O'REILLY®

Начните работу с Apache Flink, фреймворком с открытым исходным кодом, на котором основаны многие крупнейшие в мире системы для обработки потоковых данных. В данной книге вы изучите фундаментальные понятия параллельной потоковой обработки и узнаете, чем эта технология отличается от традиционной пакетной обработки данных.

Ф. Уэске и В. Калаври, занятые в проекте Apache Flink с первых дней, покажут вам, как создавать масштабируемые потоковые приложения с помощью API Flink DataStream, а также непрерывно выполнять и поддерживать эти приложения в операционных средах.

Потоковая обработка идеально подходит для многих задач: подготовка данных с малой задержкой, потоковая аналитика и информационные панели в реальном времени, раннее оповещение и обнаружение мошенничества. Вы можете обрабатывать потоковые данные любого типа, включая взаимодействия с пользователем, финансовые транзакции и данные интернета вещей, немедленно после получения.

«Отличная книга для всех, от старожилов до начинающих программистов и инженеров по обработке данных, выполняющих свои первые задания по потоковой обработке. В книге не только рассматривается Flink, в ней также преподаются основы потоковой обработки, которые помогут читателям повысить уровень знаний и развить навыки технического мышления. Рекомендую всем читателям».

Тед Маласка,
директор по IT-архитектуре предприятия в Capital One

- Концепции и проблемы распределенной потоковой обработки данных с учетом состояния.
- Системная архитектура Flink, включая режим обработки по времени событий и принципы отказоустойчивости.
- Идеология и стандартные компоненты API DataStream, включая операторы с привязкой ко времени и с учетом состояния.
- Чтение и запись данных во внешние системы с гарантией согласованности «ровно один раз».
- Развертывание и настройка кластеров Flink.
- Обслуживание непрерывно работающих потоковых приложений.

Фабиан Уэске – член консультативного совета проекта Apache Flink. Он вносит свой вклад в развитие Flink с первого дня. Фабиан является соучредителем Data Artisans (ныне Ververica) и имеет докторскую степень в области компьютерных наук Берлинского технического университета.
Василики Калаври – научный сотрудник Systems Group при Федеральной технической школе Цюриха. Она является одним из подрядчиков проекта Apache Flink. Будучи одним из первых разработчиков Flink, она работала над его библиотекой обработки графов Gelly, а также над ранними версиями Table API и потоковым SQL.

ISBN 978-5-97060-880-7

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Потоковая обработка данных с Apache Flink O'REILLY®

O'REILLY®

Потоковая обработка данных с Apache Flink

Основы разработки потоковых приложений

Фабиан Уэске
Василики Калаври

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Фабиан Уэске, Василики Калаври

Потоковая обработка данных с Apache Flink

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske
Vasiliki Kalavri

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Потоковая обработка данных с Apache Flink

Основы разработки потоковых приложений

Фабиан Уэске
Василики Калаври

ДМК Пресс
издательство
Москва, 2021

УДК 004.4
ББК 32.972
У99

Уэске Ф., Калаври В.
У99 Потоковая обработка данных с Apache Flink / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2021. – 298 с.: ил.

ISBN 978-5-97060-880-7

Начните работу с Apache Flink, фреймворком с открытым исходным кодом, на котором основаны многие крупнейшие в мире системы обработки потоковых данных. В данной книге вы изучите фундаментальные понятия параллельной потоковой обработки и узнаете, чем эта технология отличается от традиционной пакетной обработки данных.

Ф. Уэске и В. Калаври, занятые в проекте Apache Flink с первых дней, покажут вам, как создавать масштабируемые потоковые приложения с помощью API Flink DataStream, а также непрерывно выполнять и поддерживать эти приложения в операционных средах.

Потоковая обработка идеально подходит для многих задач: подготовки данных с малой задержкой, потоковой аналитики и информационных панелей в реальном времени, раннего оповещения и обнаружения мошенничества. Вы можете обрабатывать потоковые данные любого типа, включая взаимодействия с пользователем, финансовые транзакции и данные интернета вещей, немедленно после получения.

УДК 004.4
ББК 32.972

ISBN 978-1-491-97429-2 (англ.)
ISBN 978-5-97060-880-7 (рус.)

Содержание

Предисловие

Что вы узнаете из этой книги

Эта книга научит вас всему, что вам нужно знать о потоковой обработке с помощью Apache Flink. Она состоит из 11 глав, которые, как мы надеемся, расскажут вам связную историю. В то время как одни главы являются описательными и знакомят с концепциями проектирования высокого уровня, другие главы более прикладные и содержат много примеров кода.

Хотя при написании книги мы предполагали, что ее будут читать в порядке следования глав, читатели, знакомые с содержанием главы, могут пропустить ее. Другие читатели, более заинтересованные в написании кода Flink прямо сейчас, могут сначала прочитать практические главы. Далее мы кратко опишем содержание каждой главы, чтобы вы могли сразу перейти к тем, которые вас интересуют больше всего.

- В главе 1 приводится обзор потоковой обработки с учетом состояния, архитектур приложений для обработки данных, подходов к разработке приложений и преимуществ потоковой обработки по сравнению с традиционными подходами. Мы также коротко расскажем о том, как можно запустить ваше первое потоковое приложение на локальном экземпляре Flink.
- В главе 2 обсуждаются фундаментальные концепции и проблемы потоковой обработки в целом, независимо от Flink.
- Глава 3 описывает системную архитектуру и внутреннее устройство Flink. В ней обсуждается распределенная архитектура, обработка событий, зависящих от времени и состояния в потоковых приложениях, а также механизмы отказоустойчивости Flink.
- Глава 4 объясняет, как настроить среду для разработки и отладки приложений Flink.
- Глава 5 знакомит вас с основами API DataStream Flink. Вы узнаете, как реализовать приложение DataStream и какие потоковые преобразования, функции и типы данных оно поддерживает.
- В главе 6 обсуждаются операторы с привязкой ко времени события API DataStream. Сюда входят операторы оконной обработки и привязки ко времени, а также функции процессов, которые обеспечивают максимальную гибкость при работе со временем в потоковых приложениях.
- В главе 7 мы объясняем, как реализовать функции с учетом состояния, и обсуждаем все, что связано с этой темой, например быстродействие, надежность и эволюцию функций с учетом состояния. Здесь также показано, как использовать запрашиваемое состояние Flink.
- В главе 8 представлены наиболее часто используемые соединители источника и приемника данных Flink. В ней обсуждается подход Flink

к сквозной согласованности приложений и способы реализации настраиваемых коннекторов для приема и передачи данных во внешние системы.
- Глава 9 рассказывает, как устанавливать и настраивать кластеры Flink в различных средах.
- Глава 10 посвящена работе, мониторингу и обслуживанию потоковых приложений, работающих круглосуточно и без выходных.
- Наконец, глава 11 рассказывает о ресурсах, которые вы можете использовать, чтобы задавать вопросы, посещать мероприятия, связанные с Flink, и узнавать о способах применения Flink в настоящее время.

Соглашения, принятые в этой книге

В книге используются следующие типографские соглашения.

Курсив – используется для смыслового выделения важных положений, новых терминов, URL-адресов и адресов электронной почты в интернете, имен команд и утилит, а также имен и расширений файлов и каталогов.

`Моноширинный шрифт` – используется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт` – используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

`Моноширинный курсив` – используется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Совет. Такая пиктограмма обозначает совет или рекомендацию.

Примечание. Такая пиктограмма обозначает указание или примечание общего характера.

Предупреждение. Эта пиктограмма обозначает предупреждение или особое внимание к потенциально опасным ситуациям.

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» и Microsoft очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Эта книга не появилась бы на свет без помощи и поддержки множества замечательных людей. Мы хотим упомянуть и поблагодарить хотя бы некоторых из них.

В этой книге обобщены знания, полученные за годы проектирования, разработки и тестирования силами сообщества Apache Flink. Мы благодарны всем, кто внес свой вклад в Flink посредством написания кода, документации, обзоров, отчетов об ошибках, запросов функций, обсуждений в списках рассылки, тренингов, конференций, организации встреч и других мероприятий.

Мы особенно благодарны тем, кто вместе с нами принимал участие в разработке Flink. Вот их имена: Алан Гейтс, Алхоша Креттек, Андра Лунгу, Ченг Сян Ли, Чесни Шеплер, Чиван Парк, Дэниел Варнеке, Давид Высакович, Гэри Яо, Грег Хоган, Гьюла Фора, Генри Сапутра, Джейми Гриер, Джарк Ву, Джин-Чен Сун, Константинос Клоудас, Костас Цумас, Курт Янг, Мартон Баласси, Маттиас Дж. Сакс, Максимилиан Михельс, Нико Крубер, Пэрис Карбон, Роберт Мецгер, Себастьян Шелтер, Шаосю Ван, Шуй Чен, Стефан Рихтер, Стефан Юэн, Теодор Василодис, Томас Вайсе, Тиль Рорманн, Тимо Вальтер, Цзы-Ли (Гордон) Тай, Уфук Челеби, Сяоган Ши, Сяовей Цзян, Синджан Цуй. Этой книгой мы надеемся привлечь разработчиков, инженеров и энтузиастов потоковой обработки по всему миру и еще больше расширить сообщество Flink.

Мы также благодарим наших технических рецензентов, которые внесли бесчисленные ценные предложения и помогли нам сделать эту книгу лучше. Спасибо, Адам Кава, Алхоша Креттек, Кеннет Ноулз, Леа Джордано, Матиас Дж. Сакс, Стефан Юэн, Тед Маласка и Тайлер Акидау.

Наконец, мы благодарим всех сотрудников O'Reilly, которые сопровождали нас в нашем пути все два с половиной года и помогли довести этот проект до финиша. Спасибо, Алисия Янг, Колин Лобнер, Кристин Эдвардс, Кэтрин Тозер, Мари Богюро и Тим Макговерн.

Об авторах

Фабиан Уэске является коммитером и подрядчиком проекта Apache Flink и вносит свой вклад в разработку Flink с самых первых дней его создания. Фабиан является соучредителем и инженером-программистом в берлинском стартапе Ververica (ранее называвшемся Data Artisans), который поддерживает Flink и его сообщество. Он имеет докторскую степень в области компьютерных наук Берлинского технического университета.

Василики Калаври – научный сотрудник Systems Group при Фендеральной технической школе Цюриха, где она широко использует потоковые системы Apache Flink для исследования и преподавания. Василики тоже является одним из подрядчиков проекта Apache Flink. Будучи одним из первых участников Flink, она работала над его библиотекой обработки графов Gelly, а также над ранними версиями Table API и потоковым SQL.

Колофон

Животное на обложке книги – евразийская рыжая белка (Sciurus vulgaris). Большинство древесных белок в Азии, Европе и Америке в местностях с умеренным климатом относятся к роду *sciurus*. *Vulgaris* в переводе с латыни означает «обычный», а евразийские рыжие белки – обычное явление для всей Европы и Северной Азии.

У евразийских рыжих белок белое кольцо вокруг глаз, пушистый хвост и пучок меха над ушами. Их окраска на голове и спине варьируется от светло-красной до черной. Мех на животе кремового или белого цвета. Зимой мех белки вырастает немного выше и длиннее над ушами и вокруг лап, чтобы защитить их от холода. Большую часть зимы они проводят, свернувшись клубочком в гнездах.

Гнезда евразийской рыжей белки предназначены для содержания одного животного, за исключением спаривания или выращивания детенышей. Хотя они живут поодиночке, их ареалы часто пересекаются из-за большой численности. В среднем самки рожают по пять детенышей на помет дважды в год. Молодые белки покидают материнское гнездо примерно через два месяца. За бельчатами охотятся многие хищники, включая птиц, змей и млекопитающих, поэтому только четверть молодняка достигает возраста одного года.

Евразийские рыжие белки питаются семенами, желудями и орехами. Их также замечали за слизыванием древесного сока, но они нечасто экспериментируют с новой едой. Эти белки вырастают от 23 до 26 см в длину от головы до задней части, и примерно такой же длины у них хвост. Они весят от 230 до 280 г и могут жить до 12 лет. Ожидаемая продолжительность жизни в дикой природе составляет всего 4–7 лет.

Евразийские рыжие белки обладают отличной ловкостью и балансом. Сочетание острых изогнутых когтей и длинных пушистых хвостов позволяет этим обитателям деревьев взбираться по стволам, спускаться вниз головой и прыгать через нависающие ветки.

Многие животные на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира. Чтобы узнать больше о том, чем вы можете помочь, перейдите на сайт animals.oreilly.com.

Глава 1

Введение в потоковую обработку с учетом состояния

Apache Flink – это распределенный потоковый процессор с интуитивно понятными и четко структурированными API для реализации приложений потоковой обработки данных с учетом состояния. Он предоставляет надежную среду для выполнения крупномасштабных защищенных от сбоев приложений. Flink присоединился к Apache Software Foundation в качестве инкубационного проекта в апреле 2014 года и стал проектом высшего уровня в январе 2015 года. С самого начала Flink имел очень активное и постоянно растущее сообщество пользователей и участников. На сегодняшний день участниками разработки Flink стало более пятисот человек, и он превратился в один из самых сложных механизмов обработки потоковых данных с открытым исходным кодом – репутация, подкрепленная повсеместным распространением. Flink лежит в основе крупномасштабных критически важных для бизнеса приложений во многих компаниях и предприятиях в различных отраслях и по всему миру.

Технология потоковой обработки данных становится все более популярной среди больших и малых компаний, потому что она не только предлагает превосходные передовые решения для многих традиционных сценариев использования, таких как аналитика данных, ETL[1] и транзакционные приложения, но и облегчает реализацию новых приложений, архитектур программного обеспечения и бизнес-возможностей. В этой главе мы обсудим, почему *потоковая обработка с учетом состояния* (stateful stream processing) становится такой популярной, и оценим ее потенциал. Мы начнем с обзора традиционных архитектур приложений обработки данных и укажем на их ограничения. Затем мы представим читателю проекты приложений, основанные на потоковой обработке с отслеживанием состояния и демонстрирующие множество интересных свойств и преимуществ по сравнению с традиционными подходами. Наконец, мы кратко обсудим эволюцию по-

[1] Extract, transform, load – основные этапы переноса информации из одного приложения в другое. – *Прим. перев.*

токовых процессоров с открытым исходным кодом и поможем вам запустить потоковое приложение на локальном экземпляре Flink.

1.1. Традиционные инфраструктуры данных

Данные и их обработка повсеместно используются на предприятиях на протяжении многих десятилетий. С годами сбор и использование данных неуклонно росли, и компании спроектировали и построили инфраструктуры для управления этими данными. Традиционная архитектура, которую реализует большинство предприятий, различает два типа обработки данных: *транзакционная обработка* (transactional processing) и *аналитическая обработка* (analytical processing). Ниже мы обсудим оба типа и связанные с ними подходы к управлению и обработке.

1.1.1. Транзакционная обработка

В своей повседневной деятельности компании используют всевозможные приложения, такие как системы планирования ресурсов предприятия (enterprise resource planning, ERP), программное обеспечение для управления взаимодействием с клиентами (customer relationship management, CRM) и веб-приложения. В таких системах обычно предусмотрены отдельные уровни для обработки данных (само приложение) и хранения данных (система транзакционной базы данных), как показано на рис. 1.1.

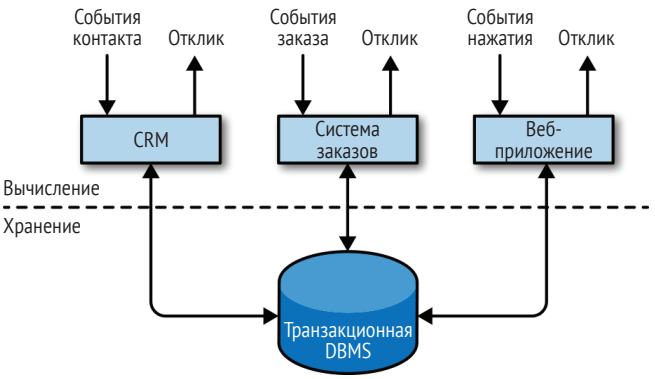

Рис. 1.1 ❖ Традиционная структура транзакционных приложений, хранящих данные в удаленной системе баз данных

Приложения обычно связаны с внешними службами или взаимодействуют с пользователями и непрерывно обрабатывают входящие события, такие как заказы, электронные письма или клики на веб-сайте. Когда происходит обработка события, приложение считывает или обновляет его состояние, выполняя транзакции с удаленной системой баз данных. Часто система баз

данных обслуживает несколько приложений, которые иногда обращаются к одним и тем же базам данных или таблицам.

Если со временем приложениям приходится развиваться или масштабироваться, такой подход может вызвать проблемы. Поскольку несколько приложений может работать с одним и тем же представлением данных или совместно использовать одну и ту же инфраструктуру, изменение схемы таблиц или масштабирование системы баз данных требует тщательного планирования и больших усилий. Новый подход к преодолению тесного связывания приложений – это идея проектирования *микросервисов*. Микросервисы представляют собой небольшие, автономные и независимые приложения. Они следуют философии UNIX: делать что-то одно и делать это хорошо. Более сложные приложения создаются соединением нескольких микросервисов, которые взаимодействуют друг с другом только через стандартизованные интерфейсы, такие как HTTP-соединения RESTful. Поскольку микросервисы строго отделены друг от друга и обмениваются данными только через четко определенные интерфейсы, каждый микросервис может быть реализован с использованием собственного технологического стека, включая язык программирования, библиотеки и хранилища данных. Микросервисы и все необходимое программное обеспечение и услуги обычно объединяются и развертываются в независимых контейнерах. На рис. 1.2 изображена архитектура микросервисов.

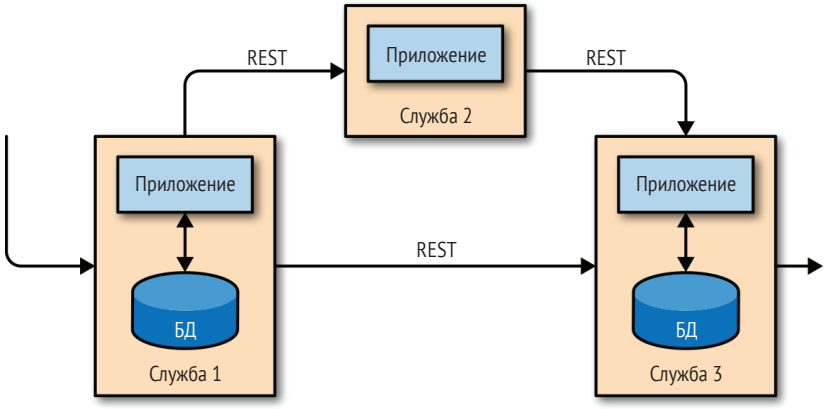

Рис. 1.2 ❖ Архитектура микросервисов

1.1.2. Аналитическая обработка

Данные, хранящиеся в различных системах транзакционных баз данных компании, могут дать ценную информацию о бизнес-операциях предприятия. Например, можно проанализировать данные системы обработки заказов, чтобы через какое-то время получить рост продаж, определить причины задержки отгрузки или спрогнозировать будущие продажи, чтобы скорректировать запасы. Однако транзакционные данные часто распределяются по

нескольким разрозненным системам баз данных и более полезны, когда их можно анализировать совместно. Более того, данные часто необходимо преобразовать в общий формат.

Вместо выполнения аналитических запросов непосредственно к транзакционным базам данных эти сырые данные обычно реплицируются в *хранилище данных* (data warehouse) – выделенную базу данных, оптимизированную под рабочие нагрузки аналитических запросов. Чтобы заполнить хранилище данных, в него необходимо скопировать данные, которыми оперируют системы транзакционных БД. Процесс копирования данных в хранилище состоит из трех стадий: *извлечение* – *преобразование* – *загрузка* (extract–transform–load, ETL). Процесс ETL извлекает данные из транзакционной БД, преобразует их в общее представление, которое может включать проверку, нормализацию значений, кодирование, удаление повторов и преобразование схемы, и наконец загружает их в аналитическую базу данных. Процессы ETL могут быть довольно сложными и часто требуют технически сложных решений для удовлетворения требований к производительности. Эти процессы необходимо запускать регулярно, чтобы синхронизировать данные в хранилище.

Как только данные окажутся в хранилище, их можно запросить и проанализировать. Обычно в хранилище данных выполняются два типа запросов. Первый тип – это запросы периодических отчетов, которые вычисляют релевантную для бизнеса статистику, такую как доход, рост числа пользователей или объем производства. Эти показатели объединяются в отчеты, которые помогают руководству оценить общее состояние бизнеса. Второй тип – это специальные запросы, которые предназначены для получения ответов на конкретные вопросы и поддержку важных для бизнеса решений, например, запрос на сбор данных о доходах и расходах на радиорекламу для оценки эффективности маркетинговой кампании. Оба типа запросов выполняются хранилищем данных в режиме *пакетной обработки* (batch processing), как показано на рис. 1.3.

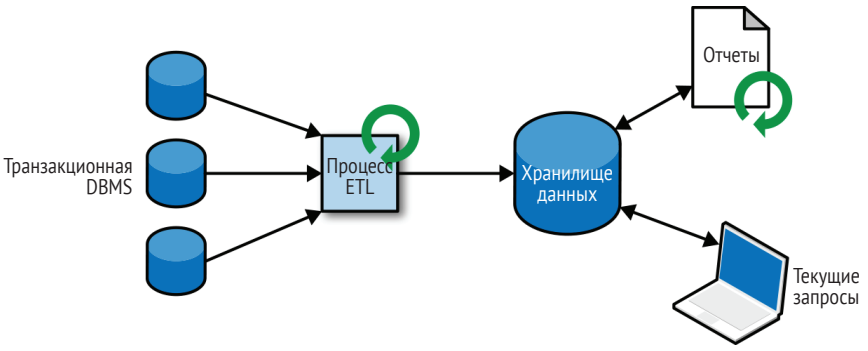

Рис. 1.3 ❖ Традиционная архитектура хранилища данных для аналитической обработки

Сегодня компоненты экосистемы Apache Hadoop являются неотъемлемыми частями информационных инфраструктур многих предприятий. Вме-

сто того, чтобы загружать все данные в систему реляционной базы данных, значительные объемы данных, такие как файлы журналов, социальные сети или журналы веб-кликов, записываются в распределенную файловую систему Hadoop (HDFS), S3 или другие хранилища массовых данных, например Apache HBase, которые обеспечивают огромную емкость хранилища по доступной цене. Данные, находящиеся в таких системах хранения, могут запрашиваться и обрабатываться механизмом SQL-on-Hadoop, например Apache Hive, Apache Drill или Apache Impala. Однако инфраструктура остается в основном такой же, как и в традиционной архитектуре хранилища данных.

1.2. Обработка потоков с учетом состояния

Практически все данные создаются как непрерывные *потоки событий*. Вспомните о взаимодействиях с пользователями на веб-сайтах или в мобильных приложениях, размещении заказов, журналах серверов или измерениях датчиков – все это потоки событий. На самом деле трудно найти примеры полных наборов данных конечного размера, которые были бы созданы сразу. Обработка потоков с учетом состояния – это подход к проектированию приложений для обработки неограниченно длинных потоков событий, который применим во множестве различных способов использования IT-инфраструктуры компании. Но прежде чем обсуждать варианты использования, мы кратко объясним, как работает потоковая обработка с учетом состояния.

Любое приложение, которое обрабатывает поток событий, а не просто выполняет тривиальные преобразования по одной записи за раз, должно учитывать состояния с возможностью хранения промежуточных данных и доступа к ним. Когда приложение получает событие, оно может выполнять произвольные вычисления, которые включают чтение или запись состояния. В принципе, состояние может быть сохранено и доступно в разных местах, включая программные переменные, локальные файлы, встроенные или внешние базы данных.

Apache Flink сохраняет состояние приложения в локальной памяти или во встроенной базе данных. Поскольку Flink является распределенной системой, локальное состояние необходимо защитить от сбоев, чтобы избежать потери данных в случае сбоя приложения или машины. Flink обеспечивает такую защиту, периодически записывая согласованную контрольную точку состояния приложения в удаленное и надежное хранилище. Состояние, согласованность состояний и механизм контрольных точек Flink будут обсуждаться более подробно в следующих главах, а пока на рис. 1.4 вы можете увидеть схему потокового приложения Flink с учетом состояния.

Приложения потоковой обработки с учетом состояния часто получают свои входящие события из *журнала событий* (event log). Журнал событий хранит и распределяет *потоки событий* (event stream). События записываются в долговременный журнал, доступный только для добавления, что означает невозможность изменить порядок событий. Поток, записанный в журнал событий, может быть прочитан много раз одним и тем же или разными по-

требителями. Благодаря тому, что журнал работает только на добавление, события всегда публикуются для всех потребителей в одном и том же порядке. Существует несколько систем журналов событий, доступных как программное обеспечение с открытым исходным кодом, наиболее популярным из которых является Apache Kafka, или как интегрированные услуги, предлагаемые поставщиками облачных вычислений.

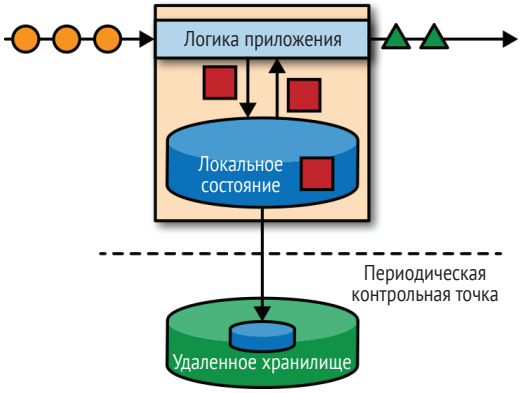

Рис. 1.4 ❖ Приложение для потоковой обработки с учетом состояния

Подключение потокового приложения, запущенного на Flink, к журналу событий интересно по нескольким причинам. В этой архитектуре журнал событий сохраняет входные события и может воспроизводить их в детерминированном порядке. В случае сбоя Flink восстанавливает потоковое приложение, извлекая его состояние из предыдущей контрольной точки и сбрасывая позицию чтения в журнале событий. Приложение будет воспроизводить (и быстро перематывать) входные события из журнала событий, пока не достигнет хвоста потока. Этот метод используется для восстановления после сбоев, но также может применяться для обновления приложения, исправления ошибок и исправления ранее выданных результатов, переноса приложения в другой кластер или выполнения A/B-тестов с разными версиями приложения.

Как мы говорили выше, потоковая обработка с учетом состояния – это универсальная и гибкая архитектура, которую можно применить во множестве различных сценариев. Далее мы представляем три класса приложений, которые обычно реализуются с использованием потоковой обработки с учетом состояния: (1) приложения, управляемые событиями, (2) приложения конвейера данных и (3) приложения для анализа данных.

Практические примеры использования и развертывания потоковой обработки

Если вы хотите узнать больше о реальных примерах использования и развертывания потоковой обработки, посетите страницу https://flink.apache.org/usecases.html, а также записи выступлений и слайд-шоу презентаций Flink Forward.

Мы разделили приложения на отдельные классы, чтобы подчеркнуть универсальность потоковой обработки с учетом состояния, но большинство реальных приложений обладает свойствами более чем одного класса.

1.2.1. Событийно-ориентированные приложения

Событийно-ориентированные приложения (event-driven application) – это потоковые приложения, которые принимают потоки событий и обрабатывают события с помощью бизнес-логики конкретного приложения. Далее для краткости мы будем называть их просто *событийными приложениями*. В зависимости от бизнес-логики событийное приложение может инициировать такие действия, как отправка предупреждения или электронного письма или запись событий в исходящий поток событий, которые будут использоваться другим событийным приложением.

Типичные варианты использования событийных приложений включают:

- рекомендации в режиме реального времени (например, для рекомендации продуктов, когда покупатели просматривают веб-сайт продавца);
- обнаружение шаблонов поведения или сложная обработка событий (например, для обнаружения мошенничества при транзакциях с кредитными картами);
- обнаружение аномалий (например, для обнаружения попыток проникновения в компьютерную сеть).

Событийные приложения – это эволюция микросервисов. Они обмениваются данными через журналы событий вместо вызовов REST и хранят данные приложения в виде *локального состояния* вместо обращения для чтения и записи ко внешнему хранилищу данных, такому как реляционная база данных или хранилище значений ключей. На рис. 1.5 показана архитектура сервиса, состоящая из событийных потоковых приложений.

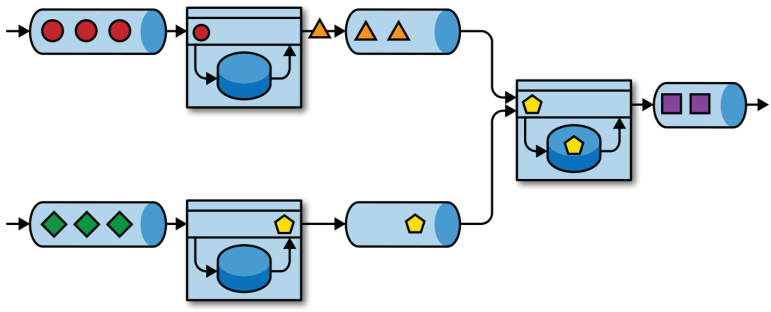

Рис. 1.5 ❖ Архитектура событийного приложения

Приложения на рис. 1.5 связаны журналами событий. Одно приложение отправляет свои выходные данные в журнал событий, а другое использует

события, созданные первым приложением. Журнал событий разделяет отправителей и получателей и обеспечивает *асинхронную неблокирующую передачу событий*. Каждое приложение может иметь учет состояния и локально управлять своим собственным состоянием без доступа к внешним хранилищам данных. Приложения также можно независимо обновлять и масштабировать.

Событийные приложения обладают несколькими преимуществами по сравнению с транзакционными приложениями или микросервисами. Доступ к локальному состоянию обеспечивает очень хорошую производительность по сравнению с запросами чтения и записи к удаленным хранилищам данных. Масштабирование и отказоустойчивость обеспечиваются потоковым процессором, а за счет использования журнала событий в качестве источника ввода полный набор входных данных приложения надежно сохраняется и может быть детерминированно воспроизведен. Кроме того, Flink может сбрасывать состояние приложения до предыдущей точки сохранения, что делает возможным развитие или масштабирование приложения без потери его состояния.

Событийные приложения предъявляют довольно высокие требования к потоковому процессору, который их запускает. Не все потоковые процессоры одинаково хорошо подходят для запуска таких приложений. Выразительная возможность[1] API, качество обработки состояний и привязка ко времени событий определяют бизнес-логику, которую можно реализовать. Этот аспект зависит от API потокового процессора, от того, какие типы примитивов состояния он поддерживает, и от качества обработки критических во времени событий. Более того, гарантия согласованности состояния «ровно один раз» и возможность масштабирования приложения являются фундаментальными требованиями для событийных приложений. Apache Flink удовлетворяет всем этим требованиям и является очень хорошим выбором для запуска приложений этого класса.

1.2.2. Конвейеры данных

Сегодняшние IT-архитектуры включают в себя множество различных хранилищ данных, таких как системы реляционных и специализированных баз данных, журналы событий, распределенные файловые системы, кеши в памяти и индексы поиска. Все эти системы хранят данные в разных форматах и структурах данных, которые обеспечивают наилучшее быстродействие для их конкретного шаблона доступа. Часто компании хранят одни и те же данные в нескольких разных системах, чтобы повысить скорость доступа к данным. Например, информация о продукте, который предлагается в интернет-магазине, может храниться в транзакционной базе данных, веб-кеше и поисковом индексе. Из-за подобной репликации данных хранилища должны быть синхронизированы.

---

[1] Выразительная возможность – мера готовности API или языка программирования реализовать идеи и потребности пользователя. – *Прим. перев.*

Традиционный подход к синхронизации данных в различных системах хранения – периодический запуск заданий ETL. Однако они не соответствуют требованиям к минимальной задержке синхронизации во многих современных сценариях использования данных. Альтернативой является использование журнала событий для распространения обновлений. Обновления записываются и распространяются в журнале событий. Пользователи журнала сами вносят обновления в затронутые хранилища данных. В зависимости от варианта использования переданные данные, возможно, потребуется нормализовать, дополнить внешними данными или агрегировать перед их записью в целевое хранилище данных.

Получение, преобразование и запись данных с малой задержкой синхронизации – еще один распространенный вариант использования приложений потоковой обработки с учетом состояния. Этот тип приложения называется *конвейером данных*. Конвейеры данных должны иметь возможность обрабатывать большие объемы данных за короткое время. Потоковый процессор, который управляет конвейером данных, также должен иметь широкий выбор соединителей (переходных интерфейсов) источника и приемника для чтения и записи данных в различные системы хранения. Опять же, все эти задачи решает Flink.

1.2.3. Потоковая аналитика

Задания ETL периодически импортируют данные в хранилище данных, а затем эти данные обрабатываются индивидуальными или регулярными запросами. В любом случае это пакетная обработка независимо от того, основана ли архитектура на хранилище данных или на компонентах экосистемы Hadoop. Хотя периодическая загрузка данных в систему анализа данных уже много лет является общепринятым подходом, она значительно увеличивает задержку в конвейере аналитики.

В зависимости от интервалов запуска заданий ETL может пройти несколько часов или дней, прежде чем точка данных будет включена в отчет. В некоторой степени задержку можно уменьшить, импортируя данные в хранилище с помощью приложения конвейера данных. Однако даже при непрерывном запуске ETL всегда будет существовать задержка до обработки события запросом. Хотя в прошлом такая задержка могла быть приемлемой, современные приложения должны иметь возможность собирать данные в режиме реального времени и немедленно реагировать на них (например, приспосабливаясь к изменяющимся условиям в мобильной игре или персонализируя взаимодействие с новым пользователем в интернет-магазине).

Вместо того чтобы бездействовать от запуска до запуска, приложение потоковой аналитики постоянно принимает потоки событий и обновляет выходные данные, добавляя в них последние события с минимальной задержкой. Это похоже на методы, которые применяются в системах баз данных для обновления материализованных представлений. Как правило, потоковые приложения сохраняют свой результат во внешнем хранилище данных, которое поддерживает быстрые обновления, например в базе данных или

хранилище пар «ключ–значение». Обновляемые в реальном времени результаты приложения потоковой аналитики можно использовать в работе приложений панели мониторинга, как показано на рис. 1.6.

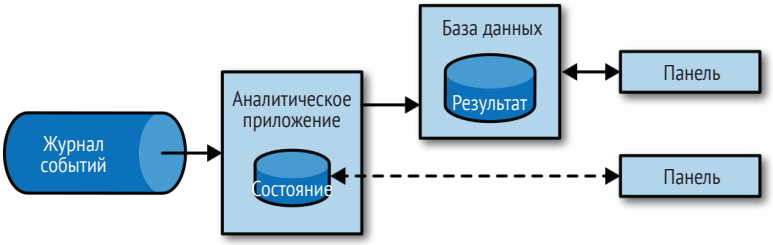

Рис. 1.6 ❖ Приложение потоковой аналитики

Помимо гораздо более короткого времени, необходимого для включения события в результат аналитики, есть еще одно, менее очевидное преимущество приложений потоковой аналитики. Традиционные конвейеры аналитики состоят из нескольких отдельных компонентов, таких как процесс ETL, система хранения, и – в случае среды на основе Hadoop – обработчика данных и планировщика для запуска заданий или запросов. Напротив, потоковый процессор, который запускает потоковое приложение с учетом состояния, берет на себя все эти этапы обработки, включая прием событий, непрерывные вычисления с поддержкой состояния и обновление результатов. Более того, потоковый процессор может восстанавливаться после сбоев с гарантией согласованности состояния «ровно один раз» и способен управлять вычислительными ресурсами приложения. Потоковые процессоры, такие как Flink, также поддерживают обработку критических по времени событий для получения правильных и детерминированных результатов и способны обрабатывать большие объемы данных за короткое время.

Приложения потоковой аналитики обычно используются для следующих целей:

- мониторинг качества сотовых сетей;
- анализ поведения пользователей в мобильных приложениях;
- специальный анализ данных в реальном времени в потребительских технологиях.

Хотя мы не рассматриваем здесь этот аспект, Flink также обеспечивает поддержку потоковых аналитических SQL-запросов.

1.3. Эволюция потоковой обработки с открытым исходным кодом

Потоковая обработка данных не новая технология. Некоторые из первых исследовательских прототипов и коммерческих продуктов относятся к кон-

цу 1990-х годов. Однако растущее распространение технологии потоковой обработки в недавнем прошлом в значительной степени было обусловлено появлением достаточно развитых потоковых процессоров с открытым исходным кодом. Сегодня распределенные потоковые процессоры с открытым исходным кодом обеспечивают работу критически важных бизнес-приложений на многих предприятиях в различных отраслях, таких как розничная торговля (включая онлайн-торговлю), социальные сети, телекоммуникации, игры и банковское дело. Программное обеспечение с открытым исходным кодом является движущей силой этой тенденции в основном по двум причинам:

1) программное обеспечение для потоковой обработки с открытым исходным кодом – это ресурс, который может оценить и использовать каждый;
2) технология масштабируемой потоковой обработки быстро развивается и совершенствуется благодаря усилиям множества сообществ открытого исходного кода.

Один лишь фонд Apache Software Foundation поддерживает в своем инвестиционном инкубаторе более десятка проектов, связанных с потоковой обработкой. Новые проекты распределенной потоковой обработки постоянно переходят на стадию открытого исходного кода и бросают вызов современным технологиям, предлагая новые функции и возможности. Сообщества с открытым исходным кодом постоянно улучшают возможности своих проектов и расширяют технические границы потоковой обработки. Мы кратко заглянем в прошлое, чтобы увидеть, откуда пришла потоковая обработка с открытым исходным кодом и где она находится сегодня.

1.3.1. Немного истории

Первое поколение распределенных потоковых процессоров с открытым исходным кодом (2011 г.) было ориентировано на обработку событий с миллисекундными задержками и предоставляло гарантии от потери событий в случае сбоев. Эти системы имели довольно низкоуровневые API-интерфейсы и не обладали встроенной поддержкой точности и целостности результатов потоковых приложений, поскольку результаты зависели от времени и порядка поступления событий. Более того, даже если события не были потеряны, их можно было обрабатывать более одного раза. В отличие от пакетных процессоров первые потоковые процессоры с открытым исходным кодом жертвовали точностью результатов ради меньшей задержки. Ситуация, когда системы обработки данных (на тот момент) могли предоставлять либо быстрые, либо точные результаты, привела к разработке так называемой *лямбда-архитектуры*, которая изображена на рис. 1.7.

