упрощающих поиск подобного рода.

Решить проблему можно двумя способами.

1. Ограничить ключи исходного набора данных, избавив их от необходимости выполнять сравнение без учета регистра символов. Это позволит обойтись без выделения памяти, а также получить самый экономный вариант сравнения.
2. Передать объект сравнения конструктору `Dictionary<TKey, TValue>`, который может использоваться для выполнения более точных операций сравнения и поиска. Класс `StringComparer` через статические свойства предоставляет ряд готовых к использованию «сравнителей».

   ```
   var keytoLookup = "myKey";
   var dict = new Dictionary <string , object >(
               StringComparer.OrdinalIgnoreCase);
   ...
   object val;
   if (dict.TryGetValue(keyToLookup , out val))
   {
       ...
   }
   ```

   Теперь словарь может быть правильно использован по назначению.

Сортировка

И еще одно важное замечание: если вам когда-нибудь придется воспользоваться собственными типами (особенно структурами) в качестве значений в списках и эти значения по своей природе могут быть упорядочены, вам нужно реализовать интерфейс `IComparable<T>`.

```
struct MyType : IComparable <MyType >
{
    public string Name { get; set; }

    public int CompareTo(MyType other)
    {
        return this.Name.CompareTo(other.Name);
    }
}
```

Если же значение может быть отсортировано произвольным способом (как в случае с объектом `Person`, сортировать который можно по дате рождения или фамилии), то лучше и вовсе обойтись без такой реализации во избежание путаницы.

Создание собственных типов коллекций

Довольно редко, но все же иногда мне требуется создать с нуля собственные типы коллекций. Если у встроенных типов отсутствует нужная семантика, то в качестве подходящей абстракции следует создать собственный тип. При этом нужно придерживаться следующих общих рекомендаций.

1. Реализуйте везде, где это имеет смысл, стандартные интерфейсы коллекций:
   - `IEnumerable<T>`;
   - `ICollection<T>`;
   - `IList<T>`;
   - `IDictionary<TKey, TValue>`.
2. Решая вопрос о внутреннем хранении данных, учитывайте, как в будущем коллекция будет использоваться. Если для нее будет характерен последовательный доступ, обратите внимание на такие вещи, как локальность ссылок и поддержка массивов.
3. Нужно ли вам добавлять синхронизацию в саму коллекцию? Или создать версию коллекции для применения в многопоточной среде?
4. Разберитесь в сложности выполнения алгоритмов добавления, вставки, обновления, поиска и удаления. Изучите описание сложности через «О» большое, рассмотренное в приложении В.
5. Реализуйте API, имеющие семантический смысл, например, `Pop` для стеков, `Dequeue` для очередей.

Строки

В среде .NET со строками связаны две проблемы.

1. Строки неизменяемы.
2. Нам требуется их изменять.

Эти два обстоятельства порождают большинство проблем.

Под неизменяемостью подразумевается, что после создания и до сборки мусора строки существуют в неизменном виде. Следовательно, любые изменения строки приводят к созданию новой строки. Быстрые высокоэффективные программы обычно вообще не изменяют строки. Если задуматься, строки представляют текстовые данные, главным образом предназначенные для восприятия человеком. Если ваша программа не предназначена для отображения или обработки текста, строки должны, насколько это возможно, рассматриваться как непрозрачные большие двоичные объекты данных. Если есть выбор, предпочтение всегда нужно отдавать нестроковому представлению данных.

Сравнение строк

Наилучшее сравнение строк — то, которое никогда не произойдет. Если получится, воспользуйтесь перечисляемыми типами или какими-нибудь другими числовыми данными для принятия решений. Если все же придется воспользоваться строками, старайтесь делать так, чтобы они были короткими, и использовать по возможности самый простой алфавитный набор.

Существует множество способов сравнения строк: просто по байтовым значениям, с учетом текущей культуры, без учета регистра символов и т. д. Пользоваться нужно самым простым из возможных способом.

Например, способ:

```
String.Compare(a, b, StringComparison.Ordinal);
```

быстрее способа:

```
String.Compare(a, b, StringComparison.OrdinalIgnoreCase);
```

который быстрее способа:

```
String.Compare(a, b, StringComparison.CurrentCulture);
```

При обработке таких программно сгенерированных строк, как, например, настройки конфигурации или какой-либо другой тесно связанный программный контракт, можно ограничиться обычными сравнениями с учетом регистра символов.

При всех сравнениях строк должны использоваться перегрузки методов, явно включающие перечисление `StringComparison`. Пренебрежение этим должно считаться ошибкой.

И наконец, метод `String.Equals` — это частный случай `String.Compare`, его нужно использовать, когда порядок сортировки неважен. В большинстве случаев он не будет быстрее, но все-таки лучше передает смысл вашего кода.

ToUpper и ToLower

Старайтесь избегать вызовов методов, подобных `ToLower` и `ToUpper`, особенно для сравнения строк. Вместо этого воспользуйтесь одним из вариантов `IgnoreCase` для метода `String.Compare`.

Этот способ, конечно, имеет некоторые недостатки, но не столь серьезные. С одной стороны, верно, что проведение сравнения строк с учетом регистра символов выполняется быстрее, однако это все-таки не повод для использования `ToUpper` или `ToLower`, так как при этом гарантированно будет обработан каждый символ, в то время как операция сравнения зачастую может принять решение при первой же встрече несовпадающего символа и даже быстрее, если строки отличаются по длине. Кроме того, такой подход приводит к созданию новой строки, выделению памяти и повышению нагрузки на сборщик мусора. Так что просто не используйте его, насколько это возможно.

Объединение

Для простого объединения (конкатенации) известного на момент компиляции количества строк обычно можно воспользоваться оператором `+` или методом `String.Concat`. Зачастую это более эффективно в смысле расхода ресурсов центрального процессора и памяти, чем применение `StringBuilder`. Есть также метод `String.Join`, но суть его заключается в использовании `StringBuilder`:

```
string result = a + b + c + d + e + f;
```

Когда ситуация становится более динамичной, все усложняется и зависит от таких факторов, как количество строк и их длина, и от возможности предварительной инициализации буфера результатов. В таком случае нужно применить класс `StringBuilder`, в котором для построения строки перед ее превращением в объект `String` используются объединенные в пул символьные буферы.

В сопровождающем книгу учебном коде есть проект StringConcatPerf, в котором проводится эталонное тестирование ряда методов конкатенации.

В качестве эталона сравниваются десять текстовых строк, каждая длиной десять литералов. На каждую итерацию уходит около 0,001 нс. С нелитералами время разбивается следующим образом.

- **Количество = 10, длина = 1:**
  - `StringBuilder` (неинициализированный) — 99 нс;
  - `StringBuilder` (инициализированный) — 125 нс;
  - `String.Concat` — 176 нс.

- **Количество = 10, длина = 10:**
  - `String.Concat` — 195 нс;
  - `StringBuilder` (инициализированный) — 246 нс;
  - `StringBuilder` (неинициализированный) — 402 нс.
- **Количество = 100, длина = 1:**
  - `StringBuilder` (инициализированный) — 771 нс;
  - `StringBuilder` (неинициализированный) — 895 нс;
  - `String.Concat` — 1714 нс.
- **Количество = 100, длина = 10:**
  - `StringBuilder` (инициализированный) — 1721 нс;
  - `String.Concat` — 1943 нс;
  - `StringBuilder` (неинициализированный) — 2243 нс.

Как видите, относительная производительность сильно зависит от входных характеристик. Разница между инициализированным и неинициализированным `StringBuilder` заключается в том, что в первом случае конструктору передается аргумент для предварительного выделения необходимого объема памяти:

```
var sb = new StringBuilder(1024); // предварительная инициализация
var sb2 = new StringBuilder(); // исходный 16-символьный буфер
```

По возможности нужно выполнять предварительную инициализацию объема памяти для `StringBuilder`, чтобы он был по крайней мере не меньше необходимого, что позволит избежать повторного выделения дополнительных буферов. Надо отметить, что `StringBuilder` выделяет дополнительные буферы в виде выстроенных в цепочку областей памяти, и это позволяет во многом избежать проблем с копированием, но все же не отличается высокой эффективностью использования центрального процессора и памяти и может вызвать выделение объема памяти, превышающего фактические потребности.

Форматирование

`String.Format` считается весьма затратным методом. Вдобавок к разбору форматируемой строки он выполняет упаковку аргументов значимого типа, потенциально вызывает пользовательские средства форматирования и выделяет больше памяти, чем необходимо для получающейся в результате строки. Не применяйте его без особой необходимости. Старайтесь не использовать его в простых ситуациях, похожих на следующую:

```
string message = String.Format(
  "The file {0} was {1} successfully.",
  filename ,
  operation);
```

Вместо этого нужно выполнить простую конкатенацию:

```
string message = "The file " + filename + " was " + operation
  + " successfully";
```

Приберегите применение `String.Format` для тех случаев, когда производительность не играет особой роли или спецификация форматирования намного сложнее (например, указано количество десятичных знаков для числового значения с двойной точностью).

В учебном проекте StringConcatPerf выдаются числовые показатели нескольких эталонных тестов для сравнения `String.Format` и `String.Concat`:

- `String.Concat` — 225 нс;
- `String.Format` — 440 нс.

На создание строки с помощью `String.Format` уходит почти вдвое больше времени, чем на создание той же строки с помощью `String.Concat`. Для выполнения поставленной задачи следует использовать самые простые и наименее затратные средства.

ToString

Для многих классов вызов `ToString` следует применять крайне осмотрительно. При удачном стечении обстоятельств будет возвращена уже существующая строка. Другие классы выполнят кэширование строки сразу же после ее создания. Например, класс `IPAddress` кэширует свою строку, но сам процесс ее генерации крайне затратен — он включает в себя использование `StringBuilder`, форматирования и упаковки. Другие типы могут создавать новую строку при каждом вызове. Это способно стать весьма расточительным действием для центрального процессора и повлиять на частоту сборки мусора.

При разработке собственных классов следует учесть особенности сценариев будущих вызовов принадлежащего вашему классу метода `ToString`. При его частых вызовах нужно обеспечить, чтобы строка генерировалась как можно реже. Если же это всего лишь вспомогательный метод для отладки, то вполне вероятно, что детали его реализации не имеют значения.

`ToString` зачастую вызывается средствами регистрации событий и различными вспомогательными методами форматирования строк наподобие `String.Format` и `Console.WriteLine` и родственными им функциями в других классах. Постарайтесь убедиться в том, что ваши собственные классы сконструированы с учетом этих обстоятельств. Некоторые классы должны создавать строку только для отладки.

Избегайте разбора строк

По возможности конструируйте свою систему таким образом, чтобы не возникало необходимости в разборе строк. Если данные должны преобразовываться из строк, нельзя ли сделать это в автономном режиме или в ходе запуска приложения? Обработка строк зачастую требует больших ресурсов центрального процессора,

периодически повторяется и создает большую нагрузку на память. Всего этого следует избегать.

Если требуется провести разбор в ходе выполнения приложения, воспользуйтесь методами, не выдающими исключений при сбоях. Например, не задействуйте метод `Int32.Parse`, поскольку он станет выдавать исключение `FormatException`. Вместо него примените метод `Int32.TryParse`, который просто возвратит значение `false`. Подобные варианты API имеются у многих основных .NET-типов.

Подстроки

Класс `String` имеет различные методы для возвращения частей строк в виде новой строки. Они приводят к выделению памяти под новые объекты. Если есть возможность оперировать подпорцией в форме массива символов, рассмотрите вероятность использования структуры `ReadOnlySpan<T>` в соответствии с описанием, рассмотренным в главе 2, для представления части базового массива, как в следующем примере:

```
{
...
    ReadOnlySpan <char> subString =
        "NonAllocatingSubstring".AsSpan().Slice(13);
    PrintSpan(subString);
...
}

private static void PrintSpan <T>(ReadOnlySpan <T> span)
{
    for (int i = 0; i < span.Length; i++)
    {
        T val = span[i];
        Console.Write(val);
        if (i < span.Length - 1) { Console.Write(", "); }
    }
    Console.WriteLine();
}
```

Этот код выведет следующее:

```
S, u, b, s, t, r, i, n, g
```

Избегайте использования API, выдающих исключения при обычных обстоятельствах

В главе 5 говорилось, что исключения обходятся слишком дорого. Поэтому их применение нужно оставить для по-настоящему исключительных ситуаций. К сожалению, ряд широко распространенных API игнорируют это основное положение.

У многих основных типов имеется метод `Parse`, который выдаст исключение `FormatException`, когда введенная строка находится в нераспознаваемом формате.

В качестве примера можно привести методы `Int32.Parse`, `DateTime.Parse` и т. д. Вместо них стоит применять метод `TryParse`, возвращающий при сбое разбора булево значение.

В качестве еще одного примера можно привести класс `System.Net.HttpWebRequest`, выдающий исключение при получении от сервера ответа с кодом, отличным от 200. К счастью, это странное поведение было исправлено в классе `System.Net.Http.HttpClient` в версии .NET 4.5.

Избегайте использования API, выделяющих память из кучи больших объектов

Вспомним материал главы 2, где говорилось, что, как правило, единственное, что может быть достаточно большим для выделения памяти в куче больших объектов, — это массивы. Если ваш собственный код выделяет под них память, то они, конечно же, должны быть собраны в пул.

К сожалению, есть такие API .NET, которые также приводят к выделениям памяти из кучи больших объектов. Единственный способ избежать применения таких API — профилирование выделения памяти в куче с помощью PerfView, которое покажет области кода, выделяющие память подобным образом. Совсем избежать этого не удастся, поэтому нужно просто быть в курсе, что существуют API .NET, которые будут так делать. Например, вызов метода `Process.GetProcesses` гарантированно приведет к выделению памяти в куче больших объектов. Вы можете избежать повторных LOH-распределений, используя кэширование результатов, вызывая этот метод реже или полностью его игнорируя и извлекая нужную информацию путем взаимодействия непосредственно с API Win32.

Некоторые API будут объединять в пул создаваемые ими большие объекты во избежание повторного выделения памяти в куче больших объектов (так, к примеру, делает метод `StringBuilder`), но вам нужно будет проанализировать их код, чтобы обнаружить именно такую особенность работы. Чтобы двигаться в правильном направлении, достаточно выполнять профилирование и отталкиваться от его результатов.

Применение ленивой инициализации

Если в вашей программе применяется большой или затратный для создания объект, используемый крайне редко или вообще не используемый в ходе того или иного вызова, то можно воспользоваться классом `Lazy<T>` в качестве оболочки для ленивой инициализации. Как только произойдет обращение к свойству `Value`, сразу же будет инициализирован реальный объект — на основе параметров конструктора, использованных при создании объекта `Lazy<T>`.

Если процесс создания объекта окажется сложнее, конструктору можно передать Func<T>.

```
var lazyObject = new Lazy<MyExpensiveObject >();
...
if (needRealObject)
{
    MyExpensiveObject realObject = lazyObject.Value;
    ...
}
```

Если конструкция сложнее, конструктору можно передать Func<T>:

```
var myObject = new Lazy<MyExpensiveObject >(
    () => Factory.CreateObject("A"));
...
MyExpensiveObject realObject = myObject.Value
```

Здесь Factory.CreateObject — фиктивный метод, создающий объект типа MyExpensiveObject.

Если к myObject.Value обращаются сразу из нескольких потоков, вполне возможно, что каждому из них понадобится инициализировать этот объект. По умолчанию Lazy<T> предназначен для работы в многопоточной среде, и только одному потоку будет разрешено выполнить делегат, порождающий объект, и установить значение свойства Value. Изменить эту особенность можно с помощью перечисления LazyThreadSafetyMode. У него имеется три значения:

- None — безопасность не обеспечивается ни одному из потоков. В этом случае может быть важно обеспечить обращение к объекту Lazy<T> только из одного потока;
- ExecutionAndPublication — вызов порождающего делегата и установка значения свойства Value разрешены только одному потоку. Если конструктор Lazy применяется без этого параметра, это значение используется по умолчанию;
- PublicationOnly — вызов порождающего делегата доступен нескольким потокам, но только один будет инициализировать свойство Value.

Lazy<T> следует задействовать вместо вашего собственного синглтона и реализаций шаблона блокировки с двойной проверкой.

При наличии большого количества объектов и, значит, при весомых издержках от использования Lazy<T> можно воспользоваться статическим методом EnsureInitialized класса LazyInitializer. Здесь для обеспечения однократности присваивания ссылки на объект используются методы Interlocked, но они не гарантируют однократность вызова порождающего делегата. В отличие от Lazy<T> метод EnsureInitialized вы должны вызвать самостоятельно:

```
static MyObject[] objects = new MyObject[1024];
static void EnsureInitialized(int index)
{
   LazyInitializer.EnsureInitialized(ref objects[index],
       () => ExpensiveCreationMethod(index));
}
```

Обратите внимание на то, что за счет применения здесь делегата произойдет дополнительное выделение памяти для объекта делегата вдобавок к выделению, которое выполняется методом `ExpensiveCreationMethod`.

Удивительно высокие издержки от использования перечислений

Возможно, для вас окажется неожиданностью то, что методы, работающие со значениями перечисления (`Enum`) — по сути своей целочисленными, — отличаются очень высокими издержками. К сожалению, из-за требований, связанных с безопасностью типов, многие простые операции обходятся дороже, чем вы ожидаете.

Возьмем, к примеру, метод `Enum.HasFlag`. Скорее всего, вы представляете себе его реализацию примерно так:

```
public static bool HasFlag(Enum value , Enum flag)
{
    return (value & flag) != 0;
}
```

К сожалению, в реальности все выглядит скорее так:

```
// Код C#, сгенерированный ILSpy
public bool HasFlag(Enum flag)
{
    if (flag == null)
    {
        throw new ArgumentNullException("flag");
    }
    if (!base.GetType().IsEquivalentTo(flag.GetType()))
    {
        throw new ArgumentException("Enum types do not match",
             new object[]
          {
            flag.GetType(),
            base.GetType()
          }));
    }
    return this.InternalHasFlag(flag);
}
```

Это хороший пример побочных эффектов использования фреймворка общего назначения. Если контролировать всю кодовую базу, можно улучшить код, сделать более рациональным с точки зрения производительности путем определенных допущений и принудительных ограничений, недоступных среде .NET Framework. Если окажется, что проверять наличие флага приходится часто, реализуйте проверку самостоятельно:

```
[Flags]
enum Options
```

```
{
  Read = 0x01,
  Write = 0x02,
  Delete = 0x04
}

...

private static bool HasFlag(Options option , Options flag)
{
  return (option & flag) != 0;
}
```

Метод `Enum.ToString` также крайне затратен для перечислений `Enum`, имеющих атрибут `[Flags]`. Один из вариантов удешевления его использования — кэширование всех вызовов `ToString` для этого типа `Enum` в простом `Dictionary<EnumType, String>`. Или можно вообще избавиться от записи этих строк и добиться более высокой производительности, применив фактическое числовое значение и автономное преобразование в строки.

В качестве любопытного упражнения посмотрите, как много кода вызывается при использовании `Enum.IsDefined`. И опять следует заметить, что существующая реализация вполне приемлема, когда чистая производительность не играет большой роли, но если обнаружится, что это действительно узкое место в программе, можно испытать настоящий ужас!

ИСТОРИЯ

Я обнаружил проблемы с производительностью, связанные с Enum, в самых жестких обстоятельствах — сразу после выпуска программы. В ходе обычного профилирования центрального процессора я заметил, что существенная доля времени ЦП — более 3 % — затрачивалась на Enum.HasFlag и Enum.ToString. Удаление всех вызовов HasFlag и использование Dictionary для кэшированных строк сделало издержки пренебрежимо малыми.

Учет времени

Под временем подразумеваются две вещи:

- абсолютное дневное время;
- временной интервал (как много времени уходит на что-то).

Для абсолютного времени в среде .NET имеется структура общего назначения `DateTime`. Но вызов `DateTime.Now` — довольно затратная операция, так как нужно учесть информацию о часовом поясе. Вместо этого следует рассмотреть возможность использования более эффективного свойства `DateTime.UtcNow`.

Но даже вызов `DateTime.UtcNow` может оказаться слишком накладным для вас, если требуется отслеживать множество отметок времени. В таком случае получите

время только один раз, а затем отслеживайте смещения, перестраивая абсолютное время в автономном режиме с помощью показанной далее технологии измерения промежутка времени.

Для измерения интервалов времени в .NET есть структура `TimeSpan`. Если вычесть одну структуру `DateTime` из другой, будет получена структура `TimeSpan`.

Но если нужно измерить очень небольшие промежутки времени с минимальными издержками, воспользуйтесь системным счетчиком производительности с помощью метода `System.Diagnostics.Stopwatch`, который возвратит 64-разрядное число, показывающее количество тактов с момента подачи напряжения на центральный процессор. Для вычисления реальной разницы во времени берутся два показания количества тактов, одно вычитается из другого, а результат делится на частоту счетчика тиков системного таймера. Следует заметить, что эта частота не обязательно связана с частотой процессора. Большинство современных процессоров часто меняют свою частоту, но на частоту счетчика тиков это не влияет.

Класс `Stopwatch` можно использовать следующим образом:

```
var stopwatch = Stopwatch.StartNew();
//...выполнение работы...
stopwatch.Stop();
TimeSpan elapsed = stopwatch.Elapsed;
long elapsedTicks = stopwatch.ElapsedTicks;
```

Существуют также статические методы получения отметки времени и частоты часов, которые могут оказаться удобнее при отслеживании большого количества отметок времени и желании избежать издержек от создания нового объекта `Stopwatch` для каждого интервала:

```
long receiveTime = Stopwatch.GetTimestamp();
long parseTime = Stopwatch.GetTimestamp();
long startTime = Stopwatch.GetTimestamp();
long endTime = Stopwatch.GetTimestamp();

double totalTimeSeconds = (endTime - receiveTime) /
    Stopwatch.Frequency;
```

И наконец, не следует забывать, что значения, полученные с помощью метода `Stopwatch.GetTimestamp`, действительны только для текущего сеанса выполнения и только для вычисления относительной разницы во времени.

Сочетая два типа времени, можно понять, как вычисляются смещения относительно базового объекта `DateTime` для получения новых значений абсолютного времени:

```
DateTime start = DateTime.Now;
long startTime = Stopwatch.GetTimestamp();
long endTime = Stopwatch.GetTimestamp();

double diffSeconds = (endTime - startTime) / Stopwatch.Frequency;
DateTime end = start.AddSeconds(diffSeconds);
```

Регулярные выражения

Регулярные выражения являются способом поиска строки с применением расширенного синтаксиса соответствия шаблону.

К числу простых примеров можно отнести следующий код:

```
Regex regex = new Regex("<value >(.*)</value >");
```

Здесь выполняется поиск любого текста между XML-тегами `<value>`. Регулярные выражения могут быть очень сложными и трудными для понимания, пока не будет как следует усвоен их синтаксис и не изучены способы их разбиения на вполне понятные блоки.

Примером более сложного на вид выражения будет:

```
static Regex regex =
   new Regex("\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)");
```

С его помощью извлекаются значения в долларовом выражении в виде `$-34.19`, `$52` и т. п.

Регулярные выражения не отличаются быстротой обработки. Внутри среды .NET создается машина, внутреннее состояние которой предназначено для обхода введенной строки и поиска ее соответствия символам шаблона. К сопутствующим издержкам относятся следующие.

- Возможная продолжительность времени вычисления. Она зависит от введенного текста и шаблона для поиска соответствий. Создавать регулярные выражения с низкой производительностью совсем не трудно. Минимальные различия выражений могут дать существенную разницу в скорости обработки, их оптимизация является обширной темой для изучения.
- Генерирование сборки. При определенных настройках при создании вами объекта `Regex` на лету создается сборка, хранящаяся в памяти. Она повышает производительность в ходе выполнения программы, но само ее создание обходится довольно дорого.
- Возможное повышение издержек на JIT-компиляцию. Код, сгенерированный из регулярного выражения, может быть очень длинным и содержать типично довольно «напряжные» для JIT-компилятора фрагменты. Хорошо, что последние версии JIT-компилятора существенно усовершенствованы в этой области.

Что касается среды .NET, то для повышения эффективности использования `Regex` можно проделать следующее.

- Обеспечить работу актуальных версий .NET и применение к ним исправлений. В частности, в среде .NET 4.6 имеется новый JIT-компилятор, существенно повышающий производительность синтаксического разбора регулярного выражения.
- Вместо использования статических методов создать переменную экземпляра `Regex`. Статические методы в любом случае создадут такой экземпляр внутри

себя, но вы не сможете воспользоваться им повторно. Они больше подходят для одноразового применения.

- Создать объект `Regex` с флагом `RegexOptions.Compiled`. Без этого флага регулярные выражения интерпретируются. А скомпилированные выражения вычисляются намного быстрее за счет того, что основные затраты приходятся на инициализацию из-за выдачи кода в сборку в памяти, которая потом подвергается JIT-компиляции. От компиляции выражений стоит отказаться, если вы постоянно используете новые форматы, а старые крайне редко применяются повторно. Это приведет к захламлению процесса массой дополнительного кода.
- Не заниматься созданием объектов `Regex` снова и снова. Создайте один объект, сохраните его и используйте многократно для проверки соответствия вновь введенным строкам. Никогда не следуйте подобному подходу:

```
class Foo
{
    private static void Evaluate(string[] inputs)
    {
        for (int i = 0; i < inputs.Length; i++)
        {
            Regex regex = new Regex("<value >(.*)</value >",
                                    RegexOptions.Compiled);
            Match match = regex.Match(input);
            ...
        }
    }
}
```

Вместо этого сделайте следующее:

```
class Foo
{
    private static readonly Regex regex =
      new Regex("<value >(.*)</value >",
                RegexOptions.Compiled);

    private static void Evaluate(string[] inputs)
    {
        for (int i = 0; i < inputs.Length; i++)
        {
            Match match = regex.Match(input);
            ...
        }
    }
}
```

Может также потребоваться задать время ожидания сложных выражений:

```
Regex regex = new Regex("<value >(.*)</value >",
                        RegexOptions.Compiled ,
                        TimeSpan.FromSeconds(5));
```

Можно также задать глобальное время ожидания, настроив домен приложения:

```
AppDomain.CurrentDomain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT",
                                TimeSpan.FromSeconds(10));
```

В качестве альтернативы регулярным выражениям можно найти или создать собственную библиотеку синтаксического анализа для таких очень простых и имеющих небольшое количество вариантов шаблонов, как даты или телефонные номера.

LINQ

Самая большая опасность, связанная с интегрированными в язык запросами (Language Integrated Query, LINQ), заключается в потенциальном сокрытии от вас кода — того самого, который невозможно принять в расчет, поскольку его нет в исходном файле!

В качестве иллюстрации рассмотрим весьма простой пример в виде метода, выполняющего сдвиг всех значений в массиве, отвечающих конкретному условию[1], на определенное число:

```
public static int[] ShiftLinq(int[] vals, int shiftAmount)
{
    var temp = from n in vals select n + shiftAmount;
    return temp.ToArray();
}
```

Во внешнем виде ничто не настораживает, но многое здесь происходит закулисно. Посмотрим на код IL, в который все это переводится:

```
.method public hidebysig static
    int32[] ShiftLinq (
        int32[] vals,
        int32 shiftAmount
    ) cil managed
{
    // Method begins at RVA 0x209c
    // Code size 37 (0x25)
    .maxstack 3
    .locals init (
         [0] class LinqCost.Program/'<>c__DisplayClass1_0'
    )

    IL_0000: newobj instance void LinqCost.Program
        /'<>c__DisplayClass1_0'::.ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: ldarg.1
```

[1] В примере нет условия, но у автора есть упоминание этого условия. — *Примеч. науч. ред.*

```
    IL_0008: stfld int32
        LinqCost.Program/'<>c__DisplayClass1_0'
            ::shiftAmount
    IL_000d: ldarg.0
    IL_000e: ldloc.0
    IL_000f: ldftn instance int32
        LinqCost.Program/'<>c__DisplayClass1_0'
            ::'<ShiftLinq >b__0'(int32)
    IL_0015: newobj instance void class
         [mscorlib]System.Func '2<int32 , int32 >
            ::.ctor(object , native int)
    IL_001a: call class [mscorlib]
        System.Collections.Generic.IEnumerable '1<!!1>
         [System.Core]System.Linq.Enumerable
        ::Select <int32 , int32 >(class
         [mscorlib]System.Collections.Generic.IEnumerable '1
            <!!0>, class [mscorlib]System.Func '2<!!0, !!1>)
    IL_001f: call !!0[]
     [System.Core]System.Linq.Enumerable
     ::ToArray <int32 >(
        class [mscorlib]
        System.Collections.Generic.IEnumerable '1<!!0>)
    IL_0024: ret
} // end of method Program::ShiftLinq
```

Как видите, здесь 37 байт кода IL, включая два выделения памяти для объектов замыканий и делегатов, а также два вызова метода. Некоторые из вызовов методов в дальнейшем будут вызывать выделение памяти. В довершение всего финальный вызов `ToArray` в самом конце получает аргумент `IEnumerable<T>`, а это означает, что в зависимости от исходной коллекции о длине массива ничего не известно, пока не будет достигнут его конец. Это влечет за собой изменение размера и копирование до самого конца, а затем финальное копирование в массив точно определенного размера. Что это означает для вас, зависит от потребностей вашей программы. Несомненно, все это намного затратнее альтернативной реализации:

```
public static void Shift(int[] vals, int shiftAmount)
{
    for (int i = 0; i < vals.Length; i++)
    {
        vals[i] += shiftAmount;
    }
}
```

Вместо синтаксиса запроса здесь используется обычный цикл `for` с инструкцией `if`. Все это переводится в следующий код IL:

```
.method public hidebysig static
    void Shift (
        int32[] vals,
        int32 shiftAmount
```

```
    ) cil managed
{
    // Method begins at RVA 0x20d0
    // Code size 27 (0x1b)
    .maxstack 3
    .locals init (
         [0] int32
    )

    IL_0000: ldc.i4.0
    IL_0001: stloc.0
    IL_0002: br.s IL_0014
    // loop start (head: IL_0014)
        IL_0004: ldarg.0
        IL_0005: ldloc.0
        IL_0006: ldelema
             [mscorlib]System.Int32
        IL_000b: dup
        IL_000c: ldind.i4
        IL_000d: ldarg.1
        IL_000e: add
        IL_000f: stind.i4
        IL_0010: ldloc.0
        IL_0011: ldc.i4.1
        IL_0012: add
        IL_0013: stloc.0

        IL_0014: ldloc.0
        IL_0015: ldarg.0
        IL_0016: ldlen
        IL_0017: conv.i4
        IL_0018: blt.s IL_0004
    // end loop

    IL_001a: ret
} // end of method Program::Shift
```

Полученный код IL меньше на 27 байт, но важнее всего абсолютное отсутствие выделений памяти или вызовов методов.

Если присмотреться, в версии кода для цикла можно заметить весьма примечательную хитрость: в ней исходный массив изменяется, а в LINQ-версии создается новый массив. Это сделано преднамеренно. LINQ подталкивает вас к созданию новых объектов, а не к повторному использованию уже существующих. Такое случается довольно часто, поскольку в LINQ весьма интенсивно используется интерфейс `IEnumerable<T>`, являющийся интерфейсом самого низкого уровня, который может быть реализован коллекцией. Это означает невозможность применения семантики, более подходящей для конкретной структуры данных, и потерю при обработке некоторой доли производительности. Впрочем, существуют методы,

которые будут проверять наличие других интерфейсов, подобных `ICollection` или `IList`, и по возможности нужно использовать именно такие методы. В LINQ также принят довольно функциональный, неизменяемый взгляд на данные. Это нас учит тому, что инструментальные средства зачастую навязывают ограничения на архитектуру и производительность.

А как все-таки изменилась производительность? Вот показатели, полученные в результате эталонного тестирования.

Метод	Среднее, нс	Ошибка, нс	Стандартное отклонение (StdDev), нс	Поколение gen 0	Выделения памяти, байт
ShiftLinq	286,78	1,8867	1,7648	0,0448	236
Shift	12,38	0,1568	0,1390	—	0

LINQ-версия в 20 раз затратнее более простой версии, и из-за выделений памяти в ней выше вероятность сборки мусора.

Я хочу уточнить, что LINQ — неплохая технология и ее есть где применить. Временами она чрезвычайно полезна, и многие LINQ-запросы вполне производительны, но LINQ также может весьма интенсивно использовать делегаты, интерфейсы и временное выделение памяти под объекты, если чрезмерно увлекаться временными динамическими объектами, объединениями или сложными условиями `where`.

Применяя Parallel LINQ, зачастую можно добиться существенного ускорения, но следует иметь в виду, что при этом объем проделываемой работы не сокращается, а перераспределяется на несколько процессоров. Это может быть вполне приемлемо для приложения, предназначенного главным образом для выполнения в одном потоке и нуждающегося в экономии времени при LINQ-запросе. Но если программа создается для сервера, задействующего для обработки данных все ядра, распределение LINQ среди этих же самых процессоров по большому счету не принесет никакой выгоды и даже может навредить. В таком случае, возможно, разумнее выполнять задачу без использования LINQ, а подобрать что-нибудь более эффективное.

Если вы подозреваете о том, что возникла некая неучтенная сложность, запустите PerfView и посмотрите на представление JITStats, чтобы увидеть размеры кода IL и показатель времени, затраченного на JIT-компиляцию методов, применяющих LINQ. Также обратите внимание на использование центрального процессора этими методами после их JIT-компиляции.

Стоит отметить, что в LINQ в .NET Core внесено несколько эффективных усовершенствований, таких как сокращение общего количества выделений памяти, уход от выделений памяти для делегатов и повсеместное улучшение алгоритмов при использовании коллекций известных размеров (в обход проблемы, связанной с `IEnumerable`), что может компенсировать некоторые из рассмотренных здесь затрат, но не все.

В завершение следует сказать, что для выполнения поставленной задачи нужно подбирать соответствующий ей инструментарий и LINQ вполне может подойти для большинства компонентов ваших программных продуктов и обеспечить удобную работу, но оказаться совершенно непригодным для других их частей. Там, где происходит выделение памяти и интенсивно используется центральный процессор, нужно проводить измерения, и обычно код, активно применяющий LINQ, можно без особых усилий преобразовать во что-то другое. При этом всегда следует ориентироваться на результаты профилирования.

Чтение и запись файлов

В классе `File` есть ряд весьма удобных методов, таких как `Open`, `OpenRead`, `OpenText` и `OpenWrite`. Они вполне подходят для тех случаев, когда не нужно добиваться особо высокой производительности.

При выполнении большого объема дискового ввода-вывода следует обращать внимание на тип доступа к диску: произвольный, последовательный или требующий того, чтобы запись на устройство произошла на физическом уровне, прежде чем приложение будет уведомлено о завершении ввода-вывода. Для такого уровня детализации следует воспользоваться классом `FileStream` и перегрузкой конструктора, принимающей перечисление `FileOptions`. Используя логическое ИЛИ, можно скомбинировать множество флагов, но не все возможные комбинации валидны. Применение этого параметра необязательно, но он может дать подсказки операционной или файловой системе, позволяющие оптимизировать доступ к файлу:

```
using (var stream = new FileStream(
            @"C:\foo.txt",
            FileMode.Open,
            FileAccess.Read,
            FileShare.Read,
            16384 /* Buffer Size*/,
            FileOptions.SequentialScan | FileOptions.Encrypted))
{
...
}
```

Доступны следующие значения:

- `None` — без дополнительных настроек;
- `Asynchronous` — показывает намерения провести чтение файла или запись в него в асинхронном режиме. Устанавливать этот флаг не обязательно для фактического выполнения асинхронных чтения и записи, но, если его не указать, ввод-вывод низкого уровня будет выполняться в синхронном режиме без портов завершения ввода-вывода (поток по-прежнему будет выполняться асинхронно). Существуют также переопределения конструктора `FileStream`, которые будут получать булев параметр, чтобы задать выполнение асинхронного доступа;

- `DeleteOnClose` — заставляет операционную систему удалять файл при закрытии его последнего дескриптора. Используется для временных файлов;
- `Encrypted` — вызывает шифрование файла с применением регистрационных данных текущей учетной записи;
- `RandomAccess` — дает подсказку файловой системе оптимизировать кэширование для произвольного доступа;
- `SequentialScan` — дает подсказку файловой системе о том, что чтение файла будет производиться последовательно;
- `WriteThrough` — приводит к игнорированию кэширования и непосредственному переходу к постоянному хранилищу устройства. Обычно параметр замедляет выполнение ввода-вывода. Этому флагу будет подчиняться кэш файловой системы, но у многих устройств хранения данных есть собственная кэш-память, и они могут проигнорировать этот флаг и отрапортовать об успешном завершении еще до записи данных в постоянное хранилище.

Произвольный доступ негативно отражается на работе любого устройства, будь то жесткий диск или ленточный накопитель, нуждающийся в поиске требуемой позиции. С точки зрения производительности последовательный доступ предпочтительнее. Впрочем, многие современные компьютеры оборудованы твердотельными накопителями (SSD), для которых разница между произвольным и последовательным доступом либо минимальна, либо вовсе отсутствует.

Большинству приложений не стоит беспокоиться о конкретном стиле ввода-вывода, пока они полностью не исчерпают возможности устройств, к которым обращаются. Как только это произойдет, придется принимать в расчет тип оборудования, а также разновидность выполняемого доступа и соответствующим образом использовать вышеупомянутые флаги.

Оптимизация настроек HTTP и сетевых соединений

Если приложение выполняет исходящие HTTP-вызовы, то для оптимизации передачи данных по сети существует ряд изменяемых настроек. Но изменения нужно вносить осторожно, поскольку их эффективность сильно зависит от топологии вашей сети и от серверов на другой стороне соединения. Также следует учитывать, где именно находятся целевые конечные точки — в контролируемом вами датацентре или где-то на просторах Интернета. Чтобы разобраться, есть ли польза от этих настроек, нужно все тщательно измерить.

Чтобы изменения по умолчанию касались всех конечных точек, внесите поправки в следующие статические свойства класса `ServicePointManager`:

- `DefaultConnectionLimit` — количество соединений, приходящихся на конечную точку. Более высокое значение этого параметра может увеличить общую про-

пускную способность, если сетевые соединения и обе конечные точки могут с этим справиться;

- `Expect100Continue` — когда клиент инициирует команду `POST` или `PUT`, она, прежде чем приступить к отправке данных, обычно ожидает от сервера сигнал на продолжение `100-Continue`. Это позволяет серверу отклонять запрос, прежде чем будут отправлены данные, экономя при этом пропускную способность. Если вы контролируете обе конечные точки и к вам данная ситуация неприменима, то для улучшения ситуации с задержками это свойство нужно отключить;
- `UseNagleAlgorithm` — оптимизация по алгоритму Nagle, описание которой находится в RFC 896 по адресу https://tools.ietf.org/html/rfc896.html, представляет собой способ сокращения издержек, связанных с передаваемыми по сети пакетами, путем объединения множества мелких пакетов в один большой. Из этого можно извлечь пользу за счет сокращения общих издержек на передачу данных по сети, но это может привести также к задержкам сетевой передачи пакетов. Для современных сетей это свойство обычно должно быть отключено. Можете поэкспериментировать с его отключением и посмотреть, насколько сократится время реагирования.

Некоторые из этих настроек могут также применяться к отдельным объектам `ServicePoint`, чем можно воспользоваться в случае, когда требуется изменять настройки для отдельных конечных точек, возможно, чтобы различать локальные конечные точки дата-центра и конечные точки в Интернете. В дополнение к перечисленным ранее класс `ServicePoint` позволяет контролировать следующие параметры:

- `ConnectionLeaseTimeout` — указывает максимальное время в миллисекундах для поддержания активного подключения в рабочем состоянии. Чтобы активность была бесконечной, для этого параметра следует установить значение `-1`. Эта настройка пригодится для соблюдения баланса нагрузки, когда понадобится периодически заставлять соединение закрываться, чтобы процесс переподключался к разным машинам. Установка для этого параметра значения `0` приведет к закрытию соединения после каждого запроса. Такой вариант не рекомендуется, поскольку создание нового HTTP-соединения обходится слишком дорого;
- `ConnectionLimit` — указывает максимальное количество подключений к данной конечной точке;
- `MaxIdleTime` — указывает максимальное время в миллисекундах, в течение которого соединение может оставаться открытым, находясь в режиме простоя. Чтобы соединение оставалось открытым бесконечно долго, независимо от наличия или отсутствия активности, этому параметру нужно присвоить значение `Timeout.Infinite`;

- `ReceiveBufferSize` — размер буфера, используемого для получения запросов. По умолчанию он составляет 8 Кбайт. Если приходится регулярно получать большие запросы, можно воспользоваться более объемным буфером;
- `SupportsPipelining` — разрешение нескольким запросам отправляться без ожидания ответа, прежде чем отправить последующий запрос. При этом ответы на такие запросы отсылаются в порядке очередности. Дополнительные сведения можно найти в спецификации RFC 2616 по адресу https://tools.ietf.org/html/rfc2616 (стандарт HTTP/1.1).

Можно также заставить отдельно взятый HTTP-запрос закрыть текущее соединение (после отправки ответа на этот запрос), установив для заголовка `KeepAlive` значение `false`.

ИСТОРИЯ

Все данные должны передаваться в оптимальной кодировке. Выполняя профилирование внутренней системы, мы заметили чрезвычайно высокий уровень выделения памяти и использования центрального процессора для конкретного компонента. Проведя некоторые исследования, мы выяснили, что он получал HTTP-ответ, преобразовывал полученные байты в строку в кодировке Base64, раскодировал ее в большой двоичный объект (blob) и в конце выполнял десериализацию blob-объекта в строго типизированный объект. Это приводило к пустым затратам пропускной способности на ненужную перекодировку blob-объекта в строку и к неоправданному расходу ресурсов центрального процессора из-за применения нескольких уровней кодирования, а вдобавок становилось причиной более высоких затрат времени на сборку мусора с выделением памяти для нескольких больших объектов. Урок заключается в том, что следует отправлять только нужные данные, оформив их как можно компактнее. Использование Base64 в наше время вряд ли принесет пользу, особенно для внутренних компонентов. Независимо от того, какой ввод-вывод выполняется, файловый или сетевой, стремитесь к идеалу при кодировании данных. Например, если нужно прочитать ряд целых чисел, не тратьте впустую время центрального процессора, память, дисковое пространство и пропускную способность сети, заключая его в формат XML.

Еще одно предостережение касается принципа, обозначенного в начале этой главы: среда .NET Framework является фреймворком общего назначения. Встроенный HTTP-клиент в целом весьма хорош и вполне приемлем для загрузки содержимого из Интернета, но может не подойти для всех приложений, в частности, при сильной чувствительности приложения к задержкам в высоких процентилях и особенно при наличии запросов внутри дата-центра. Если вас беспокоит 95-й или 99-й процентиль задержек HTTP-запросов, то, возможно, чтобы добрать недостающую производительность, придется создавать собственный HTTP-клиент поверх API WinHTTP. Чтобы сделать все правильно, оправдав потраченные усилия, необходим соответствующий опыт и в работе с HTTP, и в использовании многопоточности в среде .NET.

SIMD

Свойство процессоров, которое называется Single-Instruction, Multiple-Data (один поток команд, несколько потоков данных), позволяет выполнять один набор операций над несколькими фрагментами данных. Большинство современных процессоров платформ x64 как от компании Intel, так и от компании AMD поддерживают эти инструкции, критически важные для параллельных вычислений, требующихся в таких областях, как компьютерные игры, научные и математические вычисления. Последний на момент написания книги JIT-компилятор (4.7.1) допускает ограниченное использование этих инструкций и регистров, главным образом посредством NuGet-пакета System.Numerics.Vectors.