Лямбда-архитектура дополняет традиционную архитектуру периодической пакетной обработки за счет высокой скорости, которую обеспечивает потоковый процессор с малой задержкой. Данные, поступающие в лямбда-архитектуру, принимаются потоковым процессором, а также записываются

в пакетное хранилище. Потоковый процессор вычисляет приблизительные результаты почти в реальном времени и записывает их в «быструю» таблицу. Пакетный процессор периодически обрабатывает данные в пакетном хранилище, записывает точные результаты в «пакетную» таблицу и удаляет соответствующие неточные результаты из быстрой таблицы. Приложения используют совокупные результаты обработки, объединяя приблизительные результаты из быстрой таблицы и точные результаты из пакетной таблицы.

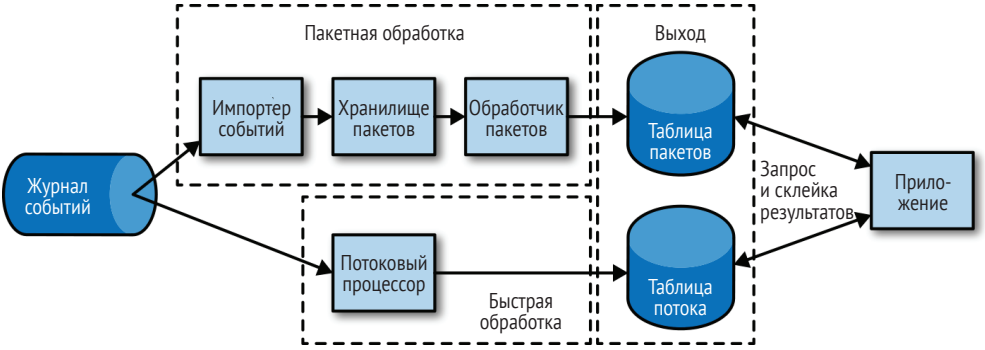

Рис. 1.7 ❖ Лямбда-архитектура

Лямбда-архитектура фактически уже устарела, но все еще используется во многих местах. Первоначальной целью этой архитектуры было уменьшение задержки выдачи результатов, свойственной исходной архитектуре пакетной аналитики. Однако у нее есть несколько заметных недостатков. Прежде всего она требует двух семантически эквивалентных реализаций логики приложения для двух отдельных систем обработки с разными API. Во-вторых, результаты, полученные потоковым процессором, являются приблизительными. В-третьих, лямбда-архитектуру сложно настраивать и поддерживать.

Следующее поколение распределенных потоковых процессоров с открытым исходным кодом (2013 г.), усовершенствованное по сравнению с первым поколением, обеспечивало более высокую отказоустойчивость и гарантировало, что в случае сбоя каждая входная запись влияет на результат только один раз. Кроме того, программные интерфейсы эволюционировали от низкоуровневых интерфейсов на основе операторов к высокоуровневым API с большим количеством встроенных примитивов. Однако некоторые улучшения, такие как более высокая пропускная способность и отказоустойчивость, были достигнуты за счет увеличения задержек обработки с миллисекунд до секунд. Более того, результаты все еще зависели от времени и порядка наступления событий.

Третье поколение распределенных потоковых процессоров с открытым исходным кодом (2015 г.) решило проблему зависимости результатов от времени и порядка поступления событий. Системы этого поколения стали первыми потоковыми процессорами с открытым исходным кодом, способными вычислять согласованные и точные результаты в сочетании с семантикой «ровно один раз». Вычисляя результаты только на основе фактических дан-

ных, эти системы могут обрабатывать исторические данные точно так же, как «живые» данные. Другим усовершенствованием стало устранение компромисса между задержкой и пропускной способностью. В то время как предыдущие потоковые процессоры обеспечивали либо высокую пропускную способность, либо низкую задержку, системы третьего поколения способны обеспечить и то и другое. Потоковые процессоры этого поколения окончательно сделали лямбда-архитектуру устаревшей.

В дополнение к давно учитываемым параметрам, таким как отказоустойчивость, быстродействие и точность результатов, потоковые процессоры также постоянно добавляли новые возможности, например высокую доступность, тесную интеграцию с диспетчерами ресурсов, такими как YARN или Kubernetes, и возможность динамического масштабирования потоковых приложений. К дополнительным, но важным функциям можно отнести поддержку обновления кода приложения или переноса задания в другой кластер или на новую версию потокового процессора без потери текущего состояния.

1.4. Обзорное знакомство с Flink

Apache Flink – это распределенный потоковый процессор третьего поколения с конкурентоспособным набором функций. Он обеспечивает точную потоковую обработку с высокой пропускной способностью и малой задержкой в масштабных системах. В частности, Flink обладает следующими характерными особенностями:

- поддержкой семантики времени события и времени обработки. Семантика времени события обеспечивает последовательные и точные результаты, несмотря на неупорядоченные события. Семантика времени обработки может использоваться для приложений с очень низкими требованиями к задержке;
- гарантией обработки «ровно один раз»;
- миллисекундными задержками при обработке миллионов событий в секунду. Приложения Flink можно масштабировать для работы на тысячах ядер;
- многоуровневыми API с различными компромиссами между выразительностью и простотой использования. В этой книге описывается API DataStream и функции обработки, которые предоставляют примитивы для обычных операций обработки потоков, таких как оконные и асинхронные операции, а также интерфейсы для точного управления состоянием и временем. Реляционные API Flink, SQL и API таблиц в стиле LINQ в этой книге не обсуждаются;
- соединителями с наиболее распространенными системами хранения, такими как Apache Kafka, Apache Cassandra, Elasticsearch, JDBC, Kinesis, и распределенными файловыми системами, такими как HDFS и S3;
- возможностью запускать потоковые приложения в режиме 24/7 с минимальным временем простоя благодаря высокой отказоустойчивости Flink (без единой точки отказа), тесной интеграции с Kubernetes, YARN

и Apache Mesos, быстрому восстановлению после сбоев и возможности динамического масштабирования заданий;
- возможностью обновлять код заданий приложения и переносить задания в разные кластеры Flink без потери состояния приложения;
- подробным и настраиваемым набором показателей системы и приложений для заблаговременного выявления проблем и реагирования на них;
- и последнее, но не менее важное: Flink также является полноценным пакетным процессором[1].

В дополнение к этим возможностям Flink – очень удобный для разработчиков фреймворк благодаря простым в использовании API. Режим встроенного выполнения запускает приложение и всю систему Flink в одном процессе JVM, который можно использовать для запуска и отладки заданий Flink в среде IDE. Эта возможность пригодится при разработке и тестировании приложений Flink.

1.4.1. Запуск вашего первого приложения Flink

Далее мы пройдем с вами через процесс запуска локального кластера и выполнения потокового приложения, чтобы вы впервые познакомились с Flink. Приложение, которое мы собираемся запустить, конвертирует и усредняет случайно сгенерированные показания датчиков температуры по времени. Для работы с этим примером в вашей системе должна быть установлена Java 8. Мы описываем шаги для среды UNIX, но, если вы работаете с Windows, мы рекомендуем настроить виртуальную машину с Linux – Cygwin (среда Linux для Windows) или подсистемой Windows для Linux, представленной в Windows 10. Следующие шаги демонстрируют, как запустить локальный кластер Flink и отправить заявку на выполнение.

1. Перейдите на веб-страницу Apache Flink и загрузите бинарный дистрибутив Apache Flink 1.7.1 для Scala 2.12 без поддержки Hadoop.
2. Распакуйте архивный файл:

```
$ tar xvfz flink-1.7.1-bin-scala_2.12.tgz
```

3. Запустите локальный кластер Flink:

```
$ cd flink-1.7.1
$ ./bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host xxx.
Starting taskexecutor daemon on host xxx.
```

[1] API пакетной обработки Flink, API DataSet и его операторы отделены от их соответствующих потоковых аналогов. Однако подход сообщества Flink состоит в том, чтобы рассматривать пакетную обработку как частный случай потоковой обработки – обработку ограниченных потоков. Постоянные усилия сообщества Flink направлены на то, чтобы превратить Flink в систему с действительно унифицированным пакетным и потоковым API и средой выполнения.

4. Откройте веб-интерфейс Flink, введя URL-адрес **http://localhost:8081** в своем браузере. Как показано на рис. 1.8, вы увидите статистику только что запущенного локального кластера Flink. Она говорит о том, что подключен один TaskManager (процесс-воркер Flink) и что доступен один слот задачи (единица ресурсов, которые предоставляет TaskManager).

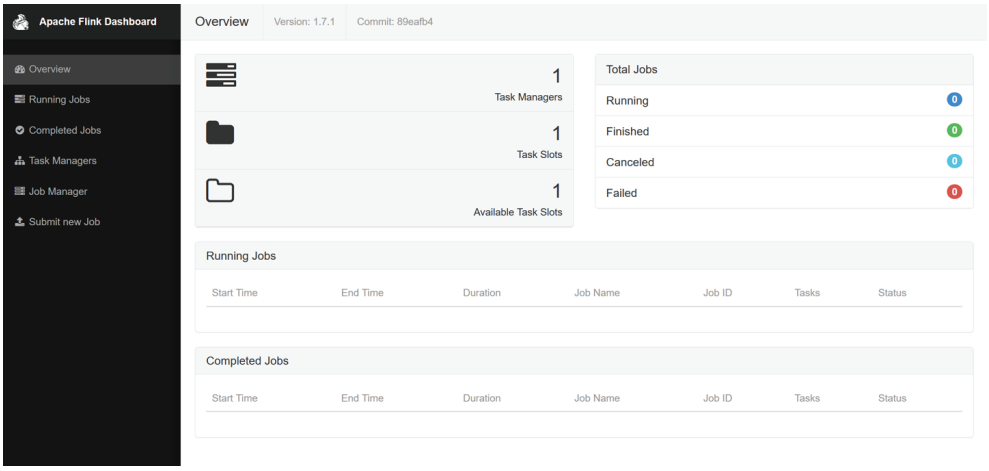

Рис. 1.8 ❖ Снимок экрана веб-панели управления Apache Flink

5. Загрузите файл JAR, содержащий примеры из этой книги:

```
$ wget https://streaming-with-flink.github.io/examples/download/examples-scala.jar
```

Вы также можете создать файл JAR самостоятельно, выполнив действия, описанные в файле README репозитория.

6. Запустите пример на локальном кластере, указав класс записи приложения и файл JAR:

```
$ ./bin/flink run \
  -c io.github.streamingwithflink.chapter1.AverageSensorReadings \
  examples-scala.jar
Starting execution of program
Job has been submitted with JobID cfde9dbe315ce162444c475a08cf93d9
```

7. Посмотрите на веб-панель. Вы должны увидеть задание в разделе **Running Jobs** (Выполняемые задания). Если вы нажмете на это задание, вы увидите поток данных и текущие параметры операторов выполняемого задания, как показано на снимке экрана на рис. 1.9.
8. Вывод задания записывается в стандартный рабочий процесс Flink, который по умолчанию перенаправляется в файл в папке `./log`. Вы

можете отслеживать состояние вывода, используя команду **`tail`** следующим образом:

```
$ tail -f ./log/flink-<user>-taskexecutor-<n>-<hostname>.out
```

Вы должны увидеть, как в файл записываются строки наподобие следующих:

```
SensorReading(sensor_1,1547718199000,35.80018327300259)
SensorReading(sensor_6,1547718199000,15.402984393403084)
SensorReading(sensor_7,1547718199000,6.720945201171228)
SensorReading(sensor_10,1547718199000,38.101067604893444)
```

Первое поле `SensorReading` – это `sensorId`, второе – это отметка времени в миллисекундах с 1970-01-01-00:00:00.000, а третье – это средняя температура, вычисленная за 5 с.

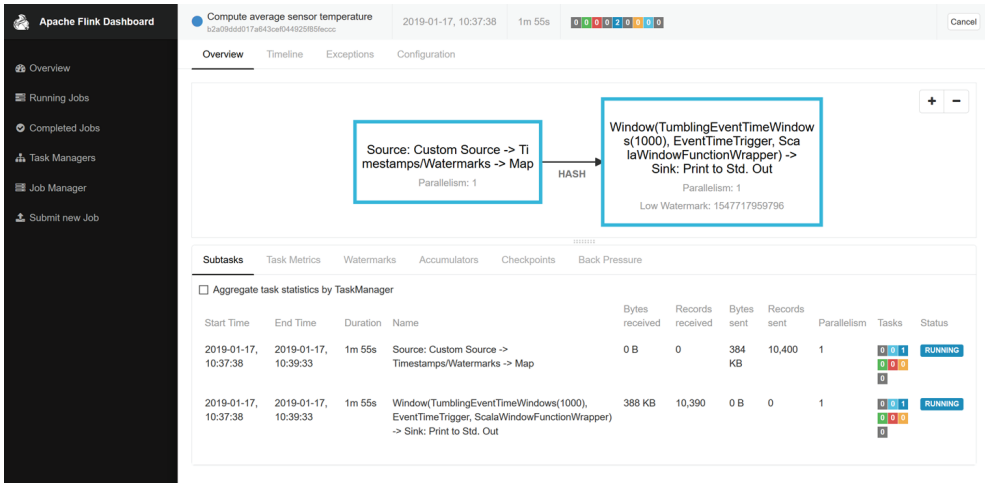

Рис. 1.9 ❖ Снимок экрана веб-панели Apache Flink, показывающий выполняемую задачу

9. Поскольку у вас запущено потоковое приложение, оно будет работать до тех пор, пока вы его не отмените. Вы можете сделать это, выбрав задание на веб-панели управления и нажав кнопку **Отмена** вверху страницы.
10. Наконец, вы должны остановить локальный кластер Flink:

```
$ ./bin/stop-cluster.sh
```

Да, это так просто. Вы только что установили и запустили свой первый локальный кластер Flink и выполнили свою первую программу Flink с API DataStream! Конечно, вам предстоит еще многое узнать о потоковой обработке с помощью Apache Flink, и именно об этом рассказывает наша книга.

1.5. Заключение

В этой главе вы познакомились с потоковой обработкой с учетом состояния, обсудили варианты ее использования и впервые встретились с Apache Flink. Мы начали с обзора традиционных инфраструктур данных – того, как обычно проектируются бизнес-приложения и как данные собираются и анализируются сегодня в большинстве компаний. Затем мы представили идею потоковой обработки с учетом состояния и объяснили, как она работает в широком спектре сценариев использования, от бизнес-приложений и микросервисов до ETL и анализа данных. Мы рассказали, как развивались системы потоковой обработки с открытым исходным кодом с момента их создания в начале 2010-х гг. и как потоковая обработка стала жизнеспособным решением для многих задач современных предприятий. Наконец, мы рассмотрели Apache Flink и его обширный набор функций, а также показали, как установить локальный кластер Flink и запустить первое приложение для обработки потока данных.

Глава 2

Основы потоковой обработки

Вы уже знаете, как потоковая обработка устраняет некоторые ограничения традиционной пакетной обработки и как она помогать создавать новые приложения и архитектуры. Вы также немного знаете об эволюции приложений потоковой обработки с открытым исходным кодом и о том, как выглядит потоковое приложение Flink. В этой главе вы окончательно войдете в мир потоковой обработки.

Цель этой главы – познакомить вас с фундаментальными концепциями потоковой обработки и требованиями к фреймворкам. Надеемся, что, прочитав эту главу, вы сможете оценить возможности современных систем обработки потоковых данных.

2.1. Введение в потоковое программирование

Прежде чем мы углубимся в основы потоковой обработки, давайте рассмотрим предысторию потокового программирования и освоим терминологию, которую мы будем использовать в этой книге.

2.1.1. Графы потока данных

Как следует из ее названия, *потоковая программа* (dataflow program) описывает, как потоки данных перемещаются между операциями. Потоковые программы обычно представлены в виде ориентированных графов, где узлы называются *операторами* и представляют вычисления, а ребра представляют *зависимости* данных. Операторы – это основные функциональные единицы потокового приложения. Они получают данные из входов, выполняют над ними вычисления и выдают данные на выходы для дальнейшей обработки. Операторы без портов ввода называются *источниками данных* (data source),

а операторы без портов вывода – *приемниками*[1] *данных* (data sink). Граф потока данных должен иметь по крайней мере один источник данных и один приемник данных. На рис. 2.1 показана потоковая программа, которая извлекает и подсчитывает хештеги из входного потока твитов.

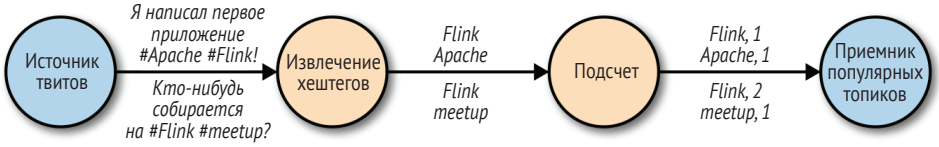

Рис. 2.1 ❖ Логический граф потока данных для непрерывного подсчета хештегов (узлы представляют операторы, а ребра обозначают зависимости данных)

Графы потоков данных, подобные изображенному на рис. 2.1, называются *логическими*, поскольку они отражают высокоуровневое представление логики вычислений. Для выполнения потоковой программы ее логический граф преобразуется в граф *физического* потока данных, который подробно определяет, как выполняется программа. Например, если мы используем механизм распределенной обработки, у каждого оператора может быть несколько параллельных задач, выполняемых на разных физических машинах. На рис. 2.2 показан граф физического потока данных для логического графа на рис. 2.1. В то время как в логическом графе потока данных узлы представляют операторы, в физическом потоке данных узлы представляют собой задачи. Операторы «Извлечь хештеги» и «Счетчик» имеют по две параллельные задачи, каждая из которых выполняет вычисление над подмножеством входных данных.

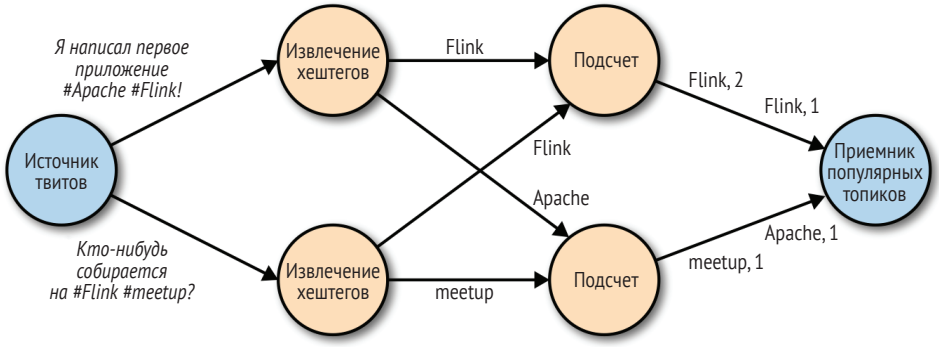

Рис. 2.2 ❖ Схема физического потока данных для подсчета хештегов (узлы представляют задачи)

[1] В контексте потоковой обработки с переводом этого термина возникает небольшая путаница, потому что data sink стоит в конце логического потока данных, то есть на *выходе*, и называть его приемником кажется слегка нелогичным. Но если представить data sink как корзинку, в которую с конвейера падают готовые изделия, становится ясно, что это действительно приемник. – *Прим. перев.*

2.1.2. Параллелизм данных и параллелизм задач

Вы можете по-разному использовать параллелизм в графах потоков данных. Во-первых, вы можете разделить свои входные данные и параллельно выполнять задачи одной и той же операции над подмножествами данных. Этот тип параллелизма называется *параллелизмом данных*. Параллелизм данных полезен, поскольку он позволяет обрабатывать большие объемы данных и распределять вычислительную нагрузку между несколькими вычислительными узлами. С другой стороны, у вас могут быть задачи от разных операторов, параллельно выполняющих вычисления с одними и теми же или разными данными. Этот тип параллелизма называется *параллелизмом задач*. Используя параллелизм задач, вы можете эффективнее использовать вычислительные ресурсы кластера.

2.1.3. Стратегии обмена данными

Стратегии обмена данными определяют, как элементы данных назначаются задачам в графе физического потока данных. Стратегии обмена данными могут быть автоматически выбраны механизмом выполнения в зависимости от семантики операторов или явно назначены программистом потоковых приложений. Здесь мы кратко рассмотрим некоторые общие стратегии обмена данными, показанные на рис. 2.3.

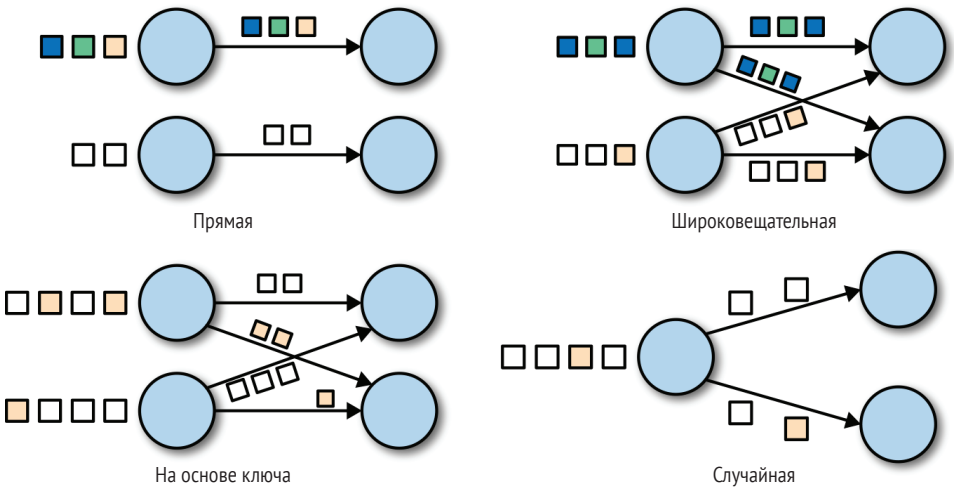

Рис. 2.3 ❖ Стратегии обмена данными

- *Стратегия передачи* отправляет данные от задачи к задаче-получателю. Если обе задачи расположены на одной физической машине (что часто обеспечивается планировщиками задач), эта стратегия обмена позволяет избежать сетевого взаимодействия.

- *Стратегия широковещания* отправляет каждый элемент данных всем параллельным задачам оператора. Поскольку эта стратегия реплицирует данные и включает сетевое взаимодействие, она обходится довольно дорого.
- *Стратегия на основе ключей* разделяет данные по ключевому атрибуту и гарантирует, что элементы данных с одинаковым ключом будут обрабатываться одной и той же задачей. На рис. 2.2 выходные данные оператора «Извлечь хештеги» разделены по ключу (хештегу), так что задачи оператора счетчика могут правильно вычислить вхождения каждого хештега.
- *Стратегия случайного выбора* равномерно распределяет элементы данных по задачам оператора, чтобы равномерно распределять нагрузку по вычислительным задачам.

2.2. Параллельная обработка потоков

Теперь, когда вы знакомы с основами потокового программирования, пора узнать, как эти концепции применяются к параллельной обработке потоков данных. Но сначала давайте определим главный термин нашей книги: *поток данных* – это потенциально неограниченная последовательность событий.

События в потоке данных могут представлять данные мониторинга, измерения датчиков, транзакции по кредитным картам, наблюдения метеостанции, интерактивные взаимодействия пользователей, поиск в интернете и т. д. В этом разделе вы узнаете, как параллельно обрабатывать бесконечные потоки, используя парадигму потокового программирования.

2.2.1. Задержка и пропускная способность

В главе 1 вы узнали, что к потоковым приложениям предъявляют иные операционные требования, чем к традиционным пакетным программам. Требования также различаются, когда дело доходит до оценки быстродействия. В случае пакетных приложений мы обычно заботимся об общем времени выполнения задачи или о том, сколько времени требуется нашему механизму обработки, чтобы прочитать ввод, выполнить вычисление и записать результат. Поскольку потоковые приложения работают непрерывно и вход потенциально бесконечен, при обработке потока данных нет понятия общего времени выполнения. Вместо этого потоковые приложения должны предоставлять результаты для входящих данных *как можно быстрее*, сохраняя при этом возможность принимать новые события, поступающие с высокой скоростью. Мы выражаем эти требования к производительности в терминах задержки и пропускной способности.

2.2.1.1. Задержка

Задержка (latency) указывает, сколько времени требуется для обработки события. По сути, это временной интервал между получением события и по-

явлением эффекта обработки этого события в выходных данных. Чтобы интуитивно понять, что такое задержка, представьте ежедневное посещение любимой кофейни. Когда вы входите в кофейню, внутри уже могут быть другие покупатели. Итак, вы стоите в очереди, а когда наступает ваша очередь, вы делаете заказ. Кассир принимает оплату и передает ваш заказ бариста, который готовит вам напиток. Когда ваш кофе будет готов, бариста называет ваше имя, и вы можете забрать свой кофе со стойки. Задержка обслуживания – это время, которое вы проводите в кафе с момента входа до первого глотка кофе.

При потоковой передаче данных задержка измеряется в единицах времени, например в миллисекундах. В зависимости от приложения вас может интересовать *средняя* задержка, *максимальная* задержка или *процентильная* задержка. Например, среднее значение задержки 10 мс означает, что события обрабатываются в среднем в течение 10 мс. С другой стороны, значение 95-процентильной задержки в 10 мс означает, что 95 % событий обрабатываются в течение 10 мс. Средние значения скрывают истинное распределение задержек обработки и могут затруднить обнаружение проблем. Если у бариста заканчивается молоко прямо перед приготовлением капучино, вам придется подождать, пока его принесут из кладовой. Хотя эта задержка может вас раздражать, большинство других клиентов останутся довольны.

Обеспечение низкой задержки критически важно для многих потоковых приложений, таких как обнаружение мошенничества, системные предупреждения, мониторинг сети и предоставление услуг со строгими соглашениями об уровне обслуживания. Низкая задержка – ключевая характеристика потоковой обработки, она позволяет работать с приложениями *реального времени*. Современные потоковые процессоры, такие как Apache Flink, могут предлагать задержки до нескольких миллисекунд. Напротив, традиционные задержки при пакетной обработке обычно составляют от нескольких минут до нескольких часов. При пакетной обработке вам сначала нужно собрать события в пакеты, и только затем вы можете их обработать. Таким образом, задержка ограничена временем прибытия последнего события в каждом пакете и, естественно, зависит от размера пакета. Правильная потоковая обработка не нуждается в подобных искусственных задержках на входе и поэтому может обеспечить действительно низкие задержки на выходе. В потоковой модели события могут обрабатываться, как только они поступают в систему, а задержка более точно отражает фактическую работу, которую необходимо выполнить для каждого события.

2.2.1.2. Пропускная способность

Пропускная способность (throughput) – это мера вычислительной мощности системы. То есть пропускная способность говорит нам, сколько событий система может обработать за единицу времени. Вернемся к примеру с кофейней: если она открыта с 7 утра до 7 вечера и способна обслужить 600 клиентов за один день, тогда ее средняя пропускная способность составит 50 клиентов в час. Обычно мы хотим, чтобы задержка была как можно меньше, а пропускная способность была как можно больше.

Пропускная способность измеряется в событиях или операциях в единицу времени. Важно отметить, что скорость обработки зависит от скорости поступления данных; низкая пропускная способность не обязательно указывает на плохую производительность. Как владелец или клиент потоковой системы, вы обычно хотите иметь уверенность, что ваша система способна обработать максимально ожидаемую частоту событий. То есть вас в первую очередь интересует определение *пиковой пропускной способности* – предела производительности, когда ваша система работает с максимальной нагрузкой. Чтобы лучше понять концепцию пиковой пропускной способности, давайте рассмотрим приложение потоковой обработки, которое не принимает никаких входящих данных и, следовательно, не потребляет никаких системных ресурсов. Когда приходит первое событие, оно будет немедленно обработано с минимально возможной задержкой. Например, если вы первый клиент, пришедший в кофейню сразу после того, как она открыла свои двери утром, вас сразу обслужат. В идеале вы хотели бы, чтобы эта задержка оставалась постоянной и независимой от скорости входящих событий. Однако, как только мы достигнем такой скорости входящих событий, при которой системные ресурсы будут полностью исчерпаны, нам придется начать буферизацию событий. В примере с кофейней вы, вероятно, увидите, что это происходит сразу после обеда. Многие люди приходят одновременно и вынуждены стоять в очереди. На этот момент система достигла максимальной пропускной способности, и дальнейшее увеличение частоты событий приведет только к ухудшению задержки. Если система продолжает получать данные с большей скоростью, чем она может обработать, буферы могут переполниться, что приведет к потере данных. Эта проблема известна как *противодавление* (backpressure), и для ее решения существуют разные стратегии.

2.2.1.3. Как связаны задержка и пропускная способность?

На этом этапе вам должно быть ясно, что задержка и пропускная способность не являются независимыми показателями. Если событиям требуется много времени для прохождения по конвейеру обработки данных, мы не сможем обеспечить высокую пропускную способность. Точно так же, если емкость системы мала, события буферизируются и должны ждать, прежде чем они будут обработаны.

Вернемся к примеру с кофейней, чтобы выяснить, как задержка и пропускная способность влияют друг на друга. Во-первых, должно быть ясно, что существует оптимальная задержка при отсутствии нагрузки. То есть вы получите самое быстрое обслуживание, если будете единственным покупателем в кофейне. Однако в часы пик клиентам придется ждать в очереди, и задержка будет увеличиваться. Другим фактором, влияющим на задержку и, следовательно, на пропускную способность, является время, необходимое для обработки события, или время, необходимое для обслуживания каждого клиента в кофейне. Представьте, что во время рождественских праздников бариста должны нарисовать Санта-Клауса на каждой чашке кофе, который они подают. Это означает, что время, необходимое для приготовления одно-

го напитка, увеличится, в результате чего каждый человек будет проводить больше времени в очереди, что снизит общую пропускную способность.

Итак, можно ли получить одновременно низкую задержку и высокую пропускную способность или это безнадежная попытка? Вы можете уменьшить задержку в нашей кофейне, наняв более опытного бариста – того, кто быстрее готовит кофе. При высокой нагрузке это изменение также увеличит пропускную способность, потому что за то же время будет обслужено больше клиентов. Другой способ добиться того же результата – нанять второго бариста и использовать параллелизм. Главный вывод здесь заключается в том, что уменьшение задержки увеличивает пропускную способность. Естественно, если система может выполнять операции быстрее, она будет выполнять больше операций за то же время. Фактически именно это и происходит, когда вы используете параллелизм в конвейере потоковой обработки. Обрабатывая несколько потоков параллельно, вы уменьшаете задержку при одновременной обработке большего количества событий.

2.2.2. Операции с потоками данных

Механизмы потоковой обработки обычно предоставляют набор встроенных операций для приема, преобразования и вывода потоков. Эти операторы могут быть объединены в графы обработки потоков данных для реализации логики потоковых приложений. В этом разделе мы расскажем про наиболее распространенные операции потоковой обработки.

Потоковые операции можно разделить на два типа: *без учета состояния* (stateless) и *с учетом состояния* (stateful). Операции без учета состояния не поддерживают никакого внутреннего состояния. То есть обработка события не зависит от каких-либо событий, наблюдаемых в прошлом, и история не сохраняется. Операции без учета состояния легко распараллелить, поскольку события могут обрабатываться независимо друг от друга и от порядка их поступления. Более того, в случае сбоя оператор, который не учитывает состояние, можно просто перезапустить и продолжить обработку с того места, где он был остановлен. Напротив, операторы с учетом состояния могут сохранять информацию о событиях, которые они получили ранее. Это состояние может обновляться входящими событиями и впоследствии использоваться в логике обработки следующих событий. Для потоковых приложений с учетом состояния труднее обеспечивать параллелизм и отказоустойчивость, потому что состояние должно быть эффективно разделено и надежно восстановлено в случае сбоев. Вы узнаете больше о потоковой обработке с отслеживанием состояния, сценариях сбоев и согласованности в конце этой главы.

2.2.2.1. Получение и выдача данных

Операции получения и выдачи данных позволяют потоковому процессору взаимодействовать с внешними системами. *Получение данных* (data ingestion) – это операция получения необработанных данных из внешних ис-

точников и их преобразования в формат, подходящий для обработки. Операторы, реализующие логику получения (извлечения) данных, называются *источниками данных*. Источник данных может извлекать данные из TCP-сокета, файла, топика Kafka или интерфейса данных датчика. *Выгрузка данных* (data egress) – это операция по созданию вывода в форме, пригодной для потребления внешними системами. Операторы, выполняющие выгрузку данных, называются *приемниками данных* (data sink) и включают файлы, базы данных, очереди сообщений и интерфейсы мониторинга.

2.2.2.2. Операции преобразования

Операции преобразования (transformation operation) – это однопроходные операции, которые обрабатывают каждое событие независимо. Эти операции используют одно событие за другим и применяют некоторые преобразования к данным события, создавая новый выходной поток. Логика преобразования может быть либо интегрирована в оператор, либо предоставлена пользовательской функцией, как показано на рис. 2.4. Функции пишутся прикладным программистом и реализуют настраиваемую логику вычислений.

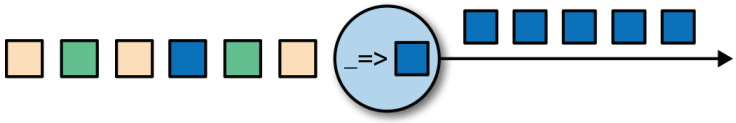

Рис. 2.4 ❖ Оператор потоковой передачи с функцией, которая превращает каждое входящее событие в событие, обозначенное более темным цветом

Операторы могут принимать несколько входных данных и создавать несколько выходных потоков. Они также могут изменять структуру графа потока данных, разделяя поток на несколько потоков или объединяя их в один. Мы обсудим семантику всех операторов, доступных во Flink, в главе 5.

2.2.2.3. Скользящее агрегирование

Скользящее агрегирование (rolling aggregation) – это агрегирование, такое как сумма, минимум и максимум, которое постоянно обновляется для каждого входного события. Операции агрегирования учитывают состояние и объединяют текущее состояние с входящим событием для получения обновленного агрегированного значения. Обратите внимание, что, для того чтобы иметь возможность эффективно комбинировать текущее состояние с событием и выдавать одно значение, функция агрегирования должна быть ассоциативной и коммутативной. В противном случае оператору пришлось бы хранить всю историю потоковой передачи. Рисунок 2.5 показывает агрегирование *скользящего минимума*. Оператор сохраняет текущее минимальное значение и соответственно обновляет его для каждого входящего события.

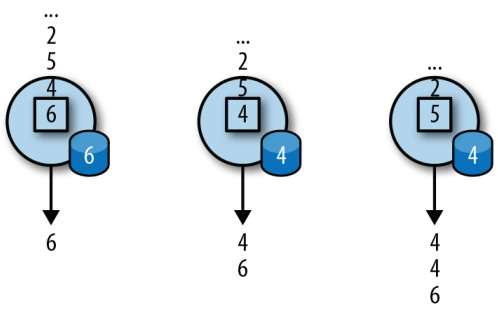

Рис. 2.5 ❖ Операция агрегирования скользящего минимума

2.2.2.4. Оконные операции

Операции преобразования и скользящего агрегирования обрабатывают одно событие за раз для создания выходных событий и, возможно, обновления состояния. Однако некоторые операции должны собирать и буферизовать определенное количество записей для вычисления результата на их основе. Рассмотрим, например, операцию потокового объединения или *целостный агрегат* (holistic aggregate), такой как медианная функция. Чтобы правильно выполнять подобные операции в неограниченных потоках, вам необходимо ограничить объем данных, которые охватывают эти операции. В этом разделе мы обсуждаем оконные операции, которые предоставляют такую услугу.

Помимо практического значения, окна также позволяют выполнять семантически интересные запросы к потокам. Вы видели, как операция скользящего агрегирования кодирует историю всего потока в единственное агрегированное значение и предоставляет нам результат с малой задержкой для каждого события. Это нормально для некоторых приложений, но что, если вас интересует только ограниченный набор самых свежих данных? Рассмотрим приложение, которое предоставляет водителям информацию о дорожной обстановке в режиме реального времени, чтобы они могли избегать перегруженных маршрутов. В этом сценарии вы хотите знать, произошла ли авария в определенном месте в течение последних нескольких минут. С другой стороны, вам вряд ли интересно знать обо всех происшествиях, которые когда-либо происходили на этом месте. Более того, сводя историю потоков к единому агрегату, вы теряете информацию о том, как ваши данные меняются с течением времени. Например, вы хотите узнать, сколько транспортных средств пересекает данный перекресток каждые 5 мин.

Оконные операции непрерывно создают конечные наборы событий, называемые *корзинами данных* (data bucket), из неограниченного потока событий и позволяют нам выполнять вычисления на этих конечных наборах. События обычно назначаются сегментам на основе свойств данных или времени. Чтобы правильно определить семантику оконного оператора, нам необходимо определить, как события назначаются сегментам и как часто окно дает результат. Поведение окон определяется набором политик. Политики

окна решают, когда создаются новые сегменты, какие события назначаются каким сегментам и когда оценивается содержимое сегмента. Последнее решение основано на условии запуска. Когда условие триггера выполнено, содержимое корзины отправляется в функцию оценки, которая применяет логику вычислений к элементам корзины. Функции вычислений могут представлять собой агрегаты, такие как сумма или минимум, или настраиваемые операции, применяемые к собранным элементам корзины. Политики могут быть основаны на времени (например, события, полученные за последние пять секунд), количестве (например, последние сто событий) или свойстве данных. Далее мы описываем семантику общих типов окон.

- *Шагающие окна* (tumbling window) помещают события в неперекрывающиеся сегменты фиксированного размера. После достижения границы окна все события отправляются в функцию оценки для обработки. Шагающие окна на основе подсчета определяют, сколько событий собирается перед запуском оценки. На рис. 2.6 показано шагающее окно на основе подсчета, которое разбивает входной поток на сегменты из четырех элементов. Шагающие окна на основе времени определяют временной интервал, в течение которого события буферизируются в сегменте. На рис. 2.7 показано шагающее окно на основе времени, в котором события собираются в сегменты и обрабатываются каждые 10 мин.

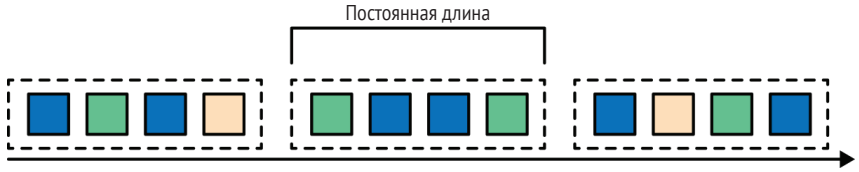

Рис. 2.6 ❖ Шагающее окно на основе подсчета

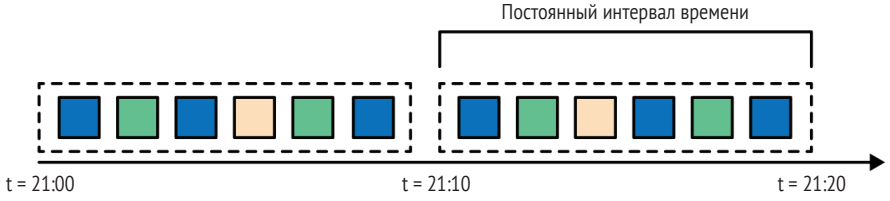

Рис. 2.7 ❖ Шагающее окно на основе времени

- *Скользящие окна* (sliding window) назначают события в перекрывающиеся сегменты фиксированного размера. Таким образом, событие может принадлежать нескольким сегментам. Мы определяем скользящие окна, указывая их длину и *сдвиг* (slide). Значение сдвига определяет интервал, через который создается новый сегмент. Скользящее окно на основе подсчета на рис. 2.8 имеет длину четыре события и сдвиг на три события.

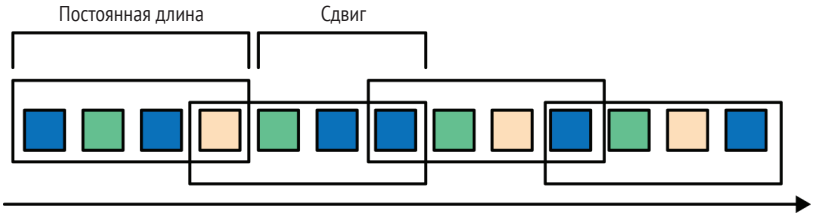

Рис. 2.8 ❖ Скользящее окно на основе подсчета с длиной четыре события и сдвигом на три события

- *Окна сеанса* (session window) полезны в обычных сценариях реального мира, где нельзя применить ни шагающие, ни скользящие окна. Рассмотрим приложение, которое анализирует поведение пользователей в интернете. В таких приложениях мы хотели бы сгруппировать события, происходящие из одного периода активности или *сеанса* пользователя. Сеансы состоят из серии событий, происходящих в смежные периоды времени, за которыми следует период бездействия. Например, взаимодействие пользователя с серией новостных статей одна за другой можно рассматривать как сеанс. Поскольку продолжительность сеанса не определяется заранее, но зависит от фактических данных, в этом сценарии нельзя применять шагающие и скользящие окна. Вместо этого нам нужна оконная операция, которая назначает события, принадлежащие одному сеансу, в одном сегменте. Окна сеанса группируют события в сеансах на основе *промежутка между сеансами*, который определяет время бездействия, достаточное, чтобы считать сеанс закрытым. На рис. 2.9 показано окно сеанса.

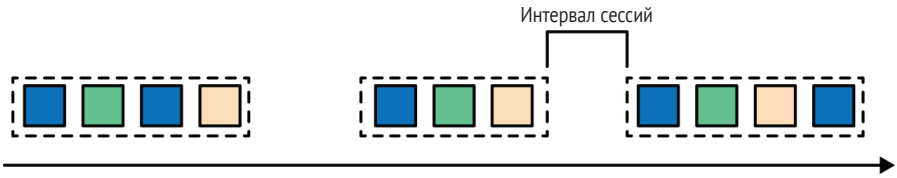

Рис. 2.9 ❖ Окно сеанса

Все типы окон, рассмотренные до сих пор, являются окнами, которые работают с полным потоком. Но на практике вы можете разделить поток на несколько логических потоков и определить *параллельные окна*. Например, если вы получаете измерения от разных датчиков, вы, вероятно, захотите сгруппировать поток по идентификатору датчика перед применением оконной операции. В параллельных окнах каждый раздел применяет оконные политики независимо от других разделов. На рис. 2.10 показано параллельное шагающее окно на основе подсчета длиной 2 с разделением потока по цвету событий.

Оконные операции тесно связаны с двумя основными концепциями потоковой обработки: семантикой времени и учетом состояния. Время, пожалуй,

является самым важным аспектом потоковой обработки. Несмотря на то что низкая задержка является привлекательным свойством потоковой обработки, ее истинная ценность выходит за рамки простой аналитики. Реальные системы, сети и каналы связи далеки от совершенства, и потоковая передача данных часто может прерываться, а события могут поступать не по порядку. Крайне важно понимать, как получить точные и детерминированные результаты в таких условиях. Более того, потоковые приложения, обрабатывающие события по мере их создания, также должны иметь возможность обрабатывать исторические события аналогичным образом, что позволяет проводить автономную аналитику или даже *анализ с путешествием во времени* (time-travel analysis). Конечно, все это не имеет значения, если ваша система не может защитить состояние от сбоев. Все типы окон, которые вы видели до сих пор, требуют буферизации данных перед получением результата. Фактически, если вы хотите вычислить что-либо интересное в потоковом приложении, даже простой подсчет, вам необходимо поддерживать состояние. Учитывая, что потоковые приложения могут работать в течение нескольких дней, месяцев или даже лет, вам необходимо убедиться, что состояние может быть надежно восстановлено при сбоях и что ваша система может гарантировать точные результаты, даже если что-то сломается. В оставшейся части этой главы мы более подробно рассмотрим концепции времени и гарантий состояния при сбоях обработки потока данных.

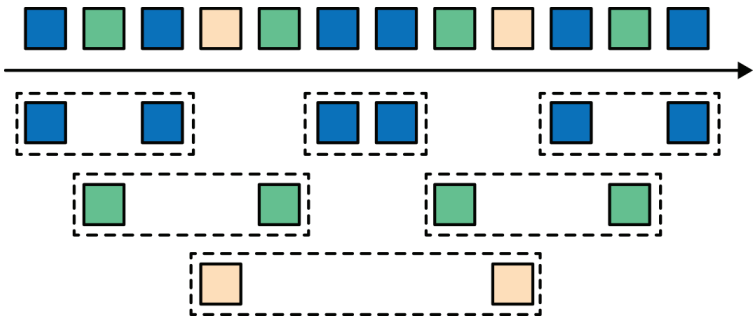

Рис. 2.10 ❖ Параллельное шагающее окно длиной 2 на основе подсчета с разделением потока по цвету событий

2.3. Семантика времени

В этом разделе мы вводим семантику времени и описываем различные нотации времени в потоковой передаче. Мы обсудим, как потоковый процессор добивается точных детерминированных результатов с неупорядоченными событиями и как вы можете выполнять обработку исторических событий и путешествия во времени при потоковой передаче данных.

2.3.1. Что означает одна минута в потоковой обработке?

При работе с потенциально неограниченным потоком постоянно прибывающих событий время становится центральным аспектом приложений. Предположим, вы хотите вычислять результаты непрерывно, возможно, каждую минуту. Что на самом деле будет означать *одна минута* в контексте вашего потокового приложения?

Рассмотрим программу, которая анализирует события пользователей, играющих в мобильную онлайн-игру. Пользователи организованы в команды, и приложение собирает данные об активности команды и раздает награды в игре, такие как дополнительные жизни и повышение уровня, в зависимости от того, насколько быстро участники команды достигают целей игры. Например, если все пользователи в команде лопают 500 пузырьков в течение одной минуты, они переходят на следующий уровень. Алиса – преданный игрок, который играет в эту игру каждое утро по дороге на работу. Проблема в том, что Алиса живет в Берлине и на работу ездит на метро. И все знают, что мобильный интернет в берлинском метро, мягко говоря, так себе. Рассмотрим случай, когда Алиса начинает лопать пузырьки, когда ее телефон подключен к сети, и отправляет события в приложение анализа. Затем поезд въезжает в туннель, и ее телефон отключается. Алиса продолжает играть, и игровые события сохраняются в ее телефоне. Когда поезд выезжает из туннеля, телефон снова подключается к сети, и ожидающие события отправляются в приложение. Что должно делать приложение? Что в этом случае означает одна минута? Включает ли она время, когда Алиса была офлайн или нет? Рисунок 2.11 иллюстрирует эту проблему.

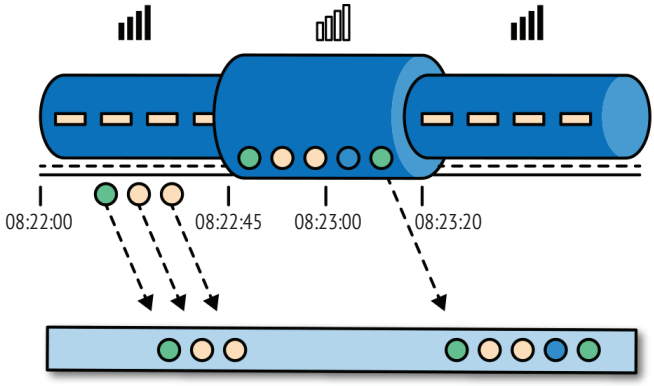

Рис. 2.11 ❖ Приложение, получающее события мобильных онлайн-игр в метро, столкнется с разрывом потока при потере сетевого подключения, но события сохраняются в памяти телефона игрока и доставляются при восстановлении соединения

Онлайн-игры – это простой сценарий, показывающий, что семантика оператора должна зависеть от времени, когда события действительно происходят, а не от времени, когда приложение получает события. В случае мобильной игры самым неприятным следствием неправильной семантики будет разочарование Алисы и ее команды, которые больше никогда не будут играть в эту игру. Но есть гораздо более критичные ко времени приложения, правильность семантики которых мы обязаны гарантировать. Если мы будем учитывать только то, сколько данных мы получаем в течение одной минуты, наши результаты будут отличаться и зависеть от скорости сетевого подключения или скорости обработки. На самом деле что действительно определяет количество событий в одной минуте, так это время самих данных.

В примере игры Алисы потоковое приложение может работать с двумя разными понятиями времени: время обработки или время события. Мы рассмотрим оба понятия в следующих разделах.

2.3.2. Время обработки

Время обработки (processing time) – это время локальных часов на машине, которая обрабатывает поток. Окно с привязкой к времени обработки включает в себя все события, которые произошли в оконном операторе за период времени, измеренный часами его машины. Как показано на рис. 2.12, в случае Алисы такое окно продолжает отсчет времени, когда ее телефон отключается, и не учитывает ее игровую активность в это время.

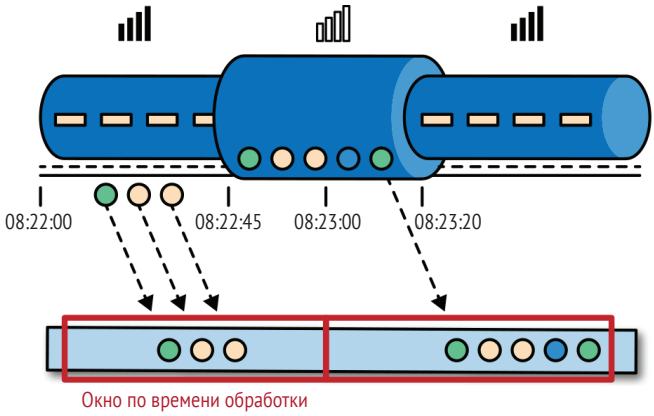

Рис. 2.12 ❖ Окно с привязкой к времени обработки продолжает отсчитывать время даже после отключения телефона Алисы

2.3.3. Время события

Время события (event time) – это время, когда событие в потоке действительно произошло. Время события основано на *метке времени* (timestamp), которая прикреплена к событию потока. Метки времени обычно суще-

ствуют внутри данных события до того, как они попадают в конвейер обработки (например, время создания события). На рис. 2.13 показано, что окно с привязкой ко времени события правильно объединяет события в соответствии с их реальным порядком, даже если некоторые события были *отложены*.

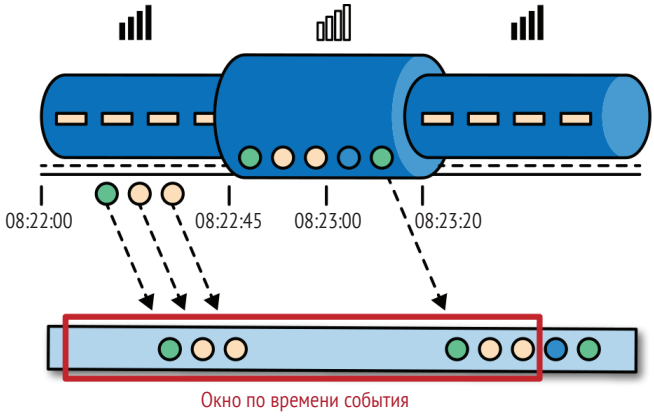

Рис. 2.13 ❖ Время события правильно помещает события в окно, отражая реальное положение дел

Время события полностью отделяет скорость обработки от результатов. Операции, основанные на времени события, предсказуемы, а их результаты детерминированы. Обработка окна с привязкой к времени события даст один и тот же результат независимо от того, насколько быстро обрабатывается поток или когда события поступают к оператору.

Обработка отложенных событий – это только одна из проблем, которую вы можете преодолеть с помощью времени события. С его помощью также может быть решена распространенная проблема неупорядоченных данных. Возьмем Боба, еще одного игрока в мобильной онлайн-игре, который оказался в одном поезде с Алисой. Боб и Алиса играют в одну игру, но у них разные операторы мобильной связи. Пока телефон Алисы теряет связь, находясь внутри туннеля, телефон Боба остается подключенным и передает события в игровое приложение.

Опираясь на время события, мы можем гарантировать правильность результата даже в случае неупорядоченных данных. Более того, в сочетании с воспроизводимыми потоками детерминизм временных меток дает вам возможность перемещаться в прошлое. То есть вы можете воспроизводить поток и анализировать исторические данные, как если бы события происходили в реальном времени. Кроме того, вы можете ускорить вычисление до настоящего момента, чтобы после того, как ваша программа догнала события, происходящие сейчас, она могла продолжить работу как приложение реального времени, используя неизменную логику программы.

2.3.4. Водяные знаки

В нашем обсуждении окон и времени событий до сих пор мы упускали из виду один очень важный аспект: *как мы решаем, когда запускать окно с привязкой к времени события*? То есть как долго нам нужно ждать, прежде чем мы сможем быть уверены, что получили все события, которые произошли до определенного момента времени? И откуда мы вообще знаем, что данные будут задержаны? Учитывая непредсказуемое поведение распределенных систем и произвольные задержки, которые могут быть вызваны внешними факторами, мы не можем дать однозначно правильные ответы на эти вопросы. В этом разделе вы узнаете, как использовать водяные знаки для настройки поведения окна с привязкой к времени события.

Водяной знак (watermark) – это глобальный показатель прогресса, который указывает момент времени, когда мы уверены, что больше не осталось отложенных событий. По сути, водяные знаки образуют логические часы, которые информируют систему о текущем времени события. Когда оператор получает водяной знак со временем T, он может предположить, что больше никогда не поступят события с меткой времени меньше T. Водяные знаки необходимы как для окон с привязкой к времени событий, так и для операторов, обрабатывающих неупорядоченные события. После поступления водяного знака операторы получают сигнал о том, что все метки времени для определенного временного интервала были учтены, и либо запускают вычисление, либо упорядочивают полученные события.

Водяные знаки обеспечивают настраиваемый компромисс между достоверностью результатов и задержкой. Частые водяные знаки обеспечивают низкую задержку, но не дают уверенности в точности. В этом случае запоздавшие события могут появиться после водяного знака, и мы должны предусмотреть код для их обработки. С другой стороны, если водяные знаки слишком неторопливы, у вас высокая уверенность в точности, но вы можете увеличить задержку обработки без реальной необходимости.

Во многих реальных приложениях системе не хватает знаний для точного определения водяных знаков. В примере с мобильными играми практически невозможно угадать, как долго пользователи могут оставаться отключенными; они могут проезжать через туннель, лететь в самолете или никогда больше не играть. Независимо от того, задаются ли водяные знаки пользователем или создаются автоматически, при наличии отложенных задач отслеживание глобального прогресса в распределенной системе остается проблематичным. Следовательно, слепая надежда на водяные знаки не всегда может быть хорошей идеей. Вместо этого крайне важно, чтобы система потоковой обработки обеспечивала механизм для работы с событиями, которые могут появиться после водяного знака. В зависимости от требований к приложению вы можете игнорировать такие события, регистрировать их или использовать для исправления предыдущих результатов.

2.3.5. Время обработки по сравнению со временем события

На этом этапе вам может быть интересно, зачем нам вообще беспокоиться о времени обработки, если время события решает все наши проблемы. На самом деле в некоторых случаях может пригодиться и время обработки. Окна с привязкой к времени обработки обеспечивают минимально возможную задержку. Поскольку вы не принимаете во внимание опоздавшие и неупорядоченные события, окно просто должно буферизовать события и немедленно запускать вычисление по достижении указанного промежутка времени. Таким образом, для приложений, где скорость важнее точности, время обработки оказывается кстати. Иногда вам нужно периодически выдавать результаты в режиме реального времени, независимо от их точности. Примером такого приложения может быть панель мониторинга в реальном времени, отображающая агрегированные данные о событиях по мере их поступления. Наконец, окна с привязкой ко времени обработки предлагают точное представление самих потоков, что желательно для некоторых случаев использования. Например, вам может быть интересно наблюдать за потоком и подсчитывать количество событий в секунду для обнаружения сбоев. Напомним, что время обработки обеспечивает низкую задержку, но результаты зависят от скорости обработки и не являются детерминированными. С другой стороны, время события гарантирует детерминированные результаты и позволяет вам иметь дело с событиями, которые опаздывают или даже нарушают порядок следования.

2.4. Модели состояния и согласованности

Теперь перейдем к другому чрезвычайно важному аспекту потоковой обработки – *состоянию* (state). Состояние широко применяется в обработке данных. Оно требуется для любых нетривиальных вычислений. Для получения результата функция накапливает состояние в течение определенного периода времени или количества событий (например, для вычисления агрегирования или обнаружения шаблона). Операторы с учетом состояния для вычисления своих выходных данных используют как входящие события, так и внутреннее состояние. Возьмем, к примеру, оператор скользящего агрегирования, выводящий текущую сумму всех событий, которые он получил до сих пор. Оператор сохраняет текущее значение суммы в качестве внутреннего состояния и обновляет его каждый раз, когда получает новое событие. Точно так же ведет себя оператор, выдающий предупреждение, когда обнаруживает событие «высокая температура», за которым в течение ближайших 10 мин следует событие «задымление». Оператору необходимо сохранить событие «высокая температура» во внутреннем состоянии до тех пор, пока он не увидит событие «задымление» или пока не истечет 10-минутный период времени.

Важность состояния становится еще более очевидной, если мы рассмотрим случай использования системы пакетной обработки для анализа неограниченного набора данных. До появления современных потоковых процессоров распространенным подходом к обработке неограниченных данных было многократное планирование заданий для небольших пакетов входящих событий в системе пакетной обработки. Когда задание завершается, результат записывается в постоянное хранилище, и все состояние оператора теряется. Задание, запланированное для обработки следующего пакета, не может получить доступ к состоянию предыдущего задания. Эта проблема обычно решается путем делегирования управления состоянием внешней системе, например базе данных. Напротив, при непрерывном выполнении потоковых заданий состояние сохраняется на протяжении всей вереницы событий, и мы можем считать его важнейшим компонентом модели программирования. В принципе, мы могли бы использовать внешнюю систему для управления состоянием и в случае потоковой обработки, даже несмотря на то что такая архитектура может привести к дополнительной задержке.

Поскольку потоковые операторы обрабатывают потенциально неограниченные данные, следует проявлять осторожность, чтобы не допустить бесконечного роста внутреннего состояния. Чтобы ограничить размер состояния, операторы обычно поддерживают некоторую сводку или *синопсис* (synopsis) событий, наблюдаемых до сих пор. Такая сводка может быть подсчетом, суммой, выборкой наблюдаемых до сих пор событий, буфером окна или настраиваемой структурой данных, которая сохраняет некоторые свойства, представляющие интерес для запущенного приложения.

Как вы можете догадаться, поддержка операторов с учетом состояния сопряжена с несколькими проблемами реализации.

Управление состоянием
: Системе необходимо эффективно управлять состоянием и гарантировать, что оно защищено от конкурентных обновлений.

Разделение состояний
: Учет состояния усложняет распараллеливание, поскольку результаты зависят как от состояния, так и от входящих событий. К счастью, во многих случаях вы можете разделить состояния по ключу и независимо управлять состоянием каждой ветви. Например, если вы обрабатываете поток измерений от набора датчиков, вы можете использовать разделенное состояние, чтобы независимо поддерживать состояние каждого датчика.

Восстановление состояния
: Третья и самая большая проблема, с которой сталкиваются операторы с учетом состояния, – это обеспечение возможности восстановления состояния и гарантии правильности результатов даже при наличии сбоев.

Далее мы подробно обсудим сбои заданий и гарантии результата.

2.4.1. Сбои заданий

Состояние оператора в потоковых заданиях очень ценно, и его следует защищать от сбоев. Если состояние потеряно во время сбоя, после восстановления мы получим неправильные результаты. Задания потоковой передачи выполняются в течение длительных периодов времени, поэтому состояние может формироваться в течение нескольких дней или даже месяцев. Повторная обработка всего ввода для воспроизведения потерянного состояния в случае сбоев обходится очень дорого и требует много времени.

В начале этой главы вы узнали, как можно моделировать потоковые программы в виде графов потоков данных. Перед выполнением они преобразуются в графы физических потоков данных связанных параллельных задач, каждая из которых выполняет некоторую логику оператора, потребляет входные потоки и создает выходные потоки для других заданий. В типичных реальных архитектурах могут запускаться сотни таких задач, выполняемых параллельно на многих физических машинах. В случае длительной работы системы каждая такая задача может завершиться ошибкой в любой момент. Как обеспечить прозрачную обработку таких сбоев, чтобы общее задание потоковой обработки могло продолжать выполняться? Фактически нам нужно, чтобы потоковый процессор не только продолжал обработку в случае сбоя задачи, но также предоставлял гарантии достоверности результата и сохранности состояния оператора. Мы обсудим эти вопросы ниже.

2.4.1.1. Что такое сбой задачи?

Для каждого события во входном потоке задача – это этап обработки, который выполняет следующие шаги: (1) принимает событие, сохраняя его в локальном буфере; (2) возможно, обновляет внутреннее состояние; (3) выполняет выходную запись. Сбой может произойти на любом из этих шагов, и система должна иметь четко определенное поведение для каждого сценария сбоя. Если сбоем завершился первый этап, событие потеряно? Если сбой случился после обновления внутреннего состояния, обновит ли его система после восстановления?

Мы предполагаем, что сетевые соединения надежны, что никакие записи не удаляются и не дублируются, и все события в конечном итоге доставляются в место назначения в порядке FIFO. Обратите внимание, что Flink использует TCP-соединения, поэтому эти требования гарантированно соблюдаются. Мы также предполагаем, что детекторы отказов являются совершенными и что ни одна задача не будет действовать злонамеренно, то есть все задачи, которые не завершились ошибкой, выполняются в соответствии с вышеуказанными шагами.

В сценарии пакетной обработки на все эти вопросы есть ответы, потому что пакетное задание можно просто перезапустить с самого начала. Следо-

вательно, никакие события не теряются, и состояние полностью создается с нуля. Однако в мире потоковой обработки устранение сбоев – нетривиальная проблема. Потоковые системы определяют свое поведение при сбоях, исходя из гарантии результата. Далее мы рассмотрим типы гарантий, предлагаемых современными потоковыми процессорами, и некоторые механизмы, на которых основаны эти гарантии.

2.4.2. Гарантии результата

Прежде чем мы опишем различные типы гарантий, нам нужно прояснить несколько моментов, которые часто являются источником путаницы при обсуждении сбоев задач в потоковых процессорах. В оставшейся части этой главы, когда мы говорим о «гарантиях результата», мы имеем в виду непротиворечивость внутреннего состояния потокового процессора. То есть нас беспокоит то, что код приложения видит как значение состояния после восстановления после сбоя. Обратите внимание, что гарантировать согласованность состояния приложения – это не то же самое, что гарантировать согласованность его вывода. Трудно гарантировать правильность результата после того, как данные были отправлены в приемник, если система приемника не поддерживает формат транзакции, не так ли?

2.4.2.1. Не более одного раза

Самое простое, что можно сделать в случае сбоя задания, – ничего не делать для восстановления потерянного состояния и воспроизведения потерянных событий. «Не более одного раза» (at-most-once) – это тривиальный подход, который гарантирует обработку каждого события не более одного раза. Другими словами, события можно просто отбросить, и ничего не будет сделано для обеспечения правильности результата. Этот тип гарантии также известен как «отсутствие гарантии», поскольку такую гарантию может предоставить даже система, теряющая каждое событие. Отсутствие каких-либо гарантий звучит ужасно, но на самом деле это неплохо, если вы согласны жить с приблизительными результатами и все, о чем вы заботитесь, – это обеспечить минимально возможную задержку.

2.4.2.2. Хотя бы один раз

В большинстве реальных приложений ожидается, что события не должны теряться. Этот тип гарантии называется «хотя бы один раз» (at-least-once), и это означает, что все события будут обработаны, однако есть вероятность, что некоторые из них будут обработаны более одного раза. Повторная обработка может быть приемлемой, если правильность результата зависит только от полноты информации. Например, с такой гарантией правильно реализуется обнаружение конкретного события во входном потоке. В худшем случае вы обнаружите событие более одного раза. Однако подсчет того, сколько раз происходит конкретное событие во входном потоке, может вернуть неправильный результат.

Чтобы гарантировать правильность результата при такой гарантии, вам необходимо иметь способ воспроизвести события – либо из источника, либо из некоторого буфера. В непрерывных журналах событий все события записываются в долговременное хранилище, чтобы их можно было воспроизвести в случае сбоя задания. Другой способ достижения эквивалентной функциональности – использование подтверждений записи. Этот метод сохраняет каждое событие в буфере до тех пор, пока его обработка не будет подтверждена всеми заданиями в конвейере, после чего событие можно отбросить.

2.4.2.3. Ровно один раз

«Ровно один раз» – это самая строгая гарантия, и ее трудно достичь. Эта гарантия означает, что не только не будет потери события, но и обновления внутреннего состояния для каждого события будут происходить ровно один раз. По сути, гарантия «ровно один раз» означает, что наше приложение предоставит правильный результат, как будто сбоя никогда не было.

Для предоставления данной гарантии требуется соблюдение гарантии «хотя бы один раз», и, следовательно, снова необходим механизм воспроизведения данных. Кроме того, потоковый процессор должен обеспечивать согласованность внутреннего состояния. То есть после восстановления он должен знать, было ли обновление события уже отражено в состоянии или нет. Одним из способов достижения этого результата могут быть постоянные транзакции обновлений, но они могут привести к значительным накладным расходам и снижению быстродействия. Вместо этого Flink использует легкий механизм моментальных снимков. Мы обсуждаем алгоритм отказоустойчивости Flink в разделе 3.5.

2.4.2.4. Сквозная гарантия «ровно один раз»

Типы гарантий, с которыми вы уже познакомились, относятся к состоянию приложения, которым управляет потоковый процессор. Однако в реальном потоковом приложении помимо потокового процессора будет по крайней мере один источник и один приемник. Сквозные гарантии относятся к правильности результатов по всему конвейеру обработки данных. Каждый компонент предоставляет свои собственные гарантии, и сквозная гарантия всего конвейера будет самым слабым местом. Важно отметить, что иногда вы можете улучшить семантику приложения с более слабыми гарантиями. Распространенный случай – когда задача выполняет идемпотентные операции, такие как поиск максимума или минимума. В этом случае для достижения результата семантики «ровно один раз» достаточно гарантии «хотя бы один раз».

2.5. Заключение

В этой главе вы изучили основы потоковой обработки данных. Вы рассмотрели модель программирования потока данных и узнали, как потоковые приложения могут быть представлены в виде распределенных графов пото-

ков данных. Затем вы узнали о требованиях к параллельной обработке бесконечных потоков и увидели важность задержки и пропускной способности для потоковых приложений. Мы рассмотрели основные операции потоковой передачи и то, как вычислить значимые результаты для неограниченных входных данных с помощью оконных операторов. Вы оценили значение времени в потоковой обработке и сравнили понятия времени события и времени обработки. Наконец, вы узнали, почему потоковым приложениям важно хранить состояние и как защитить его от сбоев и гарантировать правильные результаты.

До этого момента мы рассматривали концепции потоковой обработки независимо от Apache Flink. В оставшейся части книги мы расскажем, как Flink реализует эти концепции на практике, и как вы можете использовать его API DataStream для написания приложений, использующих все функции, которые мы уже представили.

Глава 3

Архитектура Apache Flink

В главе 2 обсуждались важные концепции распределенной потоковой обработки, такие как распараллеливание, время и состояние. В этой главе мы даем общее представление об архитектуре Flink и рассказываем, как Flink реализует аспекты потоковой обработки, которые обсуждались ранее. В частности, мы объясняем распределенную архитектуру Flink, показываем, как она обрабатывает время и состояние в потоковых приложениях, и обсуждаем его механизмы отказоустойчивости. В этой главе представлена справочная информация по успешной реализации и последующей работе с расширенными потоковыми приложениями при помощи Apache Flink. Это поможет вам понять внутреннее устройство Flink и обоснованно судить о быстродействии и поведении потоковых приложений.

3.1. Архитектура системы

Flink – это распределенная система для параллельной обработки потоков данных с учетом состояния. Обычно Flink состоит из нескольких процессов, которые выполняются на нескольких машинах. Обычные проблемы, с которыми сталкиваются распределенные системы, – это распределение вычислительных ресурсов в кластере и управление ими, координация процессов, надежное и высокодоступное хранилище данных и восстановление после сбоев.

Flink не решает все эти проблемы самостоятельно. Вместо этого он фокусируется на своей основной функции – распределенной обработке потоков данных – и использует существующую кластерную инфраструктуру и службы. Flink хорошо интегрирован с диспетчерами кластерных ресурсов, такими как Apache Mesos, YARN и Kubernetes, но также может быть настроен для работы в качестве автономного кластера. Flink не обеспечивает надежное распределенное хранилище. Вместо этого он использует преимущества распределенных файловых систем, таких как HDFS, или хранилищ объектов, таких как S3. Для выбора лидера в высокодоступных конфигурациях Flink использует Apache ZooKeeper.

В этом разделе мы опишем различные компоненты набора Flink и то, как они взаимодействуют друг с другом для выполнения приложения. Обсудим

два разных стиля развертывания приложений Flink и то, как каждый из них распределяет и выполняет задачи. Наконец, мы объясним, как работает режим высокой доступности Flink.

3.1.1. Компоненты набора Flink

Набор Flink состоит из четырех различных компонентов, которые вместе обеспечивают выполнение потоковых приложений. Эти компоненты называются JobManager, ResourceManager, TaskManager и Dispatcher. Поскольку Flink реализован на Java и Scala, все компоненты работают на виртуальных машинах Java (Java Virtual Machine, JVM). Каждый компонент имеет следующие обязанности:

- *JobManager* – это главный процесс, который контролирует выполнение одного приложения: каждое приложение управляется отдельным экземпляром JobManager. JobManager получает заявку на исполнение. Приложение состоит из так называемого JobGraph – логического графа потока данных (см. главу 1) и файла JAR, который объединяет все необходимые классы, библиотеки и другие ресурсы. JobManager преобразует JobGraph в физический граф потока данных, называемый ExecutionGraph, состоящий из задач, которые могут выполняться параллельно. JobManager запрашивает необходимые ресурсы (слоты TaskManager) для выполнения задач из ResourceManager. Как только он получает достаточно слотов TaskManager, он распределяет задачи ExecutionGraph между TaskManager, которые их выполняют. Во время выполнения JobManager отвечает за все действия, требующие централизованной координации, такие как координация контрольных точек (см. раздел 3.5);
- *ResourceManager* отвечает за управление слотами TaskManager, единицей обработки ресурсов Flink. Существует несколько вариантов ResourceManager для разных сред и поставщиков ресурсов, таких как YARN, Mesos, Kubernetes и автономные развертывания. Когда JobManager запрашивает слоты TaskManager, ResourceManager дает распоряжение TaskManager с незанятыми слотами, чтобы тот предоставил свои слоты в работу. Если ResourceManager не имеет достаточно слотов для выполнения запроса JobManager, он может обратиться к поставщику ресурсов для получения контейнеров, в которых запускаются процессы TaskManager. ResourceManager также заботится о завершении бездействующих TaskManager для освобождения вычислительных ресурсов;
- *TaskManager* – это рабочий процесс Flink. Как правило, в системе Flink работает несколько экземпляров TaskManager. Каждый TaskManager предоставляет определенное количество слотов. Количество слотов ограничивает количество заданий, которые может выполнить TaskManager. После запуска TaskManager регистрирует свои слоты в ResourceManager. По указанию ResourceManager TaskManager предлагает один или несколько своих слотов для JobManager. Затем JobManager может назначить задания слотам для их выполнения. Во время выполнения TaskManager обменивается данными с другими TaskManager,

которые запускают задания того же приложения. Выполнение заданий и концепция слотов обсуждаются в разделе 3.1.3;

- *Dispatcher* управляет работой и предоставляет интерфейс REST для отправки приложений на выполнение. После того как приложение отправлено на выполнение, он запускает JobManager и передает ему приложение. Интерфейс REST позволяет диспетчеру служить точкой входа HTTP для кластеров, находящихся за брандмауэром. Диспетчер также запускает веб-панель управления для предоставления информации о выполнении заданий. В зависимости от того, как приложение отправлено на выполнение (см. раздел 3.1.2), диспетчер может и не потребоваться.

На рис. 3.1 показано, как компоненты Flink взаимодействуют друг с другом, когда приложение отправляется на выполнение.

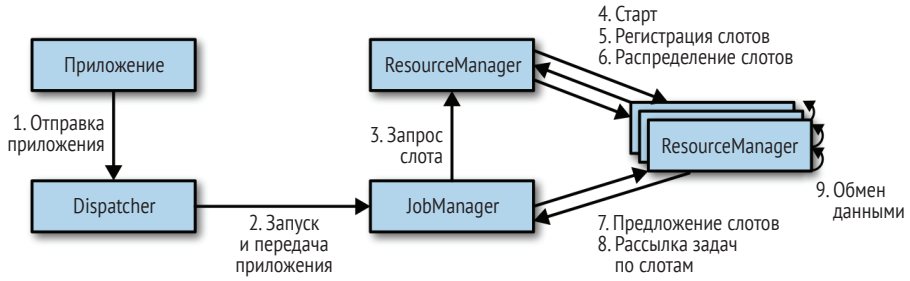

Рис. 3.1 ❖ Выполнение приложения и взаимодействие компонентов

Этот рисунок представляет собой наиболее общий набросок, и предназначен для визуализации ответственности и взаимодействия компонентов приложения. В зависимости от среды (YARN, Mesos, Kubernetes, автономный кластер) некоторые шаги могут быть пропущены, либо некоторые компоненты могут выполняться в одном процессе JVM. Например, в автономном варианте – установке без поставщика ресурсов – ResourceManager может распределять только слоты доступных компонетов TaskManager и не может запускать новые TaskManager самостоятельно. В разделе 9.1 мы обсудим, как установить и настроить Flink для различных сред.

3.1.2. Развертывание приложений

Приложения Flink можно развертывать в двух разных режимах.

Фреймворк

В этом режиме приложения Flink упаковываются в файл JAR и отправляются клиентом в работающую службу. Этой службой может быть Flink Dispatcher, Flink JobManager или YARN ResourceManager. В любом случае есть работающая служба, которая принимает приложение Flink и обеспечивает его выполнение. Если приложение было отправлено в JobManager, он немедленно начинает выполнение приложения. Если приложение было отправлено в Dispatcher или YARN ResourceManager, он запустит JobManager и передаст приложение, а JobManager начнет выполнение.

Библиотека

В этом режиме приложение Flink упаковано в зависящий от задания образ контейнера, например образ Docker. Образ также содержит код для запуска JobManager и ResourceManager. Когда контейнер запускается из образа, он автоматически запускает ResourceManager и JobManager и отправляет связанное задание на выполнение. Второй образ, не зависящий от задания, используется для развертывания контейнеров TaskManager. Контейнер, который запускается из этого образа, автоматически запускает TaskManager, который подключается к ResourceManager и регистрирует свои слоты. Как правило, внешний менеджер ресурсов, такой как Kubernetes, заботится о запуске образов и обеспечивает перезапуск контейнеров в случае сбоя.

Режим фреймворка следует традиционному подходу к отправке приложения (или запроса) через клиента в работающую службу. В библиотечном режиме нет службы Flink. Вместо этого Flink объединен как библиотека вместе с приложением в одном образе контейнера. Этот режим развертывания является общим для архитектур микросервисов. Мы обсуждаем тему развертывания приложений более подробно в разделе 10.1.

3.1.3. Выполнение задачи

TaskManager может выполнять несколько задач одновременно. Эти задачи могут быть подзадачами одного и того же оператора (параллелизм данных), другого оператора (параллелизм задач) или даже из другого приложения (параллелизм заданий). TaskManager предлагает определенное количество слотов обработки для управления количеством задач, которые он может выполнять одновременно. Слот обработки может выполнять один фрагмент приложения – по одной параллельной задаче каждого оператора приложения. На рис. 3.2 показаны отношения между экземплярами TaskManager, слотами, задачами и операторами.