Рассмотрим, к примеру, алгоритм поиска минимального значения в массиве. Простой последовательный алгоритм будет иметь следующий вид:

```
int min = int.MaxValue;
for (int i = 0; i < sourceArray.Length; i++)
{
    if (sourceArray[i] < min)
    {
        min = sourceArray[i];
    }
}
```

Эквивалентная ему SIMD-версия значительно сложнее:

```
var minVector = new Vector <int>(int.MaxValue);
int i = 0;
// обработка массива порциями, равными длине вектора
for (i=0; i < Length - Vector <int>.Count; i += Vector <int>.Count)
{
    Vector <int> subArray = new Vector <int>(this.sourceArray , i);
    minVector = Vector.Min(subArray , minVector);
}

// получение минимума из минимального вектора и оставшихся элементов
int min = Int32.MaxValue;
for (int j = 0; j < Vector <int>.Count; j++)
{
    min = Math.Min(min, minVector[j]);
}
for (i = 0; i < sourceArray.Length; i++)
{
    min = Math.Min(min, sourceArray[i]);
}
```

Поскольку `Vector` использует аппаратные регистры, длина определяется оборудованием и является статической величиной, поэтому в приведенном ранее примере задействуется статическое свойство `Vector<int>.Count`. На моей машине SIMD-версия работает не быстрее не-SIMD-версии. Но для другого алгоритма

ситуация совершенно иная. Здесь алгоритм масштабирует массив, применяя целочисленное значение. Версия, не использующая SIMD, выглядит очень просто:

```
for(int i=0; i < this.sourceArray.Length; i++)
{
    this.destinationArray[i] = this.sourceArray[i] * ScaleFactor;
}
```

А SIMD-версия, пожалуй, еще проще:

```
this.destinationVector = this.sourceVector * ScaleFactor;
```

В этом случае получается огромная разница в производительности. Вот некоторые результаты проведения эталонного тестирования для обоих типов алгоритмов.

Метод	Среднее, нс	Ошибка, нс	Стандартное отклонение (StdDev), нс
Min NonSIMD	4,312	0,1615	0,4761
Min SIMD	8,645	0,0342	0,0320
Scale NonSIMD	4,764	0,0201	0,0188
Scale SIMD	2,363	0,0052	0,0046

Здесь видно, что масштабирование вектора выполняется более чем в два раза быстрее по сравнению с работой версии, не использующей SIMD.

Гарантировать, что любая произвольно взятая SIMD-версия алгоритма обеспечит более быструю работу, чем ее не-SIMD-эквивалент, невозможно. Правильное применение алгоритмов может зависеть от контекста, оборудования, особенностей вашего центрального процессора и схем памяти. Использование SIMD-инструкций сопряжено с дополнительными издержками, и весь смысл состоит в том, чтобы эти потери не превышали выгоды, получаемой от ускорения работы, достигаемого благодаря параллельным вычислениям. Обычно эффективность применения алгоритма на SIMD-основе зависит от того, насколько удастся ограничить выполнение всех операций с данными работой исключительно со структурой `Vector` или другими структурами данных из `System.Numerics.Vectors`. Это эквивалентно выполнению вычислений непосредственно на компьютерном оборудовании. Когда значения переносятся из структуры `Vector` в стандартные структуры данных среды .NET, преимущества, получаемые от параллельных вычислений и аппаратного ускорения, утрачиваются.

Исследование причин возникновения проблем с производительностью

Многие приемы поиска проблем с производительностью среды .NET Framework точно такие же, как и применяемые при их поиске в вашем собственном коде. Когда с помощью инструментальных средств выполняется профилирование использования центрального процессора, выделений памяти, исключений, конфликтных

ситуаций и многого другого, тогда в среде выявляются проблемные участки кода аналогично тому, как они выявляются в вашем собственном коде.

Следует заметить, что PerfView создаст из большинства составляющих среды единую группу, и может понадобиться изменить настройки отображения, чтобы получить более понятную картину производительности элементов среды.

Счетчики производительности. В среде .NET множество категорий счетчиков производительности. В главах 2–4, где давалось описание сборки мусора, JIT-компиляции и асинхронного программирования, также приводились подробные сведения о соответствующих счетчиках производительности, специфичных для рассматриваемых в этих главах тем. В среде .NET есть также счетчики производительности для следующих категорий:

- .NET CLR Data — относящиеся к SQL-клиентам, пулам соединений и командам;
- .NET CLR Exceptions — относящиеся к частоте выдачи исключений;
- .NET CLR Interop — относящиеся к вызову машинного кода из управляемого кода;
- .NET CLR Networking — относящиеся к соединениям и объему переданных данных;
- .NET CLR Remoting — относящиеся к количеству удаленных вызовов, выделениям памяти под объекты, каналам и многому другому;
- .NET CLR Data Provider for SqlServer/Oracle — для различных клиентов баз данных .NET.

В зависимости от конфигурации своей системы вы можете увидеть более широкий или узкий перечень счетчиков по сравнению с представленным здесь.

Резюме

Точно так же, как и при работе с любым другим фреймворком, вы должны разобраться с подробностями реализации используемых API. Ничего не берите на веру.

Будьте осмотрительны при выборе классов коллекций. Учитывайте семантику API, локальность памяти, сложность алгоритмов и использование пространства при выборе коллекции. Отдавайте предпочтение ступенчатым, а не многомерным массивам. Полностью исключите устаревшие необобщенные коллекции вроде `ArrayList` и `HashTable`. Пользуйтесь коллекциями для многопоточной среды, только если нужна синхронизация большинства или всех обращений. Обратите внимание на исходный объем, сравнение ключей и возможность сортировки коллекций и хранящихся в них объектов.

Уделите особое внимание использованию строк и избегайте создания излишних строк. Отдавайте предпочтение более простым, а не сложным API вроде `String.Format`. Работая с подстроками, по возможности пользуйтесь `ReadOnlySpan<T>`.

Избегайте использования API, выдающих исключения в обычных обстоятельствах, провоцирующих выделение памяти в кучи больших объектов или имеющих более затратную реализацию, чем вы ожидаете.

Применяя регулярные выражения, убедитесь в том, что повторно не создаются одни и те же `Regex`-объекты, и всерьез рассматривайте возможность их компиляции, устанавливая флаг `RegexOptions.Compiled`.

Обратите внимание на тип выполняемого ввода-вывода и воспользуйтесь подходящими флагами при открытии файлов, чтобы дать операционной системе возможность оптимизировать производительность в вашу пользу. Для сетевых вызовов отключите `Nagling` и `Expect100Continue`. Передавайте только строго необходимые данные и избегайте использования лишних уровней кодирования.

Прекращайте применять API отражения для выполнения динамически загружаемого кода. Вызывайте такой код через общие интерфейсы или посредством делегатов, созданных с помощью автоматической генерации кода.

Если производятся значительные манипуляции над массивами и матрицами, задействуйте NuGet-пакет System.Numerics.Vectors, чтобы получить преимущества SIMD-команд.

7 Счетчики производительности

Счетчики производительности играют важную роль для отслеживания общей производительности вашего приложения с течением времени. Если вы отвечаете за отслеживание и повышение производительности системы, счетчики производительности помогут справиться с этой задачей. Хотя они могут (и должны) применяться для мониторинга показателей в реальном времени, дополнительную пользу можно получить, если сохранять их в базе данных для анализа на протяжении длительного периода. Это позволит оценить влияние на производительность приложения, оказываемое выпусками новых версий, моделями использования приложения и другими событиями.

Счетчики производительности могут потребляться вашим собственным кодом для самомониторинга, архивирования или автоматического анализа данных. Можно также создать и зарегистрировать собственные счетчики, которые затем станут доступными для мониторинга подобным образом. Соотнесение показателей пользовательских счетчиков вашей программы с показаниями системных счетчиков для приложения зачастую позволяет очень быстро отыскать источник проблем.

Счетчики производительности являются Windows-управляемыми объектами, отслеживающими значения с течением времени. Это могут быть произвольные числа, результаты подсчетов, скорости, промежутки времени и другие разновидности данных. У каждого счетчика есть связанные с ним категория и имя. У большинства счетчиков есть также экземпляры, являющиеся конкретными подразделами по логическим дискретным признакам. Например, счетчик % Processor Time в категории Processor имеет экземпляры для каждого запущенного в данный момент процесса. Многие счетчики также имеют метаэкземпляры, такие как _Total или <Global>, объединяющие данные, получаемые от всех экземпляров. Многие компоненты Windows создают собственные счетчики производительности, и CLR не исключение. Существуют сотни счетчиков, доступных для отслеживания почти каждого аспекта производительности программы. Они рассматривались в предыдущих главах этой книги. Чтобы увидеть все имеющиеся в системе экземпляры счетчиков производительности, воспользуйтесь, как описано в главе 1, утилитой `PerfMon.exe`, поставляемой с Windows. В данной главе рассматривается программный доступ к таким счетчикам как с точки зрения их потребления, так и для создания собственных счетчиков.

Использование существующих счетчиков

Чтобы воспользоваться счетчиком, нужно создать экземпляр класса `PerformanceCounter`, снабжая его необходимыми для отслеживания категорией и именем. Дополнительно можно указать имя экземпляра и машины. Рассмотрим пример, прикрепляющий объект счетчика к счетчику % Processor Time:

```
PerformanceCounter cpuCtr = new PerformanceCounter("Process",
    "% Processor Time", process.ProcessName);
```

Для извлечения значений нужно периодически вызывать в отношении объекта счетчика метод `NextValue`:

```
float value = cpuCtr.NextValue();
```

В документации по API рекомендуется вызывать `NextValue` не чаще одного раза в секунду, чтобы дать счетчику достаточно времени для следующего считывания. Простой пример, показывающий использование нескольких встроенных и пользовательских счетчиков, вы найдете в сопровождающем книгу учебном проекте PerfCountersTypingSpeed.

Создание пользовательского счетчика

Для создания собственного пользовательского счетчика следует получить экземпляр класса `CounterCreationData`, предоставив ему имя и тип. Этот экземпляр добавляется к коллекции, которая затем добавляется к категории:

```
const string CategoryName = "PerfCountersTypingSpeed";
if (!PerformanceCounterCategory.Exists(CategoryName))
{
    var counterDataCollection = new CounterCreationDataCollection();
    var wpmCounter = new CounterCreationData();
    wpmCounter.CounterType =
        PerformanceCounterType.RateOfCountsPerSecond32;
    wpmCounter.CounterName = "WPM";
    counterDataCollection.Add(wpmCounter);

    try
    {
        // Создание категории
        PerformanceCounterCategory.Create(
           CategoryName ,
          "Demo category to show how to create and consume counters",
          PerformanceCounterCategoryType.SingleInstance ,
          counterDataCollection);
    }
    catch (SecurityException )
    {
        // Обработка ошибки – нет прав на применение данного изменения!
    }
}
```

Для создания пользовательского счетчика вы должны иметь разрешение `PerformanceCounterPermission`. На практике это означает, что новые счетчики обычно следует создавать с программой установки, которая может запускаться с повышенными разрешениями. В среде .NET предоставляется класс `PerformanceCounterInstaller`, в который могут заключаться несколько экземпляров класса `CounterCreationData`, после чего эти счетчики будут установлены с поддержкой отката и удаления.

Как описано далее, существует множество счетчиков, сгруппированных в несколько категорий. Кроме того, для некоторых счетчиков определены 32- и 64-разрядные размеры. Можете воспользоваться наиболее подходящими из них для данных, которые записываете.

Счетчики усредненных показателей

Соответствующие счетчики показывают среднее значение двух последних измерений:

- `AverageCount64` — количество обработок во время операции;
- `AverageTimer32` — продолжительность обработки операции;
- `CountPerTimeInterval32/64` — средняя длина очереди для ресурса;
- `SampleCounter` — подсчет количества операций, выполненных за секунду.

Чтобы определить, сколько операций было выполнено с момента последнего обновления счетчика, счетчики `AverageCount64` и `AverageTimer32` нуждаются в помощи второго счетчика, `AverageBase`. Счетчик `AverageBase` должен быть проинициализирован сразу же после счетчика, к которому он применяется. Совместное создание этих двух счетчиков показано в следующем коде:

```
var counterDataCollection = new CounterCreationDataCollection();

// Фактический усредняющий счетчик
var bytesPerTx = new CounterCreationData();
bytesPerTx.CounterType = PerformanceCounterType.AverageCount64;
bytesPerTx.CounterName = "BytesPerTransmission";
counterDataCollection.Add(bytesPerTx);

// Базовый счетчик, помогающий выполнить вычисления
var bytesPerTxBase = new CounterCreationData();
bytesPerTxBase.CounterType = PerformanceCounterType.AverageBase;
bytesPerTxBase.CounterName = "BytesPerTransmissionBase";
counterDataCollection.Add(bytesPerTxBase);

PerformanceCounterCategory.Create(
  "Network Statistics",
  "Network statistics demo counters",
  PerformanceCounterCategoryType.SingleInstance ,
  counterDataCollection);
```

Для установки значений каждый счетчик меняется на количество подсчитываемых объектов и на количество операций, связанных с этими объектами. В нашем примере все это делается довольно просто:

```
bytesPerTx.IncrementBy(request.Length);
bytesPerTxBase.IncrementBy(1);
```

Счетчики мгновенных показателей

Это самые простые счетчики. Они отражают значение последней выборки.

- `NumberOfItems32/64` — показывает последнее значение простого количества.
- `NumberOfItemsHEX32/64` — показывает то же самое, что и `NumberOfItems32/64`, но отображенное в шестнадцатеричном формате.
- `RawFraction` — с базовым счетчиком `RawBase` показывает процент от общего количества. Значение общего количества присваивается базовому счетчику, а поднабор этого общего количества — данному счетчику. В качестве примера можно привести показатель использования диска в процентах.

Дельта-счетчики

Дельта-счетчики показывают разницу между последними двумя значениями счетчика.

- `CounterDelta32/64` — показывает разницу между последними двумя записанными значениями.
- `ElapsedTime` — показывает время от запуска компонента или процесса до настоящего момента. Этим можно воспользоваться, например, для отслеживания рабочего времени вашего приложения. После инициализации этому счетчику не предоставляются никакие новые значения.
- `RateOfCountsPerSecond32/64` — среднее количество операций, выполненных за секунду.

Процентные счетчики

Процентные счетчики демонстрируют процент использования ресурса. В некоторых случаях этот процент может быть больше 100. Рассмотрим многопроцессорную систему: процент использования центрального процессора может быть взят относительно одного ядра. Каждый экземпляр счетчика представляет показатели одного из ядер. Если одновременно задействуются несколько ядер, процентный показатель может быть больше 100.

- `CounterTimer` — показывает процент времени активности компонента в общем времени выборки.
- `CounterTimerInverse` — показывает то же самое, что и `CounterTimer`, за исключением того, что измеряется время, в течение которого компонент неактивен,

и затем вычитается из 100 %. Иными словами, у него будет такое же значение, как и у `CounterTimer`, но оно получается обратным образом.

- `CounterMultiTimer` — показывает то же самое, что и `CounterTimer`, но суммирует активное время всех экземпляров, что может дать процентный показатель больше 100.
- `CounterMultiTimerInverse` — показатель берется из нескольких экземпляров, но выводится из времени пребывания в неактивном состоянии.
- `CounterMultiTimer100Ns` — вместо тактов системного счетчика производительности использует такты с интервалом 100 нс.
- `CounterMultiTimer100NsInverse` — похож на счетчик `CounterMultiTimer100Ns`, но применяет обратную логику.
- `SampleFraction` — показывает отношение подмножества значений и общего количества значений. Для отслеживания общего количества значений использует базовый счетчик.
- `Timer100Ns` — показывает время активности компонента в процентах от общего времени выборки при проведении измерений с шагом по 100 нс.
- `Timer100NsInverse` — похож на счетчик `Timer100Ns`, но применяет обратную логику.

Всем счетчикам, название которых начинается с `CounterMulti-`, требуется, как и ранее рассмотренному счетчику `AverageCount64`, задействовать `CounterMultiBase`. При создании собственных счетчиков производительности следует учесть, что их не нужно обновлять слишком часто. Стоит взять за правило делать это максимум раз в секунду, поскольку данные никогда не будут предоставляться чаще. Если нужно создавать большие объемы данных о производительности, куда более подходящим будет вариант использования ETW-событий.

Резюме

Счетчики производительности — это основные строительные блоки анализа производительности. Вашей программе совсем не обязательно создавать собственные счетчики, но если в ней есть отдельные операции, фазы или элементы, влияющие на производительность, подумайте об этом.

Рассмотрите возможность автоматизированного получения и анализа показателей счетчиков через API среды .NET, чтобы обеспечить архивирование и постоянную реакцию на состояние производительности системы.

8 ETW-события

В предыдущей главе рассматривались счетчики производительности, которые великолепно подходят для отслеживания общей производительности вашего приложения. Но они не могут предоставить информацию о конкретном событии или операции, проводимой в приложении. Для этого необходимо логировать данные для каждой операции отдельно. При включении меток времени появляется возможность отслеживать производительность в программе подробнейшим образом.

Для среды .NET существует множество библиотек логирования. Среди них есть как такие популярные, например log4net, так и бесчисленное множество сторонних решений. Я же настоятельно рекомендую использовать библиотеку Event Tracing для Windows, обладающую целым рядом преимуществ.

1. Она встроена в операционную систему.
2. Работает очень быстро и хорошо вписывается в высокопроизводительные сценарии. Хотя это не обходится без затрат, особенно в некоторых сценариях. Максимальными они могут стать в крупных многопоточных приложениях, записывающих большое количество событий. Собственная система событий может также понадобиться играм, интенсивно использующим ресурсы.
3. В ней имеется автоматическая буферизация.
4. В ходе выполнения приложения вы можете включать и выключать потребление и выдачу событий в динамическом режиме.
5. В ней есть высокоизбирательный механизм фильтрации.
6. Она предоставляет возможность объединять события из нескольких источников в один поток логов для всестороннего анализа.
7. ETW-события выдаются всеми подсистемами операционной системы и среды .NET.
8. События не только имеют строковое отображение, но и делятся по типам и порядку следования.

PerfView и многие другие инструменты профилирования — не что иное, как замысловатые ETW-анализаторы. Например, в главе 2 было показано, как воспользоваться PerfView для анализа выделений памяти. Вся информация об этом поступала из ETW-событий среды CLR.

В этой главе будут исследованы способы определения и потребления своих собственных событий. Все классы находятся в пространстве имен `System.Diagnostics.Tracing` и доступны начиная с .NET 4.5.

Можно определить события, помечающие начало и завершение выполнения вашей программы, различные стадии обрабатываемых запросов, возникающие ошибки и буквально все что угодно. Вы можете полностью контролировать информацию, попадающую в событие.

Использование ETW начинается с определения того, что называется поставщиками. В понятиях среды .NET это класс с методами, определяющими события, которые вам нужно регистрировать. Эти методы могут принимать любые имеющиеся в среде .NET основные типы данных, например строки, целочисленные значения и многое другое.

События потребляются объектами, названными слушателями (listeners), которые подписываются на интересующие их события. Если подписчиков у событий нет, они перестают применяться. Этим обусловливается очень незначительный средний уровень затрат на обслуживание ETW.

Определение событий

События определяются с помощью класса, который, как в следующем примере, является производным класса `EventSource`:

```
using System.Diagnostics.Tracing;
namespace EtlDemo
{
     [EventSource(Name="EtlDemo")]
     internal sealed class Events : EventSource
     {
          ...
     }
}
```

Аргумент `Name` нужен, если требуется, чтобы слушатели находили источник по имени. Можно также предоставить GUID, но это не обязательно, и если этого не сделать, его автоматически сгенерирует процедура, спецификация которой имеется в RFC 4122 (см. https://tools.ietf.org/html/rfc4122). GUID-идентификаторы необходимы, только если требуется обеспечить уникальный источник событий. Если источник и слушатель существуют в едином процессе, то вам не нужно даже имя, и объект, происходящий из `EventSource`, можно напрямую передать слушателю.

После этого базового определения нужно при задании собственных событий соблюсти некоторые соглашения. Чтобы их проиллюстрировать, я дам определения нескольким событиям для очень простой тестовой программы, которую вы найдете в учебном проекте EtlDemo:

```
using System.Diagnostics.Tracing;

namespace EtlDemo
```

```
{
      [EventSource(Name="EtlDemo")]
     internal sealed class Events : EventSource
     {

          public static readonly Events Log = new Events();

          public class Keywords
          {
               public const EventKeywords General = (EventKeywords)1;
               public const EventKeywords PrimeOutput = (EventKeywords)2;
          }

          internal const int ProcessingStartId = 1;
          internal const int ProcessingFinishId = 2;
          internal const int FoundPrimeId = 3;

         [Event(ProcessingStartId ,
                Level = EventLevel.Informational ,
                Keywords = Keywords.General)]
        public void ProcessingStart()
        {
            if (this.IsEnabled())
            {
                this.WriteEvent(ProcessingStartId);
            }
        }

         [Event(ProcessingFinishId ,
                Level = EventLevel.Informational ,
                Keywords = Keywords.General)]
        public void ProcessingFinish()
        {
            if (this.IsEnabled())
            {
                this.WriteEvent(ProcessingFinishId);
            }
        }

         [Event(FoundPrimeId ,
                Level = EventLevel.Informational ,
                Keywords = Keywords.PrimeOutput)]
        public void FoundPrime(long primeNumber)
        {
            if (this.IsEnabled())
            {
                this.WriteEvent(FoundPrimeId , primeNumber);
            }
        }
    }
}
```

Первым делом объявляется статическая ссылка Log на экземпляр объявляемого класса. Это типовое действие, поскольку события по своей природе носят, как правило, глобальный характер и к ним нужен доступ из многих частей вашего кода. Наличие этой глобальной переменной значительно облегчает жизнь, по сравнению с необходимостью передачи ссылки на объект, унаследованный от EventSource, всему, что в нем нуждается. Этой ссылке можно присвоить любое имя, но для разборчивости нужно придумать стандартное название для всех ваших источников событий.