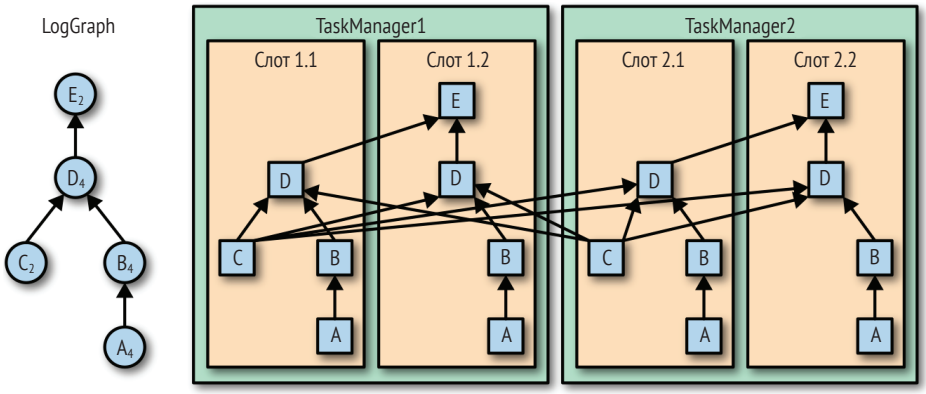

Рис. 3.2 ❖ Операторы, задачи и слоты обработки

В левой части рис. 3.2 вы видите JobGraph – непараллельное представление приложения, состоящее из пяти операторов. Операторы A и C являются источниками, а оператор E – выход. Операторы C и E распараллелены на два потока каждый. Остальные операторы разделены на четыре потока каждый. Поскольку максимальный параллелизм оператора равен четырем, приложению требуется по крайней мере четыре доступных слота обработки. Это требование выполняется при наличии двух экземпляров TaskManager с двумя слотами обработки каждый. JobManager преобразует JobGraph в ExecutionGraph и назначает задачи четырем доступным слотам. Каждому слоту назначаются задачи операторов с параллелизмом четыре. Две задачи операторов C и E назначены на слоты 1.1 и 2.1 и слоты 1.2 и 2.2 соответственно. Планирование задач в виде срезов по слотам имеет то преимущество, что многие задачи размещены в TaskManager и могут эффективно обмениваться данными в рамках одного процесса и без доступа к сети. Однако слишком большое количество одновременных задач также может перегрузить TaskManager и привести к снижению быстродействия. В разделе 10.2 мы расскажем, как управлять расписанием задач.

TaskManager выполняет свои задачи в многопоточном режиме в одном и том же процессе JVM. Потоки легче, чем отдельные процессы, и требуют меньших затрат на связь, но не обеспечивают строгую изоляцию задач друг от друга. Следовательно, одна некорректная задача может убить весь процесс TaskManager и все выполняемые в нем задачи. Установив ограничение в один слот для каждого процесса TaskManager, вы можете изолировать приложения по диспетчерам задач. Используя параллелизм потоков внутри TaskManager и развертывая несколько процессов TaskManager на каждом хосте, Flink предлагает большую гибкость в достижении компромисса между производительностью и изоляцией ресурсов при развертывании приложений. Мы подробно обсудим конфигурацию и настройку кластеров Flink в главе 9.

3.1.4. Высокодоступная конфигурация

Потоковые приложения обычно предназначены для круглосуточной работы без перерыва. Следовательно, важно, чтобы их выполнение не прекращалось даже в случае сбоя задействованного процесса. Для восстановления после сбоев системе сначала необходимо перезапустить отказавшие процессы, а во-вторых, перезапустить приложение и восстановить его состояние. В этом разделе вы узнаете, как Flink перезапускает сбойные процессы. Восстановление состояния приложения описано в разделе 3.5.2.

3.1.4.1. Сбои TaskManager

Как обсуждалось ранее, Flink нуждается в достаточном количестве слотов обработки для выполнения всех задач приложения. При настройке Flink с четырьмя экземплярами TaskManager, каждый из которых предоставляет по два слота, потоковое приложение может выполняться с максимальным параллелизмом до восьми. Если один из экземпляров TaskManager выходит из

строя, количество доступных слотов уменьшается до шести. В этой ситуации JobManager попросит ResourceManager предоставить больше слотов обработки. Если это невозможно – например, потому что приложение работает в автономном кластере, – JobManager не может перезапустить приложение, пока не станет доступно достаточное количество слотов. Стратегия перезапуска приложения определяет, как часто JobManager перезапускает приложение и как долго он ждет между попытками перезапуска[1].

3.1.4.2. Сбои JobManager

Более серьезной проблемой, чем сбои TaskManager, являются сбои JobManager. JobManager контролирует выполнение потокового приложения и хранит метаданные о его выполнении, такие как указатели на завершенные контрольные точки. Потоковое приложение не может продолжить обработку, если ответственный процесс JobManager внезапно пропадет из системы. Это делает JobManager единственной точкой отказа для приложений во Flink. Чтобы решить эту проблему, Flink поддерживает режим высокой доступности, который переносит ответственность и метаданные задания на другой экземпляр JobManager в случае, если исходный JobManager пропадает.

Режим высокой доступности Flink основан на Apache ZooKeeper, системе для распределенных сервисов, требующих координации и консенсуса. Flink использует ZooKeeper для выборов лидера и в качестве высокодоступного и надежного хранилища данных. При работе в режиме высокой доступности JobManager записывает JobGraph и все необходимые метаданные, такие как JAR-файл приложения, в удаленную систему постоянного хранения. Кроме того, JobManager записывает указатель на место хранения в хранилище данных ZooKeeper. Во время выполнения приложения JobManager получает дескрипторы состояния (места хранения) отдельных контрольных точек задач. По завершении контрольной точки – когда все задачи успешно записали свое состояние в удаленное хранилище – JobManager записывает дескрипторы состояния в удаленное хранилище и указатель на это местоположение в ZooKeeper. Следовательно, все данные, необходимые для восстановления после сбоя JobManager, хранятся в удаленном хранилище, а ZooKeeper содержит указатели на места хранения. Рисунок 3.3 иллюстрирует эту конфигурацию.

Когда JobManager выходит из строя, все задачи, относящиеся к его приложению, автоматически отменяются. Новый JobManager, который берет на себя работу отказавшего экземпляра, выполняет следующие шаги:

1) запрашивает у ZooKeeper места хранения для получения JobGraph, файла JAR и дескрипторов состояния последней контрольной точки приложения из удаленного хранилища;
2) запрашивает слоты обработки у ResourceManager для продолжения выполнения приложения;
3) перезапускает приложение и сбрасывает состояние всех его задач до последней завершенной контрольной точки.

---

[1] Стратегии перезапуска более подробно обсуждаются в главе 10.

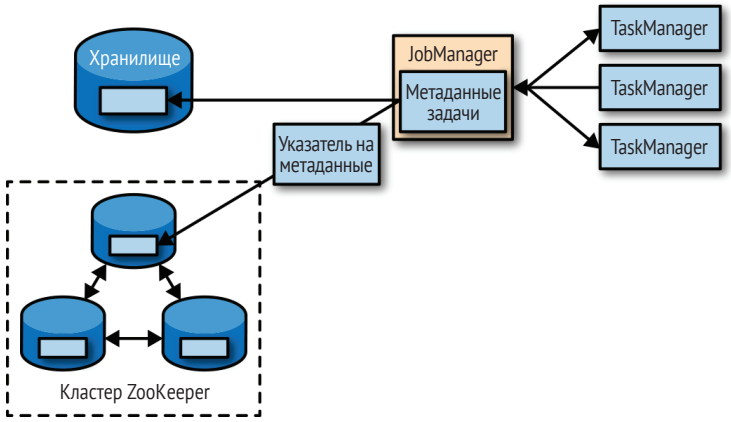

Рис. 3.3 ❖ Высокодоступная конфигурация Flink

При запуске приложения по типу развертывания библиотеки в контейнерной среде, такой как Kubernetes, отказавшие контейнеры JobManager или TaskManager обычно автоматически перезапускаются службой оркестровки контейнеров. При работе в YARN или Mesos оставшиеся процессы Flink инициируют перезапуск процессов JobManager или TaskManager. Flink не предоставляет инструменты для перезапуска сбойных процессов при работе в автономном кластере. Следовательно, было бы полезно запускать резервные диспетчеры заданий и диспетчеры задач, которые могут взять на себя работу отказавших процессов. Мы обсудим высокодоступную конфигурацию Flink позже в разделе 9.2.

3.2. Передача данных во Flink

Задачи работающего приложения постоянно обмениваются данными. Процессы TaskManager заботятся о доставке данных от передающих задач к принимающим задачам. Сетевой компонент TaskManager собирает записи в буферах перед их отправкой, то есть записи не отправляются по одной, а группируются в буферы. Это фундаментальный подход для эффективного использования сетевого ресурса и достижения высокой пропускной способности. Данный механизм аналогичен методам буферизации, используемым в сетевых протоколах или протоколах дискового ввода/вывода.

Обратите внимание, что доставка записей через буферы подразумевает, что модель обработки Flink основана на микропакетах.

Каждый TaskManager имеет пул сетевых буферов (по умолчанию размером 32 Кб) для отправки и получения данных. Если задачи отправителя и получателя выполняются в разных процессах TaskManager, они обмениваются данными через сетевой стек операционной системы. Потоковым приложениям необходимо обмениваться данными в конвейерном режиме – каждая

пара процессов TaskManager поддерживает постоянное TCP-соединение для обмена данными[1]. При перекрестной схеме подключения каждая задача-отправитель должна иметь возможность отправлять данные каждой принимающей задаче. TaskManager нуждается в одном выделенном сетевом буфере для каждой принимающей задачи, в которую любая из его задач должна отправлять данные. На рис. 3.4 показана эта архитектура.

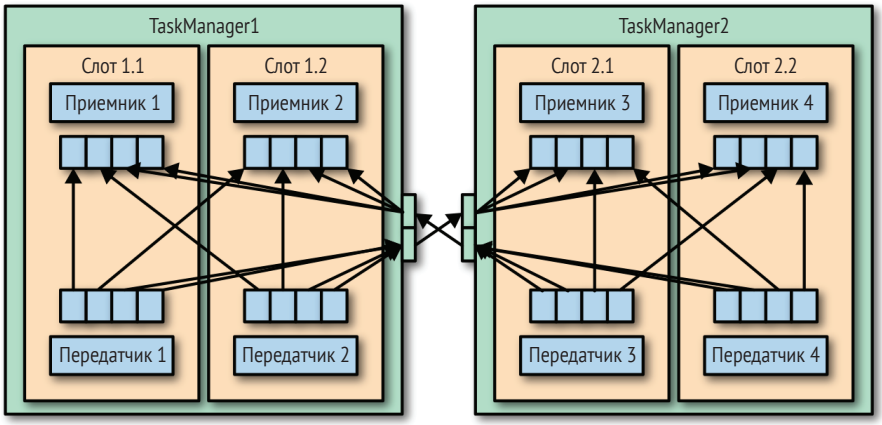

Рис. 3.4 ❖ Передача данных между TaskManager

Как показано на рис. 3.4, каждой из четырех задач-отправителей требуется по крайней мере четыре сетевых буфера для отправки данных каждой из задач-получателей, а каждой задаче-получателю требуется как минимум четыре буфера для приема данных. Буферы, которые необходимо отправить другому TaskManager, мультиплексируются через одно и то же сетевое соединение. Чтобы обеспечить неразрывный конвейерный обмен данными, TaskManager должен иметь возможность предоставлять достаточно буферов для одновременного обслуживания всех исходящих и входящих соединений. При перекрестном или широковещательном соединении каждой задаче-отправителю требуется буфер для каждой задачи-получателя; количество требуемых буферов равно квадрату количества задач задействованных операторов. Конфигурация Flink по умолчанию для сетевых буферов достаточна для систем малого и среднего размера. Для более крупных систем вам необходимо настроить конфигурацию, как описано в разделе 9.5.3.

Когда задача-отправитель и задача-получатель выполняются в одном и том же процессе TaskManager, отправитель сериализует исходящие записи в байтовый буфер и помещает буфер в очередь сразу после его заполнения. Принимающая задача берет буфер из очереди и десериализует входящие записи. Следовательно, передача данных между задачами, выполняемыми в одном TaskManager, не вызывает сетевого взаимодействия.

---

[1] Пакетные приложения могут – помимо конвейерной связи – обмениваться данными, собирая исходящие данные у отправителя. После завершения задачи отправителя данные отправляются в виде пакета по временному TCP-соединению к получателю.

Flink предлагает различные методы для снижения затрат на связь между задачами. В следующих разделах мы кратко обсудим кредитное управление потоком и цепочку задач.

3.2.1. Кредитное управление потоком

Отправка отдельных записей через сетевое соединение неэффективна и вызывает значительные накладные расходы. Буферизация необходима для полного использования пропускной способности сетевых подключений. В контексте потоковой обработки одним из недостатков буферизации является то, что она увеличивает задержку, поскольку записи собираются в буфере, а не отправляются сразу.

Flink реализует механизм *кредитного управления потоком* (credit-based flow control), который работает следующим образом. Задача-получатель предоставляет задаче-отправителю некоторый кредит – количество сетевых буферов, зарезервированных для приема ее данных. Как только отправитель получает уведомление о кредите, он отправляет столько буферов, сколько ему было предоставлено в кредите, и размер своего неисполненного журнала – количество сетевых буферов, которые заполнены и готовы к отправке, но не уместились в кредит. Получатель обрабатывает отправленные данные с помощью зарезервированных буферов и учитывает размер неисполненного журнала отправителя для определения приоритета следующих кредитов для всех подключенных отправителей.

Управление потоком на основе кредита уменьшает задержку, поскольку отправители могут отправлять данные, как только у получателя будет достаточно ресурсов для их приема. Более того, это эффективный механизм для распределения сетевых ресурсов в случае неравномерного распределения данных, поскольку кредит предоставляется в зависимости от размера отставания отправителей. Следовательно, управление потоком на основе кредита является важным компонентом Flink для достижения высокой пропускной способности и низкой задержки.

3.2.2. Цепочка задач

Flink предлагает метод оптимизации, называемый *цепочкой задач* (task chaining), который снижает накладные расходы на локальную связь при определенных условиях. Чтобы успешно сформировать цепочку задач, два или более операторов должны быть настроены на одинаковый параллелизм и соединены локальными прямыми каналами. Конвейер операторов, показанный на рис. 3.5, удовлетворяет этим требованиям. Он состоит из трех операторов, которые настроены на параллельное выполнение двух задач и связаны локальными прямыми соединениями.

На рис. 3.6 показано, как выполняется конвейер в случае цепочки задач. Функции операторов объединены в единую задачу, которая выполняется одним потоком. Записи, создаваемые функцией, по отдельности передаются следующей функции с помощью простого вызова метода. Следовательно,

при передаче записей между функциями практически отсутствуют затраты на сериализацию и связь.

Создание цепочки задач может значительно снизить затраты на обмен данными между локальными задачами, но также есть случаи, когда имеет смысл выполнить конвейер без цепочки. Например, имеет смысл разбить длинный конвейер связанных задач или разбить цепочку на две задачи, чтобы запланировать дорогостоящую функцию для разных слотов. На рис. 3.7 показан тот же конвейер, выполняемый без объединения задач. Все функции реализуются отдельной задачей, выполняемой в выделенном потоке.

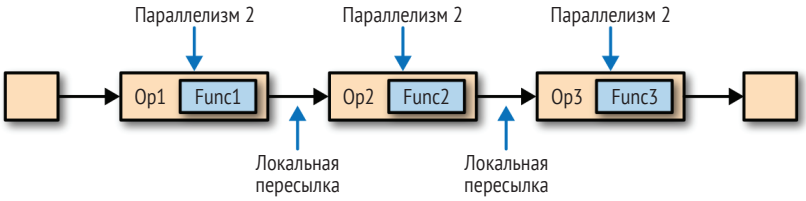

Рис. 3.5 ❖ Конвейер операторов, соответствующий требованиям цепочки задач

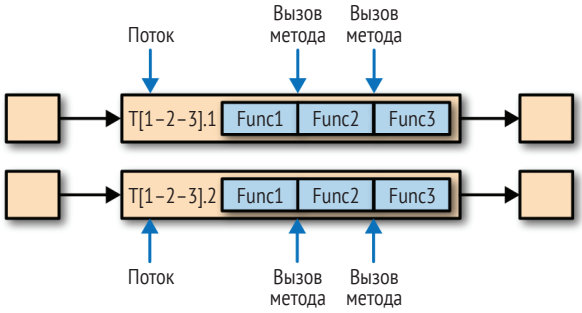

Рис. 3.6 ❖ Выполнение цепочки задач с функциями, объединенными в один поток, и передача данных через вызовы методов

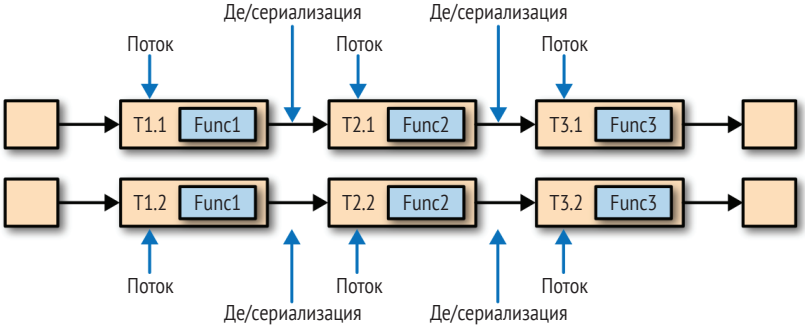

Рис. 3.7 ❖ Несвязанное выполнение задач с выделенными потоками и транспортировкой данных через буферные каналы и сериализацию

По умолчанию в Flink включен режим цепочки задач. В разделе 10.2.1 мы покажем, как отключить цепочку задач для приложения и как управлять цепочкой действий отдельных операторов.

3.3. Обработка на основе времени события

В разделе 2.3 мы подчеркнули важность семантики времени для приложений потоковой обработки и объяснили разницу между временем обработки и временем события. Хотя время обработки легко понять, поскольку оно основано на местном времени обрабатывающей машины, оно дает несколько произвольные, непоследовательные и невоспроизводимые результаты. Напротив, семантика времени события дает воспроизводимые и согласованные результаты, что является жестким требованием для многих случаев использования потоковой обработки. Однако приложения с привязкой к времени события требуют дополнительной настройки по сравнению с приложениями, привязанными ко времени обработки. Кроме того, внутренние компоненты потокового процессора, поддерживающего время события, более нагружены, чем внутренние компоненты системы, которая задействована исключительно во время обработки.

Flink предоставляет интуитивно понятные и простые в использовании примитивы для распространенных операций обработки событий с привязкой ко времени, а также выразительные API-интерфейсы для реализации более сложных приложений с привязкой к времени событий с настраиваемыми операторами. Для таких сложных приложений часто полезно, а иногда и необходимо хорошее понимание внутренней обработки времени Flink. В предыдущей главе были представлены две концепции, которые Flink использует для обеспечения семантики времени события: отметки времени записи и водяные знаки. Далее мы расскажем про внутреннюю реализацию и обработку меток времени и водяных знаков для поддержки потоковых приложений с семантикой времени события.

3.3.1. Метки времени

Все записи, которые обрабатываются потоковым приложением Flink с привязкой ко времени событий, должны сопровождаться меткой времени. *Метка времени* связывает запись с определенным моментом времени, обычно с моментом времени, когда произошло событие, представленное записью. Однако приложения могут свободно выбирать значение меток времени до тех пор, пока метки времени записей потока примерно возрастают по мере продвижения потока. Как следует из раздела 2.3, некоторая степень неупорядоченности меток времени является неизбежной данностью практически во всех реальных случаях использования.

Когда Flink обрабатывает поток данных в режиме событийного времени, он выполняет зависящие от времени операторы на основе меток времени.

Например, оператор временного окна назначает записи окнам в соответствии с их метками времени. Flink кодирует метки времени как 16-байтовые значения `Long` и прикрепляет их как метаданные к записям. Его встроенные операторы интерпретируют значение `Long` как метку времени Unix с точностью до миллисекунды – количество миллисекунд с 1970-01-01-00:00:00.000. Однако пользовательские операторы могут иметь собственную интерпретацию и, например, оперировать точностью до микросекунд.

3.3.2. Водяные знаки

В дополнение к записи меток времени приложение событийного времени Flink также должно предоставлять *водяные знаки* (watermark). Водяные знаки используются для определения текущего времени события для каждой задачи в приложении событийного времени. Операторы, зависящие от времени, используют это время для запуска вычислений и выполнения следующих шагов. Например, задача временного окна завершает вычисление окна и выдает результат, когда время события задачи проходит границу конца окна.

Во Flink водяные знаки реализованы как специальные записи, содержащие метку времени в виде значения `Long`. Водяные знаки плывут в потоке обычных записей с аннотированными отметками времени, как показано на рис. 3.8.

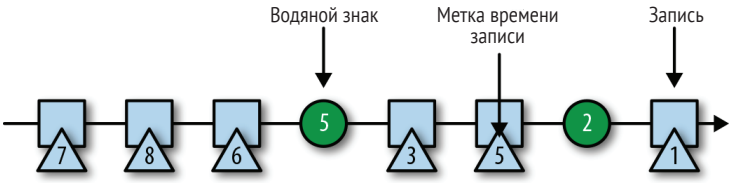

Рис. 3.8 ❖ Поток записей с метками времени и водяными знаками

Водяные знаки обладают двумя основными свойствами:

1) они должны монотонно увеличиваться, чтобы часы выполнения задач шли вперед, а не откатывались в обратном направлении;
2) они связаны с метками времени записи. Водяной знак с меткой времени T указывает, что все последующие записи должны иметь метки времени > T.

Второе свойство используется для обработки потоков с неупорядоченными метками времени записи, такими как записи с метками времени 3 и 5 на рис. 3.8. Задачи операторов, основанных на времени, собирают и обрабатывают записи с возможно неупорядоченными метками времени и завершают вычисление, когда их часы времени события, которые опережают полученные водяные знаки, показывают, что больше не ожидается записей с соответствующими отметками времени. Когда задача получает запись, которая нарушает ограничение водяного знака и имеет меньшую метку времени,

чем ранее полученный водяной знак, возможно, вычисление, которому она принадлежит, уже завершено. Такие записи называются просроченными. Flink предоставляет различные способы работы с просроченными записями, которые обсуждаются в разделе 6.5.

Интересным свойством водяных знаков является то, что они позволяют приложению контролировать полноту результатов и задержку. Очень строгие водяные знаки – близкие к меткам времени записи – приводят к низкой задержке обработки, потому что задача лишь на короткое время ожидает поступления новых записей перед завершением вычисления. В то же время может пострадать полнота результата, потому что соответствующие записи могут не попасть в результат и будут считаться просроченными. И наоборот, очень консервативные водяные знаки увеличивают задержку обработки, но улучшают полноту результата.

3.3.3. Распространение водяного знака и время события

В этом разделе мы обсудим, как операторы обрабатывают водяные знаки. Flink реализует водяные знаки как специальные записи, которые принимаются и отправляются задачами оператора. Задачи имеют внутреннюю службу времени, которая поддерживает таймеры и активируется при получении водяного знака. Задачи могут регистрировать таймеры в службе таймера для выполнения вычислений в определенный момент времени в будущем. Например, оконный оператор регистрирует таймер для каждого активного окна, который очищает состояние окна, когда время события превышает время окончания окна.

Когда задача получает водяной знак, выполняются следующие действия:

1) задача обновляет свои внутренние часы времени событий на основе метки времени водяного знака;
2) служба времени задачи выделяет все таймеры со временем, меньшим, чем обновленное время события. Для каждого истекшего таймера задача вызывает функцию обратного вызова, которая может выполнять вычисления и выдавать записи;
3) задача выдает водяной знак с обновленным временем события.

Flink ограничивает доступ к отметкам времени или водяным знакам через API DataStream. Функции не могут читать или изменять метки времени записи и водяные знаки, за исключением функций процесса, которые могут считывать метку времени текущей обрабатываемой записи, запрашивать текущее время события у оператора и регистрировать таймеры[1]. Ни одна из функций не предоставляет API для установки метки времени отправленных записей, манипулирования часами времени события задачи или выпуска водяных знаков. Вместо этого задачи оператора DataStream с привязкой ко времени настраивают временные метки отправляемых записей, чтобы гарантировать, что они правильно совмещены с отправляемыми водяными знаками. Например, зада-

[1] Функции процесса более детально рассмотрены в главе 6.

ча оператора временного окна прикрепляет время окончания окна в качестве метки времени ко всем записям, выдаваемым вычислением окна, прежде чем она выпускает водяной знак с меткой времени, которая инициировала вычисление окна.

Теперь давайте более подробно объясним, как задача генерирует водяные знаки и обновляет свои часы времени события при получении нового водяного знака. Как вы видели в разделе 2.1.2, Flink разбивает потоки данных на *разделы* (partition) и обрабатывает каждый раздел параллельно с помощью отдельной задачи оператора. Каждый раздел представляет собой независимый поток записей с метками времени и водяными знаками. В зависимости от того, как оператор связан со своими предшественниками или преемниками, его задачи могут получать записи и водяные знаки из одного или нескольких входных разделов и передавать записи и водяные знаки в один или несколько выходных разделов. Далее мы подробно расскажем, как задача передает водяные знаки нескольким задачам вывода и как она производит отсчет времени события по водяным знакам, которые она получает от своих задач ввода.

Задача поддерживает водяной знак раздела для каждого входного раздела. Когда она получает водяной знак от раздела, она обновляет водяной знак соответствующего раздела, чтобы он был максимальным из полученного значения и текущего значения. Впоследствии задача обновляет свои часы времени событий до минимума всех водяных знаков раздела. Если часы времени события продвигаются вперед, задача обрабатывает все сработавшие таймеры и наконец транслирует свое новое время события всем последующим задачам, отправляя соответствующий водяной знак всем подключенным выходным разделам.

На рис. 3.9 показано, как задача с четырьмя входными разделами и тремя выходными разделами получает водяные знаки, обновляет водяные знаки своего раздела и часы времени событий и выпускает водяные знаки.

Задачи операторов с двумя или более входными потоками, таких как `Union` или `CoFlatMap` (см. раздел 5.2.3), также вычисляют свои часы, определяемые временем события, как минимум всех водяных знаков разделов – они не различают водяные знаки разделов разных входных потоков. Следовательно, записи обоих входов обрабатываются на основе одних и тех же часов события. Такое поведение может вызвать проблемы, если время событий отдельных входных потоков приложения не выровнено.

Алгоритм обработки водяных знаков и распространения Flink гарантирует, что задачи оператора будут выдавать правильно выровненные записи с метками времени и водяные знаки. Однако он основан на том факте, что все разделы непрерывно предоставляют увеличивающиеся водяные знаки. Как только один из разделов перестанет отправлять свои водяные знаки или станет полностью бездействующим, часы задачи не возрастут и таймеры задачи не сработают. Эта ситуация может быть проблематичной для операторов, зависящих от времени, которые полагаются на возрастающие часы для выполнения вычислений и очистки своего состояния. Следовательно, задержки обработки и размер состояния операторов, зависящих от времени, могут значительно увеличиться, если задача не получает новые водяные знаки от всех задач ввода через равные промежутки времени.

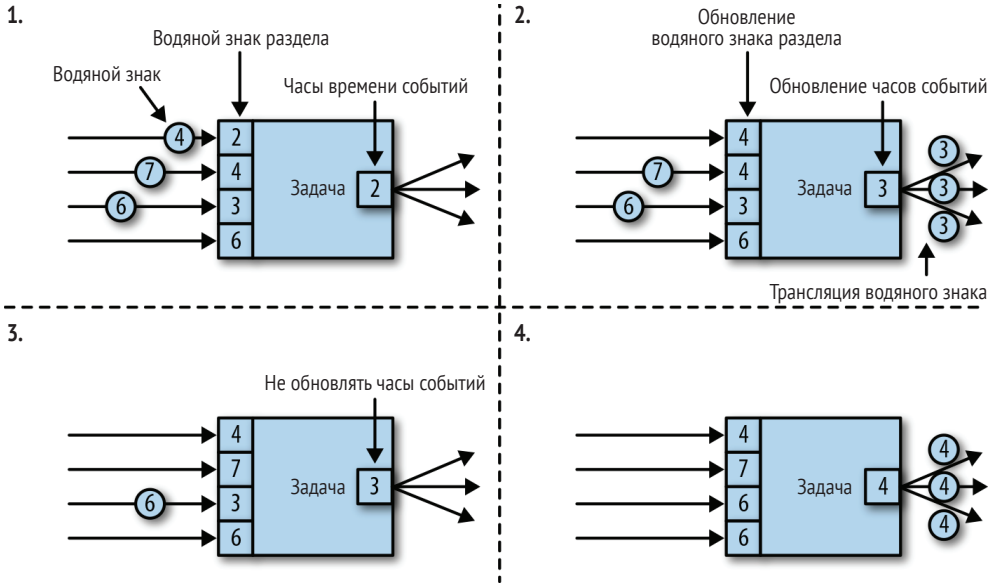

Рис. 3.9 ❖ Обновление времени события задачи с помощью водяных знаков

Аналогичный эффект появляется для операторов с двумя входными потоками, водяные знаки которых значительно расходятся. Часы времени события задачи с двумя входными потоками будут соответствовать водяным знакам более медленного потока, и обычно записи или промежуточные результаты более быстрого потока хранятся в состоянии до тех пор, пока часы времени события не позволят их обработать.

3.3.4. Назначение метки времени и создание водяных знаков

Итак, мы объяснили, что такое метки времени и водяные знаки и как они обрабатываются внутри Flink. Однако мы еще не обсуждали их происхождение. Метки времени и водяные знаки обычно назначаются и генерируются, когда поток загружается потоковым приложением. Поскольку выбор метки времени зависит от приложения, а водяные знаки зависят от меток времени и характеристик потока, приложения должны явно назначать метки времени и создавать водяные знаки. Приложение Flink DataStream может назначать метки времени и создавать водяные знаки для потока тремя способами.

1. *У источника*: метки времени и водяные знаки могут быть назначены и сгенерированы функцией `SourceFunction`, когда поток загружается в приложение. Функция источника генерирует поток записей. Записи могут быть сгенерированы вместе с соответствующей меткой времени, а водяные знаки могут быть сгенерированы в любой момент времени как специальные записи. Если функция источника (временно)

больше не генерирует водяные знаки, она может объявить себя бездействующей. Flink будет исключать разделы потока, созданные простаивающими функциями источника, из вычисления водяных знаков последующих операторов. Механизм бездействия источников можно использовать для решения проблемы отсутствия продвижения водяных знаков, как обсуждалось ранее. Функции источника более подробно обсуждаются в разделе 8.3.

2. *Периодическое присваивание*: API DataStream предоставляет определяемую пользователем функцию `AssignerWithPeriodicWatermarks`, которая извлекает метку времени из каждой записи и периодически запрашивает текущий водяной знак. Извлеченные метки времени назначаются соответствующей записи, а запрошенные водяные знаки вставляются в поток. Эта функция будет рассмотрена в разделе 6.1.1.
3. *Прерывистое присваивание*: `AssignerWithPunctuatedWatermarks` – еще одна определяемая пользователем функция, которая извлекает метку времени из каждой записи. Ее можно использовать для создания водяных знаков, которые закодированы в специальных входных записях. В отличие от функции `AssignerWithPeriodicWatermarks` эта функция может – но не обязательно – извлекать водяной знак из каждой записи. Мы подробно обсуждаем эту функцию в разделе 6.1.1.

Пользовательские функции присвоения меток времени обычно применяются как можно ближе к оператору-источнику, потому что может быть очень сложно определить порядок записей и их метки времени после того, как они были обработаны оператором. Это также причина того, почему не рекомендуется переопределять существующие метки времени и водяные знаки в середине потокового приложения, хотя это реализуемо с помощью пользовательских функций.

3.4. Управление состоянием

В главе 2 мы говорили о том, что большинство потоковых приложений поддерживают состояние. Многие операторы постоянно читают и обновляют какое-либо состояние, такое как записи, собранные в окне, позиции чтения источника ввода или настраиваемые состояния операторов для конкретного приложения, например модели машинного обучения. Flink обрабатывает все состояния – независимо от встроенных или определяемых пользователем операторов – одинаково. В этом разделе мы обсуждаем различные типы состояний, которые поддерживает Flink. Мы расскажем, как состояние хранится и поддерживается бэкендами состояния и как приложения с учетом состояния можно масштабировать с помощью перераспределения состояния.

В общем, все данные, поддерживаемые задачей и используемые для вычисления результатов функции, относятся к состоянию задачи. Вы можете воспринимать состояние как локальную переменную или переменную экземпляра, к которой обращается бизнес-логика задачи. На рис. 3.10 показано типичное взаимодействие между задачей и ее состоянием.

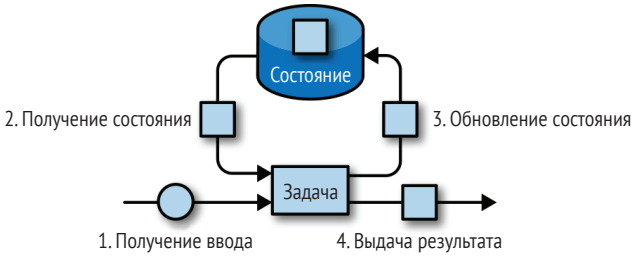

Рис. 3.10 ❖ Задача обработки потока с учетом состояния

Задача получает некоторые входные данные. При обработке данных задача может считывать и обновлять свое состояние и вычислять свой результат на основе входных данных и состояния. Простой пример – задача, которая постоянно считает, сколько записей она получает. Когда задача получает новую запись, она обращается к состоянию, чтобы получить текущий счетчик, увеличивает счетчик, обновляет состояние и выдает новый счетчик.

Логика приложения для чтения и записи в состояние часто проста. Однако эффективное и надежное управление состоянием является более сложной задачей. Оно включает в себя обработку очень больших состояний, возможно, превышающих объем памяти и обеспечение сохранности состояния в случае сбоев. Flink берет на себя все вопросы, связанные с согласованностью состояния, обработкой сбоев, эффективным хранением и доступом, так что разработчики могут сосредоточиться на логике своих приложений.

Во Flink состояние всегда связано с определенным оператором. Чтобы среда выполнения Flink знала о состоянии оператора, оператор должен зарегистрировать свое состояние. Существует два типа состояния – *состояние оператора* (operator state) и *состояние с ключевым доступом* (keyed state), которые доступны из разных областей и обсуждаются в следующих разделах.

3.4.1. Состояние оператора

Состояние оператора привязано к задаче оператора. Это означает, что все записи, обрабатываемые одной параллельной задачей, имеют доступ к одному и тому же состоянию. Состояние оператора не может быть доступно другой задаче того же или другого оператора. На рис. 3.11 показано, как задачи получают доступ к состоянию оператора.

Flink предлагает три примитива для состояния оператора.

Списочное состояние (list state)

Представляет состояние в виде списка записей.

Каталожное состояние (union list state)

Также представляет состояние в виде списка записей. Но оно отличается от обычного списочного состояния тем, как оно восстанавливается в случае сбоя или когда приложение запускается из точки сохранения. Мы обсудим это различие позже в этой главе.

Широковещательное состояние (broadcast state)

Предназначено для особого случая, когда состояние каждой задачи оператора идентично. Это свойство можно использовать во время обслуживания контрольных точек и при изменении масштаба оператора. Оба аспекта обсуждаются в следующих разделах этой главы.

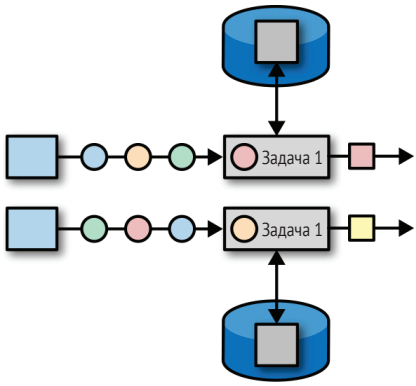

Рис. 3.11 ❖ Задачи с состоянием оператора

3.4.2. Состояние с ключевым доступом

Обращение к состоянию с ключевым доступом осуществляется с использованием ключа, определенного в записях входного потока оператора. Flink поддерживает один экземпляр состояния для каждого значения ключа и разделяет все записи с одним и тем же ключом для задачи оператора, которая поддерживает состояние для этого ключа. Когда задача обрабатывает запись, она автоматически связывает доступ состояния с ключом текущей записи. Следовательно, все записи с одним и тем же ключом обращаются к одному и тому же состоянию. На рис. 3.12 показано, как задачи взаимодействуют с ключевым доступом к состоянию.

Вы можете думать о состоянии с ключевым доступом как о карте «ключ–значение», которая разделена (или сегментирована) по ключу для всех параллельных задач оператора. Flink предоставляет различные примитивы для состояния с ключевым доступом, которые определяют тип значения, хранимого для каждого ключа в этой распределенной карте «ключ–значение». Мы кратко обсудим наиболее распространенные примитивы.

Одиночное значение (value state)

Хранит одно значение произвольного типа для каждого ключа. Сложные структуры данных также могут быть сохранены как состояние значения.

Список значений (list state)

Хранит список значений для каждого ключа. Записи в списке могут быть произвольного типа.

Карта значений (map state)

Хранит карту «ключ–значение» для каждого ключа. Ключ и значение карты могут быть произвольного типа.

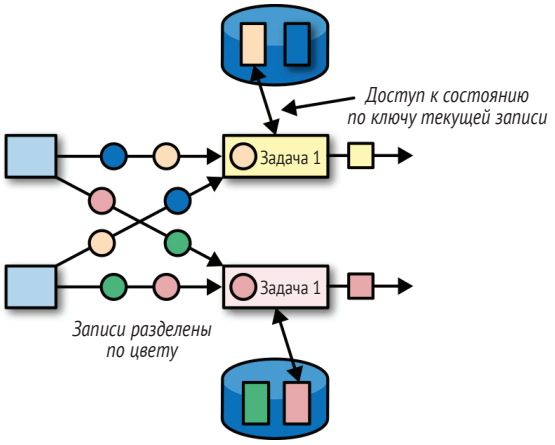

Рис. 3.12 ❖ Задачи с ключевым доступом к состоянию

Примитивы состояния открывают структуру состояния для Flink и обеспечивают более эффективный доступ к состоянию. Они обсуждаются далее в разделе 7.1.1.

3.4.3. Бэкенд состояния

Задача оператора с учетом состояния обычно считывает и обновляет свое состояние для каждой входящей записи. Поскольку эффективный доступ к состоянию имеет решающее значение для обработки записей с малой задержкой, каждая параллельная задача локально поддерживает свое состояние для обеспечения быстрого доступа к состоянию. Как именно состояние сохраняется, к которому осуществляется доступ и поддерживается, определяется подключаемым компонентом, который называется *бэкендом состояния* (state backend). Бэкенд состояния отвечает за две вещи: управление локальным состоянием и состояние контрольной точки в удаленном месте.

Что касается управления локальным состоянием, бэкенд состояния хранит все состояния с ключевым доступом и гарантирует, что все обращения правильно привязаны к текущему ключу. Flink предоставляет бэкенды состояния, которые управляют состоянием с ключом как объектом, хранящимся в структурах данных в памяти в куче (heap) JVM. Другой вариант бэкенда состояния сериализует объекты состояния и помещает их в RocksDB, которая записывает их на локальные жесткие диски. Хотя первый вариант обеспечивает очень быстрый доступ к состоянию, он ограничен размером памяти. Доступ к состоянию, сохраненному бэкендом RocksDB, происходит медленнее, но зато это состояние может стать очень большим.

Контрольные точки состояния важны, потому что Flink – это распределенная система, и состояние поддерживается только локально. Процесс TaskManager (а вместе с ним и все выполняемые в нем задачи) может выйти из строя в любой момент времени. Следовательно, его хранилище следует считать уязвимым. Бэкенд состояния заботится о том, чтобы контрольные точки состояния задачи передавались в удаленное и надежное хранилище. Удаленное хранилище для контрольных точек может быть распределенной файловой системой или системой баз данных. Бэкенды состояния различаются по способу проверки состояния. Например, бэкенд состояния RocksDB поддерживает инкрементные контрольные точки, что может значительно снизить накладные расходы при установке контрольных точек для состояний очень большого размера.

Мы обсудим различные бэкенды состояния, их преимущества и недостатки более подробно в разделе 7.4.1.

3.4.4. Масштабирование операторов с учетом состояния

Обычным требованием для потоковых приложений является подстройка параллелизма операторов при увеличении или уменьшении скорости ввода. В то время как масштабирование операторов без состояния тривиально, изменение параллелизма операторов с сохранением состояния намного сложнее, поскольку их состояние необходимо перераспределить и назначить большему или меньшему количеству параллельных задач. Flink поддерживает четыре шаблона масштабирования различных типов состояния.

Операторы с ключевым доступом к состоянию масштабируются путем перераспределения ключей на меньшее или большее количество задач. Однако для повышения эффективности передачи необходимого состояния между задачами Flink не перераспределяет отдельные ключи. Вместо этого Flink объединяет ключи в так называемые группы ключей. *Группа ключей* (key group) – это раздел ключей и способ Flink назначать ключи задачам. На рис. 3.13 показано, как состояние с ключевым доступом перераспределяется по группам ключей.

Операторы со списком состояний масштабируются путем перераспределения записей списка. По идее, записи списка всех параллельных задач оператора собираются и равномерно перераспределяются между меньшим или большим количеством задач. Если записей в списке меньше, чем новый параллелизм оператора, некоторые задачи начнутся с пустого состояния. На рис. 3.14 показано перераспределение списочного состояния оператора.

Операторы с каталожным состоянием масштабируются путем широковещательной рассылки полного списка записей состояния каждой задаче. Затем задача может выбрать, какие записи использовать, а какие отбросить. На рис. 3.15 показано, как перераспределяется каталожное состояние оператора.

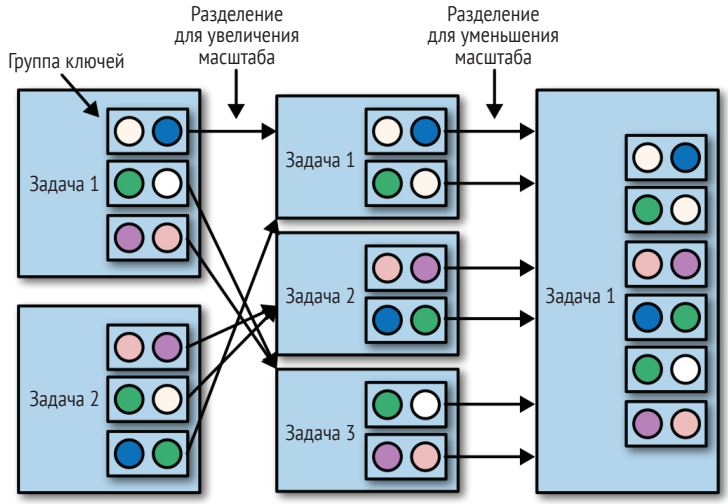

Рис. 3.13 ❖ Масштабирование оператора с ключевым доступом к состоянию

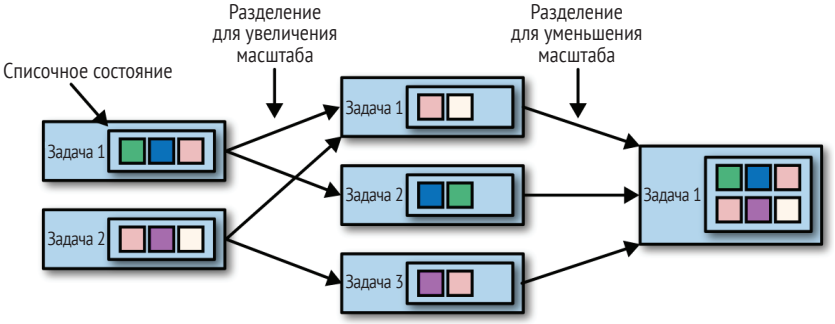

Рис. 3.14 ❖ Масштабирование оператора со списочным состоянием

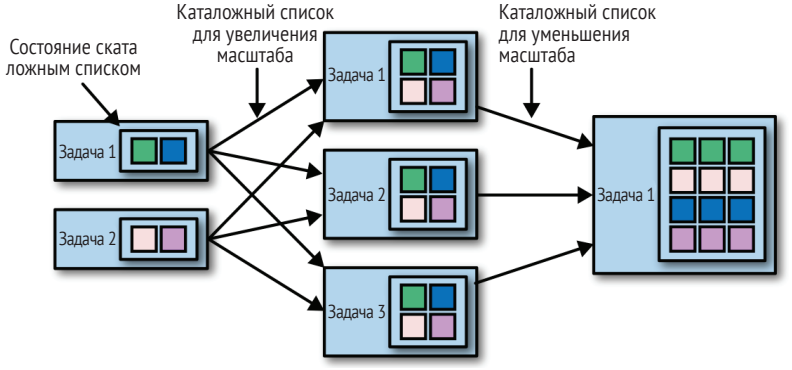

Рис. 3.15 ❖ Масштабирование оператора с каталожным состоянием

Операторы с широковещательным состоянием масштабируются путем копирования состояния в новые задачи. Это работает, потому что широковещательное состояние гарантирует, что все задачи имеют одинаковое состояние. В случае масштабирования избыточные задачи просто отменяются, поскольку состояние уже реплицировано и не будет потеряно. На рис. 3.16 показано перераспределение широковещательного состояния оператора.

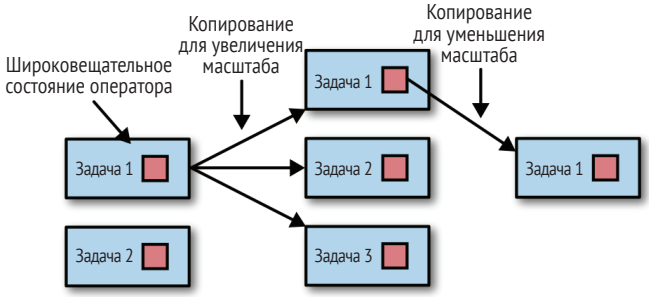

Рис. 3.16 ❖ Масштабирование оператора с широковещательным состоянием

3.5. Контрольные точки, точки сохранения и восстановление состояния

Flink – это распределенная система обработки данных, и поэтому она имеет дело со сбоями, такими как остановленные процессы, сбойные машины и прерванные сетевые соединения. Поскольку задачи поддерживают свое состояние локально, Flink должен гарантировать, что это состояние не будет потеряно и останется неизменным в случае сбоя.

В этом разделе мы представляем механизм контрольных точек и восстановления Flink, гарантирующий согласованность состояния «ровно один раз». Мы также расскажем про уникальную функцию точки сохранения Flink – инструмент, похожий на «швейцарский армейский нож», который решает многие проблемы работы с потоковыми приложениями.

3.5.1. Согласованные контрольные точки

Механизм восстановления Flink основан на сохранении состояния приложения в контрольных точках. *Согласованная контрольная точка* (consistent checkpoint) приложения потоковой передачи с учетом состояния – это согласованная копия состояния каждой из его задач в момент, когда все задачи обработали один и тот же ввод. Это понятие можно объяснить на примере шагов наивного алгоритма, который использует согласованную контрольную точку приложения. Шаги этого наивного алгоритма будут следующими.

1. Приостановите прием всех входных потоков.
2. Подождите, пока все задействованные данные будут полностью обработаны, то есть все задачи обработают все свои входные данные.
3. Создайте контрольную точку, скопировав состояние каждой задачи в удаленное постоянное хранилище. Контрольная точка считается созданной, когда все задачи завершили свои копии.
4. Возобновите прием всех потоков.

Имейте в виду, что Flink не реализует этот наивный механизм. Позже в этом разделе мы представим более сложный алгоритм контрольной точки Flink.

На рис. 3.17 показана согласованная контрольная точка простого приложения.

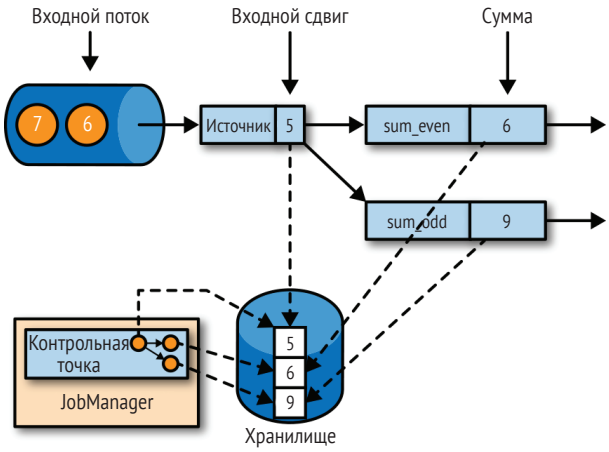

Рис. 3.17 ❖ Согласованная контрольная точка потокового приложения

Приложение имеет единую исходную задачу, которая потребляет поток возрастающих чисел – 1, 2, 3 и т. д. Поток чисел делится на поток четных и нечетных чисел. Две задачи оператора суммы вычисляют текущие суммы всех четных и нечетных чисел. Исходная задача хранит текущее смещение входного потока как состояние. Суммирующие задачи сохраняют текущее значение суммы как состояние. На рис. 3.17 Flink создал контрольную точку, когда входное смещение было 5, а суммы были 6 и 9.

3.5.2. Восстановление из сохраняющей контрольной точки

Во время выполнения потокового приложения Flink периодически создает согласованные контрольные точки состояния приложения. В случае сбоя Flink использует последнюю контрольную точку для согласованного вос-

становления состояния приложения и перезапускает обработку. На рис. 3.18 показан процесс восстановления.

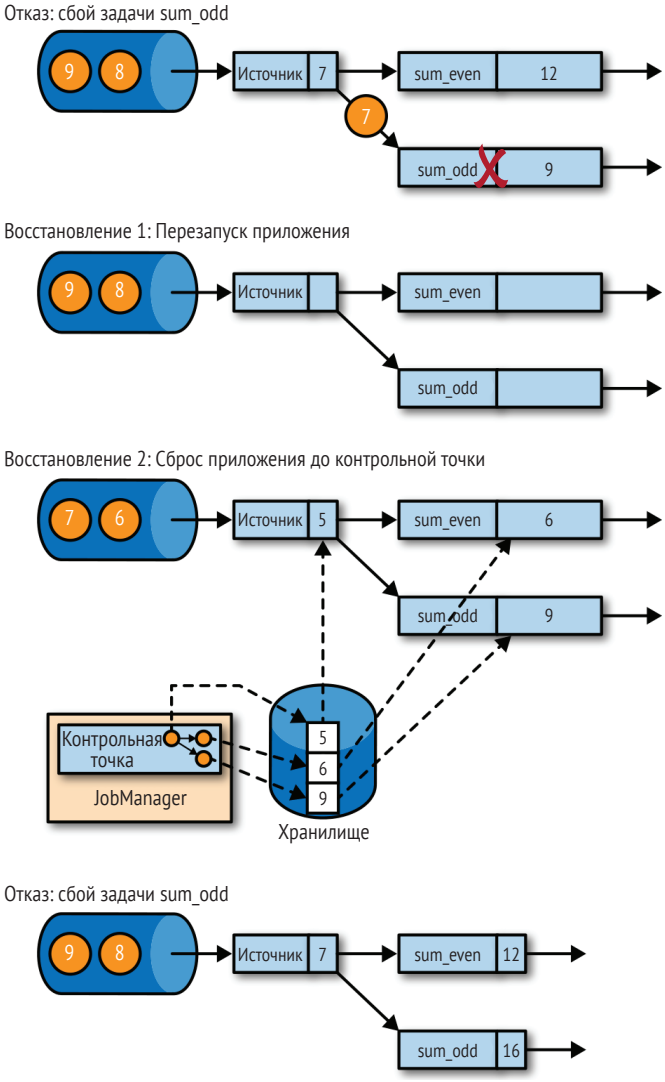

Рис. 3.18 ❖ Восстановление приложения из согласованной контрольной точки

Приложение восстанавливается в три этапа.

1. Перезапуск всех приложений.
2. Сброс состояния всех задач, учитывающих состояние, до последней контрольной точки.
3. Возобновление обработки всех задач.

Этот механизм расстановки контрольных точек и последующего восстановления обеспечивает выполнение приложения «ровно один раз», поскольку все операторы сохраняют и восстанавливают все свои состояния, а все входные потоки сбрасываются в положение, до которого они были использованы при установке контрольной точки. Может ли источник данных сбросить свой входной поток, зависит от его реализации и внешней системы или интерфейса, из которого поток потребляется. Например, журналы событий, такие как Apache Kafka, могут предоставлять записи потока со сдвигом назад. Напротив, поток, потребляемый из сокета, не может быть сброшен, потому что сокеты удаляют данные сразу после того, как они были использованы. Следовательно, приложение может реализовать обработку «ровно один раз» только в том случае, если все входные потоки предоставлены сбрасываемыми (отматываемыми назад) источниками данных.

После перезапуска приложения из контрольной точки его внутреннее состояние точно такое же, как и в момент установки контрольной точки. Затем приложение начинает потреблять и обрабатывать все данные, которые были обработаны между контрольной точкой и моментом сбоя. Хотя фактически Flink обрабатывает некоторые сообщения дважды (до и после сбоя), механизм по-прежнему обеспечивает обработку «ровно один раз», потому что состояние всех операторов было сброшено до точки, для которой эти данные еще не существовали.

Мы должны отметить, что механизм контрольных точек и восстановления Flink сбрасывает только *внутреннее* состояние потокового приложения. В зависимости от операторов приемника приложения во время восстановления некоторые записи результатов могут быть отправлены несколько раз в нижестоящие системы, такие как журнал событий, файловая система или база данных. Для некоторых систем хранения Flink предоставляет функции приемника, которые обеспечивают вывод типа «ровно один раз», например, путем фиксации отправленных записей при завершении контрольной точки. Другой подход, который работает для многих систем хранения, – это идемпотентные обновления. Проблемы сквозных приложений с обработкой «ровно один раз» и подходы к их решению подробно обсуждаются в разделе 8.1.

3.5.3. Алгоритм создания контрольной точки Flink

Механизм восстановления Flink основан на согласованных контрольных точках приложения. Наивный подход к созданию контрольной точки из потокового приложения – приостановка, запись контрольной точки и возобновление работы приложения – неприемлем для приложений, которые имеют даже умеренные требования к задержке из-за его поведения в стиле «остановите мир, я сойду». Вместо этого Flink реализует контрольную точку на основе *алгоритма Ченди–Лампорта* (Chandy-Lamport algorithm) для распределенных снимков. Алгоритм не приостанавливает работу всего приложения, а отделяет контрольную точку от обработки, так что некоторые задачи продолжают обработку, а другие сохраняют свое состояние. Далее мы объясним, как работает этот алгоритм.

Алгоритм контрольной точки Flink использует специальный тип записи, называемый *барьером контрольной точки* (checkpoint barrier). Подобно водяным знакам, барьеры контрольных точек вводятся операторами источника в обычный поток записей, и другие записи не могут обогнать или преодолеть этот барьер. Барьер контрольной точки содержит идентификатор контрольной точки для определения контрольной точки, которой он принадлежит, и логически разделяет поток на две части. Все изменения состояния из-за записей, которые предшествуют барьеру, включаются в контрольную точку текущего барьера, а все изменения из-за записей, следующих за барьером, включаются в более позднюю контрольную точку.

Для пошагового объяснения алгоритма мы воспользуемся примером простого потокового приложения. Приложение состоит из двух входных задач, каждая из которых потребляет поток возрастающих чисел. Вывод входных задач разделяется на потоки четных и нечетных чисел. Каждый раздел обрабатывается задачей, которая вычисляет сумму всех полученных чисел и пересылает обновленную сумму в приемник. Приложение схематически изображено на рис. 3.19.

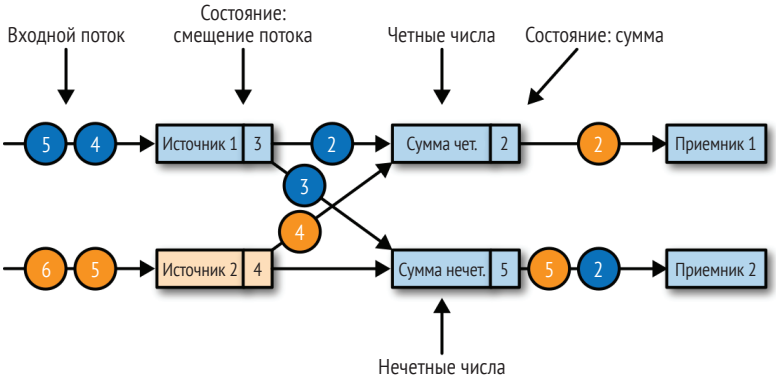

Рис. 3.19 ❖ Потоковое приложение с двумя источниками с сохранением состояния, двумя задачами с сохранением состояния и двумя приемниками без сохранения состояния

Контрольная точка инициируется процессом JobManager путем отправки сообщения с новым идентификатором контрольной точки каждой задаче источника данных, как показано на рис. 3.20.

Когда задача источника данных получает сообщение, она приостанавливает отправку записей, запускает запись контрольную точку своего локального состояния на бэкенде состояния и транслирует барьеры контрольной точки с идентификатором контрольной точки через все разделы исходящего потока. Бэкенд состояния уведомляет задачу, когда ее контрольная точка состояния завершена, а задача подтверждает контрольную точку в JobManager. После того как все барьеры обработаны, источник продолжает свою обычную работу. Внедряя барьер в свой выходной поток, функция источника определяет позицию потока, на которой ставится контрольная точка. На рис. 3.21

показано потоковое приложение после того, как обе входные задачи установили контрольные точки своего локального состояния и сгенерировали барьеры контрольных точек.

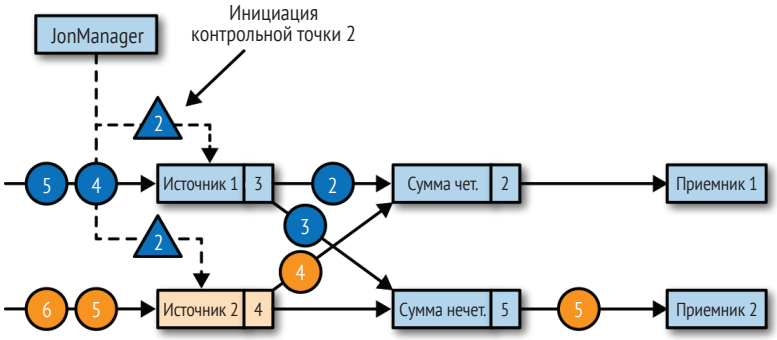

Рис. 3.20 ❖ JobManager инициирует создание контрольной точки, отправляя сообщение всем источникам

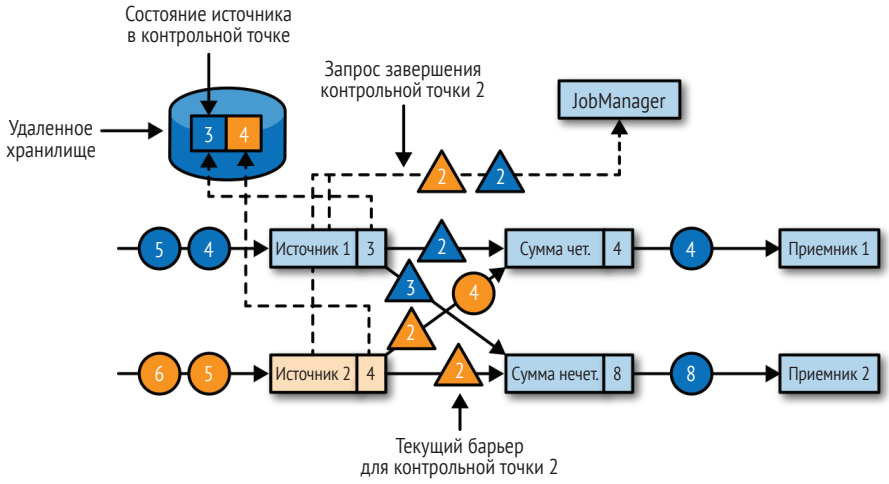

Рис. 3.21 ❖ Источники проверяют свое состояние и создают барьер контрольной точки

Барьеры контрольных точек, создаваемые входными задачами, доставляются подключенным задачам. Подобно водяным знакам, барьеры контрольных точек транслируются на все подключенные параллельные задачи, чтобы гарантировать, что каждая задача получит барьер из каждого из своих входных потоков. Когда задача получает барьер для новой контрольной точки, она ожидает прибытия барьеров со всех своих входных разделов для контрольной точки. Во время ожидания она продолжает обрабатывать записи из потоковых разделов, которые еще не создали барьер. Записи, поступающие в разделы, которые уже передали барьер, теперь не могут быть обработаны

и помещаются в буфер. Процесс ожидания прибытия всех барьеров называется *выравниванием барьера* (barrier alignment) и изображен на рис. 3.22.

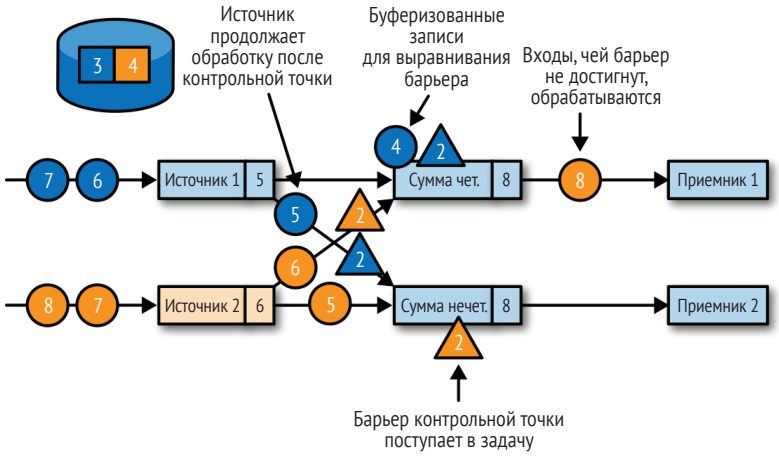

Рис. 3.22 ❖ Задачи ожидают получения барьера на каждом входном разделе; записи из входных потоков, для которых уже достигнут барьер, буферизуются; все остальные записи обрабатываются как обычно

Как только задача получила барьеры от всех входных разделов, она инициирует запись контрольной точки на бэкенде состояния и транслирует барьер контрольной точки на все связанные с ней нижестоящие задачи, как показано на рис. 3.23.

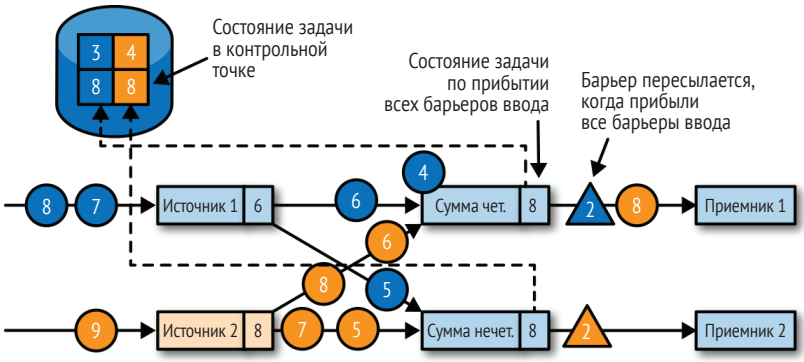

Рис. 3.23 ❖ Задачи сохраняют в контрольную точку свое состояние после получения всех барьеров, затем они пересылают барьер контрольной точки

После того как все барьеры контрольной точки были переданы дальше, задача начинает обработку буферизованных записей. Завершив обработку буфера записей, задача продолжает обработку своих входных потоков в обычном режиме. На рис. 3.24 показано приложение на этом этапе.

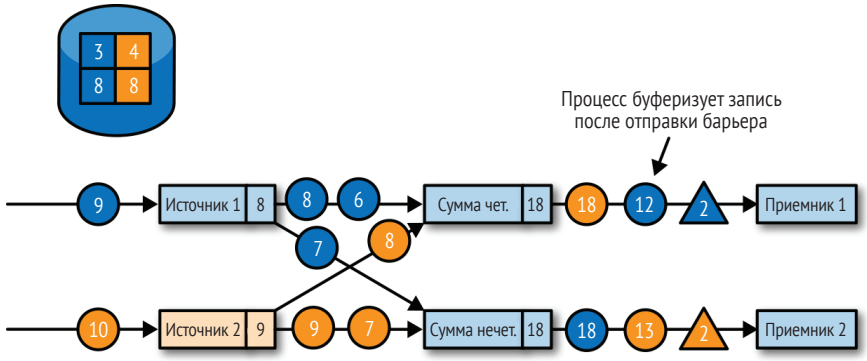

Рис. 3.24 ❖ Задачи продолжают обычную обработку после пересылки барьера контрольной точки

В конце концов барьеры контрольных точек достигают задачи выходного лотка. Когда задача лотка получает барьер, она выполняет выравнивание барьера, записывает свое состояние в контрольную точку и подтверждает обработку барьера в JobManager. JobManager помечает контрольную точку приложения как завершенную после того, как он получил подтверждение контрольной точки от всех задач приложения. На рис. 3.25 показан последний шаг алгоритма контрольной точки. Завершенную контрольную точку можно использовать для восстановления приложения после сбоя, как описано ранее.

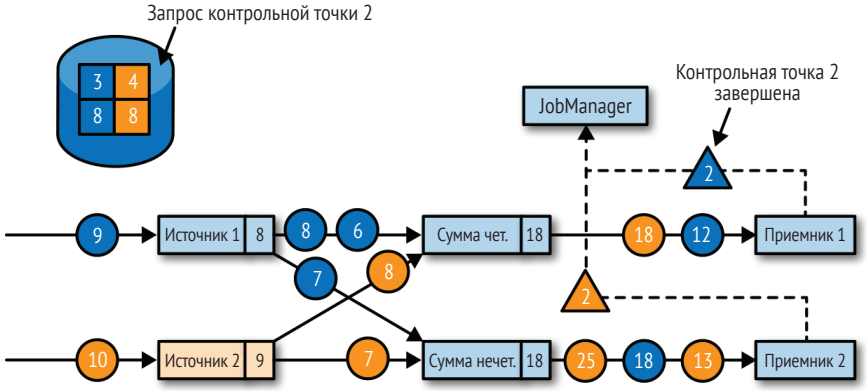

Рис. 3.25 ❖ Выходные лотки подтверждают прием барьера контрольной точки в JobManager, и контрольная точка помечается как завершенная, когда все задачи подтвердили успешное создание контрольной точки своего состояния

3.5.4. Значение контрольных точек для производительности

Алгоритм контрольных точек Flink создает согласованные распределенные контрольные точки из потоковых приложений без остановки всего приложения. Однако это может увеличить задержку приложения. Flink содержит настройки, которые при определенных условиях могут уменьшить негативное влияние на быстродействие.

Пока задача сохраняет свое состояние, она блокируется, а ее ввод буферизируется. Поскольку состояние может стать довольно большим, а контрольная точка требует записи данных по сети в удаленную систему хранения, создание контрольной точки запросто может занять от нескольких секунд до минут – слишком долго для приложений, чувствительных к задержкам. В архитектуре Flink ответственность за выполнение контрольной точки возлагается на бэкенд состояния. Как именно копируется состояние задачи, зависит от реализации бэкенд-части хранилища состояния. Например, бэкенд состояния FileSystem и RocksDB поддерживают *асинхронные* контрольные точки. Когда срабатывает контрольная точка, создается локальная копия состояния. Как только создание локальной копии завершено, задача продолжает свою обычную обработку. Фоновый поток асинхронно копирует локальный моментальный снимок в удаленное хранилище и уведомляет задачу после завершения контрольной точки. Механизм асинхронной контрольной точки значительно сокращает время простоя, пока задача не продолжит обработку данных. Кроме того, бэкенд-часть RocksDB также поддерживает инкрементные контрольные точки, что уменьшает объем данных для передачи.

Еще один способ уменьшить влияние алгоритма контрольной точки на задержку обработки – настроить шаг выравнивания барьера. Для приложений, которым требуется очень низкая задержка и которым достаточно гарантии обработки «хотя бы один раз», Flink можно настроить для обработки всех поступающих записей во время выравнивания буфера вместо буферизации тех записей, для которых барьер уже достигнут. После того как все барьеры для данной контрольной точки достигнуты, оператор сохраняет в контрольную точку состояние, которое теперь может также включать изменения, вызванные записями, которые фактически принадлежат следующей контрольной точке. В случае восстановления после сбоя эти записи будут обработаны снова, что означает, что контрольная точка предоставляет гарантию обработки типа «хотя бы один раз», а не «ровно один раз».

3.5.5. Точки сохранения

Алгоритм восстановления Flink основан на контрольных точках состояния. Контрольные точки периодически создаются и автоматически удаляются в соответствии с настраиваемой политикой. Поскольку назначением контрольных точек является обеспечение возможности перезапуска приложения в случае сбоя, они удаляются, когда работу приложения сознательно останав-

ливают[1]. Однако согласованные снимки состояния приложения могут использоваться для многих других целей, чем просто восстановление после сбоя.

Одна из наиболее ценных и уникальных функций Flink – это *точки сохранения* (savepoint). В принципе, точки сохранения создаются с использованием того же алгоритма, что и контрольные точки, и в основном являются контрольными точками с некоторыми дополнительными метаданными. Flink не создает точки сохранения автоматически, поэтому пользователь (или внешний планировщик) должен явно инициировать их создание. Flink также не удаляет точки сохранения автоматически. В главе 10 рассказано, как создавать и удалять точки сохранения.

3.5.5.1. Использование точек сохранения

Имея приложение и совместимую точку сохранения, вы можете запустить приложение из точки сохранения. Это инициализирует состояние приложения до состояния точки сохранения и запустит приложение с момента, когда была создана точка сохранения. Хотя такое поведение кажется в точности таким же, как восстановление приложения после сбоя с помощью контрольной точки, восстановление после сбоя на самом деле является просто частным случаем, когда вы запускаете то же самое приложение с такой же конфигурацией в том же кластере. Запуск приложения из точки сохранения позволяет делать гораздо больше:

- вы можете запустить другое, но совместимое приложение из точки сохранения. Следовательно, вы можете исправить ошибки в логике своего приложения и повторно обработать столько событий, сколько может предоставить источник потоковой передачи, чтобы исправить свои результаты. Модифицированные приложения также можно использовать для запуска A/B-тестов или сценариев «что, если»[2] с другой бизнес-логикой. Обратите внимание, что приложение и точка сохранения должны быть совместимы – приложение должно иметь возможность загружать состояние точки сохранения;
- вы можете запустить одно и то же приложение с разным параллелизмом и масштабировать приложение по мере увеличения или уменьшения нагрузки;
- вы можете запустить одно и то же приложение в другом кластере. Это позволяет вам перенести приложение на более новую версию Flink или в другой кластер или центр обработки данных;
- вы можете использовать точку сохранения, чтобы приостановить приложение и возобновить его позже. Это дает возможность высвободить ресурсы кластера для приложений с более высоким приоритетом или в ситуациях, когда входные данные не поступают непрерывно;

---

[1] Можно настроить приложение таким образом, чтобы оно сохраняло свою последнюю контрольную точку при остановке.

[2] Сценарий «что, если» – это один из приемов так называемого хаос-инжиниринга – передовой технологии обеспечения устойчивости масштабных систем. Издательство «ДМК Пресс» выпустило отдельную книгу про эту технологию: «Хаос-инжиниринг. Революция в разработке устойчивых систем». – *Прим. перев.*

- вы также можете просто создать точку сохранения для текущей версии и заархивировать состояние приложения.

Поскольку точки сохранения – это очень мощная функция, многие пользователи периодически создают точки сохранения, чтобы иметь возможность вернуться в прошлое. Одно из самых интересных применений точек сохранения, которое мы встречали на практике, – это безразрывная миграция потокового приложения в центр обработки данных, который обеспечивает самые низкие цены на экземпляры машин.

3.5.5.2. Запуск приложения из точки сохранения

Все ранее упомянутые варианты использования точек сохранения следуют одному шаблону. Сначала берется точка сохранения работающего приложения, а затем она используется для восстановления состояния в запущенном приложении. В этом разделе мы описываем, как Flink инициализирует состояние приложения, запущенного из точки сохранения.

Приложение состоит из нескольких операторов. Каждый оператор может определять одно или несколько состояний с ключом и состояний операторов. Операторы выполняются параллельно одной или несколькими задачами операторов. Следовательно, типичное приложение состоит из нескольких состояний, распределенных между несколькими задачами, которые могут выполняться в разных процессах TaskManager.

На рис. 3.26 показано приложение с тремя операторами, каждый из которых выполняет две задачи. Один оператор (OP-1) имеет одно состояние оператора (OS-1), а другой оператор (OP-2) имеет два состояния с ключом (KS-1 и KS-2). Когда создается точка сохранения, состояния всех задач копируются в постоянное хранилище.

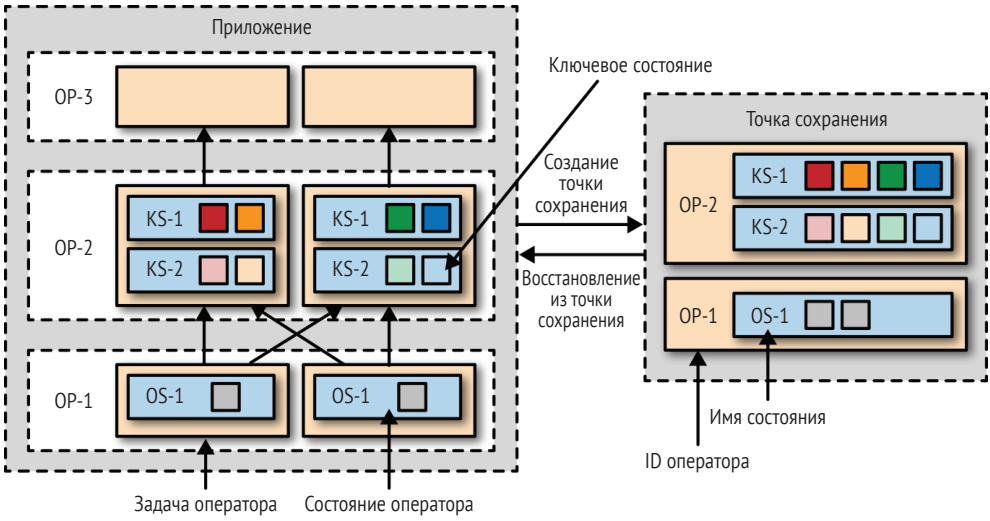

Рис. 3.26 ❖ Создание точки сохранения в приложении и восстановление приложения из точки сохранения

Копии состояний в точке сохранения организованы по идентификатору оператора и имени состояния. Идентификатор оператора и имя состояния необходимы, чтобы иметь возможность отображать данные состояния точки сохранения в состояния операторов запускаемого приложения. Когда приложение запускается из точки сохранения, Flink перераспределяет данные точки сохранения для задач соответствующих операторов.

Обратите внимание, что точка сохранения не содержит информации о задачах оператора. Это связано с тем, что количество задач может измениться, когда приложение запускается с другим параллелизмом. Ранее в этом разделе мы обсуждали стратегии Flink для масштабирования операторов с отслеживанием состояния.

Если измененное приложение запускается из точки сохранения, состояние в точке сохранения может быть сопоставлено приложению, только если оно содержит оператор с соответствующим идентификатором и именем состояния. По умолчанию Flink присваивает уникальные идентификаторы операторов. Однако идентификатор оператора детерминировано генерируется на основе идентификаторов предшествующих ему операторов. Следовательно, идентификатор оператора изменяется при изменении одного из его предшественников, например при добавлении или удалении оператора. Как следствие, приложение с идентификаторами операторов по умолчанию очень ограничено в способах развития без потери состояния. Поэтому мы настоятельно рекомендуем *вручную назначать уникальные идентификаторы операторам* и не полагаться на присвоение Flink по умолчанию. Мы подробно рассказываем, как назначать идентификаторы операторов в разделе 7.3.1.

3.6. Заключение

В этой главе мы обсудили высокоуровневую архитектуру Flink и внутреннее устройство его сетевого стека, режим обработки с привязкой ко времени событий, управление состоянием и механизм восстановления после сбоев. Эта информация пригодится при разработке сложных потоковых приложений, настройке и конфигурировании кластеров и эксплуатации потоковых приложений, а также при оценке их быстродействия.