За этим объявлением следует внутренний класс, в котором определяются некоторые значения констант Keywords (ключевые слова), являющихся необязательными и имеющих произвольные значения, полностью зависящие от вас. Они служат способом разбиения событий по категориям, имеющим смысл для вашего приложения. Слушатели могут фильтровать то, что им необходимо, на основе ключевых слов. Следует заметить, что ключевые слова обрабатываются как битовые флаги, поэтому им следует присваивать степени числа два в качестве значений. Тогда в слушателях можно будет легко указать сразу несколько отслеживаемых ключевых слов.

Далее следуют объявления констант для идентификаторов событий. Объявлять константы не требуется, но они удобны в случае, если и источнику, и слушателю нужно ссылаться на идентификаторы.

И наконец, за ними идет перечень событий. Они определяются с помощью метода, возвращающего пустое значение (void), получающего любое количество произвольных аргументов. Этим методам предшествует атрибут, указывающий идентификатор, уровень события и любые ключевые слова, которые нужно применить. Можно использовать сразу несколько ключевых слов, объединив их с помощью логического оператора ИЛИ: Keywords = Keywords.General | Keywords.PrimeOutput.

Существует пять уровней событий:

- LogAlways — логируется всегда, вне зависимости от каких-либо обстоятельств, не обращая внимания на установленный уровень логирования;
- Critical — очень серьезная ошибка, возможно служащая признаком невозможности восстановления нормальной работы программы;
- Error — обычная ошибка;
- Warning — нельзя считать полноценной ошибкой, но кому-то может понадобиться среагировать на это событие;
- Informational — чисто информационное сообщение, не указывает на ошибку;
- Verbose — во многих ситуациях вообще не должно логироваться. Используется для отладки конкретных проблемных мест или при запуске в определенных режимах.

Эти уровни имеют накопительный характер. Указание уровня логирования означает, что вы будете получать все события этого и всех вышестоящих уровней. Например, если задан уровень логирования Warning, вы будете также получать события для Error, Critical и LogAlways.

Тело события устроено просто. Сначала проверяем, включены ли события вообще (они могут быть отключены в первую очередь в целях оптимизации

производительности). Если они включены, вызываем метод `WriteEvent`, наследуемый из `EventSource`, с идентификатором события и вашими аргументами.

ПРИМЕЧАНИЕ

Остерегайтесь логирования значений null. Прежние версии системы EventSource не знали, как их правильно интерпретировать, из-за отсутствия информации о типе. Это было характерно в основном для строковых значений. Похоже, что в самых последних версиях среды .NET такие значения обрабатываются правильно, при этом null-строки без оповещения заменяются пустыми строками. Чтобы проявить максимум осторожности, выполните проверку на null и замените на разумное значение по умолчанию:

```
[Event(5,
       Level = EventLevel.Informational ,
       Keywords = Keywords.General)]
public void Error(string message)
{
  if (IsEnabled())
  {
    this.WriteEvent(5, message ?? string.Empty);
  }
}
```

Существует свыше десяти перегрузок метода `WriteEvent`, принимающих различные комбинации типов параметров. Если вызову не соответствует ни одна из перегрузок, используется вариант по умолчанию `WriteEvent(int eventId, params object[] args)`. Этого метода следует избегать — его выполнение влечет за собой выделения памяти и отражение с целью выяснения правильного типа объектов.

Чтобы записать собственные события, в вашем коде нужно просто сделать нечто подобное:

```
Events.Log.ProcessingStart();
Events.Log.FoundPrime(7);
```

Потребление пользовательских событий в PerfView

Теперь, когда приложение выдает ETW-события, их можно захватить в любом ETW-слушателе, например PerfView, Windows Performance Analyzer или встроенной в Windows утилите PerfMon.

Для захвата событий из пользовательских источников в PerfView нужно поместить имя, предварив его символом звездочки (*), в текстовое поле Additional Providers (Дополнительные поставщики), которое находится в окне Collect (Сбор) (рис. 8.1).

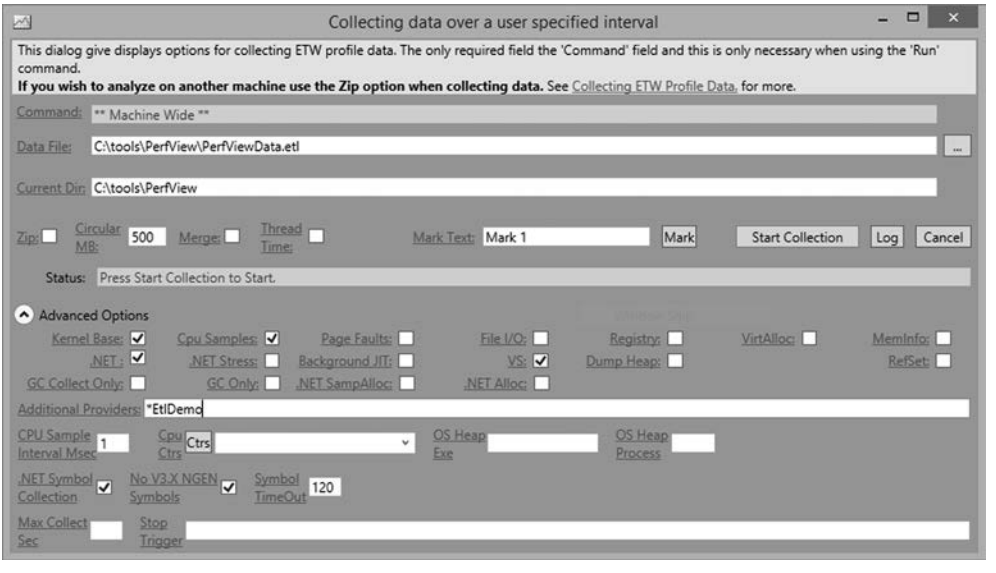

Рис. 8.1. Окно Collect в PerfView показывает, куда вводить дополнительные ETW-поставщики

Вписав в текстовое поле `*EtlDemo`, вы сообщаете PerfView о необходимости выполнить рассмотренное ранее в этой главе автоматическое вычисление GUID. Дополнительную информацию можно увидеть после щелчка на заголовке-ссылке Additional Providers (Дополнительные поставщики).

Начните собирать выборки, запустите `EtlDemo`, а затем нажмите кнопку Stop Collection (Остановить сбор). После того как итоговые события будут записаны, откройте простой узел Events (События). Будет показан список всех захваченных событий, включая следующие:

- `EtlDemo/FoundPrime`;
- `EtlDemo/ManifestData`;
- `EtlDemo/ProcessingStart`;
- `EtlDemo/ProcessingFinish`.

Если выделить все события в списке и нажать кнопку Update (Обновить), чтобы актуализировать представление, появится список интересующих вас событий (рис. 8.2).

В нем пользовательские события показаны в контексте всех остальных захваченных событий. К примеру, можно увидеть JIT-события, предшествующие событиям `FoundPrime`. Это намекает, какие огромные возможности дает вам грамотный ETW-анализ. В контексте сценария вашего собственного приложения можно проделать скрупулезные исследования производительности. Простой пример будет показан в этой главе чуть позже.

EtlDemo/ProcessingStart	2,701.303	EtlDemo (8296)	ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,701.345	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,701.424	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,701.466	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,701.975	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Windows Kernel/PerfInfo/SampleProf	2,702.102	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,702.342	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,702.533	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
EtlDemo/FoundPrime	2,702.564	EtlDemo (8296)	ThreadID="8,556" primeNumber="0
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,702.596	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Windows Kernel/PerfInfo/SampleProf	2,703.167	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,703.191	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
EtlDemo/FoundPrime	2,704.007	EtlDemo (8296)	ThreadID="8,556" primeNumber="0

Рис. 8.2. Отсортированный список событий Windows, .NET и приложения

Создание собственного слушателя ETW-событий

При разработке большинства приложений вам вряд ли потребуется создавать собственные ETW-слушатели. Почти всегда достаточно определить свои события и использовать приложение вроде PerfView для их сбора и анализа. Но может понадобиться создать слушатель при необходимости иметь собственное средство логирования или, к примеру, для проведения близкого к реальному времени анализа событий.

В среде .NET слушатель событий представляет собой класс, производный от класса `EventListener`. Чтобы показать несколько способов обработки данных о событиях, я определю базовый класс для обобщенного управления слушателями.

Этому классу нужно знать, какие события должны отслеживаться и по каким уровням и ключевым словам отфильтровываться, поэтому сначала следует определить простую структуру для инкапсуляции этой информации:

```
class SourceConfig
{
    public string Name { get; set; }
    public EventLevel Level { get; set; }
    public EventKeywords Keywords { get; set; }
}
```

Затем можно определить конструктор слушателя, который будет принимать коллекцию объектов этого класса (по одному для каждого источника события):

```
abstract class BaseListener : EventListener
{
    List<SourceConfig > configs = new List<SourceConfig >();
    protected BaseListener(
        IEnumerable <SourceConfig > sources)
    {
```

```
        this.configs.AddRange(sources);

        foreach (var source in this.configs)
        {
            var eventSource = FindEventSource(source.Name);
            if (eventSource != null)
            {
                this.EnableEvents(eventSource ,
                        source.Level ,
                        source.Keywords);
            }
        }
    }

    private static EventSource FindEventSource(string name)
    {
        foreach (var eventSource in EventSource.GetSources())
        {
            if (string.Equals(eventSource.Name, name))
            {
                return eventSource;
            }
        }
        return null;
    }
}
```

После сохранения источников во внутренний список выполняется их последовательный обход и предпринимается попытка найти существующий источник события `EventSource`, совпадающий с нужным нам по именам. Если такой источник найден, происходит подписка путем вызова унаследованного метода `EnableEvents`.

Но этого недостаточно. Вполне возможно, что `EventSource` создан после установки слушателя. Учитывая это, можно переопределить метод `OnEventSourceCreated` и провести точно такую же проверку, чтобы понять, представляет ли новый `EventSource` для нас интерес:

```
protected override void OnEventSourceCreated(
    EventSource eventSource)
{
    base.OnEventSourceCreated(eventSource);

    foreach (var source in this.configs)
    {
        if (string.Equals(source.Name, eventSource.Name))
        {
            this.EnableEvents(eventSource ,
                              source.Level ,
                              source.Keywords);
        }
    }
}
```

И последнее, что нужно сделать, — обработать событие `OnEventWritten`, которое выдается при каждой записи нового события, сделанной источниками, представляющими интерес для данного слушателя:

```
protected override void OnEventWritten(
    EventWrittenEventArgs eventData)
{
    this.WriteEvent(eventData);
}

protected abstract void WriteEvent(
    EventWrittenEventArgs eventData);
```

В данном случае я просто полагаюсь на абстрактный метод, которому придется взять на себя всю тяжелую работу.

Обычно определяют несколько типов слушателей, которые выводят данные о событиях различными способами. Для этого примера я определил один слушатель, который выводит сообщения в консоль, и еще один, который выполняет логирование в файл.

Класс `ConsoleListener` имеет следующий вид:

```
class ConsoleListener : BaseListener
{
    public ConsoleListener(
        IEnumerable <SourceConfig > sources)
            :base(sources)
    {
    }

    protected override void WriteEvent(
            EventWrittenEventArgs eventData)
    {
        string outputString;
        switch (eventData.EventId)
        {
            case Events.ProcessingStartId:
                outputString = string.Format("ProcessingStart ({0})",
                                eventData.EventId);
                break;
            case Events.ProcessingFinishId:
                outputString = string.Format("ProcessingFinish ({0})",
                                eventData.EventId);
                break;
            case Events.FoundPrimeId:
                outputString = string.Format("FoundPrime ({0}): {1}",
                                eventData.EventId ,
                                (long)eventData.Payload[0]);
                break;
            default:
                throw new InvalidOperationException("Unknown event");
        }
```

```
        Console.WriteLine(outputString);
    }
}
```

Свойство `EventId` определяет, с каким событием вы имеете дело. К сожалению, получение имени события сложнее, но, как будет показано чуть позже, оно возможно при выполнении некоторой предварительной работы. Свойство `Payload` предоставляет вам массив значений, переданных в метод исходного события.

Слушатель `FileListener` чуть сложнее:

```
class FileListener : BaseListener
{
    private StreamWriter writer;

    public FileListener(IEnumerable <SourceConfig > sources ,
                        string outputFile)
      :base(sources)
    {
        writer = new StreamWriter(outputFile);
    }

    protected override void WriteEvent(
        EventWrittenEventArgs eventData)
    {
        StringBuilder output = new StringBuilder();
        DateTime time = DateTime.Now;
        output.AppendFormat("{0:yyyy-MM-dd-HH:mm:ss.fff} - {1} - ",
                  time, eventData.Level);
        switch (eventData.EventId)
        {
            case Events.ProcessingStartId:
                output.Append("ProcessingStart");
                break;
            case Events.ProcessingFinishId:
                output.Append("ProcessingFinish");
                break;
            case Events.FoundPrimeId:
                output.AppendFormat("FoundPrime - {0:N0}",
                            eventData.Payload[0]);
                break;
            default:
                throw new InvalidOperationException("Unknown event");
        }
        this.writer.WriteLine(output.ToString());
    }

    public override void Dispose()
    {
        this.writer.Close();

        base.Dispose();
    }
}
```

Этот фрагмент кода из проекта EtlDemo показывает способ использования обоих слушателей и подписывает их на различные ключевые слова и уровни:

```
var consoleListener = new ConsoleListener(
    new SourceConfig[]
    {
        new SourceConfig(){
            Name = "EtlDemo",
            Level = EventLevel.Informational ,
            Keywords = Events.Keywords.General}
    });

var fileListener = new FileListener(
    new SourceConfig[]
    {
        new SourceConfig(){
        Name = "EtlDemo",
        Level = EventLevel.Verbose ,
        Keywords = Events.Keywords.PrimeOutput}
    },
    "PrimeOutput.txt");

long start = 1000000;
long end = start + 10000;

Events.Write.ProcessingStart();
for (long i = start; i < end; i++)
{
    if (IsPrime(i))
    {
        Events.Write.FoundPrime(i);
    }
}

Events.Write.ProcessingFinish();
consoleListener.Dispose();
fileListener.Dispose();
```

Сначала создаются два типа слушателей, которые подписываются на различные наборы событий. Затем регистрируются определенные события и выполняется программа.

В консоли получается вывод только таких данных:

```
ProcessingStart (1)
ProcessingFinish (2)
```

В то же время вывод в файл содержит следующие строки:

```
2014-03-08-15:21:31.424 - Informational - FoundPrime - 1,000,003
2014-03-08-15:21:31.425 - Informational - FoundPrime - 1,000,033
2014-03-08-15:21:31.425 - Informational - FoundPrime - 1,000,037
```

Обработка событий в абсолютно реальном времени невозможна. Во-первых, получение событий происходит не в тех потоках, в которых они генерируются. Во-вторых, события из нескольких источников могут быть собраны в одном слушателе, это предполагает, что в какие-то моменты времени они выстраиваются в очередь.

Получение подробных данных об EventSource

В предыдущих разделах можно было заметить кое-что интересное: наш собственный слушатель событий не знает имени получаемого им события, а вот PerfView как-то с этим справляется. Такое возможно благодаря тому, что с каждым `EventSource` связан манифест, который представляет собой простое XML-описание источника события.

К счастью, среда .NET упрощает создание этого манифеста из класса `EventSource`:

```
string xml =
    EventSource
    .GenerateManifest(typeof(Events), string.Empty);
```

Вот как выглядит манифест для наших собственных ранее определенных событий:

```
<instrumentationManifest
    xmlns="http://schemas.microsoft.com/win/2004/08/events">
        <instrumentation
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
            xmlns:win=...>
                <events xmlns=...>
                    <provider name="EtlDemo"
                        guid="{458d4a63 -7cc9 -5239-62c4-f8aebbe597ac}"
                        resourceFileName=""
                        messageFileName=""
                        symbol="EtlDemo">
                    <tasks>
                        <task name="FoundPrime"
                            value="65531"/>
                        <task name="ProcessingFinish"
                            value="65532"/>
                        <task name="ProcessingStart"
                            value="65533"/>
                    </tasks>
                    <opcodes>
                    </opcodes>
                    <keywords>
                        <keyword name="General"
                                message="$(string.keyword_General)"
                                mask="0x1"/>
                        <keyword name="PrimeOutput"
                                message="$(string.keyword_PrimeOutput)"
                                mask="0x2"/>
                    </keywords>
                    <events>
```

```
                    <event value="1"
                        version="0"
                        level="win:Informational"
                        keywords="General"
                        task="ProcessingStart"/>
                    <event value="2"
                        version="0"
                        level="win:Informational"
                        keywords="General"
                        task="ProcessingFinish"/>
                    <event value="3"
                        version="0"
                        level="win:Informational"
                        keywords="PrimeOutput"
                        task="FoundPrime"
                        template="FoundPrimeArgs"/>
                </events>
                <templates>
                    <template tid="FoundPrimeArgs">
                        <data name="primeNumber"
                            inType="win:Int64"/>
                    </template>
                </templates>
            </provider>
        </events>
    </instrumentation>
    <localization>
        <resources culture="en-US">
            <stringTable>
                <string id="keyword_General"
                        value="General"/>
                <string id="keyword_PrimeOutput"
                        value="PrimeOutput"/>
            </stringTable>
        </resources>
    </localization>
</instrumentationManifest>
```

Чтобы обеспечить возможность исследования используемых вами типов и в результате этого создать манифест, средой .NET выполняются очень полезные неявные действия. Для системы логирования с расширенными функциональными возможностями можно проанализировать XML, чтобы получить имена событий и сравнить их с идентификаторами, а также типами всех аргументов.

Потребление событий CLR и системы

Если нужно потреблять события .NET или другие системные события Windows, сделать это можно с помощью доступной и очень надежной библиотеки. Это TraceEvent — тот же самый механизм работы с событиями, который обеспечивает эффективность применения PerfView. Библиотека была издана в качестве NuGet-пакета

Microsoft.Diagnostics.Tracing.TraceEvent. Она предоставляет довольно простые средства для обработки многих типов CLR-событий и событий операционной системы в вашем коде. Можно воспользоваться ею для создания собственного аналитического инструмента или даже поместить ее в вашу производственную систему, что позволит ей реагировать на поведение сборщика мусора практически в реальном масштабе времени.

Вот небольшой пример, отслеживающий в системе Start- и Stop-события сборщика мусора:

```
using Microsoft.Diagnostics.Tracing.Parsers;
using Microsoft.Diagnostics.Tracing.Session;
using System;
using Microsoft.Diagnostics.Tracing.Parsers.Clr;

namespace GCListener
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press Ctrl-C to exit");

            using (var session =
              new TraceEventSession("GCListenSession"))
            {
                Console.CancelKeyPress += (a,b) => session.Stop();
                session.EnableProvider(
             ClrTraceEventParser.ProviderGuid ,
             Microsoft.Diagnostics.Tracing.TraceEventLevel.Informational ,
             (ulong)(ClrTraceEventParser.Keywords.GC));

                session.Source.Clr.GCStart += Clr_GCStart;
                session.Source.Clr.GCStop += Clr_GCStop;

                // Это будет работать, пока не будет вызван метод Session.Stop()
                session.Source.Process();
            }
        }

        private static void Clr_GCStart(
          Microsoft.Diagnostics.Tracing.Parsers.Clr.GCStartTraceData
          gcStartData)
            {
                Console.WriteLine(
            $"GCStart: Process: {gcStartData.ProcessID}, " +
            $"Gen: {gcStartData.Depth}, Type: {gcStartData.Type}"
            }

            private static void Clr_GCStop(GCEndTraceData gcEndData)
            {
                Console.WriteLine(
```

```
                $"GCStop: Process: {gcEndData.ProcessID}, " +
                $"Gen: {gcEndData.Depth}");
            }
        }
    }
```

Вывод будет выглядеть примерно так:

```
Press Ctrl-C to exit
GCStart: Process: 84592, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 0
GCStart: Process: 84592, Gen: 1, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 1
GCStart: Process: 84592, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 0
GCStart: Process: 97844, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 97844, Gen: 0
```

Пользовательские аналитические расширения PerfView

Если TraceEvent не способен полностью удовлетворить ваши потребности, можно подстроить под них средство PerfView для автоматизации анализа на более высоком уровне, воспользовавшись его весьма впечатляющими возможностями группировки и свертки для создания отфильтрованных, соответствующих вашим потребностям информационных стеков для вашего приложения.

Чтобы было с чего начать, PerfView поставляется с собственным встроенным в сам исполняемый файл PerfView образцом проекта для создания расширения. Чтобы сгенерировать решение-образец, наберите в приглашении командной строки:

```
PerfView.exe CreateExtensionProject MyProjectName
```

В результате будут созданы файл решения, файл проекта и файл образца исходного кода в комплекте с некоторыми образцами кода, позволяющими вам приступить к работе. Вот кое-что из того, что вы можете реализовать.

- Создать отчет, показывающий, какие сборки используют наибольшее количество ресурсов ЦПУ. В образце уже есть демонстрационная команда, занимающаяся именно этим.
- Автоматизировать анализ работы центрального процессора, экспортируя XML-файл, показывающий самые затратные составляющие вашей программы, основываясь на каких-либо критериях.
- Создать представление со сложными схемами свертки и группировки, которые вы часто используете.
- Создать представление, показывающее выделение памяти для конкретной операции в программе, где операция определяется вашими собственными пользовательскими ETW-событиями.

Задействуя пользовательские расширения и режим командной строки PerfView (без GUI), можно создать сценарный инструмент профилирования, предоставляющий вам удобные для анализа отчеты о наиболее интересных областях приложения.