Глава 4

Настройка рабочей среды для Apache Flink

Теперь, когда вы получили столько новых знаний, пришло время заняться разработкой приложений Flink! В этой главе вы узнаете, как настроить рабочую среду для разработки, запуска и отладки приложений Flink. Мы начнем с обсуждения необходимого программного обеспечения и того, где вы можете получить примеры кода из этой книги. Используя эти примеры, мы покажем, как приложения Flink выполняются и отлаживаются в среде IDE. Наконец, мы покажем, как запустить проект Flink Maven – отправную точку для нового приложения.

4.1. Необходимое ПО

Прежде всего давайте обсудим программное обеспечение, необходимое для разработки приложений Flink. Вы можете разрабатывать и запускать приложения Flink в Linux, macOS и Windows. Однако среда на основе UNIX обладает богатейшей инструментальной поддержкой, потому что эту ОС предпочитает большинство разработчиков Flink. В оставшейся части этой главы мы будем подразумевать использование среды на основе UNIX. Как пользователь Windows, вы можете использовать подсистему Windows для Linux (WSL), Cygwin или виртуальную машину Linux для запуска Flink в среде UNIX.

API Flink DataStream доступен для Java и Scala. Следовательно, для реализации приложений Flink DataStream требуется Java JDK версии 8 (или выше). Java JRE недостаточно.

Мы предполагаем, что на вашем компьютере также установлено следующее программное обеспечение, хотя оно не является строго обязательным для разработки приложений Flink:

- Apache Maven 3.x. Примеры кода в книге используют сборщик проектов Maven. Более того, Flink предоставляет архетипы Maven для запуска новых проектов Flink Maven;

- IDE для разработки на Java и/или Scala. Обычно разработчики предпочитают IntelliJ IDEA, Eclipse или Netbeans с соответствующими подключаемыми модулями (например, для поддержки Maven, Git и Scala). Мы рекомендуем использовать IntelliJ IDEA. Для загрузки и установки IntelliJ IDEA вы можете воспользоваться инструкциями на веб-сайте.

4.2. Запуск и отладка приложений Flink в среде IDE

Несмотря на то что Flink является системой распределенной обработки данных, вы обычно разрабатываете и запускаете начальные тесты на своем локальном компьютере. Это упрощает разработку приложения и развертывание кластера, поскольку вы можете запускать тот же код в кластерной среде без каких-либо изменений. Далее мы расскажем, как получить примеры кода, которые мы используем в книге, как импортировать их в IntelliJ, как запустить пример приложения и как его отладить.

4.2.1. Импорт примеров книги в IDE

Примеры кода из этой книги размещены на GitHub. На странице книги на GitHub вы найдете один репозиторий с примерами на языке Scala и один репозиторий с примерами на языке Java. Мы будем использовать репозиторий Scala для установки, но вы сможете следовать тем же инструкциям, если предпочитаете Java.

Откройте терминал и выполните следующую команду Git, чтобы клонировать репозиторий `examples-scala` на ваш локальный компьютер[1]:

```
> git clone https://github.com/streaming-with-flink/examples-scala
```

Вы также можете скачать исходный код примеров в виде zip-архива с GitHub:

```
> wget https://github.com/streaming-with-flink/examples-scala/archive/master.zip
> unzip master.zip
```

Примеры книг представлены как проект Maven. Вы найдете исходный код в каталоге `src/`, сгруппированном по главам:

```
.
└── main
    └── scala
        └── io
            └── github
```

[1] Мы также предлагаем читателям репозиторий examples-Java с примерами на языке Java.

```
└── streamingwithflink
    ├── chapter1
    │   └── AverageSensorReadings.scala
    ├── chapter5
    │   └── ...
    ├── ...
    │   └── ...
    └── util
        └── ...
```

Теперь откройте свою среду IDE и импортируйте проект Maven. Шаги импорта аналогичны для большинства IDE. Далее мы подробно объясняем этот шаг для IntelliJ.

Перейдите из **Existing Sources** (существующие источники) в меню **File ⇨ New ⇨ Project**, выберите папку примеров книг `examples-scala` и нажмите **OK**. Убедитесь, что выбраны опции **Import project from external model** (Импортировать проект из внешней модели) и **Maven**, и нажмите кнопку **Next**.

Мастер импорта проекта проведет вас через следующие шаги, такие как выбор проекта Maven для импорта (должен быть только один проект), выбор SDK и присвоение имени проекту. На рис. 4.1–4.3 показан процесс импорта.

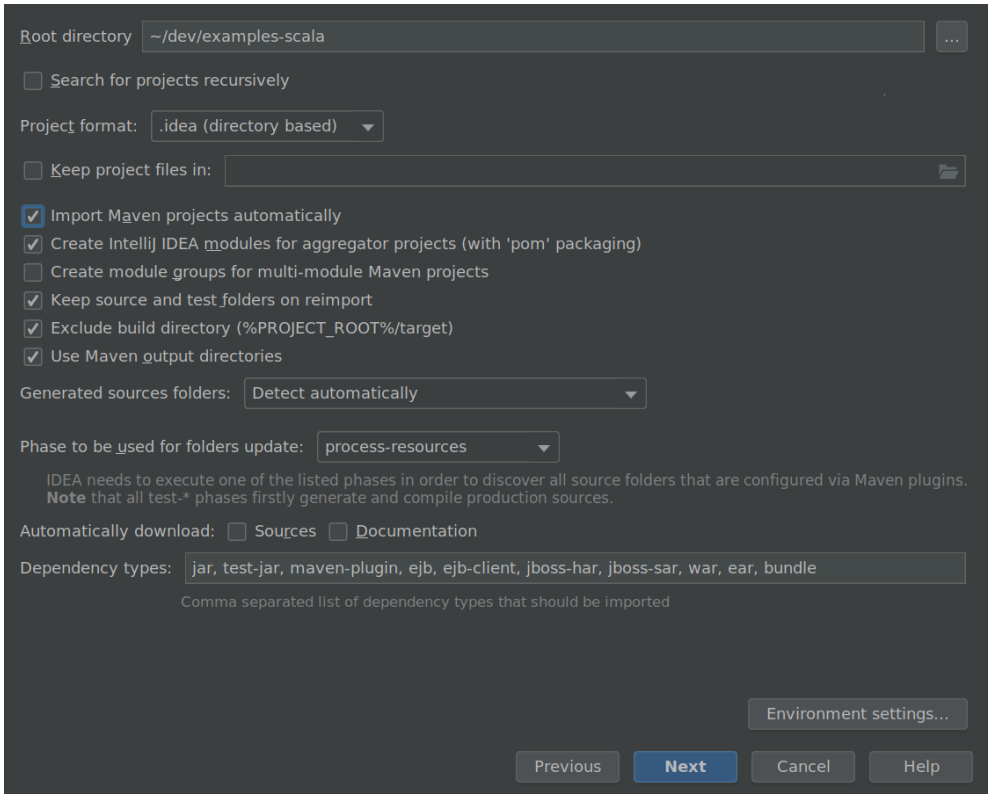

Рис. 4.1 ❖ Импортируйте репозиторий примеров книги в IntelliJ

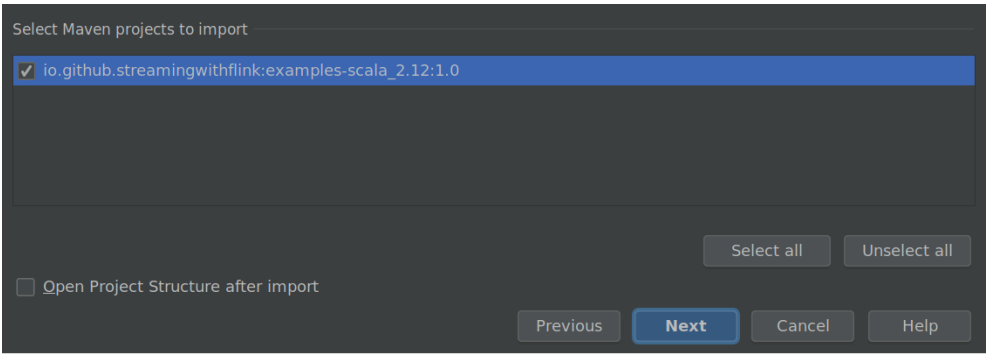

Рис. 4.2 ❖ Выберите проект Maven для импорта

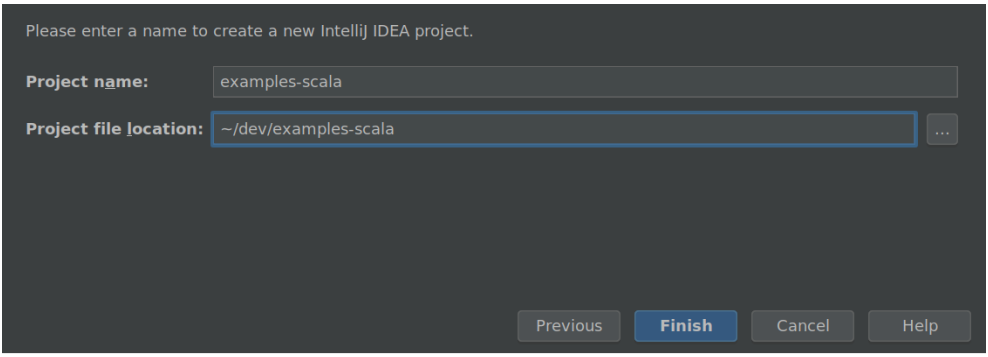

Рис. 4.3 ❖ Дайте вашему проекту имя и нажмите **Finish**

Готово! Теперь вы можете просматривать и проверять код примеров книг.

4.2.2. Запуск приложений Flink в среде IDE

Теперь давайте запустим в вашей среде IDE один из примеров приложений из книги. Найдите класс `AverageSensorReadings` и откройте его. Как было сказано в разделе 1.4, программа генерирует события чтения для нескольких термодатчиков, конвертирует температуру событий из градусов Фаренгейта в градусы Цельсия и вычисляет среднюю температуру каждого датчика каждую секунду. Результаты программы отправляются на стандартный вывод. Как и во многих приложениях `DataStream`, источник, приемник и операторы программы собраны в методе `main()` класса `AverageSensorReadings`.

Чтобы выполнить приложение, запустите метод `main()`. Вывод программы записывается в стандартное окно вывода (или консоль) вашей IDE. Вывод начинается с нескольких строк журнала о состояниях, через которые проходят задачи параллельных операторов, таких как `SCHEDULING`, `DEPLOYING` и `RUNNING`. После того как все задачи будут извлечены и запущены, программа начинает выдавать свои результаты, которые должны выглядеть примерно так:

```
2> SensorReading(sensor_31,1515014051000,23.924656183848732)
4> SensorReading(sensor_32,1515014051000,4.118569049862492)
1> SensorReading(sensor_38,1515014051000,14.781835420242471)
3> SensorReading(sensor_34,1515014051000,23.871433252250583)
```

Программа будет продолжать генерировать новые события, обрабатывать их и выдавать новые результаты каждую секунду, пока вы не завершите ее.

Теперь давайте бегло обсудим, что происходит под капотом. Как мы говорили в разделе 3.1.1, приложение Flink отправляется в JobManager (руководитель), который распределяет задачи выполнения между одним или несколькими процессами TaskManager (исполнители). Поскольку Flink является распределенной системой, JobManager и TaskManager обычно запускаются как отдельные процессы JVM на разных машинах.

Обычно метод `main()` программы собирает поток данных и отправляет его удаленному JobManager при вызове метода `StreamExecutionEnvironment.execute()`.

Однако существует также режим, в котором вызов метода `execute()` запускает JobManager и TaskManager (по умолчанию с таким количеством слотов, сколько доступных потоков ЦП) как отдельные потоки в одной JVM. Следовательно, все приложение Flink является многопоточным и выполняется в одном процессе JVM. Этот режим используется для выполнения программы Flink в среде IDE.

4.2.3. Отладка приложений Flink в среде IDE

Благодаря одиночному режиму выполнения JVM приложения Flink можно отлаживать в среде IDE почти как любую другую программу. Вы можете определять точки остановки в коде и отлаживать приложение, как обычно.

Однако при отладке приложения Flink в среде IDE следует учитывать несколько моментов:

- если вы не укажете параллелизм, программа будет выполняться таким количеством потоков, сколько потоков ЦП доступно в вашей рабочей машине. Следовательно, вы должны быть готовы к отладке многопоточной программы;
- в отличие от выполнения программы Flink путем отправки ее удаленному JobManager при отладке программа выполняется в одной JVM. Поэтому некоторые проблемы, такие как проблемы с загрузкой классов, не могут быть должным образом отлажены;
- хотя программа выполняется в одной JVM, записи сериализуются для межпоточного взаимодействия и, возможно, сохранения состояния.

4.3. Развертывание проекта Flink для сборщика Maven

Импорт репозитория `examples-scala` в вашу среду IDE для экспериментов с Flink – хороший первый шаг. Однако вы также должны знать, как создать новый проект Flink с нуля.

Flink предоставляет архетипы Maven для создания проектов Maven для приложений Flink на Java или Scala. Откройте терминал и выполните следующую команду, чтобы создать проект Flink Maven Quickstart Scala в качестве отправной точки для вашего приложения Flink:

```
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.flink \
   -DarchetypeArtifactId=flink-quickstart-scala \
   -DarchetypeVersion=1.7.1 \
   -DgroupId=org.apache.flink.quickstart \
   -DartifactId=flink-scala-project \
   -Dversion=0.1 \
   -Dpackage=org.apache.flink.quickstart \
   -DinteractiveMode=false
```

Эта длинная команда создаст проект Maven для Flink 1.7.1 в папке с именем `flink-scala-project`. Вы можете изменить версию Flink, идентификаторы группы и артефакта, версию и сгенерированный пакет, изменив соответствующие параметры указанной выше команды `mvn`. Созданная папка содержит папку `src/` и файл `pom.xml`. Папка `src/` имеет следующую структуру:

```
src/
└── main
    ├── resources
    │   └── log4j.properties
    └── scala
        └── org
            └── apache
                └── flink
                    └── quickstart
                        ├── BatchJob.scala
                        └── StreamingJob.scala
```

Проект содержит два файла-скелета, `BatchJob.scala` и `StreamingJob.scala`, в качестве отправной точки для ваших собственных программ. Вы также можете удалить их, если они вам не нужны.

Вы можете импортировать проект в свою среду IDE, выполнив шаги, описанные в предыдущем разделе, или выполнить следующую команду для создания файла JAR:

```
mvn clean package -Pbuild-jar
```

Если команда выполнена успешно, вы найдете новую папку `target` в папке вашего проекта. В папке находится файл `flink-scala-project-0.1.jar`, который является файлом JAR вашего приложения Flink. Сгенерированный файл `pom.xml` также содержит инструкции о том, как добавить новые зависимости в ваш проект.

4.4. Заключение

В этой главе вы узнали, как настроить среду для разработки и отладки приложений Flink DataStream и как создать проект Maven с использованием архетипа Maven Flink. Следующий очевидный шаг – узнать, как на практике реализовать программу DataStream.

Глава 5 познакомит вас с основами API DataStream, а главы 6, 7 и 8 расскажут обо всем, что нужно знать об операторах, основанных на времени, функциях с учетом состояния и соединителях источника и приемника.

Глава 5

API DataStream (v1.7)

В этой главе мы рассказываем об основах API DataStream. Мы продемонстрируем структуру и компоненты типичного потокового приложения Flink, обсудим системы типов Flink и поддерживаемые типы данных, а также представим функции преобразования данных и разделения потоков. Операторы окна, преобразования на основе времени, операторы с учетом состояния и соединители обсуждаются в следующих главах. Прочитав эту главу, вы узнаете, как реализовать приложение потоковой обработки с базовыми функциями. В наших примерах кода для краткости используется Scala, но API Java в основном аналогичен (исключения или особые случаи будут указаны отдельно). Мы также предоставляем полные примеры приложений, реализованных на Java и Scala, в наших репозиториях GitHub.

5.1. Hello, Flink!

Начнем с простого примера, чтобы получить первое впечатление о написании потоковых приложений с помощью API DataStream. Мы будем использовать этот пример, чтобы продемонстрировать базовую структуру программы Flink и представить некоторые важные функции API DataStream. Наш пример приложения принимает поток измерений температуры от нескольких датчиков.

Во-первых, давайте посмотрим на тип данных, который мы будем использовать для представления показаний датчика:

```
case class SensorReading(
  id: String,
  timestamp: Long,
  temperature: Double)
```

Программа в примере 5.1 преобразует температуру из Фаренгейта в Цельсия и каждые 5 с вычисляет среднюю температуру для каждого датчика.

Пример 5.1 ❖ Вычисление средней температуры каждые 5 с для потока данных от датчиков

```
// Объект Scala, который определяет программу DataStream в методе main().
object AverageSensorReadings {
```

```
  // main() определяет и выполняет программу DataStream.
  def main(args: Array[String]) {

    // Определяем среду потокового выполнения.
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // Используем в приложении время события.
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    // Создаем DataStream[SensorReading] из источника потока.
    val sensorData: DataStream[SensorReading] = env
      // Загружаем данные датчиков при помощи экземпляра SourceFunction.
      .addSource(new SensorSource)
      // Устанавливаем метки времени и водяные знаки (нужны для времени события).
      .assignTimestampsAndWatermarks(new SensorTimeAssigner)
    val avgTemp: DataStream[SensorReading] = sensorData
      // Конвертируем шкалу Фаренгейта в шкалу Цельсия.
      .map( r => {
          val celsius = (r.temperature - 32) * (5.0 / 9.0)
          SensorReading(r.id, r.timestamp, celsius)
        } )
       // Упорядочиваем измерения по ID датчика.
       .keyBy(_.id)
       // Группируем измерения по 5 секунд при помощи шагающего окна.
       .timeWindow(Time.seconds(5))
       // Вычисляем среднюю температуру при помощи определяемой пользователем функции.
       .apply(new TemperatureAverager)

     // Направляем поток результатов в стандартный вывод.
     avgTemp.print()

     // Выполняем приложение.
     env.execute("Compute average sensor temperature")
 }
}
```

Вы, наверное, уже заметили, что программы Flink определяются и отправляются на выполнение в обычные методы Scala или Java. Чаще всего это делается в статическом методе `main()`. В нашем примере мы определяем объект `AverageSensorReadings` и включаем большую часть логики приложения в `main()`.

Структура алгоритма типичного потокового приложения Flink выглядит следующим образом.

1. Настройка среды выполнения.
2. Считывание одного или нескольких потоков из источников данных.
3. Применение потоковых преобразований для реализации логики приложения.
4. При необходимости вывод результата в один или несколько приемников данных.
5. Запуск программы.

Теперь рассмотрим эти части подробнее.

5.1.1. Настройка среды выполнения

Первое, что нужно сделать приложению Flink, – это настроить *среду выполнения*. Среда выполнения определяет, выполняется ли программа на локальном компьютере или в кластере. В API DataStream среда выполнения приложения представлена как `StreamExecutionEnvironment`. В нашем примере мы получаем среду выполнения, вызывая статический метод `getExecutionEnvironment()`. Этот метод возвращает локальную или удаленную среду в зависимости от контекста, в котором вызывается метод. Если метод вызывается из клиента отправки с подключением к удаленному кластеру, возвращается среда удаленного выполнения. В противном случае он возвращает локальную среду.

Также можно явно создать локальную или удаленную среду выполнения следующим образом:

```
// Создаем локальную среду потокового выполнения.
val localEnv: StreamExecutionEnvironment.createLocalEnvironment()

// Создаем удаленную среду потокового выполнения.
val remoteEnv = StreamExecutionEnvironment.createRemoteEnvironment(
  "host", // Имя хоста JobManager.
  1234, // Порт процесса JobManager.
  "path/to/jarFile.jar") // Файл JAR для отправки в JobManager.
```

Затем мы используем `env.setStreamTimeCharacteristic (TimeCharacteristic.EventTime)`, чтобы указать нашей программе использовать семантику времени с использованием времени события. Среда выполнения предлагает и другие параметры конфигурации, такие как настройка параллелизма программ и включение отказоустойчивости.

5.1.2. Чтение входного потока

После того как среда выполнения настроена, пора приступить к реальной работе и начать обработку потоков. `StreamExecutionEnvironment` предоставляет методы для создания источников потока, которые принимают потоки данных в приложение. Потоки данных могут быть получены из таких источников, как очереди сообщений или файлы, а также могут быть созданы на лету.

В нашем примере мы используем

```
val sensorData: DataStream[SensorReading] =
  env.addSource(new SensorSource)
```

для подключения к источнику измерений датчика и создания начального потока данных типа `SensorReading`. Flink поддерживает множество типов данных, которые мы опишем в следующем разделе. Здесь мы используем case-класс Scala в качестве типа данных, который мы определили ранее. `SensorReading` содержит идентификатор датчика, метку времени, обозначающую, когда было выполнено измерение, и измеренную температуру. Метод `assignTimestampsAndWatermarks(new SensorTimeAssigner)` назначает метки време-

ни и водяные знаки, необходимые для времени события. Детали реализации `SensorTimeAssigner` сейчас нас не интересуют.

5.1.3. Применение преобразований

Как только у нас появился `DataStream`, мы можем применить к нему преобразование. Есть разные типы преобразований. Одни преобразования могут создавать новый поток данных, возможно, другого типа, в то время как другие преобразования не изменяют записи потока данных, а реорганизуют его путем разделения или группировки. Логика приложения определяется цепочкой преобразований.

В нашем примере мы сначала применяем преобразование `map()`, которое преобразует температуру каждого показания датчика в градусы Цельсия. Затем мы используем преобразование `keyBy()`, чтобы разделить показания датчиков по их идентификаторам. Затем мы определяем преобразование `timeWindow()`, которое группирует показания каждого раздела по идентификатору датчика в шагающие окна по 5 с:

```
val avgTemp: DataStream[SensorReading] = sensorData
     .map( r => {
           val celsius = (r.temperature - 32) * (5.0 / 9.0)
           SensorReading(r.id, r.timestamp, celsius)
       } )
     .keyBy(_.id)
     .timeWindow(Time.seconds(5))
     .apply(new TemperatureAverager)
```

Оконные преобразования подробно описаны в разделе 6.3. Наконец, мы применяем пользовательскую функцию, которая вычисляет среднюю температуру в каждом окне. Мы обсудим реализацию пользовательской функции в следующем разделе этой главы.

5.1.4. Вывод результата

Потоковые приложения обычно отправляют свои результаты в какую-либо внешнюю систему, такую как Apache Kafka, файловую систему или базу данных. Flink предоставляет хорошо поддерживаемый набор приемников потока, которые можно использовать для записи данных в разные системы. Также возможно реализовать свои собственные приемники потоковой передачи. Встречаются приложения, которые не выдают результаты, а хранят их внутри, чтобы обслуживать их с помощью функции запросов состояния Flink.

В нашем примере результатом является запись `DataStream[SensorReading]`. Каждая запись содержит среднюю температуру датчика за 5 с. Поток результатов записывается в стандартный вывод с помощью вызова `print()`:

```
avgTemp.print ()
```

Обратите внимание, что выбор приемника потоковой передачи влияет на сквозную согласованность приложения независимо от того, обеспечивается ли результат приложения семантикой хотя бы один или ровно один раз. Сквозная согласованность приложения зависит от интеграции выбранных приемников потока с алгоритмом контрольной точки Flink. Мы обсудим эту тему более подробно в разделе 8.1.

5.1.5. Выполнение

Когда приложение полностью определено, его можно запустить, вызвав `StreamExecutionEnvironment.execute()`. Это последний вызов в нашем примере:

```
env.execute("Compute average sensor temperature")
```

Программы Flink выполняются *лениво*. То есть вызовы API, которые создают источники потоков и преобразования, не сразу запускают какую-либо обработку данных. Вместо этого вызовы API создают план выполнения в среде выполнения, который состоит из источников потока, созданных из среды, и всех преобразований, которые были транзитивно применены к этим источникам. Только при вызове метода `execute()` система запускает выполнение программы.

Построенный план переводится в `JobGraph` и передается в `JobManager` для выполнения. В зависимости от типа среды выполнения `JobManager` запускается как локальный поток (локальная среда выполнения) или `JobGraph` отправляется удаленному процессу `JobManager`. Если `JobManager` работает удаленно, `JobGraph` должен поставляться вместе с файлом JAR, который содержит все классы и необходимые зависимости приложения.

5.2. Преобразования

В этом разделе мы даем обзор основных преобразований API DataStream. Операторы, связанные со временем, такие как оконные операторы и другие специализированные преобразования, описаны в следующих главах. Преобразование потока применяется к одному или нескольким потокам и преобразует их в один или несколько выходных потоков. Написание программы API DataStream по существу сводится к объединению таких преобразований для создания графа потока данных, реализующего логику приложения.

Большинство потоковых преобразований основано на пользовательских функциях. Функции инкапсулируют логику пользовательского приложения и определяют, как элементы входного потока преобразуются в элементы выходного потока. Функции, такие как `MapFunction`, в дальнейшем определяются как классы, реализующие интерфейс функции, зависящий от преобразования:

```
class MyMapFunction extends MapFunction[Int, Int] {
  override def map(value: Int): Int = value + 1
}
```

Интерфейс функции определяет метод преобразования, который должен быть реализован пользователем, например метод `map()` в приведенном выше примере.

Большинство интерфейсов функций разработаны как интерфейсы SAM (single abstract method, «единый абстрактный метод») и могут быть реализованы как лямбда-функции Java 8. API DataStream Scala также имеет встроенную поддержку лямбда-функций. При представлении преобразований API DataStream мы показываем интерфейсы для всех классов функций, но для краткости в примерах кода в основном используются лямбда-функции вместо функциональных классов.

API DataStream предоставляет преобразования для наиболее распространенных операций преобразования данных. Если вы знакомы с API пакетной обработки данных, функциональными языками программирования или SQL, вам будет очень легко понять концепции API. Мы представляем преобразования API DataStream в четырех категориях.

1. Базовые преобразования – это преобразования отдельных событий.
2. Преобразования `KeyedStream` – это преобразования, которые применяются к событиям в контексте ключа.
3. Многопоточные преобразования объединяют несколько потоков в один или разделяют один поток на несколько потоков.
4. Преобразования распределения реорганизуют потоковые события.

5.2.1. Основные преобразования

Основные преобразования обрабатывают отдельные события, то есть каждая выходная запись создается из одной входной записи. Примерами распространенных основных функций являются простое преобразование значений, разделение записей или фильтрация записей. Далее мы объясним их семантику и покажем примеры кода.

5.2.1.1. Map

Преобразование `map` определяется путем вызова метода `DataStream.map()` и создает новый экземпляр `DataStream`. Он передает каждое входящее событие определенному пользователем преобразователю, который возвращает ровно одно выходное событие, возможно, другого типа. На рис. 5.1 показано преобразование `map`, при котором каждый квадрат превращается в круг.

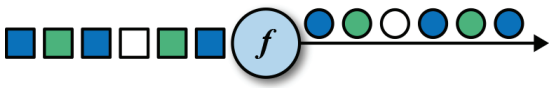

Рис. 5.1 ❖ Операция `map`, превращающая каждый квадрат в круг того же цвета

Функция MapFunction типизирована для типов входных и выходных событий и может быть указана с помощью интерфейса MapFunction. Он определяет метод map(), который преобразует входное событие ровно в одно выходное событие:

```
// T: тип входных элементов.
// O: тип выходных элементов.
MapFunction[T, O]
    > map(T): O
```

Ниже показан простой преобразователь, который извлекает первое поле (id) каждого SensorReading во входном потоке:

```
val readings: DataStream[SensorReading] = ...
val sensorIds: DataStream[String] = readings.map(new MyMapFunction)

class MyMapFunction extends MapFunction[SensorReading, String] {
  override def map(r: SensorReading): String = r.id
}
```

При использовании Scala API или Java 8 преобразователь также может быть выражен как лямбда-функция:

```
val readings: DataStream[SensorReading] = ...
val sensorIds: DataStream[String] = readings.map(r => r.id)
```

5.2.1.2. Filter

Преобразование фильтра отбрасывает или пересылает события потока, проверяя логическое условие для каждого входного события. Возвращаемое значение true сохраняет входное событие и перенаправляет его на выход, а false приводит к отбрасыванию события. Преобразование фильтра указывается путем вызова метода DataStream.filter() и создает новый DataStream того же типа, что и входной DataStream. На рис. 5.2 показана операция фильтрации, при которой сохраняются только белые квадраты.

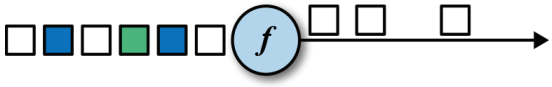

Рис. 5.2 ❖ Операция filter, сохраняющая только белые значения

Логическое условие реализуется как функция с использованием интерфейса FilterFunction или лямбда-функции. Интерфейс FilterFunction типизирован для типа входного потока и определяет метод filter(), который вызывается с входным событием и возвращает логическое значение:

```
// T: тип элементов.
FilterFunction[T]
    > filter(T): Boolean
```

В следующем примере показан фильтр, который сбрасывает все показания датчика при температуре ниже 25 °F:

```
val readings: DataStream[SensorReadings] = ...
val filteredSensors = readings
    .filter( r => r.temperature >= 25 )
```

5.2.1.3. FlatMap

Преобразование `flatMap` похоже на `map`, но может генерировать ноль, одно или несколько выходных событий для каждого входящего события. Фактически преобразование `flatMap` является обобщением преобразований `filter` и `map` и может использоваться для реализации обеих этих операций. На рис. 5.3 показана операция `flatMap`, которая меняет свой вывод в зависимости от цвета входящего события. Если вход представляет собой белый квадрат, она выводит событие без изменений. Черные квадраты дублируются, а серые отфильтровываются.

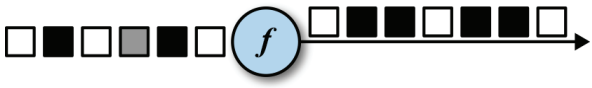

Рис. 5.3 ❖ Операция `flatMap`, которая выводит белые квадраты, дублирует черные и отбрасывает серые

Преобразование `flatMap` применяет функцию к каждому входящему событию. Соответствующая функция `FlatMapFunction` определяет метод `flatMap()`, который может возвращать ноль, одно или несколько событий в качестве результатов, передав их объекту `Collector`:

```
// T: тип входных элементов.
// O: тип выходных элементов.
FlatMapFunction[T, O]
    > flatMap(T, Collector[O]): Unit
```

В этом примере показано преобразование `flatMap`, которое обычно встречается в учебниках по обработке данных. Функция применяется к потоку предложений, разделяет каждое предложение символом пробела и выдает каждое полученное слово как отдельную запись:

```
val sentences: DataStream[String] = ...
val words: DataStream[String] = sentences
   .flatMap(id => id.split(" "))
```

5.2.2. Преобразования KeyedStream

Общим требованием многих приложений является обработка групп событий, которые вместе имеют определенное свойство. В API DataStream реализована абстракция `KeyedStream`, которая представляет собой DataStream, логически разделенный на непересекающиеся подпотоки событий, имеющих один и тот же ключ.

Преобразования с учетом состояния, которые применяются к `KeyedStream`, читают из состояния и записывают в состояние в контексте ключа текущего обрабатываемого события. Это означает, что все события с одним и тем же ключом обращаются к одному и тому же состоянию и, следовательно, могут обрабатываться вместе.

Обратите внимание, что преобразования с учетом состояния и агрегаты с ключами следует использовать с осторожностью. Если занятая ключами область непрерывно растет – например, потому что ключ является уникальным идентификатором транзакции, – вам необходимо очистить состояние ключей, которые больше не активны, чтобы избежать проблем с памятью. Обратитесь к разделу 7.1, где подробно рассматриваются функции с учетом состояния.

`KeyedStream` можно обрабатывать с помощью преобразований `map`, `flatMap` и `filter`, которые вы видели ранее. В дальнейшем мы будем использовать преобразование `keyBy` для конвертации `DataStream` в `KeyedStream` и преобразований с ключом, таких как скользящее агрегирование и сокращение.

5.2.2.1. keyBy

Преобразование `keyBy` преобразует `DataStream` в `KeyedStream` путем назначения ключа. События потока назначаются разделам на основе ключа, поэтому все события с одним и тем же ключом обрабатываются одной и той же задачей последующего оператора. События с разными ключами тоже могут обрабатываться одной и той же задачей, но доступ к ключевому состоянию функции задачи всегда осуществляется в области действия ключа текущего события.

Принимая во внимание цвет входного события в качестве ключа, на рис. 5.4 события черного цвета назначаются одному разделу, а все остальные события – другому разделу.

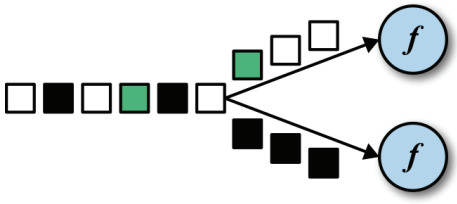

Рис. 5.4 ❖ Операция keyBy, разделяющая события по цвету

Метод keyBy() получает аргумент, указывающий ключ (или ключи) для группировки, и возвращает KeyedStream. Есть разные способы указать ключи. Мы рассмотрим их в разделе 5.5. Следующий код объявляет поле id как ключ потока записей SensorReading:

```
val readings: DataStream[SensorReading] = ...
val keyed: KeyedStream[SensorReading, String] = readings
  .keyBy(r => r.id)
```

Лямбда-функция r => r.id извлекает поле id датчика, считывающего r.

5.2.2.2. Скользящее агрегирование

Преобразования скользящего агрегирования применяются к KeyedStream и создают поток данных агрегатов, таких как сумма, минимум и максимум. Оператор скользящего агрегирования сохраняет агрегированное значение для каждого наблюдаемого ключа. Для каждого входящего события оператор обновляет соответствующее агрегированное значение и генерирует событие с обновленным значением. Скользящее агрегирование не требует написания пользовательской функции, но получает аргумент, указывающий, по какому полю вычисляется агрегат. API DataStream предоставляет следующие методы скользящего агрегирования:

`sum()`
: Скользящая сумма входящего потока по указанному полю.

`min()`
: Скользящий минимум входного потока по указанному полю.

`max()`
: Скользящий максимум входного потока по указанному полю.

`minBy()`
: Скользящий минимум входного потока, который возвращает событие с наименьшим наблюдаемым значением.

`maxBy()`
: Скользящий максимум входящего потока, который возвращает событие с наибольшим наблюдаемым значением.

Невозможно комбинировать несколько методов скользящего агрегирования – за один раз можно вычислить только один скользящий агрегат.

Рассмотрим следующий пример кортежа Tuple3[Int, Int, Int] с ключом в первом поле и вычислением скользящей суммы по второму полю:

```
val inputStream: DataStream[(Int, Int, Int)] = env.fromElements(
  (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

val resultStream: DataStream[(Int, Int, Int)] = inputStream
  .keyBy(0) // Ключ в первом поле кортежа.
  .sum(1) // Сумма значений второго поля в текущей позиции.
```

В этом примере входной поток кортежа привязан к первому полю, а скользящая сумма вычисляется по второму полю. Результатом примера будет `(1,2,2)`, за которым следует `(1,7,2)` для ключа «1» и `(2,3,1)`, за которым следует `(2,5,1)` для ключа «2». Первое поле – это общий ключ, второе поле – сумма, а третье поле не определено.

Используйте скользящие агрегаты только для ограниченных ключевых пространств

Оператор скользящего агрегата сохраняет состояние для каждого обрабатываемого ключа. Поскольку это состояние никогда не очищается, вам следует применять оператор скользящего агрегирования только к потоку с ограниченным пространством ключей.

5.2.2.3. Reduce

Преобразование `reduce` – это обобщение скользящего агрегирования, оператор скользящего сокращения. Он применяет `ReduceFunction` к `KeyedStream`, который объединяет каждое входящее событие с текущим уменьшенным значением и создает `DataStream`. Преобразование `reduce` не изменяет тип потока. Тип выходного потока такой же, как и тип входного потока.

Функцию можно определить с помощью класса, реализующего интерфейс `ReduceFunction`. `ReduceFunction` определяет метод `reduce()`, который принимает два входных события и возвращает событие того же типа:

```
// T: тип элемента.
ReduceFunction[T]
  > reduce(T, T): T
```

В приведенном ниже примере ключом потока является язык, и в результате получается постоянно обновляемый список слов для каждого языка:

```
val inputStream: DataStream[(String, List[String])] = env.fromElements(
  ("en", List("tea")), ("fr", List("vin")), ("en", List("cake")))

val resultStream: DataStream[(String, List[String])] = inputStream
  .keyBy(0)
  .reduce((x, y) => (x._1, x._2 ::: y._2))
```

Лямбда-функция `reduce` пересылает первое поле кортежа (ключевое поле) и объединяет значения `List[String]` второго поля кортежа.

Используйте скользящее уменьшение только для ограниченных ключевых пространств

Оператор скользящего сокращения сохраняет состояние для каждого обрабатываемого ключа. Поскольку это состояние никогда не очищается, вам следует применять оператор скользящего сокращения только к потоку с ограниченным набором ключей.

5.2.3. Многопоточные преобразования

Многие приложения принимают несколько потоков, которые необходимо совместно обрабатывать или разделять, чтобы применить разную логику к разным подпотокам. Далее мы обсуждаем преобразования API DataStream,

которые обрабатывают несколько входных потоков или генерируют несколько выходных потоков.

5.2.3.1. Union

Метод `DataStream.union()` объединяет два или более `DataStream` одного типа и создает новый `DataStream` того же типа. Последующие преобразования обрабатывают элементы всех входных потоков. На рис. 5.5 показана операция `union`, которая объединяет черные и серые события в один выходной поток.

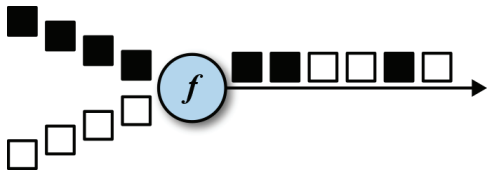

Рис. 5.5 ❖ Операция `union`, объединяющая два входных потока в один

События объединяются в стиле FIFO – оператор не формирует определенный порядок событий. Более того, оператор объединения не устраняет повторы. Каждое событие ввода передается следующему оператору.

Ниже показано, как объединить три потока типа `SensorReading` в один:

```
val parisStream: DataStream[SensorReading] = ...
val tokyoStream: DataStream[SensorReading] = ...
val rioStream: DataStream[SensorReading] = ...
val allCities: DataStream[SensorReading] = parisStream
  .union(tokyoStream, rioStream)
```

5.2.3.2. Connect, coMap и coFlatMap

Объединение событий двух потоков – очень распространенное требование при потоковой обработке. Рассмотрим приложение, которое контролирует лесной массив и выдает предупреждение, когда есть высокий риск пожара. Приложение получает поток показаний датчиков температуры, который вы видели ранее, и дополнительный поток измерений уровня дыма. Когда температура превышает заданный порог и уровень задымленности высок, приложение выдает сигнал пожарной тревоги.

Для поддержки таких вариантов использования[1] API DataStream обеспечивает преобразование `connect`. Метод `DataStream.connect()` получает `DataStream` и возвращает объект `ConnectedStreams`, который представляет два связанных потока:

---

[1] Flink предлагает специальные операторы для объединения потоков по времени, которые обсуждаются в главе 6. Преобразование объединения и совместные функции, обсуждаемые в этом разделе, носят более общий характер.

```
// Первый поток.
val first: DataStream[Int] = ...
// Второй поток.
val second: DataStream[String] = ...

// Объединение потоков.
val connected: ConnectedStreams[Int, String] = first.connect(second)
```

Объект ConnectedStreams предоставляет методы map() и flatMap(), которые ожидают CoMapFunction и CoFlatMapFunction в качестве аргумента соответственно[1].

Обе функции типизированы для типов первого и второго входного потока и для типа выходного потока и определяют два метода – по одному для каждого входа. Методы map1() и flatMap1() вызываются для обработки события первого ввода, а map2() и flatMap2() – для обработки события второго ввода:

```
// IN1: Тип первого входного потока.
// IN2: Тип второго входного потока.
// OUT: Тип выходных элементов.
CoMapFunction[IN1, IN2, OUT]
    > map1(IN1): OUT
    > map2(IN2): OUT

// IN1: Тип первого входного потока.
// IN2: Тип второго входного потока.
// OUT: Тип выходных элементов.
CoFlatMapFunction[IN1, IN2, OUT]
    > flatMap1(IN1, Collector[OUT]): Unit
    > flatMap2(IN2, Collector[OUT]): Unit
```

Функция не может выбирать, какие ConnectedStreams читать

Невозможно контролировать порядок, в котором вызываются методы CoMapFunction или CoFlatMapFunction. Вместо этого метод вызывается, как только событие поступает через соответствующий вход.

Совместная обработка двух потоков обычно требует, чтобы события обоих потоков детерминированно маршрутизировались на основе некоторого условия, которое должно быть обработано одним и тем же параллельным экземпляром оператора. По умолчанию connect() не устанавливает связи между событиями обоих потоков, поэтому события обоих потоков случайным образом назначаются экземплярам операторов. Такое поведение дает недетерминированные результаты и обычно нежелательно. Для достижения детерминированных преобразований ConnectedStreams, connect() можно комбинировать с keyBy() или broadcast(). Сначала мы покажем пример для keyBy():

```
val one: DataStream[(Int, Long)] = ...
val two: DataStream[(Int, String)] = ...
```

[1] Вы также можете применить к ConnectedStreams функцию CoProcessFunction. Мы обсуждаем функцию CoProcessFunction в главе 6.

```
// Применяем keyBy к двум связанным потокам.
val keyedConnect1: ConnectedStreams[(Int, Long), (Int, String)] = one
  .connect(two)
  .keyBy(0, 0) // Ключ обоих входных потоков в первом атрибуте.

// Альтернативный вариант: применяем connect к обоим потокам с ключом.
val keyedConnect2: ConnectedStreams[(Int, Long), (Int, String)] = one.keyBy(0)
  .connect(two.keyBy(0))
```

Независимо от того, применяете ли вы `keyBy()` к `ConnectedStreams` или `connect()` к двум потокам с ключом, преобразование `connect()` направит все события из обоих потоков с одним и тем же ключом в один и тот же экземпляр оператора. Обратите внимание, что ключи обоих потоков должны относиться к одному и тому же классу сущностей, как и предикат соединения в запросе SQL. Оператор, который применяется к связанному и ключевому потоку, имеет доступ к состоянию с ключом[1].

В следующем примере показано, как связать `DataStream` (без ключа) с транслируемым потоком:

```
val first: DataStream[(Int, Long)] = ...
val second: DataStream[(Int, String)] = ...

// Связывание потоков с транслированием.
val keyedConnect: ConnectedStreams[(Int, Long), (Int, String)] = first
  // Транслирование второго входного потока.
  .connect(second.broadcast())
```

Все события вещаемого потока реплицируются и отправляются всем экземплярам параллельного оператора последующей функции обработки. События невещаемого потока просто перенаправляются. Следовательно, элементы обоих входных потоков могут обрабатываться совместно.

Вы можете использовать состояние широковещания для соединения ключевого и широковещательного потоков. Состояние широковещания – это улучшенная версия преобразования `broadcast()` – `connect()`. Она также поддерживает соединение ключевого и широковещательного потоков и сохранение широковещательных событий в состоянии с доступом по ключу. Это позволяет вам реализовать операторы, которые динамически настраиваются через поток данных (например, для добавления или удаления правил фильтрации или обновления моделей машинного обучения). Состояние широковещательной передачи подробно обсуждается в разделе 7.1.3.

5.2.3.3. Split и select

`Split` (разделение) – это преобразование, обратное преобразованию `union` (объединение). Оно делит входной поток на два или более выходных потока того же типа, что и входной поток. Каждое входящее событие может быть направлено в ноль, один или несколько выходных потоков. Следовательно, `split` также можно использовать для фильтрации или репликации событий.

---

[1] См. главу 8 для получения подробной информации о состоянии с доступом по ключу.

На рис. 5.6 показан оператор split, который направляет все белые события в отдельный поток от остальных.

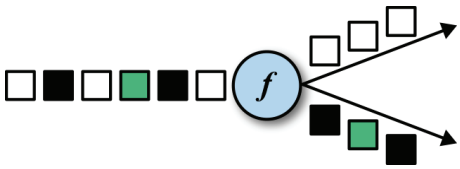

Рис. 5.6 ❖ Операция split, разделяющая входной поток на поток белых событий и поток остальных событий

Метод DataStream.split() получает OutputSelector, который определяет, как элементы потока назначаются именованным выходам. OutputSelector определяет метод select(), который вызывается для каждого входного события и возвращает java.lang.Iterable[String]. Значения String, возвращаемые для записи, определяют потоки вывода, в которые направляется запись.

```
// IN: тип разделяемых элементов.
OutputSelector[IN]
    > select(IN): Iterable[String]
```

Метод DataStream.split() возвращает SplitStream, который предоставляет метод select() для выбора одного или нескольких потоков из SplitStream путем указания имен вывода.

Код примера 5.2 разделяет поток чисел на поток больших чисел и поток маленьких чисел.

Пример 5.2 ❖ Разделение кортежа входного потока на поток больших чисел и поток маленьких чисел

```
val inputStream: DataStream[(Int, String)] = ...

val splitted: SplitStream[(Int, String)] = inputStream
  .split(t => if (t._1 > 1000) Seq("large") else Seq("small"))

val large: DataStream[(Int, String)] = splitted.select("large")
val small: DataStream[(Int, String)] = splitted.select("small")
val all: DataStream[(Int, String)] = splitted.select("small", "large")
```

Одним из ограничений преобразования split является то, что все исходящие потоки имеют тот же тип, что и входной тип. В разделе 6.2.2 мы представляем опцию побочного вывода функций обработки, которая позволяет выдавать несколько потоков разных типов из одной функции.

5.2.4. Преобразования распределения

Преобразования распределения (секционирования) соответствуют стратегиям обмена данными, которые мы представили в разделе 2.1.3. Эти операции

определяют, как события назначаются задачам. При создании приложений с помощью API DataStream система автоматически выбирает стратегии разделения данных и направляет данные в правильное место назначения в зависимости от семантики операции и настроенного параллелизма. Иногда необходимо или желательно управлять стратегиями распределения на уровне приложения или определять собственные распределители. Например, если мы знаем, что нагрузка параллельных разделов DataStream перекошена, мы можем принять решение повторно сбалансировать данные, чтобы равномерно распределить вычислительную нагрузку последующих операторов. В качестве альтернативы логика приложения может требовать, чтобы все задачи операции получали одни и те же данные или чтобы события распределялись в соответствии с настраиваемой стратегией. В этом разделе мы представляем методы DataStream, которые позволяют пользователям управлять стратегиями распределения или определять свои собственные.

Обратите внимание, что `keyBy()` отличается от преобразований распределения, обсуждаемых в этом разделе. Все преобразования в этом разделе создают `DataStream`, тогда как `keyBy()` приводит к `KeyedStream`, к которому можно применить преобразование с доступом к ключевому состоянию.

Случайный обмен
: Стратегия случайного обмена данными реализуется методом `DataStream.shuffle()`. Метод распределяет записи случайным образом в соответствии с равномерным распределением по параллельным задачам следующего оператора.

Круговой обход (ребаланс)
: Метод `rebalance()` разделяет входной поток таким образом, чтобы события равномерно распределялись между последующими задачами в циклическом режиме. На рис. 5.7 показано преобразование циклического распределения.

Масштабирование
: Метод масштабирования `rescale()` также распределяет события циклически, но только для подмножества последующих задач. По сути, стратегия разделения масштабирования предлагает способ выполнить упрощенную перебалансировку нагрузки, когда количество задач отправителя и получателя неодинаково. Преобразование изменения масштаба более эффективно, если количество задач-получателей является множеством задач-отправителей, или наоборот.

  Основное различие между `rebalance()` и `rescale()` заключается в способе формирования связей между задачами. В то время как `rebalance()` будет создавать каналы связи между всеми отправляющими задачами и всеми принимающими задачами, `rescale()` будет создавать только каналы от каждой задачи к некоторым задачам нижестоящего оператора. Схема подключения преобразования масштабирования распределения показана на рис. 5.7.

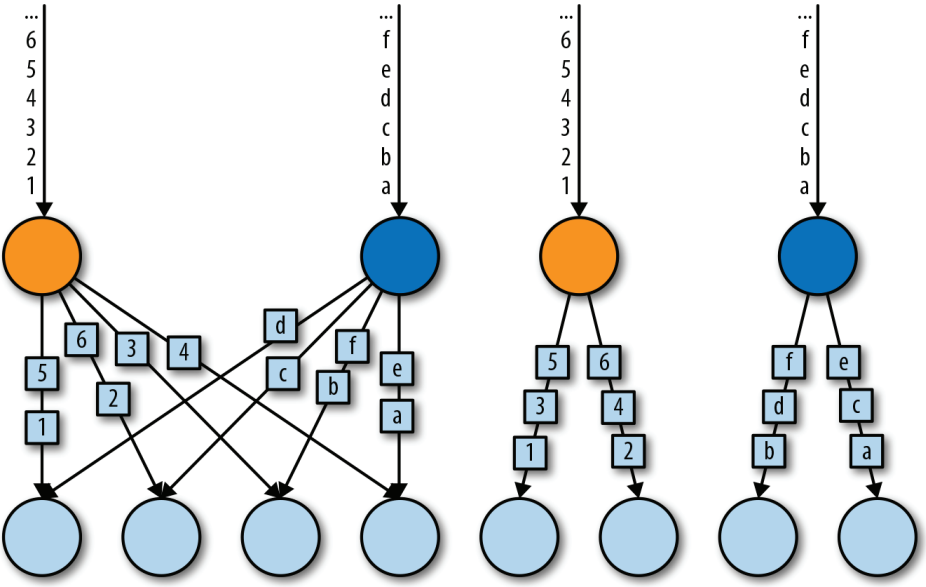

Рис. 5.7 ❖ Преобразования ребаланса и масштабирования

Широковещание

Метод `broadcast()` реплицирует поток входных данных, так что все события отправляются во все параллельные задачи нижестоящего оператора.

Глобальное распределение

Метод `global()` отправляет все события потока входных данных в первую параллельную задачу нижестоящего оператора. Эту стратегию разделения необходимо использовать с осторожностью, поскольку перенаправление всех событий одной и той же задаче может повлиять на быстродействие приложения.

Пользовательское распределение

Если ни одна из встроенных стратегий распределения не подходит, вы можете определить свою собственную стратегию с помощью метода `partitionCustom()`. Этот метод получает объект `Partitioner`, который реализует логику разделения, и поле или позицию ключа, по которой поток должен быть разделен. В следующем примере поток целых чисел разбивается на части, так что все отрицательные числа отправляются в первую задачу, а все остальные – в случайную задачу:

```
val numbers: DataStream[(Int)] = ...
numbers.partitionCustom(myPartitioner, 0)

object myPartitioner extends Partitioner[Int] {
  val r = scala.util.Random
```

```
    override def partition(key: Int, numPartitions: Int): Int = {
      if (key < 0) 0 else r.nextInt(numPartitions)
    }
  }
```

5.3. Настройка параллельной обработки

Приложения Flink выполняются параллельно в распределенной среде, такой как кластер машин. Когда программа DataStream отправляется в JobManager для выполнения, система создает граф потока данных и подготавливает операторы к выполнению. Каждый оператор распараллеливается на одну или несколько задач. Каждая задача будет обрабатывать подмножество входного потока оператора. Количество параллельных задач оператора называется параллелизмом оператора. Он определяет, насколько могут быть распределены усилия оператора по обработке, а также сколько данных он может обработать.

Параллелизмом оператора можно управлять на уровне среды выполнения или для отдельного оператора. По умолчанию параллелизм всех операторов приложения установлен как параллелизм среды выполнения приложения. Параллелизм среды (и, следовательно, параллелизм всех операторов по умолчанию) автоматически инициализируется в зависимости от контекста, в котором запускается приложение. Если приложение выполняется в локальной среде выполнения, параллелизм устанавливается в соответствии с количеством ядер ЦП. При отправке приложения в работающий кластер Flink для параллелизма среды устанавливается значение параллелизма кластера по умолчанию, если это не указано в явном виде через клиент отправки (см. раздел 10.1 для получения дополнительных сведений).

В целом рекомендуется определять параллелизм ваших операторов относительно параллелизма среды по умолчанию. Это позволяет легко масштабировать приложение, регулируя его параллелизм через программу-клиент отправки. Вы можете получить доступ к параллелизму среды по умолчанию, как показано в следующем примере:

```
val env: StreamExecutionEnvironment.getExecutionEnvironment
// Получение параллелизма по умолчанию, настроенного в конфигурации кластера
// или явно указанного через программу-клиент отправки.
val defaultP = env.env.getParallelism
```

Вы также можете переопределить параллелизм среды по умолчанию, но вы больше не сможете контролировать параллелизм вашего приложения через программу-клиент отправки:

```
val env: StreamExecutionEnvironment.getExecutionEnvironment
// Устанавливаем параллелизм окружения.
env.setParallelism (32)
```

Параллелизм по умолчанию для оператора можно изменить, указав его явно. В следующем примере оператор источника будет выполняться с па-

раллелизмом среды по умолчанию, преобразование `map` имеет вдвое больше задач, чем источник, а операция приемника всегда будет выполняться двумя параллельными задачами:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// Получение параллелизма по умолчанию.
val defaultP = env.getParallelism

// Источник выполняется с параллелизмом по умолчанию.
val result: = env.addSource(new CustomSource)
  // Параллелизм map равен удвоенному параллелизму по умолчанию.
  .map(new MyMapper).setParallelism(defaultP * 2)
  // Параллелизм приемника print равен фиксированному значению 2.
  .print().setParallelism(2)
```

Когда вы отправляете приложение через программу-клиент отправки и указываете параллелизм равным 16, источник будет работать с параллелизмом 16, преобразователь будет работать с 32 задачами, а приемник будет работать с двумя задачами. Если вы запустите приложение в локальной среде – или, например, из своей IDE – на машине с 8 ядрами, исходная задача будет выполняться с 8 задачами, преобразователь `map` – с 16 задачами, а приемник – с двумя задачами.

5.4. Типы

Приложения DataStream обрабатывают потоки событий, которые представлены в виде объектов данных. Функции вызываются с объектами данных и выдают объекты данных. Flink должен уметь обрабатывать эти объекты. Их необходимо сериализовать и десериализовать, чтобы отправлять их по сети, записывать в них или считывать их из бэкендов состояния, контрольных точек и точек сохранения. Для того чтобы делать это эффективно, Flink нуждается в детальных знаниях о типах данных, обрабатываемых приложением. Flink использует концепцию информации о типах для представления типов данных и создания конкретных сериализаторов, десериализаторов и компараторов для каждого типа данных.

Flink также имеет систему извлечения типов, которая анализирует входные и возвращаемые типы функций для автоматического получения информации о типе и, следовательно, сериализаторов и десериализаторов. Однако в определенных ситуациях, таких как лямбда-функции или универсальные типы, необходимо явно предоставить информацию о типе, чтобы заставить приложение работать или улучшить его быстродействие.

В этом разделе мы расскажем о типах, поддерживаемых Flink, как создать информацию о типе для типа данных и как помочь системе типов Flink с подсказками, если она не может автоматически определить возвращаемый тип функции.

5.4.1. Поддерживаемые типы данных

Flink поддерживает все распространенные типы данных, доступные в Java и Scala. Наиболее широко используемые типы можно сгруппировать в следующие категории:

- примитивы;
- кортежи Java и Scala;
- классы Scala;
- POJO[1], включая классы, созданные Apache Avro;
- некоторые особые типы.

Типы, которые не обрабатываются специальным образом, рассматриваются как универсальные типы и сериализуются с использованием фреймворка сериализации Kryo.

Используйте Kryo только в качестве запасного решения

Обратите внимание, что вам следует избегать использования Kryo, если это возможно. Поскольку Kryo – сериализатор общего назначения, он обычно не очень эффективен. Flink предоставляет параметры конфигурации для повышения эффективности за счет предварительной регистрации классов в Kryo. Более того, Kryo не предоставляет хороший способ миграции для развития типов данных.

Давайте рассмотрим каждую категорию типов.

Примитивы

Поддерживаются все примитивные типы Java и Scala, такие как `Int` (или `Integer` для Java), `String` и `Double`. Вот пример, который обрабатывает поток длинных значений и увеличивает каждый элемент:

```
val numbers: DataStream[Long] = env.fromElements(1L, 2L, 3L, 4L)
numbers.map( n => n + 1)
```

Кортежи Java и Scala

Кортежи – это составные типы данных, которые состоят из фиксированного числа типизированных полей.

В API DataStream для Scala используются обычные кортежи Scala. Следующий пример фильтрует DataStream кортежей с двумя полями:

```
// DataStream Tuple2[String, Integer] для Person(name, age).
val persons: DataStream[(String, Integer)] = env.fromElements(
  ("Adam", 17),
  ("Sarah", 23))

// Фильтр персон с возрастом > 18.
persons.filter(p => p._2 > 18)
```

[1] Plain Old Java Object – простой объект языка Java, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов, кроме тех, которые нужны для модели предметной области. – *Прим. перев.*

Flink предоставляет эффективные реализации кортежей Java. Кортежи Java в Flink могут иметь до 25 полей, каждая длина которых реализована как отдельный класс – `Tuple1`, `Tuple2`, вплоть до `Tuple25`. Классы кортежей строго типизированы.

Мы можем переписать пример фильтрации в API DataStream на языке Java следующим образом:

```
// DataStream Tuple2<String, Integer> для Person(name, age).
DataStream<Tuple2<String, Integer>> persons = env.fromElements(
  Tuple2.of("Adam", 17),
  Tuple2.of("Sarah", 23));

// Фильтр для персон с возрастом > 18.
persons.filter(p -> p.f1 > 18);
```

Доступ к полям кортежей можно получить по имени их общедоступных полей – `f0`, `f1`, `f2` и т. д., как показано ранее – или по позиции с помощью метода `getField(int pos)`, где индексы начинаются с 0:

```
Tuple2<String, Integer> personTuple = Tuple2.of("Alex", "42");
Integer age = personTuple.getField(1); // Возраст = 42.
```

В отличие от своих коллег на языке Scala кортежи Flink на языке Java являются изменяемыми, поэтому значения полей можно переназначать. Функции могут повторно использовать кортежи Java, чтобы снизить нагрузку на сборщик мусора. В следующем примере показано, как обновить поле кортежа Java:

```
personTuple.f1 = 42; // Устанавливаем второе поле в 42.
personTuple.setField(43, 1); // Устанавливаем второе поле в 43.
```

Case-классы Scala

Flink поддерживает case-классы Scala. Доступ к полям case-класса осуществляется по имени. Далее мы определяем case-класс `Person` с двумя полями: `name` и `age`. Что касается кортежей, мы фильтруем `DataStream` по возрасту:

```
case class Person(name: String, age: Int)

val persons: DataStream[Person] = env.fromElements(
  Person("Adam", 17),
  Person("Sarah", 23))

// Фильтр для персон с возрастом > 18.
persons.filter(p => p.age > 18)
```

POJO

Flink анализирует каждый тип, который не попадает ни в одну категорию, и проверяет, можно ли его идентифицировать и обрабатывать как тип POJO. Flink принимает класс как POJO, если он удовлетворяет следующим условиям:

- это публичный класс;

- он имеет открытый конструктор без аргументов – конструктор по умолчанию;
- все поля являются общедоступными или доступны через геттеры и сеттеры. Функции геттера и сеттера должны соответствовать схеме именования по умолчанию: `Y getX()` и `setX(Y x)` для поля `x` типа `Y`.

Все поля имеют типы, поддерживаемые Flink.
Например, следующий класс Java будет идентифицирован Flink как POJO:

```
public class Person {
  // Оба поля публичные.
  public String name;
  public int age;

  // Конструктор по умолчанию.
  public Person() {}

  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }
}

DataStream<Person> persons = env.fromElements(
   new Person("Alex", 42),
   new Person("Wendy", 23));
```

Сгенерированные Avro классы автоматически идентифицируются Flink и обрабатываются как POJO.

Array, List, Map, Enum и другие специальные типы
Flink поддерживает несколько специализированных типов, таких как примитивы и типы объекта `Array`; типы Java `ArrayList`, `HashMap` и `Enum`; а также типы `Writable` Hadoop. Кроме того, он предоставляет информацию о типах для типов Scala `Either`, `Option` и `Try`, а также типа `Either` для Java-версии Flink.

5.4.2. Создание информации о типах для типов данных

Центральный класс в системе типов Flink – `TypeInformation`. Он предоставляет системе необходимую информацию для создания сериализаторов и компараторов. Например, когда вы соединяете данные или группируете их по какому-либо ключу, `TypeInformation` позволяет Flink выполнять семантическую проверку допустимости полей, используемых в качестве ключей.

Когда приложение отправлено на выполнение, система типов Flink пытается автоматически получить `TypeInformation` для каждого типа данных, который обрабатывается платформой. Так называемый экстрактор типов анализирует универсальные типы и возвращаемые типы всех функций для

получения соответствующих объектов TypeInformation. Следовательно, вы можете какое-то время использовать Flink, не беспокоясь о TypeInformation для ваших типов данных. Однако иногда экстрактор типов не работает, или вы можете определить свои собственные типы и сообщить Flink, как эффективно с ними обращаться. В таких случаях вам необходимо сгенерировать TypeInformation для определенного типа данных.

Flink предоставляет два служебных класса для Java и Scala со статическими методами для генерации TypeInformation. Для Java вспомогательным классом является org.apache.flink.api.common.typeinfo.Types, и он используется, как показано в следующих примерах:

```
// TypeInformation для примитивов.
TypeInformation<Integer> intType = Types.INT;

// TypeInformation для кортежей Java.
TypeInformation<Tuple2<Long, String>> tupleType =
  Types.TUPLE(Types.LONG, Types.STRING);

// TypeInformation для POJO.
TypeInformation<Person> personType = Types.POJO(Person.class);
```

Вспомогательный класс TypeInformation для API на языке Scala – org.apache.flink.api.scala.typeutils.Types, и он используется, как показано ниже:

```
// TypeInformation для примитивов.
val stringType: TypeInformation[String] = Types.STRING

// TypeInformation для кортежей Scala.
val tupleType: TypeInformation[(Int, Long)] = Types.TUPLE[(Int, Long)]

// TypeInformation для case-классов.
val caseClassType: TypeInformation[Person] = Types.CASE_CLASS[Person]
```

Информация о типе в API Scala

В API на языке Scala применяются макросы компилятора Scala, которые генерируют объекты TypeInformation для всех типов данных во время компиляции. Чтобы получить доступ к функции макроса createTypeInformation, всегда добавляйте в приложение Scala следующий оператор импорта:

```
import org.apache.flink.streaming.api.scala
```

5.4.3. Явное предоставление информации о типе

В большинстве случаев Flink может автоматически определять типы и генерировать правильную информацию о типе. Средство извлечения типов Flink использует отражение и анализирует сигнатуры функций и информацию о подклассах, чтобы получить правильный тип вывода для определяемой пользователем функции. Однако иногда необходимая информация не может быть извлечена (например, из-за того, что Java стирает информацию общего типа). Более того, в некоторых случаях Flink может не выбирать TypeInforma-

tion, который генерирует наиболее эффективные сериализаторы и десериализаторы. Следовательно, вам может потребоваться явное предоставление объектов TypeInformation для Flink для некоторых типов данных, используемых в вашем приложении.

Есть два способа предоставить TypeInformation. Во-первых, вы можете расширить класс функции, чтобы явно предоставить TypeInformation его возвращаемого типа, реализовав интерфейс ResultTypeQueryable. В следующем примере показана функция MapFunction, которая предоставляет свой возвращаемый тип:

```
class Tuple2ToPersonMapper extends MapFunction[(String, Int), Person]
    with ResultTypeQueryable[Person] {

  override def map(v: (String, Int)): Person = Person(v._1, v._2)

  // Предоставляет TypeInformation для типа выходных данных.
  override def getProducedType: TypeInformation[Person] = Types.CASE_CLASS[Person]
}
```

В API DataStream на языке Java вы также можете использовать метод returns(), чтобы явно указать тип возвращаемого значения оператора при определении потока данных, как показано ниже:

```
DataStream<Tuple2<String, Integer>> tuples = ...
DataStream<Person> persons = tuples
   .map(t -> new Person(t.f0, t.f1))
   // Предоставляет TypeInformation для вывода лямбда-функции map.
   .returns(Types.POJO(Person.class));
```

5.5. Определение ключей и полей ссылок

Некоторые из преобразований, которые вы видели в предыдущем разделе, требуют спецификации по ключу или ссылки на поле типа входного потока. Во Flink ключи не предопределены в типах входных данных, как в системах, которые работают с парами «ключ–значение». Вместо этого ключи определены как функции над входными данными. Следовательно, нет необходимости определять типы данных для хранения ключей и значений, что позволяет избежать использования большого количества стереотипного кода.

Далее мы обсудим различные методы ссылки на поля и определения ключей для типов данных.

5.5.1. Позиции поля

Если тип данных является кортежем, ключи можно определить, просто используя позицию поля соответствующего элемента кортежа. В следующем примере входной поток задается вторым полем входного кортежа:

```
val input: DataStream[(Int, String, Long)] = ...
val keyed = input.keyBy(1)
```

Также можно определить составные ключи, состоящие из более чем одного поля кортежа. В этом случае позиции представлены в виде списка, одна за другой. Мы можем указать входной поток по второму и третьему полю следующим образом:

```
val keyed2 = input.keyBy(1, 2)
```

5.5.2. Выражения поля

Другой способ определения ключей и выбора полей – использование строковых выражений полей. Выражения полей работают для кортежей, POJO и case-классов. Они также поддерживают выбор вложенных полей.

Во вводном примере этой главы мы определили следующий case-класс:

```
case class SensorReading(
  id: String,
  timestamp: Long,
  temperature: Double)
```

Чтобы управлять потоком по идентификатору датчика, мы можем передать имя поля `id` функции `keyBy()`:

```
val sensorStream: DataStream[SensorReading] = ...
val keyedSensors = sensorStream.keyBy("id")
```

Поля POJO или case-класса выбираются по имени поля, как в приведенном выше примере. На поля кортежей ссылаются либо по их имени поля (смещение 1 для кортежей Scala, смещение 0 для кортежей Java), либо по индексу их поля со смещением 0:

```
val input: DataStream[(Int, String, Long)] = ...
val keyed1 = input.keyBy("2") // Ключ по 3-му полю.
val keyed2 = input.keyBy("_1") // Ключ по 1-му полю.
DataStream<Tuple3<Integer, String, Long>> javaInput = ...
javaInput.keyBy("f2") // Ключ кортежа Java по 3-му полю.
```

Вложенные поля в POJO и кортежах выбираются путем обозначения уровня вложенности знаком «.» (символ точки). Рассмотрим следующие case-классы:

```
case class Address(
  address: String,
  zip: String
  country: String)


case class Person(
  name: String,
  birthday: (Int, Int, Int), // Год, месяц, день.
  address: Address)
```

Если мы хотим сослаться на почтовый индекс человека, мы можем использовать выражение поля:

```
val persons: DataStream[Person] = ...
persons.keyBy("address.zip") // Ключ по вложенному полю POJO.
```

Также можно вкладывать выражения в смешанные типы. Следующее выражение обращается к полю кортежа, вложенного в POJO:

```
persons.keyBy("birthday._1") // Ключ по полю вложенного кортежа.
```

Полный тип данных можно выбрать с помощью выражения поля с подстановочным знаком «_» (символ подчеркивания):

```
persons.keyBy("birthday._") // Ключ по всем полям вложенного кортежа.
```

5.5.3. Ключевые селекторы

Третий вариант указания ключей – это функции `KeySelector`. Функция `KeySelector` извлекает ключ из события ввода:

```
// T: тип входных элементов.
// KEY: тип ключа.
KeySelector[IN, KEY]
  > getKey(IN): KEY
```

Вводный пример фактически использует простую функцию `KeySelector` в методе `keyBy()`:

```
val sensorData: DataStream[SensorReading] = ...
val byId: KeyedStream[SensorReading, String] = sensorData
  .keyBy(r => r.id)
```

Функция `KeySelector` получает элемент ввода и возвращает ключ. Ключ необязательно должен быть полем входного события, но может быть получен с помощью произвольных вычислений. Далее функция `KeySelector` возвращает максимум полей кортежа в качестве ключа:

```
val input : DataStream[(Int, Int)] = ...
val keyedStream = input.keyBy(value => math.max(value._1, value._2))
```

По сравнению с позициями полей и выражениями полей преимущество функций `KeySelector` заключается в том, что результирующий ключ строго типизирован универсальными типами класса `KeySelector`.

5.6. Реализация функций

Вы уже видели пользовательские функции в действии в примерах кода в этой главе. В этом разделе мы более подробно объясняем различные способы определения и параметризации функций в API DataStream.

5.6.1. Функциональные классы

Flink предоставляет все интерфейсы для пользовательских функций, таких как `MapFunction`, `FilterFunction` и `ProcessFunction`, как интерфейсы или абстрактные классы.

Функция реализуется путем реализации интерфейса или расширения абстрактного класса. В следующем примере мы реализуем `FilterFunction`, которая фильтрует строки, содержащие слово «`flink`»:

```
class FlinkFilter extends FilterFunction[String] {
  override def filter(value: String): Boolean = {
    value.contains("flink")
  }
}
```

Затем экземпляр класса функции может быть передан в качестве аргумента преобразованию фильтра:

```
val flinkTweets = tweets.filter(new FlinkFilter)
```

Функции также могут быть реализованы как анонимные классы:

```
val flinkTweets = tweets.filter(
  new RichFilterFunction[String] {
    override def filter(value: String): Boolean = {
      value.contains("flink")
    }
  })
```

Функции могут получать параметры через свой конструктор. Мы можем параметризовать приведенный выше пример и передать `String "flink"` в качестве параметра конструктору `KeywordFilter`, как показано ниже:

```
val tweets: DataStream[String] = ???
val flinkTweets = tweets.filter(new KeywordFilter("flink"))

class KeywordFilter(keyWord: String) extends FilterFunction[String] {
  override def filter(value: String): Boolean = {
    value.contains(keyWord)
  }
}
```

Когда программа отправляется на выполнение, все объекты функций сериализуются с использованием сериализации Java и отправляются на все параллельные задачи соответствующих операторов. Следовательно, после десериализации объекта все значения конфигурации остаются неизменными.

Функции должны быть сериализуемыми для Java

Flink сериализует все функциональные объекты с сериализацией Java, чтобы отправить их рабочим процессам. Все, что содержится в пользовательской функции, должно быть `Serializable`.

Если вашей функции требуется экземпляр несериализуемого объекта, вы можете либо реализовать его как расширенную функцию и инициализировать несериализуемое поле в методе `open()`, либо переопределить методы сериализации и десериализации Java.

5.6.2. Лямбда-функции

Большинство методов API DataStream принимает лямбда-функции. Лямбда-функции доступны для Scala и Java и предлагают простой и лаконичный способ реализации логики приложения, когда не требуются дополнительные операции, такие как доступ к состоянию и конфигурации. В следующем примере показана лямбда-функция, которая фильтрует твиты, содержащие слово «flink»:

```
val tweets: DataStream[String] = ...
// Фильтрующая лямбда-функция, которая проверяет, содержит ли твит слово "flink".
val flinkTweets = tweets.filter(_.contains("flink"))
```

5.6.3. Расширенные функции

Часто возникает необходимость инициализировать функцию перед обработкой первой записи или получить информацию о контексте, в котором она выполняется. API DataStream предоставляет расширенные функции, которые предоставляют больше возможностей, чем обычные функции, обсуждаемые до сих пор.

Существуют расширенные версии всех функций преобразования API DataStream, и вы можете использовать их там же, где вы можете использовать обычную функцию или лямбда-функцию. Расширенные функции можно параметризовать так же, как обычные классы функций. Имя расширенной функции начинается с `Rich`, за которым следует имя преобразования – `RichMapFunction`, `RichFlatMapFunction` и т. д.

При использовании расширенной функции вы можете реализовать два дополнительных метода жизненного цикла функции:

- метод `open()` – это метод инициализации расширенной функции. Он вызывается один раз для каждой задачи перед вызовом метода преобразования, такого как фильтр или карта. Метод `open()` обычно используется для настройки, которую нужно выполнить только один раз. Обратите внимание, что параметр `Configuration` используется только API DataSet, а не API DataStream. Следовательно, его следует игнорировать;
- метод `close()` является методом завершения для функции и вызывается один раз для каждой задачи после последнего вызова метода преобразования. Таким образом, он обычно используется для очистки и освобождения ресурсов.

Кроме того, метод `getRuntimeContext()` обеспечивает доступ к контексту времени выполнения функции. `RuntimeContext` может использоваться для получения такой информации, как параллелизм функции, индекс ее подзадачи

и имя задачи, которая выполняет функцию. Кроме того, он включает методы для доступа к разделенному состоянию. Обработка потока с сохранением состояния в Flink подробно обсуждается в разделе 7.11 В примере 5.3 показано, как использовать методы функции `RichFlatMapFunction`.

Пример 5.3 ❖ Методы `open()` и `close()` функции `RichFlatMapFunction`