Рассмотрим простой пример анализа частоты событий `FoundPrime` из учебной программы EtlDemo. Сначала я захватил события с помощью PerfView, собирая их обычным способом с использованием поставщика `*EtlDemo`:

```
public void AnalyzePrimeFindFrequency(string etlFileName)
{
    using (var etlFile = OpenETLFile(etlFileName))
    {
        var events = GetTraceEventsWithProcessFilter(etlFile);

        const int BucketSize = 10000;
        // Каждая запись представляет собой простые числа
        // и время, затраченное на их поиск
        List<double > primesPerSecond = new List<double >();

        int numFound = 0;
        DateTime startTime = DateTime.MinValue;

        foreach (TraceEvent ev in events)
        {
            if (ev.ProviderName == "EtlDemo")
            {
                if (ev.EventName == "FoundPrime")
                {
                    if (numFound == 0)
                    {
                        startTime = ev.TimeStamp;
                    }

                    var primeNumber = (long)ev.PayloadByName("primeNumber");
                    if (++numFound == BucketSize)
                    {
                        var elapsed = ev.TimeStamp - startTime;
                        double rate = BucketSize / elapsed.TotalSeconds;
                        primesPerSecond.Add(rate);
                        numFound = 0;
                    }
                }
            }
        }

        var htmlFileName = CreateUniqueCacheFileName(
                    "PrimeRateHtmlReport", ".html");
        using (var htmlWriter = File.CreateText(htmlFileName))
        {
            htmlWriter.WriteLine("<h1>Prime Discovery Rate </h1>");
```

```
            htmlWriter.WriteLine("<p>Buckets: {0}</p>",
                primesPerSecond.Count);
            htmlWriter.WriteLine("<p>Bucket Size: {0}</p>", BucketSize);
            htmlWriter.WriteLine("<p>");
            htmlWriter.WriteLine("<table border=\"1\">");
            for (int i = 0; i < primesPerSecond.Count; i++)
            {
                htmlWriter.WriteLine(
                    "<tr><td>{0}</td><td>{1:F2}/sec</td></tr>",
                    i,
                    primesPerSecond[i]);
            }
            htmlWriter.WriteLine("</table >");
        }

        OpenHtmlReport(htmlFileName , "Prime Discovery Rate");
    }
}
```

Расширение можно запустить, воспользовавшись следующей командной строкой:

```
PerfView userCommand MyProjectName.AnalyzePrimeFindFrequency
    PerfViewData.etl
```

Все, что следует за именем расширения, передается в метод в качестве аргументов.

Выводом станет окно в PerfView, которое выглядит как веб-страница, где появится введенная вами информация (рис. 8.3).

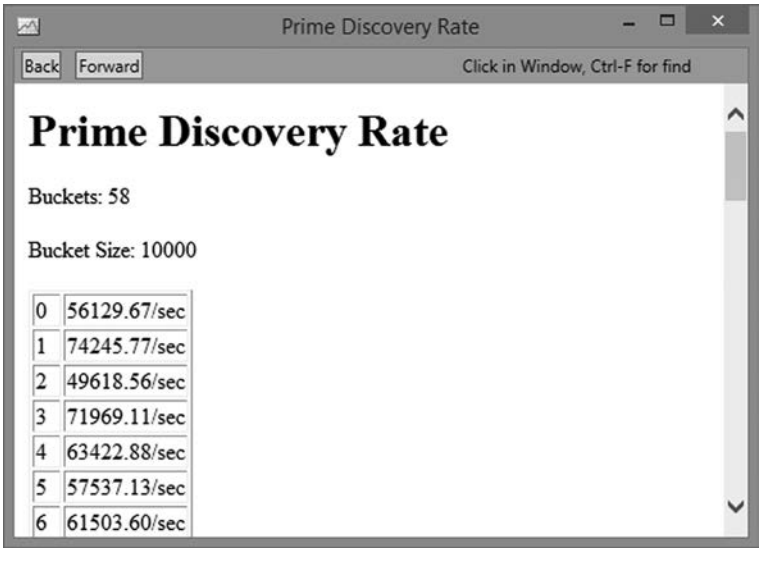

Рис. 8.3. HTML-вывод нашего пользовательского анализа ETW

Следует заметить, что возможность применения расширений официально API не поддерживается. Во внутренний API PerfView в прошлом вносились существенные изменения, несовместимые с предыдущими версиями и, вполне возможно, это повторится и в будущем.

Резюме

ETW-события — предпочтительный метод логирования дискретных событий в приложении. Они идеально подходят как для ведения лога приложения, так и для отслеживания подробной информации, касающейся производительности.

В большинстве случаев все необходимые инструменты для исследования могут быть предоставлены PerfView или другими приложениями для анализа ETW, но если вам нужна специализированная аналитика, воспользуйтесь библиотекой TraceEvent или напишите собственное средство для анализа с помощью EventListener. Чтобы воспользоваться преимуществами расширенной группировки и свертки, создайте расширение PerfView.

9 Безопасность и анализ кода

Разработчики программных продуктов часто говорят, что сначала нужно заставить код работать, а уж потом заботиться о быстроте его выполнения. Эта книга посвящена главным образом повышению производительности, но данная глава немного отклоняется от основной темы в сторону решения важных вопросов, не имеющих прямого отношения к производительности, но способных помочь вам создать высокопроизводительные масштабируемые приложения. Ежедневная работа по обеспечению стабильности и надежности кода позволяет избавиться от необходимости существенно изменять его, чтобы повысить производительность. А если при этом возникают проблемы, становится намного проще понять, что их вызвало.

Представление об операционной системе, API и оборудовании

При глубокой оптимизации производительности придется отказаться от всевозможных абстракций, вводимых по вашему желанию в программный продукт. Как уже неоднократно упоминалось в этой книге, вы должны понимать используемые вами API, чтобы осознанно принимать решения о том, как их применять и применять ли их вообще.

Но одного этого недостаточно. Возьмем, к примеру, многопоточность. Хотя в различные версии среды .NET поверх потоков были добавлены абстракции, упрощающие асинхронное программирование, получить максимальную выгоду от этих функций можно, лишь досконально разбираясь в том, как они взаимодействуют с потоками базовой операционной системы и каков используемый ими алгоритм диспетчеризации потоков. Это справедливо и для отладки операций с памятью. Исследовать кучу сборщика мусора просто, но если имеется очень крупный процесс, загружающий тысячи типов из сотен сборок, можно столкнуться с проблемами и за пределами управляемого мира, решение которых потребует от вас представления о полной структуре памяти.

И наконец, не менее важную роль играет оборудование. В главе, посвященной JIT-компиляции, уже упоминались такие явления, как локальность ссылок — помещение битов кода и данных, используемых вместе, физически близко друг к другу в памяти, чтобы они могли эффективно включаться в кэш-память процессора. Если повезет, ваш код будет ориентирован на одну аппаратную платформу. А если нет, то

нужно будет разобраться, в чем разница выполнения кода на каждой из платформ. Увас могут быть разные лимиты памяти, размеры кэша или даже более существенные различия, например совершенно разные модели памяти. Оборудование влияет также на то, какие могут быть обнаружены проблемы с многопоточностью, — некоторые платформы будут более снисходительны к небрежной синхронизации, а другие ее не простят.

Все это нужно сопоставить с тем, что оптимизировать абсолютно все невозможно, — профилирование нужно использовать, чтобы сфокусироваться на тех областях кода, которые требуют наибольшего внимания. Затрачиваемые усилия должны быть строго пропорциональны реальной потребности повышения производительности. В то же время чем лучше вы разбираетесь в строительных блоках, с которыми работаете, тем меньших времени и усилий потребует весьма затратная оптимизация производительности.

Ограничение использования API в определенных областях кода

Позволять всем компонентам активно использовать каждый API среды или системы нет никакого смысла. Например, при наличии модели обработки, основанной исключительно на применении задач `Task`, следует сосредоточиться на этой функциональности и запретить другим компонентам обращаться к чему-либо в пространстве имен `System.Threading`.

Такие правила рода важны, в частности, для систем с моделью расширения. Обычно требуется, чтобы платформа выполняла весь жесткий опасный код, а на долю расширений оставались простые действия в соответствующих им областях.

Один из способов предотвратить использование опасных подходов к написанию кода или преодолеть проблемы заведомо низкой производительности — статический анализ кода. В среде .NET есть два основных инструментальных средства для выполнения этой задачи, которые можно свободно использовать, — FxCop и .NET Compiler Code Analyzers.

FxCop — более старое средство, оно работает с уже скомпилированными DLL-библиотеками, а значит, может разбираться только в языке MSIL. А средство .NET Compiler Code Analyzers (вы могли слышать его прежнее название — Roslyn Code Analyzers) имеет доступ ко всему дереву синтаксиса в самом компиляторе и может запускаться в ходе разработки. Я приведу примеры использования обоих средств, но для всех будущих разработок рекомендую .NET Compiler Code Analyzers.

Пользовательские правила FxCop

FxCop — статический анализатор кода, свободно поставляемый с Visual Studio. У него есть стандартные правила в таких категориях, как Performance («Производительность»), Globalization («Глобализация»), Security («Безопасность») и др., но вы можете добавить библиотеку собственных правил. Многие правила

производительности, уже рассмотренные в книге, могут быть представлены в виде правил FxCop, например:

- запрет на использование «опасных» пространств имен;
- запрет на применение типов или API, вызывающих выделения памяти в LOH;
- запрет на использование API, у которых есть более подходящие альтернативы, например `TryParse` взамен `Parse`;
- поиск случаев двойного приведения типов;
- поиск случаев упаковки;
- применение конкретных правил написания кодов для некоторых типов (например, все `Regex`-объекты должны быть статическими, предназначенными только для чтения и созданными с установленным флагом `RegexOptions.Compiled`).

Создавая правила, следует учесть, что FxCop может анализировать только IL и метаданные. Это средство не понимает код C# или любого другого языка высокого уровня. Поэтому вы не сможете навязать соблюдение статических правил, зависящих от специфических шаблонов того или иного языка. Написание собственных правил FxCop дается довольно легко, но официальная документация, регламентирующая эту область, практически отсутствует и вам придется полагаться на анализ IL своих программ и на интенсивное использование автодополнения IntelliSense, чтобы продраться через FxCop API. Чем глубже вы будете понимать язык IL, тем более сложные правила сможете разработать.

Сначала нужно установить FxCop SDK, и это сложнее, чем должно быть. Если в вашем распоряжении есть Visual Studio Professional или более совершенная версия, значит, в среде IDE уже имеется переработанное средство Code Analysis (под ним по-прежнему скрывается средство FxCop). На моей машине соответствующие файлы размещены в каталоге `C:\Program Files (x86)\Microsoft Visual Studio 14.0\Team Tools\Static Analysis Tools\FxCop`.

Если получить доступ к нужной версии Visual Studio невозможно, есть и другие варианты. Самый простой способ заключается в загрузке этого средства CodePlex по адресу https://fxcopinstaller.codeplex.com. Если в то время, когда вы читаете книгу, данных строк проекта там уже не будет, попробуйте воспользоваться Windows 7.1 SDK, в котором, по-видимому, неисправен веб-установщик, но можно получить ISO-образ, расположенный по адресу: https://www.microsoft.com/download/details.aspx?id=8442. Загрузите его и извлеките установщик из архива `\Setup\WinSDKNetFxTools\cab1.cab`. В этом архиве есть файл, имя которого начинается с `WinSDK_FxCopSetup.exe`. Извлеките его, переименуйте в `FxCopSetup.exe`, и все будет в порядке.

Проекты, относящиеся к FxCop, можно найти в исходном коде, сопровождающем эту книгу. Их разместили в собственном файле решения, чтобы не ломать сборку для остальных учебных проектов. В FxCopRules содержатся правила, которые будут загружены обработчиком FxCop и запущены в отношении некоторых целевых сборок. В FxCopViolator содержится класс, имеющий ряд нарушений, наличие которых будут проверять правила. Разбирайте эти проекты шаг за шагом по мере того, как я буду объяснять назначение определенных компонентов.

Прежде чем приступить к созданию правил, вам может понадобиться отредактировать файл `FxCopRules.csproj`, чтобы в нем были указаны правильные пути SDK. Действующие значения должны быть похожи на такие:

```
<PropertyGroup>
    <FxCopSdkDir>C:\...\Microsoft Fxcop 10.0</FxCopSdkDir>
</PropertyGroup>
<ItemGroup>
    <Reference Include="$(FxCopSdkDir)\FxCopSdk.dll" />
    <Reference Include="$(FxCopSdkDir)\Microsoft.CCi.dll" />
</ItemGroup>
```

Обновите значение `FxCopSdkDir`, чтобы оно указывало на каталог установки FxCop (по умолчанию он должен находиться в каталоге `Program Files (x86)`) или на то место, куда вы поместили соответствующие DLL-библиотеки.

Затем нужно создать файл `Rules.xml`, содержащий метаданные для каждого правила. Первое правило будет иметь следующий вид:

```
<?xml version="1.0" encoding="utf-8" ?>
<Rules FriendlyName="Custom Rules">
    <Rule TypeName="DisallowStaticFieldsRule"
        Category="Custom.Arbitrary"
        CheckId="HP100">
    <Name>Static fields are not allowed</Name>
    <Description>Static fields are not allowed because...
    </Description>
    <Url>http://internaldocumentationsite/FxCop/HP100</Url>
    <Resolution>Make the static field '{0}' either
      readonly or const.
    </Resolution>
    <MessageLevel Certainty="90">Error</MessageLevel>
    <FixCategories>Breaking</FixCategories>
    <Email>feedback@high -perf.net</Email>
    <Owner>Ben Watson</Owner>
    </Rule>
</Rules>
```

Следует заметить, что атрибут `TypeName` должен соответствовать имени определяемого ниже класса с правилом. Этот XML-файл должен быть включен в проект при установленном для параметра Build Action (Действия при сборке) значении Embedded Resource (Внедренный ресурс).

Каждое определяемое нами правило должно быть производным класса, предоставляемого FxCop SDK, и содержать общую информацию, такую как местонахождение XML-манифеста правил. Можно для удобства создать базовый класс для всех ваших правил, обеспечивающих общие функциональные возможности.

```
using Microsoft.FxCop.Sdk;
using System.Reflection;

namespace FxCopRules
{
    public abstract class BaseCustomRule : BaseIntrospectionRule
```

```
    {
        // Имя манифеста – это пространство имен, используемое по умолчанию,
        // плюс имя XML-файла правил, указанное без расширения.
        private const string ManifestName = "FxCopRules.Rules";

        // Сборка, в которую внедрен манифест правил
        // (в данном случае это текущая сборка).
        private static readonly Assembly ResourceAssembly =
                            typeof(BaseCustomRule).Assembly;
        protected BaseCustomRule(string ruleName)
            :base(ruleName , ManifestName , ResourceAssembly)
        {
        }
    }
}
```

После этого нужно определить класс, являющийся производным класса `BaseCustomRule` и предназначенный для конкретного нарушения, наличие которого нужно проверить. В первом примере будут запрещены все статические поля, но разрешены поля `const` и `readonly`:

```
public class DisallowStaticFieldsRule : BaseCustomRule
{
    public DisallowStaticFieldsRule()
        : base(typeof(DisallowStaticFieldsRule).Name)
    {
    }

    public override ProblemCollection Check(Member member)
    {
        var field = member as Field;
        if (field != null)
        {
          // Поиск всех статических данных, не являющихся const или readonly
            if (field.IsStatic && !field.IsInitOnly && !field.IsLiteral)
            {
              // field.FullName является дополнительным аргументом,
              // который будет использоваться для форматирования
              // параметра {0} строки Resolution.
                var resolution = this.GetResolution(field.FullName);
                var problem = new Problem(resolution ,
                                          field.SourceContext);
                this.Problems.Add(problem);
            }
        }
        return this.Problems;
    }
}
```

Класс `BaseIntrospectionRule` предоставляет ряд перегрузок виртуального метода `Check` с различными типами аргументов, которые можно переопределить, чтобы обеспечить нужное функционирование. В исходном состоянии эти методы ничего

не делают. Автодополнение IntelliSense — хорошее подспорье при написании правил FxCop, и оно показывает наличие следующих методов `Check`:

- `Check(ModuleNode moduleNode);`
- `Check(Parameter parameter);`
- `Check(Resource resource);`
- `Check(TypeNode typeNode);`
- `Check(string namespaceName, TypeNodeCollection types).`

Из любого метода можно исследовать отдельные строки кода IL. Вот как выглядит правило, запрещающее изменение регистра строк:

```
public class DisallowStringCaseConversionRule : BaseCustomRule
{
    public DisallowStringCaseConversionRule()
        : base(typeof(DisallowStringCaseConversionRule).Name)
    { }

    public override ProblemCollection Check(Member member)
    {
        var method = member as Method;
        if (method != null)
        {
            foreach (var instruction in method.Instructions)
            {
                if (instruction.OpCode == OpCode.Call
                    || instruction.OpCode == OpCode.Calli
                    || instruction.OpCode == OpCode.Callvirt)
                {
                    var targetMethod = instruction.Value as Method;
                    if (targetMethod != null
                        && (targetMethod.FullName == "System.String.ToUpper"
                        || targetMethod.FullName == "System.String.ToLower"))
                    {
                        var resolution = this.GetResolution(method.FullName);
                        var problem = new Problem(resolution ,
                        method.SourceContext);
                        this.Problems.Add(problem);
                    }
                }
            }
        }

        return this.Problems;
    }
}
```

В качестве заключительного примера рассмотрим другой способ, позволяющий сообщить средству FxCop о необходимости обхода кода. Вдобавок к ранее рассмотренным методам `Check` можно переопределить десятки методов `Visit*`.

Они вызываются рекурсивно, проходя через каждый узел в графе программы, начиная с выбранного узла. Переопределяются только нужные вам методы `Visit`. Рассмотрим пример, использующий данный прием для добавления правила, относящегося к созданию объекта `Thread`:

```
public class DisallowThreadCreationRule : BaseCustomRule
{
    public DisallowThreadCreationRule()
        : base(typeof(DisallowThreadCreationRule).Name) { }

    public override ProblemCollection Check(Member member)
    {
        var method = member as Method;
        if (method != null)
        {
            VisitStatements(method.Body.Statements);
        }

        return base.Check(member);
    }

    public override void VisitConstruct(Construct construct)
    {
        if (construct != null)
        {
            var binding = construct.Constructor as MemberBinding;
            if (binding != null)
            {
                var instanceInitializer =
                binding.BoundMember as InstanceInitializer;
                if (instanceInitializer.DeclaringType.FullName
                    == "System.Threading.Thread")
                {
                    var problem = new Problem(this.GetResolution(),
                    construct.SourceContext);
                    this.Problems.Add(problem);
                }
            }
        }

        base.VisitConstruct(construct);
    }
}
```

Чтобы воспользоваться этими правилами в Visual Studio, создайте сборку библиотеки `FxCopRules.dll` и скопируйте ее в папку `Rules`, находящуюся в установочной папке FxCop (путь к моей папке выглядит так: `C:\Program Files (x86)\Microsoft Visual Studio 14.0\Team Tools\Static Analysis Tools\FxCop\Rules`). Перейдите в Visual Studio к свойствам другого проекта (тестирование можно

провести в учебном проекте FxCopViolator) и просмотрите вкладку Code Analysis (Анализ кода) (рис. 9.1). В меню Rule Set (Набор правил) можно выбрать набор пользовательских правил или создать собственный набор, включающий нужные вам правила.

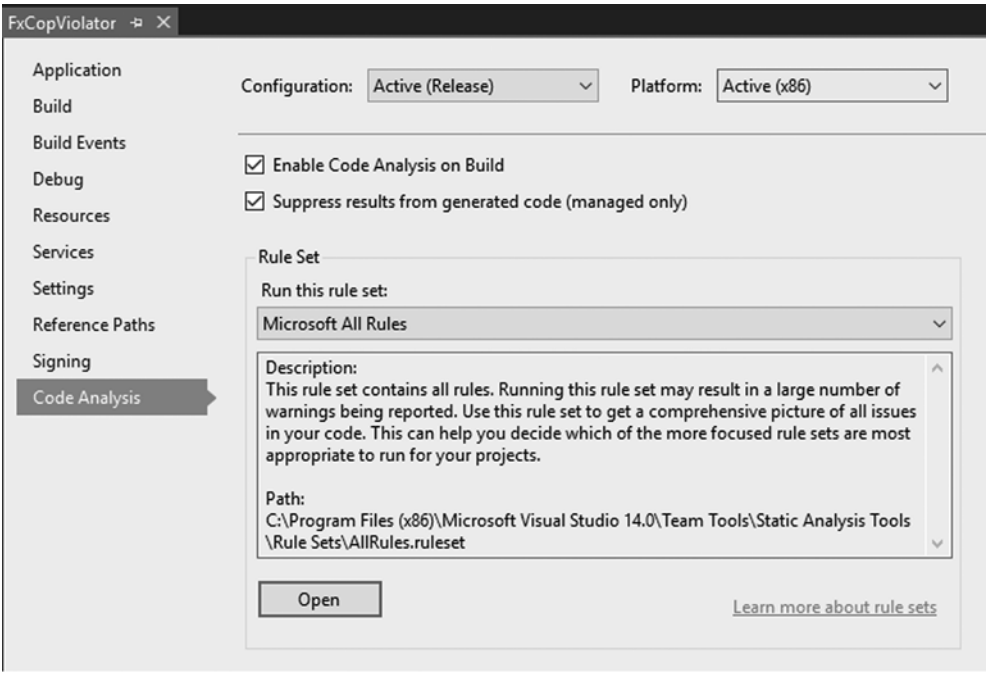

Рис. 9.1. Настроить правила, требующие применения в Visual Studio, можно с помощью свойств проекта на вкладке Code Analysis (Анализ кода)

Теперь при сборке проекта с соответствующими правилами будут появляться сообщения, показывающие, какие из определенных вами правил нарушены. Как и при работе со стандартными правилами для сборки или анализа кода, вы можете, дважды щелкнув кнопкой мыши на этих сообщениях, перейти к источнику нарушения (рис. 9.2).

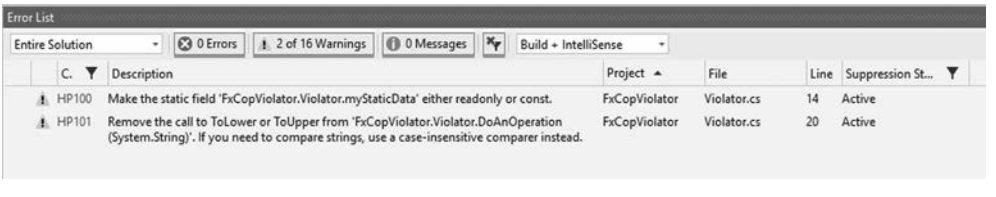

Рис. 9.2. Заданные пользователем предупреждения FxCop будут показаны на панели Error List (Список ошибок), как и другие предупреждения о проблемах при сборке и анализе кода

В учебный проект включен также пример запуска FxCop с пользовательскими правилами из командной строки:

```
>"C:\Program Files (x86)\Microsoft Fxcop 10.0\FxCopCmd.exe"
    /out:.\FxCopOutput.xml /rule:FxCopRules.dll
    /file:FxCopViolator.dll

Microsoft (R) FxCop Command-Line Tool, Version 14.0 (14.0.25420.1)
Copyright (C) Microsoft Corporation, All Rights Reserved.

Loaded fxcoprules.dll...
Loaded FxCopViolator.dll...
Initializing Introspection engine...
Analyzing...
Analysis Complete.
Writing 3 messages...
Writing report to .\FxCopOutput.xml...
Done:00:00:00.8200342
```

Достаточно изучить работу с FxCop один раз, чтобы в дальнейшем все давалось без особого труда. Самым большим препятствием при создании собственных правил станет скудость официальной документации. Чтобы получить дополнительные сведения о правилах FxCop, определяемых пользователем, обратитесь к замечательному обзору, составленному Джейсоном Крейсовати (Jason Kresowaty) и находящемуся по адресу http://www.binarycoder.net/fxcop/.

.NET Compiler Code Analyzers

В отличие от FxCop, при работе с которым вы столкнетесь с ограниченным пониманием кода, .NET Compiler Code Analyzers способен не только взамен анализа кода IL анализировать код высокого уровня, но и вести его непосредственно из IDE-среды Visual Studio и даже выдавать предложения и править код. Этот тип анализатора — более эффективная альтернатива FxCop. В этом подразделе будет продемонстрирован процесс создания двух правил, одно из которых потребует от статических полей `static` иметь пометку «только для чтения» (`readonly`), а второе станет выдавать предупреждения, препятствующие вызову методов `String.ToLower` и `String.ToUpper`.

В Visual Studio 2015 нужно из установщика поставить на компьютер компонент Visual Studio Extensibility Tools (инструменты расширяемости Visual Studio). В Visual Studio 2017 у этого компонента немного другое название — Visual Studio extension development (разработка расширений Visual Studio).

В обеих версиях следует установить инструментарий .NET Compiler Platform SDK, что можно сделать непосредственно из Visual Studio, пройдя по пунктам меню File ▸ New Project ▸ Visual C# ▸ Extensibility (Файл ▸ Новый проект ▸ Visual C# ▸ Расширяемость) и выбрав в списке типов проектов вариант Download the .NET Compiler Platform SDK. После завершения установки нужно перезапустить Visual Studio. Затем для создания собственного анализатора можно выбрать в окне New Project (Новый проект) вариант Analyzer with Code Fix (NuGet + VSIX).

Разработанный здесь пример доступен в учебном решении CodeAnalyzers, имеющемся в исходном коде книги. Там есть три проекта:

- **SampleCodeAnalyzer.csproj** — содержит код для самого анализа и внесения исправлений;
- **SampleCodeAnalyzer.Test.csproj** — модульные тесты для того, чтобы поупражняться с вашим кодом без необходимости отладки в Visual Studio;
- **SampleCodeAnalyzer.Vsix** — проект надстройки Visual Studio, позволяющий анализатору размещаться в самой среде Visual Studio.

Чтобы протестировать анализатор, нужно убедиться, что в качестве проекта по умолчанию выбран Vsix, и нажать клавишу F5. В результате будет запущен еще один экземпляр Visual Studio с загруженным анализатором. Чтобы протестировать анализ кода, здесь можно создать новый проект.

Первый анализатор кода будет обнаруживать любое статическое поле `static` и рекомендовать для него установку метки `readonly`. Кроме того, он будет содержать средство исправления для самостоятельного выполнения этого действия.

Как это делается, можно понять из содержимого файла `StaticFieldAnalyzer.cs`:

```
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;

namespace SampleCodeAnalyzer
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class StaticFieldAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId = "StaticFieldAnalyzer";

        private static readonly LocalizableString Title =
            new LocalizableResourceString(
                    nameof(Resources.AnalyzerTitle),
                    Resources.ResourceManager,
                    typeof(Resources));

        private static readonly LocalizableString MessageFormat =
            new LocalizableResourceString(
                    nameof(Resources.AnalyzerMessageFormat),
                    Resources.ResourceManager,
                    typeof(Resources));

        private static readonly LocalizableString Description =
            new LocalizableResourceString(
                    nameof(Resources.AnalyzerDescription),
                    Resources.ResourceManager,
                    typeof(Resources));

        private const string Category = "Thread Safety";

        private static DiagnosticDescriptor Rule =
```

```
            new DiagnosticDescriptor(DiagnosticId ,
                                     Title ,
                                     MessageFormat ,
                                     Category ,
                                     DiagnosticSeverity.Info,
                                     isEnabledByDefault: true,
                                     description: Description);

        public override ImmutableArray <DiagnosticDescriptor >
            SupportedDiagnostics
        {
            get
            {
                return ImmutableArray.Create(Rule);
            }
        }

        public override void Initialize(AnalysisContext context)
        {
            context.RegisterSymbolAction(AnalyzeFieldSymbol ,
                                         SymbolKind.Field);
        }

        private void AnalyzeFieldSymbol(
            SymbolAnalysisContext context)
        {
            IFieldSymbol field = (IFieldSymbol)context.Symbol;
            if (field.IsStatic && !field.IsReadOnly)
            {
                var diagnostic = Diagnostic.Create(
                                     Rule,
                                     field.Locations[0],
                                     field.Name);
                context.ReportDiagnostic(diagnostic);
            }
        }
    }
}
```

Поля в верхней части класса являются стандартными шаблонными метаданными, которые вы должны настроить для каждого правила. Шаблон проекта по умолчанию помещает эти строки в файл `Resources.resx` (рис. 9.3), чтобы было проще их локализовать, но делать это не обязательно.

Метод `Initialize` сообщает среде Visual Studio, что требуется проанализировать. В данном случае анализируются только поля, но позже будет показан и другой вариант. Метод `AnalyzeFieldSymbol` — именно то место, где протекает действие. Он вызывается для каждого найденного символа требуемого типа. Код проверяет, является ли поле статическим (`static`), но без пометки `readonly`, и если это так, выдает новое диагностическое сообщение, появляющееся в пользовательском интерфейсе в виде зеленой волнистой линии, подчеркивающей проблемный символ (рис. 9.4). Зеленый цвет обусловливается тем, что правило было помечено нами как `DiagnosticSeverity.Info`.

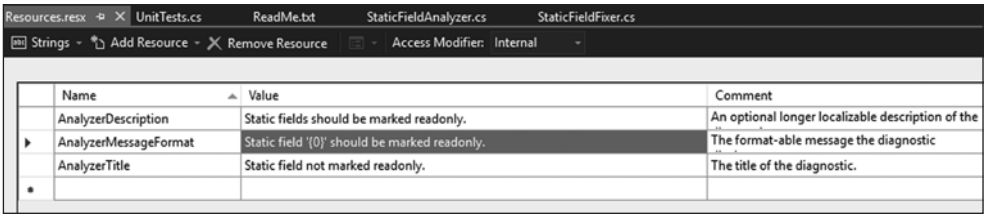

Рис. 9.3. В файле Resources.resx содержатся локализуемые строки для вашего анализатора кода

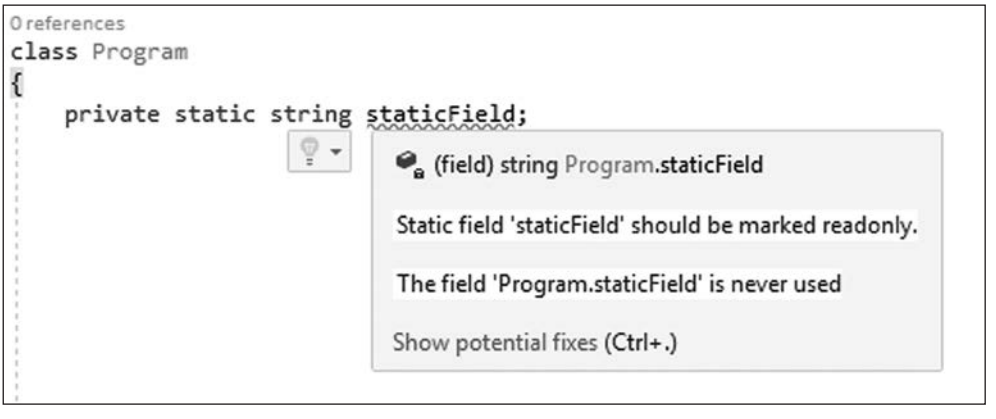

Рис. 9.4. Правило анализа кода приводит к появлению волнистой линии под рассматриваемым нами синтаксисом

В некоторых случаях возможно автоматическое исправление кода в соответствии с вашими рекомендациями. Делать это не обязательно, но все же хорошо, если эта возможность существует. Файл `StaticFieldFixer.cs` содержит код для автоматической реализации исправления — добавления пометки `readonly`:

```
using System.Collections.Immutable;
using System.Composition;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SampleCodeAnalyzer
{
    [ExportCodeFixProvider(LanguageNames.CSharp,
                           Name = nameof(StaticFieldFixer)),
                           Shared]
    public class StaticFieldFixer : CodeFixProvider
```

```
{
    private const string title = "Make readonly";

    public sealed override ImmutableArray <string >
        FixableDiagnosticIds
    {
        get
        {
            return ImmutableArray.Create(
            StaticFieldAnalyzer.DiagnosticId);
        }
    }

    public sealed override FixAllProvider GetFixAllProvider()
    {
        return WellKnownFixAllProviders.BatchFixer;
    }

    public sealed override async Task RegisterCodeFixesAsync(
        CodeFixContext context)
    {
        var root = await context.Document.GetSyntaxRootAsync(
            context.CancellationToken).ConfigureAwait(false);

        var diagnostic = context.Diagnostics.First();
        var diagnosticSpan = diagnostic.Location.SourceSpan;

        // Нахождение идентифицированной при диагностике
        // декларации в типе
        var declaration = root.FindToken(diagnosticSpan.Start)
                              .Parent.AncestorsAndSelf()
                              .OfType <FieldDeclarationSyntax >()
                              .First();

        // Регистрация действия над кодом, которое будет вызывать исправление
        context.RegisterCodeFix(
            CodeAction.Create(
                title: title ,
                createChangedDocument:
                    c => MakeReadOnlyAsync(context.Document ,
                                           declaration ,
                                           c),
                equivalenceKey: title),
            diagnostic);
    }

    private async Task<Document > MakeReadOnlyAsync(
    Document document ,
        FieldDeclarationSyntax fieldDecl ,
        CancellationToken cancellationToken)
    {
```

```
            // Нахождение поля и обновление его модификаторов
            var newFieldDecl = fieldDecl.AddModifiers(
                   SyntaxFactory.Token(
                     SyntaxKind.ReadOnlyKeyword));

            var root = await document.GetSyntaxRootAsync();

            // Замена старого узла новым
            var newRoot = root.ReplaceNode(fieldDecl ,
                                           newFieldDecl);

            var newDocument = document.WithSyntaxRoot(newRoot);

            // Возвращение нового документа, теперь с полем readonly static
            return newDocument;
        }
    }
}
```

Свойство `FixableDiagnosticIds` связывает анализатор с этим средством исправления, и теперь среда Visual Studio знает, какое действие может предпринять. Метод `RegisterCodeFixesAsync` находит места, где сработала диагностика, и регистрирует делегат, вызываемый для исправления кода. Метод `MakeReadOnlyAsync` проделывает реальную работу. Он возвращает объект `Document`, представляющий новый кодовый документ после сгенерированных этим методом исправлений. В данном случае он берет объявление поля и добавляет `readonly` к списку модификаторов. Класс `SyntaxFactory` содержит множество вариантов создания новых фрагментов кода.

Этот код работает, изменяя отдельные узлы в дереве синтаксиса документа. Дерево синтаксиса неизменяемое, поэтому внесение любых исправлений приводит к созданию новой версии возвращаемого вам объекта. Метод `MakeReadOnlyAsync` последовательно извлекает новые версии поля, узла синтаксиса и документа (рис. 9.5).

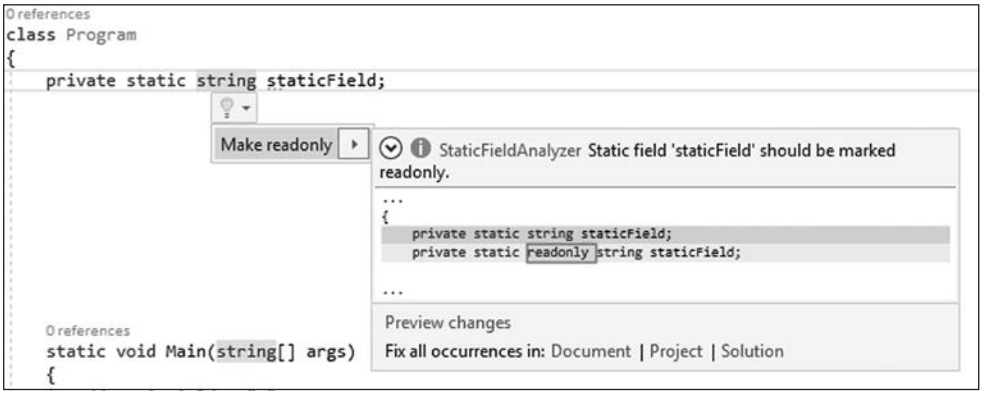

Рис. 9.5. Щелчок на значке подсказок Code Tips приводит к отображению варианта автоматического исправления с предварительным просмотром того, что будет изменено

Рассмотрим еще один простой пример анализатора, рекомендующего не вызывать методы `String.ToLower` и `String.ToUpper`:

```
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SampleCodeAnalyzer
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class StringToUpperToLowerAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId =
            "StringToUpperToLowerAnalyzer";

        private static readonly LocalizableString Title =
            new LocalizableResourceString(
                nameof(Resources.ToUpperToLowerAnalyzerTitle),
                Resources.ResourceManager ,
                typeof(Resources));

        private static readonly LocalizableString MessageFormat =
            new LocalizableResourceString(
                nameof(Resources.ToUpperToLowerAnalyzerMessageFormat),
                Resources.ResourceManager ,
                typeof(Resources));

        private static readonly LocalizableString Description =
            new LocalizableResourceString(
                nameof(Resources.ToUpperToLowerAnalyzerDescription),
                Resources.ResourceManager ,
                typeof(Resources));

        private const string Category = "Performance";

        private static DiagnosticDescriptor Rule =
            new DiagnosticDescriptor(DiagnosticId ,
                                     Title ,
                                     MessageFormat ,
                                     Category ,
                                     DiagnosticSeverity.Warning ,
                                     isEnabledByDefault: true,
                                     description: Description);

        public override ImmutableArray <DiagnosticDescriptor >
            SupportedDiagnostics
        {
            get
            {
```

```
            return ImmutableArray.Create(Rule);
        }
    }

    public override void Initialize(AnalysisContext context)
    {
        context.RegisterSyntaxNodeAction(
            AnalyzeNode ,
            SyntaxKind.InvocationExpression);
    }

    private void AnalyzeNode(
        SyntaxNodeAnalysisContext context)
    {
        var invocationExpression =
             (InvocationExpressionSyntax)context.Node;
        var memberAccessExpression =
            invocationExpression.Expression
                as MemberAccessExpressionSyntax;
        var memberName =
            memberAccessExpression?.Name.ToString();
        if (memberName == "ToUpper"
            || memberName == "ToLower")
        {
            var diagnostic = Diagnostic.Create(
                Rule,
                memberAccessExpression.GetLocation());

            context.ReportDiagnostic(diagnostic);
        }
    }
  }
}
```

Еще одно средство, помогающее разрабатывать анализаторы, — инструмент визуализации синтаксиса Syntax Visualizer. Его можно установить в Visual Studio, воспользовавшись пунктами меню Tools ▸ Extensions and Updates (Инструменты ▸ Расширения и обновления). Найдите там .NET Compiler Platform SDK, включающий Syntax Visualizer. После установки откройте это средство, пройдя по пунктам меню View ▸ Other Windows ▸ Syntax Visualizer (Вид ▸ Другие окна ▸ Синтаксический визуализатор). Когда все будет сделано, можно щелкнуть на любом месте в коде файла и увидеть обновленное дерево синтаксиса (рис. 9.6).

В анализаторах кода можно делать практически все что угодно. Они исключительно гибки и обеспечивают вам почти полную свободу для анализа собственного кода (рис. 9.7). Приведу несколько примеров:

- разбор, компиляция и предварительное выполнение кода по мере его набора с анализом результатов на основании произвольных правил;
- анализ строковых литералов на семантическую корректность (например, на недопустимое наличие учетных данных или правильность регулярных выражений);

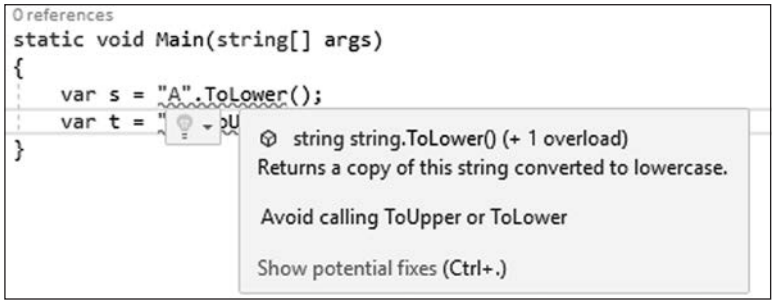

Рис. 9.6. С помощью анализаторов кода можно внедрять разумные приемы программирования с предоставлением информации об этом непосредственно в среде Visual Studio

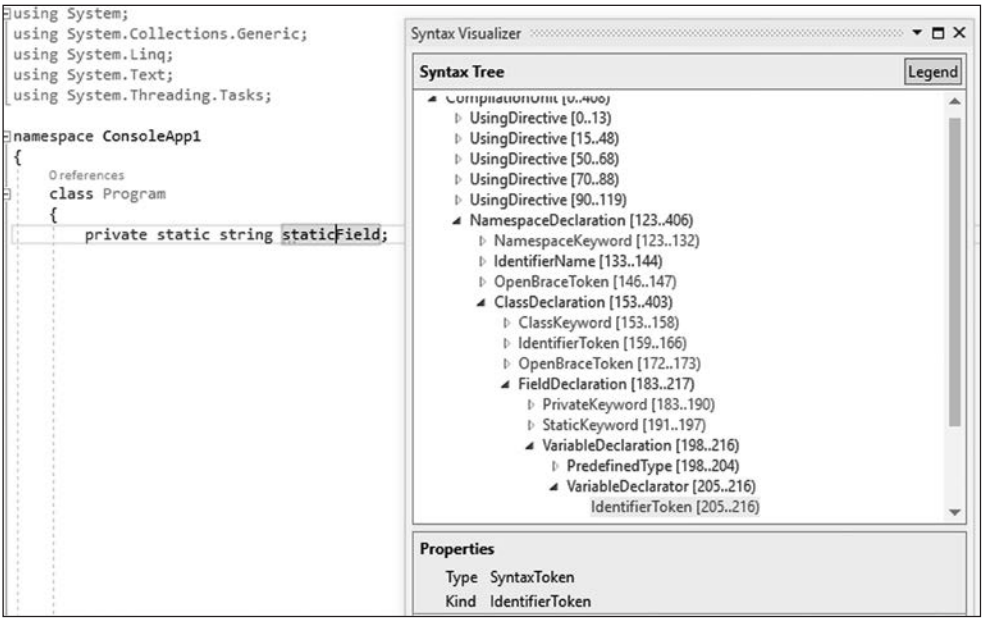

Рис. 9.7. Средство визуализации синтаксиса может помочь разобраться в структуре кода при разработке правил его анализа

- принуждение к соблюдению правил программирования, принятых в вашей команде или компании;
- принуждение к соблюдению стандартов производительности, характерных для вашего программного продукта.

По мере приобретения опыта и получения углубленного представления о вашем собственном коде и способах представления его синтаксиса компилятором можно будет повторно применять анализаторы для проведения более сложного и глубокого анализа.

К счастью, вам не придется все создавать самостоятельно. Есть множество анализаторов кода, пригодных для повторного использования, включая следующие:

- Roslyn Analyzers — поставляются с компилятором. Исследование их исходного кода позволит вам изучить множество особенностей написания анализаторов;
- Clr Heap Allocation Analyzer — подсвечивает каждый источник выделения памяти в куче, включая неявные выделения;
- StyleCopAnalyzers — выдают рекомендации по стилю программирования.

Существует также множество других анализаторов.
Все их можно найти с помощью поисковой системы, которой вы пользуетесь.

Выполняйте централизацию и абстрагирование сложного и важного для повышения производительности кода

Чтобы упростить поддержку, нужно как можно больше кода, важного для обеспечения высокой производительности, хранить в одном месте, преимущественно за API, используемыми всеми остальными частями вашего приложения. Например, если приложение загружает файлы через HTTP, этот код можно заключить в API, который раскрывает только те части загружаемого, о которых должны знать остальные части вашей программы (например, запрошенный URL-адрес и загруженное содержимое). API управляет сложностью HTTP-вызова, а все приложение проходит через этот API, как только возникает потребность в совершении HTTP-вызова. Если обнаружится проблема с производительностью, связанная с загрузкой, либо потребуется обеспечить очередность загрузки или внести другое изменение, то сделать это «под прикрытием» API будет несложно. Следует помнить, что такие API должны поддерживать асинхронный характер операции.

Изолируйте неуправляемый и небезопасный код

Есть много причин, по которым нужно избегать использования неуправляемого кода. Как говорилось во введении, преимущества неуправляемого кода зачастую преувеличены, зато опасность повреждения памяти вполне реальна.

Короче говоря, если приходится иметь дело с неуправляемым кодом (скажем, для взаимодействия с устаревшей системой, когда затраты на полный переход всего интерфейса на управляемый код слишком велики), его нужно изолировать. Сделать это можно множеством способов, при этом следует стремиться избавиться даже от случайных системных вызовов откуда бы то ни было в область неуправляемого кода. Это станет предпосылкой хаоса.

В идеале нужно выделить неуправляемому коду собственный процесс, чтобы обеспечить строгую изоляцию на уровне операционной системы (рис. 9.8). Если это невозможно и требуется, чтобы неуправляемый код загружался в тот же самый

процесс, постарайтесь свести его к минимально возможному количеству DLL-библиотек и пропускать все вызовы к нему через централизованный API, способный обеспечить соблюдение стандартных мер безопасности.

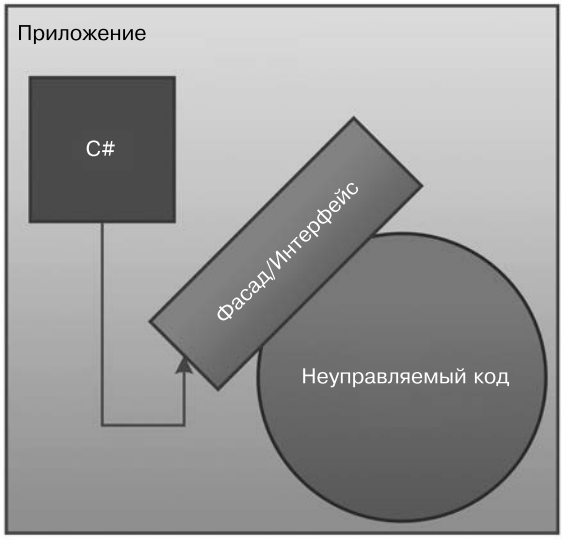

Рис. 9.8. Изолируйте небезопасный код в известных областях вашего приложения. Уделяйте им пристальное внимание

Неуправляемый код вносит в ваш процесс существенный риск. Любые ошибки, повреждающие память на неуправляемой стороне процесса, могут разрушить в последнем все что угодно, включая память на управляемой стороне. При этом утрачиваются гарантии безопасности, предоставляемые средой CLR.

Считайте управляемый код, помеченный как небезопасный, точно таким же, как неуправляемый код, и изолируйте его в наименьшей области видимости. Нужно также разрешить использовать небезопасный код в настройках проекта.

Отдавайте приоритет ясности кода, а не получению высокой производительности, пока нет веских причин для обратного

Возможность легко читать и поддерживать код важнее его производительности, пока не возникнут веские причины изменить приоритеты на противоположные. Осознав необходимость внести глубокие изменения, чтобы повысить производительность, выполняйте эту работу так, чтобы она была как можно более прозрачной для вышестоящего кода.

Если же в угоду производительности код перестал быть ясно читаемым, обеспечьте в нем документирование всех своих действий, чтобы вашим последователям не пришлось «прояснять» для себя вашу утонченную оптимизацию путем ее упрощения.

Резюме

Чтобы гарантировать безопасность своего кода, нужно иметь представление о подробностях его реализации на всех уровнях. Для ограничения внешнего воздействия изолируйте в особых модулях самый рискованный код, особенно машинный или незащищенный. Откажитесь от проблемных API и шаблонов программирования и принуждайте к соблюдению разумных стандартов программирования, поощряя безопасные приемы. Для этого используйте анализаторы кода или другие инструменты статического анализа. Не приносите ясность кода или возможность его поддержки в жертву повышению производительности без веских на то оснований.

10 Формирование команды, нацеленной на достижение высокой производительности

Наиболее интересные программные продукты создают не одиночки. Скорее всего, вы входите в команду, стремящуюся создать какой-нибудь полезный и высокопроизводительный программный продукт. Если вы числитесь в команде знатоком способов повышения производительности и даже если это не так, в ваших силах предпринять ряд действий, чтобы нацелить всех остальных на достижение высоких результатов в этой сфере.

Основная часть рекомендаций, изложенных в этой книге, предназначена для организации, считающей, что разработка программных продуктов — работа истинного специалиста. К сожалению, многие люди вынуждены трудиться далеко не в идеальных условиях. Если это относится и к вам, не стоит отчаиваться. Возможно, ряд советов, которые я дам в этой главе, помогут повысить уровень оценки инженеров и компетентность в вашей компании.

Выявление областей, требующих особо высокой производительности

Оптимизировать все невозможно по определению. Это возвращает нас к рассмотренным в главе 1 принципам, относящимся к измерениям и поиску областей, требующих особо высокой производительности. В командной работе нужно прийти к согласию относительно того, какие области считать определяющими, а какие можно не трогать.

Разработчики должны гордиться своим трудом и выполнять работу максимально хорошо, а в программировании не должно быть областей, в которых можно пустить все на самотек. Но реалии делового мира диктуют ограничения по времени и человеческим ресурсам, с учетом которых приходится выполнять работу. Исходя из них, нужно потратить время на выявление критических областей системы — не забываем: измерение, измерение и еще раз измерение! — и убедиться в том, что в этих областях к деталям отнеслись с должным вниманием.

Производительность — не единственный показатель оценки кода. На принятие решений должны влиять также возможность поддержки, масштабирования,

обеспечения безопасности, настройки конфигурации и другие важные факторы. Но подозреваю, что на измерение производительности и ее повышение вы всегда будете тратить основную часть своего времени.

Эффективное тестирование

Это не книга по тестированию, но без всяких слов понятно, что эффективные тесты на всех уровнях существенно повысят вашу уверенность при внесении в код значительных изменений. Если у вас есть модульные тесты с высокой степенью охвата кода, то резкое изменение основного алгоритма или структуры данных для существенного повышения эффективности не должно вас пугать.

Если вы считаете, что производительность для вас важна, то связанные с ней показатели нужно отслеживать, применяя инструментальные средства и приемы, рассмотренные в книге. Наряду с функциональными тестами можете проводить тесты производительности. Они могут быть как простыми, отслеживающими количество операций, выполняемых компонентом за секунду, так и сложными, оценивающими производительность по тысячам показателей между группой серверов предварительного выпуска и группой производственных серверов.

Провалы теста производительности следует рассматривать не менее серьезно, чем провал функционального теста, и они должны становиться блокировщиками поставок. Скорее всего, окажется, что создать надежные повторяемые тесты производительности намного сложнее, чем функциональные тесты. Производительность слишком сильно переплетается с состоянием машины, другим программным продуктом, историей запущенных процессов и бесчисленным множеством других переменных. Есть три основных подхода работы с этим шумом, который непременно проявит себя.

1. **Устранение шума.** «Чистые» машины перезапустите перед тестированием, проконтролируйте все запущенные процессы, различия в составе оборудования и многое другое. Этот подход применяется при сравнении производительности двух машин. Важно также выполнить калибровку путем запуска базового теста на контрольной и целевой машинах, чтобы убедиться, что при абсолютно одинаковой конфигурации оборудования и операционной системы запуск на них одного и того же теста дает одинаковые показатели. В машины с одинаковой спецификацией довольно часто закрадывается разница, влияющая на показатели тестов, что будет сбивать вас с толку.
2. **Запуск широкомасштабных тестов.** Если устранять весь шум нецелесообразно, игнорируйте его и запускайте тесты в масштабе, достаточном для уменьшения влияния самых значительных источников шума. Это может обойтись весьма недешево, особенно при крупных инфраструктурах. Чтобы получить статистически значимый результат, могут понадобиться десятки, сотни или даже тысячи машин. Если расширить масштабы использования оборудования нельзя, можно раздвинуть временные рамки и сотни раз перезапускать тесты, но при этом не учитывается такое же большое количество переменных.

3. **Тестирование производительности в производственном режиме.** Этот вариант во многих отношениях предпочтителен. Существует множество разновидностей проблем производительности, ничем себя не проявляющих вне производственной среды или при тестировании в менее крупных масштабах. Отслеживать производительность в производственном режиме следует обязательно, тогда почему бы не расширить эту инфраструктуру, включив в нее A/B-тестирование, анализ выборок запросов, непрерывное профилирование, мониторинг процесса поставки и иные средства, позволяющие увидеть изменение производительности с течением времени? Любое тестирование в производственном режиме требует уверенной в себе команды и организации, опыта работы и достаточных навыков, а также гораздо более развитой инфраструктуры обеспечения безопасности, откатов, оповещений и т. д.

В любом случае нужно заняться A/B-тестированием, то есть сравнением производительности двух сборок, в качестве идеального подконтрольного вам сценария.

Инфраструктура и автоматизация для оценки производительности

Для сбора данных о производительности вам, скорее всего, потребуется создать некую пользовательскую инфраструктуру, инструментарий и автоматизированную поддержку (все инструментальные средства для чтения показателей уже рассмотрены в данной книге). К счастью, почти все полезные средства оценки производительности в той или иной степени поддерживают скриптование.

Существует множество способов отслеживания производительности, и вам придется решить, какой из них окажется наилучшим для вашего программного продукта. Вот несколько мыслей на этот счет.

- **PerfMon.** Если все данные представлены в виде счетчиков производительности и работа выполняется на одной машине, то этого средства будет вполне достаточно.
- **Объединение показателей счетчиков производительности.** Если работа ведется на нескольких машинах, то вам, вероятно, нужно будет объединить показатели счетчиков в централизованной базе данных. Преимуществом этого способа является то, что можно сохранить данные о производительности, чтобы анализировать полученные ранее.
- **Эталонное тестирование.** Ваше приложение обрабатывает стандартный набор данных, получившиеся показатели производительности сравниваются с прежними результатами. Эталонные тесты полезны, но нужно проявлять осторожность: при смене сценариев старые данные могут оказаться непригодными. Чтобы сравнение было корректным, эталонные результаты тестирования при смене производственных данных нужно обновлять.

- **Автоматизированное профилирование.** Произвольное профилирование центрального процессора и выделения памяти в ходе работы либо с тестовыми, либо с реальными данными.
- **Оповещения, выдаваемые на основе данных о производительности.** Например, отправка автоматического оповещения в службу поддержки, если центральный процессор испытывает длительную перегрузку или увеличилось количество задач в очереди.
- **Автоматизированный анализ ETW-событий.** С его помощью можно заметить некоторые нюансы, которые способны пропустить счетчики производительности.

То, что вы делаете сейчас для создания инфраструктуры производительности, воздастся сторицей в будущем, как только поддержка высокой производительности станет высокоавтоматизированной. Создание такой инфраструктуры зачастую важнее устранения любых самопроизвольно возникающих проблем с производительностью, поскольку качественная инфраструктура способна найти и высветить эти проблемы намного раньше, чем любые предпринятые людьми меры. Качественная инфраструктура убережет вас от сюрпризов падения производительности в самые неподходящие моменты. Она также послужит в качестве великолепного сервиса регрессионного тестирования, что гарантирует поддержание приемлемого уровня производительности при дальнейшей разработке.

Наиболее важным при формировании инфраструктуры станет вопрос о том, насколько много вмешательства человека ей требуется. Если вы в той же ситуации, что и большинство разработчиков программного обеспечения, то у вас работы намного больше, чем времени на ее выполнение. Если полагаться на анализ производительности, выполняемый вручную, это будет означать, что его просто никто не будет делать. Таким образом, автоматизация становится ключом к эффективной стратегии повышения производительности. Первоначальные вложения будут изо дня в день экономить бессчетное количество часов. Хорошее средство экономии времени может быть не сложнее сценариев, запускающих вспомогательные программы и создающие для вас отчеты, как только они потребуются, но, скорее всего, его масштаб придется подгонять под размер вашего приложения. Более крупному приложению, запущенному в дата-центре, потребуются иные типы анализа производительности, чем приложению, выполняемому на настольном компьютере, и более серьезная инфраструктура оценки производительности.

Подумайте, какой будет наиболее подходящая инфраструктура для вашего случая, и приступайте к ее созданию. Относитесь к ней во всех смыслах как к проекту первостепенной важности, реализуемому поэтапно, на который выделено достаточно ресурсов, а архитектура и код подвергаются пересмотру. Как можно раньше доведите инфраструктуру до работоспособного состояния и постепенно наращивайте в ней объем автоматизации.

Порой для того, чтобы убедить руководство в необходимости внедрения всех этих идей, требуется особое упорство. Вам могут помочь следующие советы.

- **Упирайте на возврат вложений.** Руководители мыслят финансовыми категориями. Большие расходы (деньги) в данный момент означают меньшие расходы (деньги) в дальнейшем.
- **Напомните об общих затратах владельца.** Речь опять идет о деньгах и времени. Если вложения способны уменьшить расходы в других областях, значит, они становятся выгодными.
- **Не скатывайтесь к мелочам.** Разговаривайте на языке, понятном руководству. Если интересуют технические подробности, то и обсуждайте их, в иных случаях придерживайтесь вопросов, волнующих руководство.
- **Придерживайтесь определенной политики.** Выбор правильного решения не всегда неизбежен. Зачастую на решения влияют факторы, не имеющие ничего общего с выделением ресурсов или техническими аспектами. Будьте в курсе политической ситуации в организации и постарайтесь соотносить с ней свои действия. Возможно, тут придется идти на компромисс.
- **Привлекайте сторонников.** Чем больше людей с различным опытом или стоящих на разных позициях будет увлечено вашими предложениями, тем серьезнее их станут воспринимать.
- **Приводите фактические обоснования.** Если в качестве обоснований необходимости внедрения ваших идей можно привести конкретные данные, примеры или случившиеся в прошлом инциденты и даже аварийные ситуации, обязательно воспользуйтесь этим.

Доверяйте только конкретным числовым показателям

Во многих командах, откладывавших решение проблем с производительностью, приступают к этому только тогда, когда они начинают серьезно влиять на восприятие программы конечными пользователями. Это означает, что все сводится к следующей ситуации.

Пользователь: «Ваше приложение работает очень медленно!»

Разработчик: «Из-за чего?»

Пользователь: «Я не знаю! Просто устраните недоделки!»

Разработчик: «*Ушел ускорять приложение, если повезет*».

Никто бы не хотел участвовать в таком разговоре. Всегда нужно иметь числа, полученные при измерении всего, что вы оцениваете. Необходимы данные, подводящие базу под все, что вы делаете. Людское доверие безгранично повышается при виде числовых показателей и графиков. Конечно, прежде чем выносить эти показатели на публику, необходимо убедиться в их достоверности!

Другим аспектом числовых показателей является гарантированное наличие у вашей команды официальных, реалистичных, ощутимых целей. В приведенном примере единственным «показателем» было слово «ускорить». Это неформальная, неконкретная и, по сути, бесполезная цель. Нужны же реальные, официальные

цели повышения производительности, к постановке которых следует привлечь всю цепочку руководства. Нужны результаты работы в виде конкретных показателей. Всех необходимо поставить в известность о том, что, как только они будут определены, неприемлемо неофициально требовать добиться более высокой производительности.

Подробнее о постановке разумных целей повышения производительности говорилось в главе 1.

Эффективная система просмотра кода

Идеальных разработчиков не бывает, но, когда за чьим-то кодом следит несколько пар глаз, его качество может существенно возрасти. Наверное, весь код должен пройти через процесс просмотра — это будет либо просмотр разницы между двумя версиями, с рассылкой по электронной почте, либо просмотр на совещании, на котором присутствует вся команда.

Нужно понимать, что не весь код одинаково важен для достижения бизнес-целей. Конечно, соблазнительно заявить, что самым высоким стандартам должен отвечать весь код, но эта планка может быть слишком высока в начале. Можно продумать особые категории оценок, применяемых к программным средствам, наиболее сильно влияющим на бизнес, где ошибка в функциональных свойствах или уровне производительности может привести к реальным денежным потерям (или к потере кем-то работы!). Например, можно перед отправкой кода потребовать подписи двух разработчиков, один из которых — старший разработчик или специалист в предметной области. Для широкомасштабного и сложного просмотра кода можно посадить всех в комнату с собственными ноутбуками, чтобы кто-то стал первым демонстрировать свой код через проектор. Как все это будет происходить, зависит от ресурсов и организации производства в вашей компании, но процесс должен развиваться и внедряться в практику, меняясь по мере необходимости.

Полезным может оказаться просмотр кода с концентрацией на его конкретных аспектах, например на функциональной корректности, безопасности или производительности. Можно попросить специалистов прокомментировать код исключительно в рамках их компетентности.

Эффективный просмотр кода не имеет ничего общего с придирками. Стилистические различия зачастую следует игнорировать. Иногда нужно обходить молчанием даже довольно крупные вопросы, если они не играют важной роли и существуют более серьезные проблемы, на которых следует сосредоточиться. То, что код отличается от того, какой написали бы вы, еще не означает, что он обязательно хуже. Нет ничего более деструктивного, чем, затеяв просмотр кода и ожидая критического разбора какого-нибудь сложного многопоточного программного решения, вместо этого убить кучу времени на споры о правильном синтаксисе комментариев или о других мелочах. Не тратьте время попусту. Задайте направление критического просмотра кода и заставьте его придерживаться. Если есть несомненные нарушения стандартов, не игнорируйте их, но сначала сосредоточьтесь на более важных вопросах.

В то же время не принимайте сомнительных отговорок вроде: «Да, я понимаю, что данная строка неэффективна, но разве, по большому счету, это имеет какое-то значение?» Правильным будет такой ответ: «Вы что, спрашиваете, насколько плохо можно писать код?» Нужно выдерживать баланс между игнорированием мелких недочетов и необходимостью формирования культуры повышения производительности, чтобы в следующий раз разработчик все делал правильно на автомате. Следует также иметь доказательство низкой производительности — либо на основе предшествующего опыта действий в подобных ситуациях, либо фактическим проведением эталонного тестирования. Приберегайте критику для очевидных проблем.

И наконец, нужно правильно понимать смысл общего «владения» кодом. Каждый должен чувствовать свою ответственность за весь проект. Нет независимых конкурирующих вотчин, и никто не должен рьяно отстаивать «свой» код. Наличие владельцев, отвечающих за раннее предотвращение проблем и просмотр кода, — фактор положительный, но каждый должен чувствовать себя уполномоченным на внесение улучшений в любую область кода. Оставьте самомнение за дверью.

Обучение

Выработка мышления, нацеленного на производительность, требует обучения. Учеба может носить неформальный характер, когда информацию получают от опытного специалиста команды или из книг, подобных данной, или облекаться в формальные рамки с платными занятиями и привлечением признанных специалистов в этой области.

Следует иметь в виду, что даже тем, кто уже знаком с .NET-программированием, придется сменить свои программистские привычки, как только они начнут серьезно вникать в проблемы повышения производительности.

А тем, кто хорошо разбирается в C или C++, нужно будет осознать, что здесь правила достижения высокой производительности зачастую совершенно другие или противоречащие их прежним представлениям, связанным с миром неуправляемого кода.

Изменяться нелегко, и большинство людей будут сопротивляться этому процессу, поэтому стоит проявить чуткость при внедрении новых приемов работы. И, как всегда, важно заручиться поддержкой руководства в том, чего вы пытаетесь добиться.

Если вам нужно подтолкнуть коллег к обсуждению проблем повышения производительности программных продуктов, могу подсказать, как это сделать.

- Начните за обедом непринужденную беседу, чтобы поделиться мыслями о том, что вы изучаете.
- Заведите внутренний или публичный блог, чтобы поделиться знаниями или обсудить проблемы повышения производительности, обнаруженные в программных продуктах.

- Подберите кого-нибудь из команды в качестве своего постоянного рецензента по вопросам повышения производительности.
- Продемонстрируйте преимущества, получаемые от повышения производительности, с помощью простых эталонных тестов или программ, подтверждающих работоспособность концепции.
- Назначьте кого-нибудь специалистом по повышению производительности. Он будет постоянно следить за производительностью, просматривать код, научит других действенным приемам и будет в курсе всех актуальных изменений в этой области и общего уровня ее развития. Если вы читаете эти строки, значит, вы уже вызвались добровольцем на эту роль.
- Проявите инициативу там, где возможны улучшения. Совет: лучше всего начать с собственного кода!
- Уговорите руководителей своей организации купить эту книгу для всех специалистов. (Беззастенчивая реклама!)

Резюме

Нацеливая команду на достижение высокой производительности программного продукта, начните с малого. Прежде всего возьмите собственный код и потратьте время на то, чтобы разобраться, какие области в нем действительно влияют на производительность. Добейтесь одинаковой нетерпимости к падению производительности и функциональным сбоям. Чтобы уменьшить нагрузку на команду, автоматизируйте как можно больше действий. Оценивайте показатели производительности точными цифрами, а не на уровне интуиции или субъективного восприятия.

Создайте эффективную систему просмотра кода, поощряющую качественный стиль программирования, сконцентрированность на реально значимых аспектах и коллективную ответственность за код.

Осознайте, что изменить уже сложившуюся ситуацию нелегко и здесь нужен чуткий подход. Скорее всего, сменить взгляды понадобится даже тем, кто хорошо разбирается в среде .NET. А ветеранам C++ и Java, чтобы освоиться в .NET-программировании, может понадобиться некоторое время.

Найдите способы внедрения в обычную практику команды регулярные обсуждения вопросов повышения производительности программ и подыщите или подготовьте специалистов для распространения информации на эту тему.

Приложение А. Начало работы над повышением производительности приложения

В этой книге упомянуты сотни подробностей, касающихся проблем, которые могут возникнуть в вашем приложении, но если вы в самом начале пути, ознакомьтесь с изложенными далее общими направлениями движения к анализу и последующему повышению производительности ваших программ.

Определение метрик

- Определите интересующие вас метрики.
- Решите, какой тип статистических данных вам необходим: по средним показателям, по минимуму, по максимуму, в процентном выражении или еще более сложный вариант.
- Каковы ограничения на ресурсы, с которыми вам приходится иметь дело? Возможные их примеры: центральный процессор, использование памяти, интенсивность выделения памяти, сетевой ввод-вывод, задействование диска, интенсивность записи на диск и т. д.
- Какова цель применения метрики или ресурса?

Анализ использования центрального процессора

- Используйте PerfView или Visual Studio Standalone Profiler для получения результатов профилирования работы центрального процессора при выполнении вашего приложения.
- Проанализируйте стеки для особо выделяющихся функций.
- Обработка данных занимает много времени?

- Можно ли изменить структуру данных, чтобы она была представлена в формате, требующем меньшей обработки? Например, вместо разбора XML воспользуйтесь простым двоичным форматом сериализации.
- Есть ли альтернативные API?
- Можете ли вы распараллелить работу с применением `Task`-делегатов или `Parallel.For`?

Анализ использования памяти

- Выберите подходящий тип сборки мусора.
  - Серверный — ваша программа является единственным значимым приложением на машине и нуждается в самой низкой задержке для сборок мусора.
  - Рабочей станции — у вас имеется пользовательский интерфейс или вы делите машину с другим важным процессом.
- Проведите профилирование памяти с помощью PerfView.
  - Найдите среди результатов самых активных «выделителей» памяти — насколько они ожидаемы и приемлемы?
  - Уделите пристальное внимание на выделения памяти в куче больших объектов.
- Если сборки мусора в поколении gen 2 происходят слишком часто.
  - Не слишком ли часто выделяется память в LOH? Удалите или объедините в пулы выделяемые в этой куче объекты.
  - Не чрезмерен ли объем объектов, перемещаемых в старшее поколение? Сокращайте время существования объектов, чтобы они попадали под сборки мусора в предыдущих поколениях. Выделяйте память под объекты только при крайней надобности и избавляйтесь от них, когда они больше не нужны.
  - Если объекты существуют слишком долго, объединяйте их в пул.
- Если сборка мусора в поколении gen 2 занимает слишком много времени.
  - Оцените возможность применения GC-уведомлений для получения сигнала готовности сборки мусора к запуску. Используйте эту возможность для остановки обработки.
  - Сократите частоту полных сборок мусора, понизив объем перемещения объектов в старшее поколение и снизив количество выделений памяти в LOH.
- Если наблюдается большое количество сборок мусора в поколениях gen 0/1.
  - Найдите область с наиболее активными выделениями памяти с помощью профилирования. Найдите способ сокращения надобности в выделении памяти.
  - Минимизируйте время существования объектов.

- Если сборка мусора в поколениях gen 0/1 выполняется с большими паузами.
  - Сократите общее количество выделений памяти.
  - Минимизируйте время существования объектов.
  - Были ли объекты закреплены? Удалите их, если возможно, или сократите область видимости закрепления.
  - Упростите объекты за счет удаления ссылок между ними.
- Если LOH разрастается.
  - Проверьте степень фрагментации с помощью WinDbg или CLR Profiler.
  - Периодически уплотняйте кучи больших объектов.
  - Проверьте, нет ли пулов объектов с безудержным ростом.

Анализ JIT-компиляции

- Если на запуск уходит слишком много времени.
  - Действительно ли причина в JIT-компиляции? Чаще всего слишком долгий запуск вызван загрузкой данных, характерных для приложения. Следует убедиться, что причина действительно в JIT-компиляции.
  - Используйте PerfView для анализа, позволяющего выявить методы, на JIT-компиляцию которых уходит слишком много времени.
  - Применяйте Profile Optimization для ускорения JIT-компиляции при загрузке приложения.
  - Проанализируйте возможность использования NGEN.
  - Проанализируйте возможность применения настраиваемой предварительной подготовки путем выполнения вашего кода.
- Появляются ли в профиле те методы, код которых вы ожидали увидеть встроенным?
  - Найдите код, препятствующий встраиванию кода методов, например циклы, обработка исключений, рекурсии и т. д.

Анализ производительности в асинхронном режиме

- Используйте PerfView для выявления высокого уровня конфликтности.
  - Устраните конфликты путем реструктуризации кода таким образом, чтобы требовалось меньше блокировок.
  - Используйте в нужных местах методы `Interlocked` или гибридные блокировки.

- Захватывайте события категории Thread Time с помощью PerfView для выяснения того, на что затрачивается время. Анализируйте соответствующие области кода, чтобы убедиться, что потоки не блокируются на операциях ввода-вывода.
  - Возможно, чтобы избежать ожидания на Task-объектах или операциях ввода-вывода, придется внести в программу существенные изменения, придавая ей бóльшую асинхронность на всех уровнях.
  - Удостоверьтесь в применении асинхронных потоковых API.
- Возникают ли паузы перед тем, как ваша программа приступает к эффективному использованию пула потоков? Это может проявляться в виде начального замедления, исчезающего через несколько минут.
  - Удостоверьтесь, что минимальный размер пула потоков соответствует рабочей нагрузке.

Приложение Б. Увеличение производительности на более высоком уровне

Эта книга призвана главным образом помочь вам разобраться в основах обеспечения производительности с точки зрения основных элементов .NET. Прежде чем складывать из своих строительных блоков более крупные приложения, очень важно понять, какие затраты будут приходиться на эти самые блоки. Все, что до сих пор рассматривалось в книге, применимо к большинству типов .NET-приложений, включая и те, речь о которых пойдет в этом приложении.

Данное приложение станет шагом наверх и даст вам несколько кратких советов по поводу популярных типов приложений. Я не стану рассматривать эти темы так же подробно, как делал на других страницах книги, поэтому считайте это общим обзором, вдохновляющим вас на дальнейшие исследования. Советы большей частью не связаны со средой .NET и ориентированы на архитектуры, предметные области или библиотеки.

ASP.NET

- Отключите ненужные HTTP-модули.
- Удалите неиспользуемые View Engines.
- Не проводите компиляцию производственной версии как для отладки (проверьте наличие `<compilation debug="true"/>`).
- Сократите количество обращений браузера к серверу.
- Убедитесь в том, что буферизация страниц включена (по умолчанию включена).
- Освойте интенсивное использование кэшей:
  - `OutputCache` — кэширует вывод страницы;
  - `Cache` — кэширует произвольные объекты по вашему желанию.
- Оцените объем своих страниц с точки зрения клиента.
- Уберите на страницах ненужные и пробельные символы.

- Применяйте HTTP-сжатие.
- Производите валидацию данных на стороне клиента, чтобы уменьшить обмен данными с сервером. Но также проводите валидацию на стороне сервера для надежности.
- Отключите или ограничьте использование `ViewState` для малых объектов. Если без него не обойтись, воспользуйтесь сжатием.
- Выключите состояние сессии, если в нем нет необходимости.
- Не пользуйтесь методом `Page.DataBind`.
- Объединяйте в пул подключения к внутренним серверам, например, к базам данных.
- Выполняйте предварительную компиляцию веб-сайта.
- Воспользуйтесь свойством `Page.IsPostBack`, чтобы запустить код, который должен выполняться только один раз на каждой странице, например для инициализации.
- Применяйте `Server.Transfer` вместо `Response.Redirect`.
- Не используйте задач с большим временем выполнения.
- Избегайте конфликтов при блокировках или блокировок потоков по какой-либо причине.

ADO.NET

- Сохраняйте объекты подключений, команд, параметров и другие объекты, связанные с базами данных, в повторно используемых полях, а не в экземплярах, создаваемых заново при каждом вызове часто применяемого метода.
- Объединяйте сетевые подключения в пулы.
- Убедитесь в том, что структура и индексация базы данных верны.
- Сократите количество гуляющих туда и обратно запросов к базе данных.
- Кэшируйте как можно больше данных локально, в памяти.
- Используйте где только возможно сохраненные процедуры.
- Для больших наборов данных задействуйте постраничную выборку, то есть не возвращайте целиком весь набор данных.
- По возможности объединяйте запросы в пакеты.
- Применяйте объекты `DataView` в качестве надстройки над объектами `DataSet` вместо повторных запросов одной и той же информации.
- Если можно обойтись представлением данных с кратковременным однонаправленным их перебором, воспользуйтесь `DataReader`.
- Профилируйте производительность запросов с помощью SQL Query Analzyer.

WPF

- Работайте с самой свежей версией среды .NET, поскольку за последние годы произошло значительное повышение ее производительности.
- Никогда не выполняйте большой объем обработки данных в потоке пользовательского интерфейса.
- Убедитесь в отсутствии ошибок привязки.
- Обходитесь только абсолютно необходимыми визуальными представлениями. Избыточные преобразования и уровни замедлят отображение данных.
- Сократите размер и глубину видимого дерева.
- Используйте минимально приемлемую частоту кадров анимации.
- Для отображения только видимых объектов применяйте виртуальные представления и списки.
- Предусмотрите при необходимости возможность отложенной прокрутки длинных списков.
- `StreamGeometry` работает быстрее `PathGeometry`, но поддерживает меньше функций.
- `Drawing`-объекты обрабатываются быстрее `Shape`-объектов, но поддерживают меньше функций.
- По возможности не заменяйте, а обновляйте преобразования визуализации.
- Заставьте WPF явным образом загружать изображения нужного вам размера, если размер их отображения будет меньше полного.
- Удалите обработчики событий из объектов, чтобы обеспечить их удаление при сборке мусора.
- Переопределите метаданные `DependencyProperty`, чтобы перенастроить их при изменении значений, вызывающих повторное отображение.
- Заморозьте объекты, если нужно избежать издержек на уведомления об изменениях.
- Отдавайте предпочтение не динамическим, а статическим ресурсам.
- Чтобы показывать минимальный набор свойств, выполняйте привязку к CLR-объектам с несколькими свойствами или создайте объекты-оболочки.
- Отключите хит-тестирование для больших 3D-объектов, если в нем нет необходимости.
- Перекомпилируйте код для универсальной платформы Windows с прицелом на Windows 10, чтобы без каких-либо затрат получить существенное повышение производительности.

Приложение B. Нотация «"O" большое»

На уровень выше непосредственного профилирования производительности лежит алгоритмический анализ. Обычно он проводится на предмет абстрактных операций относительно величины задачи. В компьютерной науке есть стандартный способ обозначения затратности алгоритмов, который называется «O» большим.

«O» большое

Нотация «"O" большое», также известная как асимптотическая нотация, представляет собой способ обобщения производительности алгоритмов на основе величины задачи. Обычно величина задачи обозначается *n*. То, какое у алгоритма «O» большое, является показателем его сложности. Определение «асимптотическое» используется для того, чтобы описать поведение функции, когда размер ее входных данных приближается к бесконечности.

Рассмотрим в качестве примера неотсортированный массив, содержащий значение, которое нам надо найти. Поскольку он не отсортирован, нам придется вести поиск в каждом элементе до тех пор, пока это значение не будет найдено. Если массив имеет размер *n*, в худшем случае придется искать среди *n* элементов. Поэтому говорится, что этот алгоритм линейного поиска имеет сложность O(*n*).

Это самый худший случай. Но в среднем алгоритм предполагает, что искать придется в *n* / 2 элементах. Можно уточнить оценку и сказать, что алгоритм в среднем имеет сложность O(*n* / 2), но фактически это не такое уж большое изменение, если рассматривать фактор роста (*n*). Константы отбрасываются, оставляя нас с той же степенью сложности O(*n*).

Нотация «"O" большое» выражается через функции от *n*, где *n* — размер входных данных, определяемый алгоритмом и структурой данных, с которыми он работает. Для коллекции он может быть количеством элементов в коллекции, для алгоритма поиска в строке — длиной строки.

Нотация «"O" большое» указывает рост времени, необходимого для выполнения алгоритма, растет с увеличением размера входных данных. В примере с массивом

ожидается, что при удвоении размера массива время, необходимое для поиска, также удвоится. Это означает, что алгоритм имеет линейные характеристики производительности.

Алгоритм со сложностью $O(n^2)$ будет показывать производительность ниже линейной. При удвоении входных данных время учетверяется. Если размер задачи вырастает в восемь раз, время увеличивается в 64 раза, всегда являясь квадратом размера. Этот тип алгоритма демонстрирует квадратичную сложность. Хорошим примером может послужить алгоритм пузырьковой сортировки (фактически сложность $O(n^2)$ имеют большинство простых алгоритмов сортировки):

```
private static void BubbleSort(int[] array)
{
    bool swapped;
    do
    {
        swapped = false;
        for (int i = 1; i < array.Length; i++)
        {
            if (array[i - 1] > array[i])
            {
                int temp = array[i - 1];
                array[i - 1] = array[i];
                array[i] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}
```

Когда встречаются вложенные циклы, то, скорее всего, алгоритм будет квадратичным или полиномиальным, если не хуже. В случае с пузырьковой сортировкой внешний цикл может запускаться до n раз, а внутренний будет проверять до n элементов на каждой итерации, поэтому сложность можно обозначить как $O(n^2)$.

При анализе ваших собственных алгоритмов можно придумать формулу, содержащую несколько факторов, как в случае с $O(8n^2 + n + C)$ (квадратичная часть, умноженная на 8, линейная часть и часть с постоянным временем). Для целей нотации «"O" большое» оставляют только наиболее значимый фактор, а мультипликативные константы игнорируют. Этот алгоритм будет трактоваться как $O(n^2)$. Следует также помнить, что нотация «"O" большое» определяет рост времени по мере приближения величины задачи к бесконечности. Даже притом, что $8n^2$ в восемь раз больше n^2, это совсем не актуально по сравнению с ростом фактора n^2, который для больших значений n сильно превосходит любой другой фактор. И наоборот, если n небольшое, разница между $O(n \log n)$, $O(n^2)$ или $O(2^n)$ незначительна и неинтересна. Обратите внимание на то, что у вас могут применяться сложные значимые факторы, например $O(n^2 \cdot 2^n)$, и ни один компонент,

включающий n, не будет удален, пока он действительно не станет незначительным.

У многих алгоритмов имеется несколько входных данных, и их сложность может быть обозначена несколькими переменными, например $O(mn)$ или $O(m + n)$. К примеру, многие алгоритмы с графами зависят от количества ребер и количества вершин.

Наиболее часто встречаются следующие типы сложностей:

- $O(1)$ (постоянная) — требуемое время не зависит от размера входных данных. Сложностью $O(1)$ обладают многие хеш-таблицы;
- $O(\log n)$ (логарифмическая) — время увеличивается как доля размера входных данных. Любой алгоритм, делящий пространство задачи пополам при каждой итерации, имеет логарифмическую сложность. Обратите внимание на то, что основание этого логарифма не указано;
- $O(n)$ (линейная) — время увеличивается пропорционально размеру входных данных;
- $O(n \log n)$ (логарифмически линейная) — время увеличивается квазилинейно, то есть во времени доминирует линейный фактор, но при этом происходит умножение на долю размера входных данных;
- $O(n^2)$ (квадратическая) — время увеличивается с квадратом размера входных данных;
- $O(n^C)$ (полиномиальная) — C больше или равно 2;
- $O(C^n)$ (экспоненциальная) — C больше 1;
- $O(n!)$ (факториальная) — попробуйте каждую перестановку.

Алгоритмическая сложность обычно описывается в понятиях ее средней и наихудшей производительности. Наилучшая производительность не слишком интересна, поскольку на многие алгоритмы может повлиять удача (например, для нашего анализа фактически все равно, что наилучшая производительность линейного поиска обозначается $O(1)$, поскольку это означает, что нам просто повезло).

На графике на рис. B.1 показано, как быстро может расти время на основе величины задачи. Обратите внимание на то, что разница между $O(1)$ и $O(\log n)$ практически неразличима даже для относительно больших задач. Алгоритм со сложностью $O(n!)$ практически непригоден для всего, кроме самых мелких задач.

Хотя время — самая распространенная размерность сложности, по такой же методологии может быть проанализировано и пространство (использование памяти). Например, большинство алгоритмов сортировки по времени имеют сложность $O(\log n)$, а по пространству — $O(n)$. С точки зрения сложности, совсем немногие структуры данных занимают больше места, чем то количество элементов, которое в них содержится.

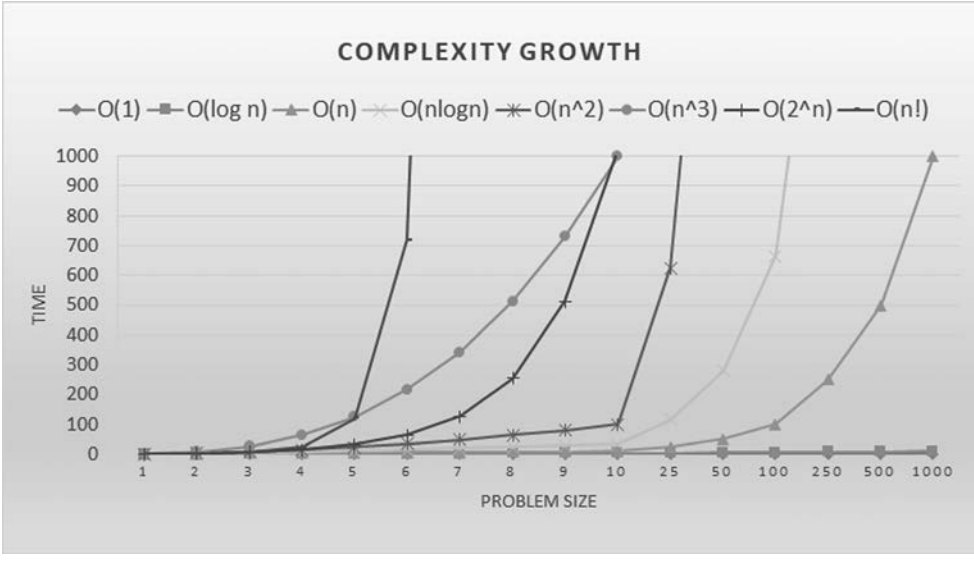

Рис. B.1. Влияние величины задачи на скорость возрастания сложности разных типов алгоритмов

Самые распространенные алгоритмы и их сложность

Сортировка

- Быстрая сортировка (Quicksort) — O($n \log n$), O(n^2) — худший случай.
- Сортировка методом слияния (Merge sort) — O($n \log n$).
- Древовидная сортировка (Heap sort) — O($n \log n$).
- Пузырьковая сортировка (Bubble sort) — O(n^2).
- Сортировка методом вставок (Insertion sort) — O(n^2).
- Сортировка методом выбора (Selection sort) — O(n^2).

Графы

- Поиск преимущественно в глубину (Depth-first search) — O($E + V$), где E — ребра, V — вершины.
- Поиск преимущественно в ширину (Breadth-first search) — O($E + V$).
- Кратчайший путь (Shortest-path с использованием Min-heap) — O(($E + V$)log V).

Поиск

- Неотсортированный массив — O(n).
- Отсортированный массив с двоичным поиском — O($\log n$).
- Дерево двоичного поиска — O($\log n$).
- Хеш-таблица — O(1).

Особый случай

- Вычисление каждой перестановки строки — O($n!$).
- Коммивояжер — O($n!$). Надо признать, это наихудший случай. Существует способ решения со сложностью O($n^2 \cdot 2^n$), предусматривающий использование технологии динамического программирования.

Зачастую O($n!$) фактически является сокращением для понятия «решение в лоб, опробование каждой возможности».

Приложение Г. Библиография

Ценные источники информации

- *Hewardt M., Dussud P.* Advanced .NET Debugging. Addison-Wesley Professional, 2009.
- *Richter J.* CLR via C#. 4th ed. Microsoft Press, 2012.
- *Russinovich M., Solomon D., Ionescu A.* Windows Internals. 6th ed. Microsoft Press, 2012.
- *Rasmussen B.* High-Performance Windows Store Apps. Microsoft Press, 2014.
- Стандарты ECMA C# и CLI. https://www.visualstudio.com/license-terms/ecma-c-common-language-infrastructure-standards/, Microsoft.
- Закон Амдала. http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.

Люди и блоги

Кроме вышеупомянутых источников информации, стоит прислушаться к мнениям людей, излагаемым либо в блогах, либо в статьях в различных изданиях.

- Блог .NET Framework — сообщения, новости, обсуждения и серьезные статьи. http://blogs.msdn.com/b/dotnet/.
- Маони Стивенс (Maoni Stephens) — разработчик CLR и специалист по сборке мусора. Ее блог находится по адресу http://blogs.msdn.com/b/maoni/, обновляется довольно редко, но там много полезной информации и временами появляются важные сообщения.
- Вэнс Моррисон (Vance Morrison) — архитектор производительности .NET. Автор инструментальных средств PerfView, MeasureIt и многочисленных статей и презентаций по вопросу производительности .NET. Блоги по адресу http://blogs.msdn.com/b/vancem/.
- Мэтт Уоррен (Matt Warren) — энтузиаст повышения производительности в среде .NET, особо ценный специалист компании Microsoft, блогер и соучастник

многих .NET-проектов с открытым кодом, включая BenchmarkDotNet. http://mattwarren.org/.

- Брендан Грегг (Brendan Gregg): http://www.brendangregg.com/. Не имеет отношения к .NET, но в этом источнике масса ценной информации, относящейся к производительности.
- Журнал MSDN Magazine: http://msdn.microsoft.com/magazine. Содержит множество великолепных статей, освещающих самые сокровенные подробности внутреннего устройства среды CLR.

Бен Уотсон

Высокопроизводительный код на платформе .NET

2-е издание

Перевел с английского Н. Вильчинский

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*М. Сагалович*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

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: 06.2019. 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 28.05.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87