```
class MyFlatMap extends RichFlatMapFunction[Int, (Int, Int)] {
  var subTaskIndex = 0

  override def open(configuration: Configuration): Unit = {
    subTaskIndex = getRuntimeContext.getIndexOfThisSubtask
    // Выполнение инициализации,
    // т.е. установление связи с внешней системой.
  }

  override def flatMap(in: Int, out: Collector[(Int, Int)]): Unit = {
    // Подзадачи индексируются по 0.
    if(in % 2 == subTaskIndex) {
      out.collect((subTaskIndex, in))
    }
    // Остальная обработка.
  }

  override def close(): Unit = {
    // Очистка, т.е. закрытие связей с внешними системами.
  }
}
```

5.7. Добавление внешних и Flink-зависимостей

Добавление внешних зависимостей – обычное требование при реализации приложений Flink. Существует множество популярных библиотек, таких как Apache Commons или Google Guava, для различных случаев использования. Более того, большинство приложений Flink зависит от одного или нескольких коннекторов Flink для приема или передачи данных во внешние системы, такие как Apache Kafka, файловые системы или Apache Cassandra. Некоторые приложения также используют библиотеки Flink для конкретных предметных областей, такие как Table API, SQL или CEP. Следовательно, большинство приложений Flink зависит не только от зависимостей Flink API DataStream и Java SDK, но и от дополнительных сторонних и внутренних зависимостей Flink.

Когда приложение выполняется, все его зависимости должны быть доступны приложению. По умолчанию кластер Flink загружает только основные зависимости API (API DataStream и DataSet). Все остальные зависимости, необходимые приложению, должны быть явно указаны.

Причина этого состоит в том, чтобы поддерживать на низком уровне количество зависимостей по умолчанию[1]. Большинство коннекторов и библиотек полагается на одну или несколько библиотек, которые обычно имеют несколько дополнительных транзитивных зависимостей. Зачастую это широко используемые библиотеки, такие как Apache Commons или Google Guava. Многие проблемы возникают из-за несовместимости между разными версиями одной и той же библиотеки, которые извлекаются из разных соединителей или непосредственно из пользовательского приложения.

Есть два способа гарантировать, что все зависимости доступны приложению при его выполнении:

1) объединение всех зависимостей в JAR-файл приложения. В результате получается автономный, но обычно довольно большой JAR-файл приложения;
2) JAR-файл зависимости можно добавить в папку ./lib в среде Flink. В этом случае зависимости подключаются в путь к классам при запуске процессов Flink. Зависимость, которая добавляется в путь к классам данным способом, доступна (и может мешать) всем приложениям, запускаемым в среде Flink.

Создание так называемого толстого JAR-файла – предпочтительный способ поддержки зависимостей приложения. Архетипы Flink Maven, которые мы представили в разделе 4.3, генерируют проекты Maven, настроенные для создания файлов JAR с расширенным набором функций, включая все необходимые зависимости. Зависимости, включенные в путь к классам процессов Flink по умолчанию, автоматически исключаются из файла JAR. Файл pom.xml сгенерированного проекта Maven содержит комментарии, объясняющие, как добавить дополнительные зависимости.

5.8. Заключение

В этой главе мы познакомились с основами API DataStream. Мы изучили структуру программ Flink и узнали, как объединить операторы преобразования данных и распределения для создания потоковых приложений. Мы также рассмотрели поддерживаемые типы данных и различные способы указания ключей и определяемых пользователем функций. Если вы сделаете шаг назад и еще раз прочитаете вводный пример, вы, надеюсь, лучше поймете, что происходит. В главе 6 все станет еще интереснее – вы узнаете, как обогатить наши программы оконными операторами и семантикой времени.

---

[1] Flink также стремится свести к минимуму свои собственные внешние зависимости и скрывает большинство из них (включая транзитивные зависимости) от пользовательских приложений, чтобы предотвратить конфликты версий.

Глава 6

Операторы на основе времени и оконные операторы

В этой главе мы рассмотрим методы API DataStream для обработки времени и основанные на времени операторы, такие как окна. Как вы узнали из раздела 2.3, операторы Flink, основанные на времени, могут работать с различными понятиями времени.

Сначала мы узнаем, как определять временные характеристики, метки времени и водяные знаки. Затем мы рассмотрим функции обработки – низкоуровневые преобразования, которые обеспечивают доступ к меткам времени и водяным знакам и могут регистрировать таймеры. Затем мы воспользуемся оконным API Flink, который предоставляет встроенные реализации наиболее распространенных типов окон. Вы также получите введение в настраиваемые, определяемые пользователем оконные операции и основные оконные конструкции, такие как назначители, триггеры и вытеснители. Наконец, мы расскажем, как присоединяться к потокам в нужное время и что делать с опоздавшими событиями.

6.1. Настройка показателей времени

Чтобы определить зависящие от времени операции в приложении для обработки распределенного потока, важно понимать значение времени. Когда вы указываете окно для сбора событий в одноминутные сегменты, какие именно события будет содержать каждый сегмент? В API DataStream вы можете использовать *показатель времени* (time characteristic), чтобы сообщить Flink, как определять время при создании окон. Показатель времени является свойством `StreamExecutionEnvironment` и принимает следующие значения.

`ProcessingTime`
: Указывает, что операторы определяют текущее время потока данных в соответствии с системными часами машины, на которой они выполняются. Окна, привязанные ко времени обработки, запускаются в зависимости

от машинного времени и включают в себя все элементы, которые случайно были доставлены оператору до этого момента времени. В целом использование времени обработки для оконных операций приводит к недетерминированным результатам, потому что содержимое окон зависит от скорости, с которой прибывают элементы. Этот параметр предлагает очень низкую задержку, поскольку задачам обработки не нужно ждать водяных знаков, чтобы продвинуть время события.

`EventTime`
: Указывает, что операторы определяют текущее время, используя информацию из самих данных. Каждое событие имеет отметку времени, а логическое время системы определяется водяными знаками. Как вы узнали из раздела 3.3.1, метки времени либо существуют в данных до входа в конвейер обработки данных, либо назначаются приложением в источниках. Временно̀е окно события запускается, когда водяные знаки объявляют, что все метки времени для определенного временно̀го интервала уже получены. Окна, привязанные ко времени событий, вычисляют детерминированные результаты, даже когда события прибывают не по порядку. Результат окна не будет зависеть от скорости чтения или обработки потока.

`IngestionTime`
: Указывает время обработки исходного оператора в виде отметки времени события для каждой принятой записи и автоматически создает водяные знаки. Это гибрид `EventTime` и `ProcessingTime`. *Время поступления события* (ingestion time) – это время, когда оно поступило в потоковый процессор. Время поступления не имеет большого практического значения по сравнению со временем события, поскольку оно не дает детерминированных результатов и обеспечивает такое же быстродействие, как время события.

В примере 6.1 показано, как настроить показатель времени в коде приложения потоковой передачи датчика, с которым вы ознакомились в разделе 5.1.

Пример 6.1 ❖ Настройка показателя времени для использования времени события

```
object AverageSensorReadings {

  // main() определяет и выполняетпрограмму DataStream.
  def main(args: Array[String]) {
    // Настройка среды выполнения.
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // Используем в приложении время события.
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    // Входящий поток данных датчика.
    val sensorData: DataStream[SensorReading] = env.addSource(...)
  }
}
```

Установка для показателя времени значения `EventTime` включает обработку отметок времени и водяных знаков и, как следствие, операции с привязкой к времени события. Конечно, вы все равно можете использовать временные окна обработки и таймеры, если выберете показатель `EventTime`.

Чтобы использовать время обработки, замените `TimeCharacteristic.EventTime` на `TimeCharacteristic.ProcessingTime`.

6.1.1. Назначение меток времени и создание водяных знаков

Как было сказано в разделе 3.3, чтобы работать с привязкой ко времени события, ваше приложение должно предоставить Flink важную информацию. Каждое событие должно быть связано с меткой времени, которая обычно указывает, когда событие действительно произошло. Потоки, привязанные ко времени событий, также должны нести водяные знаки, по которым операторы определяют текущее время события.

Метки времени и водяные знаки указаны в миллисекундах с эпохи 1970-01-01T00:00:00Z. Водяной знак сообщает операторам, что больше событий с меткой времени, меньшей или равной водяному знаку, не ожидается. Метки времени и водяные знаки могут быть назначены и сгенерированы функцией `SourceFunction` или с помощью явного определяемого пользователем назначителя меток времени и генератора водяных знаков. Назначение меток времени и создание водяных знаков в функции `SourceFunction` обсуждается в разделе 8.3.2. Здесь мы объясняем, как это сделать с помощью пользовательской функции.

Переопределение меток времени и водяных знаков, созданных источником

Если используется назначитель меток времени, все существующие метки времени и водяные знаки будут перезаписаны.

API DataStream предоставляет интерфейс `TimestampAssigner` для извлечения меток времени из элементов после того, как они были загружены в приложение потоковой передачи. Как правило, назначитель метки времени вызывается сразу после входной функции, потому что большинство назначителей при создании водяных знаков делают предположения о порядке элементов относительно их меток времени. Поскольку элементы обычно загружаются параллельно, любая операция, которая заставляет Flink перераспределять элементы по разделам параллельного потока, например изменения параллелизма, `keyBy()` или другие явные перераспределения, перемешивает порядок меток времени элементов.

Лучше всего назначать метки времени и создавать водяные знаки как можно ближе к источникам или даже внутри `SourceFunction`. В зависимости от варианта использования приложения перед назначением меток времени к входному потоку можно применить начальную фильтрацию или преобразование, если такие операции не вызывают перераспределения элементов.

Чтобы гарантировать, что операции с привязкой ко времени события ведут себя должным образом, назначитель должен быть вызван перед любым преобразованием, зависящим от времени события (например, перед первым окном с привязкой к времени события).

Назначители меток времени работают так же, как и другие операторы преобразования. Они вызываются в потоке элементов и создают новый поток элементов с метками времени и водяными знаками. Назначители меток времени не изменяют тип данных `DataStream`.

Код в примере 6.2 показывает, как использовать назначитель меток времени. В этом примере после чтения потока мы сначала применяем преобразование фильтра, а затем вызываем метод `assignTimestampsAndWatermarks()`, в котором мы определяем назначитель метки времени `MyAssigner()`.

Пример 6.2 ❖ Использование назначителя меток времени

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// Настройка параметра привязки ко времени события.
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

// Входящий поток данных.
val readings: DataStream[SensorReading] = env
  .addSource(new SensorSource)
  // Назначение метки времени и генерация водяного знака.
  .assignTimestampsAndWatermarks(new MyAssigner())
```

В приведенном выше примере `MyAssigner` может иметь тип `AssignerWithPeriodicWaterMarks` или `AssignerWithPunctuatedWatermarks`. Эти два интерфейса расширяют интерфейс `TimestampAssigner`, предоставляемый API DataStream. Первый интерфейс определяет назначители, которые периодически генерируют водяные знаки, а второй вводит водяные знаки на основе свойства входных событий. Далее мы подробно опишем оба интерфейса.

6.1.1.1. Назначитель с периодическими водяными знаками

Периодическое назначение водяных знаков означает, что мы даем системе указание выдавать водяные знаки и увеличивать время события с фиксированными интервалами машинного времени. По умолчанию установлен интервал в 200 мс, но его можно настроить с помощью метода ExecutionConfig. setAutoWatermarkInterval ():

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
// Генерация водяного знака каждые 5 секунд.
env.getConfig.setAutoWatermarkInterval(5000)
```

В предыдущем примере вы указываете программе выпускать водяные знаки каждые 5 с. Фактически каждые 5 с Flink вызывает метод `getCurrentWatermark()` класса `AssignerWithPeriodicWatermarks`. Если метод возвращает ненулевое значение с меткой времени, превышающей метку времени предыдущего

водяного знака, пересылается новый водяной знак. Эта проверка необходима для обеспечения непрерывного увеличения времени события; в противном случае водяной знак не создается.

В примере 6.3 показан назначитель с периодическими метками времени, который создает водяные знаки, отслеживая максимальную метку времени элемента, которую он видел до сих пор. При запросе нового водяного знака назначитель возвращает водяной знак с максимальной отметкой времени минус одноминутный интервал допуска.

Пример 6.3 ❖ Периодический назначитель водяных знаков

```
class PeriodicAssigner
    extends AssignerWithPeriodicWatermarks[SensorReading] {

  val bound: Long = 60 * 1000 // 1 минута в миллисекундах.
  var maxTs: Long = Long.MinValue // Наибольшая наблюдаемая метка времени.

  override def getCurrentWatermark: Watermark = {
    // Метка времени с 1-минутным допуском.
    new Watermark(maxTs - bound)
  }

  override def extractTimestamp(
      r: SensorReading,
      previousTS: Long): Long = {
    // Обновление максимальной метки времени.
    maxTs = maxTs.max(r.timestamp)
    // Возвращает записанную метку времени.
    r.timestamp
  }
}
```

API DataStream предоставляет реализации для двух распространенных случаев назначителей меток времени с периодическими водяными знаками. Если ваши элементы ввода имеют метки времени, которые монотонно увеличиваются, вы можете использовать метод быстрого вызова `assignAscendingTimeStamps`. Этот метод использует текущую метку времени для создания водяных знаков, поскольку более ранние метки времени недоступны. Ниже показано, как создавать водяные знаки для возрастающих отметок времени:

```
val stream: DataStream[SensorReading] = ...
val withTimestampsAndWatermarks = stream
  .assignAscendingTimestamps(e => e.timestamp)
```

Другой распространенный случай периодической генерации водяных знаков – это когда вы знаете максимальную задержку, с которой столкнетесь во входном потоке, – максимальную разницу между меткой времени элемента и самой большой меткой времени из всех ранее загруженных элементов. Для таких случаев Flink предоставляет `BoundedOutOfOrdernessTimeStampExtractor`, который принимает в качестве аргумента максимальную ожидаемую задержку:

```
val stream: DataStream[SensorReading] = ...
val output = stream.assignTimestampsAndWatermarks(
new BoundedOutOfOrdernessTimestampExtractor[SensorReading](
  Time.seconds(10))(e =>.timestamp)
```

В этом коде элементы могут опаздывать до 10 с. Это означает, что, если разница между временем события элемента и максимальной отметкой времени всех предыдущих элементов превышает 10 с, элемент может поступить на обработку после завершения соответствующего вычисления и выдачи результата. Flink предлагает различные стратегии для обработки таких опоздавших событий, и мы обсудим их позже в разделе 6.5.

6.1.1.2. Назначитель прерывистых водяных знаков

Иногда входной поток содержит специальные кортежи или маркеры, указывающие на прогресс потока. Flink предоставляет интерфейс `AssignerWithPunctuatedWatermarks` для таких случаев или когда водяные знаки могут быть определены на основе какого-либо другого свойства входных элементов. Он определяет метод `checkAndGetNextWatermark()`, который вызывается для каждого события сразу после `extractTimestamp()`. Этот метод может решить, создавать новый водяной знак или нет. Новый водяной знак создается, если метод возвращает ненулевой водяной знак, размер которого превышает размер последнего отправленного водяного знака.

В примере 6.4 показан назначитель *прерывистых водяных знаков* (punctuated watermark), который выпускает водяной знак при каждом считывании, полученном от датчика с идентификатором `"sensor_1"`.

Пример 6.4 ❖ Назначитель прерывистых водяных знаков

```
class PunctuatedAssigner
    extends AssignerWithPunctuatedWatermarks[SensorReading] {

  val bound: Long = 60 * 1000 // 1 минута в миллисекундах.

  override def checkAndGetNextWatermark(
      r: SensorReading,
      extractedTS: Long): Watermark = {
    if (r.id == "sensor_1") {
      // Выпускаем водяной знак, если данные с датчика sensor_1.
      new Watermark(extractedTS - bound)
    } else {
      // Не выпускаем водяной знак.
      null
    }
  }

  override def extractTimestamp(
      r: SensorReading,
      previousTS: Long): Long = {
    // Назначаем метку времени записи.
    r.timestamp
  }
}
```

6.1.2. Водяные знаки, задержка и полнота

До сих пор мы обсуждали, как создавать водяные знаки с помощью `TimestampAssigner`. Что мы еще не обсуждали, так это влияние водяных знаков на ваше потоковое приложение.

Водяные знаки используются для достижения баланса задержки и полноты результата. Они определяют, как долго ждать прибытия данных перед выполнением вычислений, таких как завершение вычисления окна и выдача результата. Оператор, основанный на времени события, использует водяные знаки для определения полноты полученных записей и хода выполнения операции. На основе полученных водяных знаков оператор вычисляет момент времени, до которого он ожидает получить соответствующие входные записи.

Однако реальность такова, что у нас никогда не может быть идеальных водяных знаков, потому что это означало бы нашу уверенность в отсутствии отложенных записей. На практике для создания водяных знаков в ваших приложениях нужно сделать обоснованное предположение и использовать эвристику. Вам необходимо использовать любую имеющуюся у вас информацию об источниках, сети и разделах для оценки прогресса и верхней границы задержки ваших входных записей. Оценки означают, что есть место для ошибок, и в этом случае вы можете создать неточные водяные знаки, что приведет к задержке данных или ненужному увеличению задержки приложения. Имея это в виду, вы можете использовать водяные знаки, чтобы сбалансировать задержку и полноту результата.

Если вы генерируете *разреженные водяные знаки* (loose watermarks), когда водяные знаки намного отстают от меток времени обработанных записей, вы увеличиваете задержку получаемых результатов. Возможно, вы могли бы получить результат раньше, но пришлось дождаться водяного знака. Более того, размер состояния обычно увеличивается, потому что приложению необходимо буферизовать больше данных, пока оно не сможет выполнить вычисление. Однако вы можете быть уверены, что при выполнении вычислений доступны все соответствующие данные.

С другой стороны, если вы создаете очень жесткие водяные знаки – водяные знаки, которые могут быть больше, чем метки времени некоторых более поздних записей, – вычисления на основе времени могут быть выполнены до того, как будут получены все соответствующие данные. Хотя это может привести к неполным или неточным результатам, эти результаты выдаются своевременно и с меньшими задержками.

В отличие от пакетных приложений, которые построены на предположении, что все данные доступны, компромисс между задержкой и полнотой является фундаментальной характеристикой приложений потоковой обработки, которые обрабатывают неограниченные данные по мере их поступления. Водяные знаки – мощный способ контролировать поведение приложения во времени. Помимо водяных знаков, Flink имеет множество опций для настройки точного поведения операций, зависящих от времени, таких как функции обработки и триггеры окон, и предлагает различные способы обработки опоздавших данных, которые обсуждаются в разделе 6.5.

6.2. Функции процесса

Несмотря на то что информация о времени и водяные знаки имеют решающее значение для многих потоковых приложений, вы могли заметить, что мы не можем получить к ним доступ с помощью основных преобразований API DataStream, которые мы видели до сих пор. Например, функция `MapFunction` не имеет доступа к меткам времени или времени текущего события.

API DataStream предоставляет семейство низкоуровневых преобразований, *функций обработки* (process function), которые также могут получать доступ к отметкам времени и водяным знакам записи и регистрировать таймеры, срабатывающие в определенное время в будущем. Более того, функции обработки имеют боковые выходы для передачи записей в несколько выходных потоков. Функции обработки обычно используются для создания приложений, управляемых событиями, и для реализации пользовательской логики, для которой предопределенные окна и преобразования могут не подходить. Например, большинство операторов Flink для поддержки SQL реализовано с использованием функций обработки.

В настоящее время Flink предоставляет восемь различных функций процессов: `ProcessFunction`, `KeyedProcessFunction`, `CoProcessFunction`, `ProcessJoinFunction`, `BroadcastProcessFunction`, `KeyedBroadcastProcessFunction`, `ProcessWindowFunction` и `ProcessAllWindowFunction`. Как видно из названий, эти функции применимы в разных контекстах. Однако у них очень похожий набор опций. Мы продолжим обсуждение этих общих опций, подробно рассмотрев функцию `KeyedProcessFunction`.

`KeyedProcessFunction` – это очень универсальная функция, которая может применяться к `KeyedStream`. Функция вызывается для каждой записи потока и возвращает ноль, одну или несколько записей. Все функции процесса реализуют интерфейс `RichFunction` и, следовательно, предлагают методы `open()`, `close()` и `getRuntimeContext()`. Кроме того, `cessFunctionKeyedPro[KEY, IN, OUT]` предоставляет следующие два метода:

1) `processElement(v: IN, ctx: Context, out: Collector[OUT])` вызывается для каждой записи потока. Как правило, записи результатов передаются в `Collector`. Объект `Context` – это характерная особенность именно функций обработки. Он дает доступ к метке времени и ключу текущей записи, а также к TimerService. Более того, контекст может отправлять записи на побочные выходы;
2) `onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT])` – это функция обратного вызова, которая вызывается при срабатывании ранее зарегистрированного таймера. Аргумент `timestamp` дает метку времени сработавшего таймера, а `Collector` разрешает выпуск записей. `OnTimerContext` предоставляет те же услуги, что и объект `Context` метода `processElement()`, а также возвращает область времени (время обработки или время события) запускающего триггера.

6.2.1. TimerService и таймеры

`TimerService` объектов `Context` и `OnTimerContext` предлагает следующие методы:

- `currentProcessingTime(): Long`: возвращает текущее время обработки;
- `currentWatermark(): Long`: возвращает метку времени текущего водяного знака;
- `registerProcessingTimeTimer(timestamp: Long)`: регистрирует таймер времени обработки для текущего ключа. Таймер сработает, когда время обработки исполняющей машины достигнет указанной отметки времени;
- `registerEventTimeTimer(timestamp: Long)`: регистрирует таймер времени события для текущего ключа. Таймер срабатывает, когда водяной знак обновляется до отметки времени, которая равна отметке времени таймера или превышает ее;
- `deleteProcessingTimeTimer(timestamp: Long)`: модуль удаляет таймер времени обработки, который был ранее зарегистрирован для текущего ключа. Если такого таймера нет, метод не действует;
- `deleteEventTimeTimer(timestamp: Long)`: модуль удаляет таймер времени события, который был ранее зарегистрирован для текущего ключа. Если такого таймера нет, метод не действует.

Когда срабатывает таймер, вызывается функция обратного вызова `onTimer()`. Методы `processElement()` и `onTimer()` синхронизируются для предотвращения одновременного доступа и манипулирования состоянием.

Таймеры в потоках без ключа

Таймеры могут быть зарегистрированы только для потоков с ключом. Обычным вариантом использования таймеров является очистка состояния с ключом после некоторого периода бездействия или реализация настраиваемой логики управления окнами на основе времени. Чтобы использовать таймеры в потоке без ключа, вы можете создать поток с ключом, используя `KeySelector` с постоянным фиктивным ключом. Обратите внимание, что это переместит все данные в одну задачу, так что оператор будет фактически выполняться с параллелизмом 1.

Для каждого ключа и метки времени может быть зарегистрирован ровно один таймер, что означает, что каждый ключ может иметь несколько таймеров, но только по одному для каждой метки времени. По умолчанию `KeyedProcessFunction` хранит метки времени всех таймеров в очереди приоритетов в куче. Однако вы можете настроить бэкенд состояния RocksDB так, чтобы она также сохраняла таймеры.

Таймеры сохраняются в контрольной точке вместе с любым другим состоянием функции. Если приложению необходимо восстановиться после сбоя, все таймеры времени обработки, истекшие во время перезапуска приложения, сработают немедленно, когда приложение возобновит работу. Это также верно для таймеров времени обработки, которые сохраняются в точке

сохранения. Таймеры всегда устанавливаются асинхронно, за одним исключением. Если вы используете бэкенд состояния RocksDB с инкрементными контрольными точками и хранением таймеров в куче (настройка по умолчанию), они устанавливаются синхронно. В этом случае рекомендуется использовать таймеры не слишком активно, чтобы избежать длительного времени установки контрольных точек.

Таймеры, зарегистрированные для отметки времени в прошлом, не отбрасываются, а также обрабатываются. Таймеры времени обработки срабатывают сразу после возврата из метода регистрации. Таймеры времени события срабатывают при обработке следующего водяного знака.

В следующем коде показано, как применить `KeyedProcessFunction` к `KeyedStream`. Функция контролирует температуру датчиков и выдает предупреждение, если температура датчика монотонно увеличивается в течение 1 с во время обработки:

```
val warnings = readings
  // Ключ по id датчика.
  .keyBy(_.id)
  // Используем ProcessFunction для мониторинга температуры.
  .process(new TempIncreaseAlertFunction)
```

Реализация `TempIncreaseAlterFunction` показана в примере 6.5.

Пример 6.5 ❖ Функция `KeyedProcessFunction`, которая выдает предупреждение, если температура датчика монотонно увеличивается в течение 1 с во время обработки

```
/** Выдает предупреждение, если температура датчика
монотонно увеличивается в течение 1 секунды (во время обработки).
*/
class TempIncreaseAlertFunction
    extends KeyedProcessFunction[String, SensorReading, String] {
  // Сохранение последнего считанного значения температуры.
  lazy val lastTemp: ValueState[Double] = getRuntimeContext.getState(
      new ValueStateDescriptor[Double]("lastTemp", Types.of[Double]))
  // Сохранение метки времени текущего активного таймера.
  lazy val currentTimer: ValueState[Long] = getRuntimeContext.getState(
      new ValueStateDescriptor[Long]("timer", Types.of[Long]))

  override def processElement(
      r: SensorReading,
      ctx: KeyedProcessFunction[String, SensorReading, String]#Context,
      out: Collector[String]): Unit = {
    // Получение предыдущей температуры.
    val prevTemp = lastTemp.value()
    // Обновление последнего значения температуры.
    lastTemp.update(r.temperature)

    val curTimerTimestamp = currentTimer.value();
```

```
    if (prevTemp == 0.0 || r.temperature < prevTemp) {
    // Температура уменьшилась; удаляем текущий таймер.
    ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp)
    currentTimer.clear()
  } else if (r.temperature > prevTemp && curTimerTimestamp == 0) {
    // Температура увеличилась, а у нас еще не установлен таймер.
    // Устанавливаем таймер времени обработки на сейчас + 1 секунда.
    val timerTs = ctx.timerService().currentProcessingTime() + 1000
    ctx.timerService().registerProcessingTimeTimer(timerTs)
    // Запоминаем текущий таймер.
    currentTimer.update(timerTs)
  }
}

override def onTimer(
    ts: Long,
    ctx: KeyedProcessFunction[String, SensorReading, String]#OnTimerContext,
    out: Collector[String]): Unit = {
  out.collect("Температура датчика '" + ctx.getCurrentKey +
    "' непрерывно возрастает в течение 1 секунды.")
  currentTimer.clear()
 }
}
```

6.2.2. Передача потоков на боковые выходы

Большинство операторов API DataStream имеют один выход – они создают один поток результатов с определенным типом данных. Только оператор `split` позволяет разделить поток на несколько потоков одного типа. *Боковые выходы* (side outputs) – это особая возможность функций процесса, позволяющая выводить несколько потоков из функции, возможно, разных типов. Боковой выход идентифицируется объектом `OutputTag[X]`, где `X` – тип результирующего бокового выходного потока. Функции процесса могут отправлять запись на один или несколько боковых выходов через объект `Context`.

В примере 6.6 показано, как передавать данные из `ProcessFunction` через `DataStream` бокового выхода.

Пример 6.6 ❖ Применение `ProcessFunction`, которая передает поток на боковой выход

```
val monitoredReadings: DataStream[SensorReading] = readings
  // Отслеживание потока на элементы с температурой замерзания.
  .process(new FreezingMonitor)

// Получение и печать сигнал о замерзании на боковом выходе.
monitoredReadings
  .getSideOutput(new OutputTag[String]("freezing-alarms"))
  .print()

// Печать главного выхода.
readings.print()
```

В примере 6.7 показана функция FreezingMonitor, которая отслеживает поток показаний датчика и выдает предупреждение на боковой выход для показаний с температурой ниже 32 °F.

Пример 6.7 ❖ Функция ProcessFunction, которая передает записи на боковой выход

```
/** Передает сигнал о замерзании при появлении измерения
  * с температурой ниже 32F. */
class FreezingMonitor extends ProcessFunction[SensorReading, SensorReading] {

  // Определяем тег побочного выхода.
  lazy val freezingAlarmOutput: OutputTag[String] =
    new OutputTag[String]("freezing-alarms")

  override def processElement(
      r: SensorReading,
      ctx: ProcessFunction[SensorReading, SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {
    // Передаем сигнал о замерзании, если температура ниже 32 °F.
    if (r.temperature < 32.0) {
      ctx.output(freezingAlarmOutput, s"Freezing Alarm for ${r.id}")
    }
    // Пересылаем все измерения на обычный выход.
    out.collect(r)
  }
}
```

6.2.3. CoProcessFunction

Для низкоуровневых операций с двумя входами API DataStream также предоставляет CoProcessFunction. Подобно CoFlatMapFunction, CoProcessFunction предлагает метод преобразования для каждого ввода, processElement1() и processElement2(). Подобно ProcessFunction, оба метода вызываются с объектом Context, который предоставляет доступ к метке времени элемента или таймера, TimerService и боковым выходам. CoProcessFunction также предоставляет метод обратного вызова onTimer(). В примере 6.8 показано, как применить функцию CoProcessFunction для объединения двух потоков.

Пример 6.8 ❖ Применение CoProcessFunction

```
// Входной поток датчика.
val sensorData: DataStream[SensorReading] = ...

// Управляемая пересылка данных от датчиков.
val filterSwitches: DataStream[(String, Long)] = env
  .fromCollection(Seq(
    ("sensor_2", 10 * 1000L), // Перенаправление sensor_2 в течение 10 секунд.
    ("sensor_7", 60 * 1000L)) // Перенаправление sensor_7 в течение 1 минуты.
  )
val forwardedReadings = readings
```

```
  // Соединение датчиков и переключателя.
  .connect(filterSwitches)
  // Ключ по индексам датчиков.
  .keyBy(_.id, _._1)
  // Применяем фильтрующую CoProcessFunction.
  .process(new ReadingFilter)
```

Реализация функции `ReadingFilter`, которая динамически фильтрует поток показаний датчиков на основе потока переключателей фильтров, показана в примере 6.9.

Пример 6.9 ❖ Реализация функции `CoProcess`, которая динамически фильтрует поток показаний датчиков

```
class ReadingFilter
    extends CoProcessFunction[SensorReading, (String, Long), SensorReading] {

  // Разрешаем перенаправление.
  lazy val forwardingEnabled: ValueState[Boolean] = getRuntimeContext.getState(
      new ValueStateDescriptor[Boolean]("filterSwitch", Types.of[Boolean]))

  // Удерживаем метку времени для текущего активного disableTimer.
  lazy val disableTimer: ValueState[Long] = getRuntimeContext.getState(
      new ValueStateDescriptor[Long]("timer", Types.of[Long]))

  override def processElement1(
      reading: SensorReading,
      ctx: CoProcessFunction[SensorReading, (String, Long), SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {
  // Проверяем, можно ли перенаправить данные.
  if (forwardingEnabled.value()) {
    out.collect(reading)
  }
}

override def processElement2(
    switch: (String, Long),
    ctx: CoProcessFunction[SensorReading, (String, Long), SensorReading]#Context,
    out: Collector[SensorReading]): Unit = {
  // Разрешаем пересылку данных.
  forwardingEnabled.update(true)
  // Устанавливаем disableTimer.
  val timerTimestamp = ctx.timerService().currentProcessingTime() + switch._2
  val curTimerTimestamp = disableTimer.value()
    if (timerTimestamp > curTimerTimestamp) {
    // Удаляем текущий таймер и регистрируем новый таймер.
    ctx.timerService().deleteEventTimeTimer(curTimerTimestamp)
    ctx.timerService().registerProcessingTimeTimer(timerTimestamp)
    disableTimer.update(timerTimestamp)
  }
}

  override def onTimer(
```

```
      ts: Long,
      ctx: CoProcessFunction[SensorReading, (String, Long), SensorReading]
                            #OnTimerContext,
      out: Collector[SensorReading]): Unit = {
    // Удаляем состояние; пересылка будет отключена по умолчанию.
    forwardingEnabled.clear()
    disableTimer.clear()
  }
}
```

6.3. Оконные операторы

Окна – это обычные операции в потоковых приложениях. Они позволяют выполнять преобразования, такие как агрегирование на *ограниченных интервалах* неограниченного потока. Обычно эти интервалы определяются с использованием временнòй логики. Оконные операторы позволяют группировать события в сегменты конечного размера и применять вычисления к ограниченному содержимому этих сегментов. Например, оконный оператор может сгруппировать события потока в окна по 5 мин и подсчитать для каждого окна, сколько событий было получено.

API DataStream предоставляет встроенные методы для наиболее распространенных оконных операций, а также очень гибкий механизм управления окнами для определения пользовательской логики работы с окнами. В этом разделе мы покажем вам, как определять оконные операторы, представим встроенные типы окон API DataStream, обсудим функции, которые могут быть применены к окну, и наконец объясним, как определить пользовательскую логику работы с окнами.

6.3.1. Определение оконных операторов

Оконные операторы могут применяться как к потоку с ключом, так и к потоку без ключа. Оконные операторы в окнах с ключами выполняются параллельно, а окна без ключей обрабатываются в одном потоке.

Чтобы создать оконный оператор, вам необходимо указать два оконных компонента.

1. *Назначитель окна*, определяющий, как элементы входного потока группируются в окна. Назначитель окна создает `WindowedStream` (или `AllWindowedStream`, если применяется к потоку данных без ключа).
2. *Оконная функция*, которая применяется к `WindowedStream` (или `AllWindowedStream`) и обрабатывает элементы, назначенные окну.

В следующем коде показано, как указать назначитель окна и оконную функцию для потока с ключом и без ключа:

```
// Определяем оконный оператор с ключом.
stream
```

```
.keyBy(...)
.window(...) // Определяем назначитель окна.
.reduce/aggregate/process(...) // Определяем оконную функцию.
// Определяем оконный оператор без ключа.
stream
.windowAll(...) // Определяем назначитель окна.
.reduce/aggregate/process(...) // Определяем оконную функцию.
```

В оставшейся части главы мы сосредоточимся только на окнах с ключами. Окна без ключей (так называемые all-windows в DataStream API) ведут себя точно так же, за исключением того, что они собирают все данные и не оцениваются параллельно.

Обратите внимание, что вы можете настроить оконный оператор, предоставив настраиваемый триггер или вытеснитель и объявив стратегии для работы с опоздавшими элементами. Пользовательские оконные операторы окна подробно рассмотрены ниже в этом разделе.

6.3.2. Встроенные средства назначения окон

Flink предоставляет встроенные средства назначения окон для наиболее распространенных случаев использования окон. Все обсуждаемые здесь назначители основаны на времени и были представлены в разделе 2.2.2. Данные назначители окон назначают элемент на основе его метки времени события или текущего времени обработки для окон. Временные окна имеют отметку времени начала и конца.

Все встроенные средства назначения окон предоставляют триггер по умолчанию, который запускает оценку окна после того, как время (обработки или события) доходит до конца окна. Важно отметить, что окно создается, когда ему назначается первый элемент. Flink никогда не будет обрабатывать пустые окна.

Окна на основе подсчета

В дополнение к окнам, основанным на времени, Flink также поддерживает окна, основанные на подсчете, – окна, которые группируют фиксированное количество элементов в том порядке, в котором они прибывают к оператору окна. Поскольку они зависят от порядка приема, окна на основе подсчета не являются детерминированными. Более того, они могут вызывать проблемы, если используются без настраиваемого триггера, который в какой-то момент отбрасывает неполные и устаревшие окна.

Встроенные средства назначения окон Flink создают окна типа `TimeWindow`. Этот тип окна по существу представляет собой временной интервал между двумя временными метками, где начало включено, а конец – исключен. Он предоставляет методы для получения границ окна, проверки пересечения окон и объединения перекрывающихся окон.

Далее мы покажем различные встроенные средства назначения окон в API DataStream и способы их использования для определения оконных операторов.

6.3.2.1. Шагающие окна

Назначение шагающего окна помещает элементы в неперекрывающиеся окна фиксированного размера, как показано на рис. 6.1.

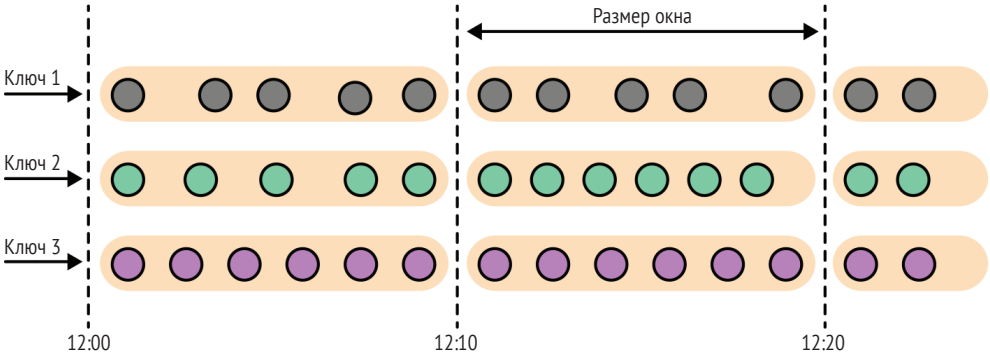

Рис. 6.1 ❖ Назначение шагающего окна помещает элементы в неперекрывающиеся окна фиксированного размера

API DataStream предоставляет два назначителя – `TumblingEventTimeWindows` и `TumblingProcessingTimeWindows` – для шагающих окон времени события и времени обработки соответственно. Назначитель шагающего окна получает один параметр – размер окна в единицах времени; это можно указать с помощью метода назначителя `of(Time size)`. Временной интервал может быть установлен в миллисекундах, секундах, минутах, часах или днях.

В следующем коде показано, как определить окна с привязкой ко времени события и времени обработки в потоке данных измерений датчика:

```
val sensorData: DataStream[SensorReading] = ...

val avgTemp = sensorData
  .keyBy(_.id)
    // Групповые данные в окне времени события.
  .window(TumblingEventTimeWindows.of(Time.seconds(1)))
  .process(new TemperatureAverager)

val avgTemp = sensorData
  .keyBy(_.id)
    // Групповые данные в окне времени обработки.
  .window(TumblingProcessingTimeWindows.of(Time.seconds(1)))
  .process(new TemperatureAverager)
```

Определение окна выглядело немного иначе в нашем первом примере API DataStream в разделе 2.2.2. Там мы определили шагающее окно времени события, используя метод `timeWindow(size)`, который является ярлыком для любого `window.(TumblingEventTimeWindows.of(size))` или для `window.(TumblingProcessingTimeWindows.of(size))` в зависимости от настройки показателя времени. В следующем коде показано, как использовать этот ярлык:

```
val avgTemp = sensorData
  .keyBy(_.id)
    // Ярлык для window.(TumblingEventTimeWindows.of(size)).
  .timeWindow(Time.seconds(1))
  .process(new TemperatureAverager)
```

По умолчанию шагающие окна отсчитываются от времени эпохи 1970-01-01-00:00:00.000. Например, назначитель размером один час будет определять окна в 00:00:00, 01:00:00, 02:00:00 и т. д. В качестве альтернативы вы можете указать смещение в качестве второго параметра назначителя. Следующий код показывает окна со смещением 15 мин, которые начинаются в 00:15:00, 01:15:00, 02:15:00 и т. д.

```
val avgTemp = sensorData
  .keyBy (_. id)
    // Группируем показания окна размером 1 час со смещением 15 минут.
  .window (TumblingEventTimeWindows.of (Время.часы (1), Время.минуты (15)))
  .process (новый TemperatureAverager)
```

6.3.2.2. Скользящие окна

Назначитель скользящего окна назначает элементы окнам фиксированного размера, которые сдвигаются на заданный интервал скольжения, как показано на рис. 6.2.

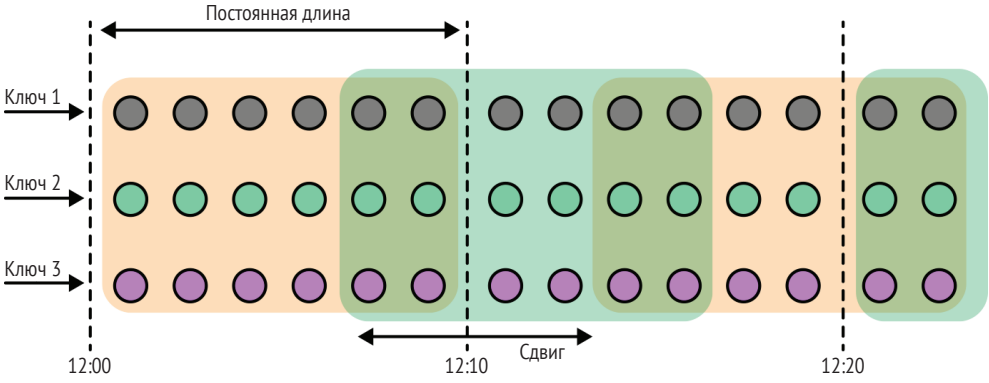

Рис. 6.2 ❖ Назначитель скользящего окна помещает элементы в окна фиксированного размера, возможно, в перекрывающиеся окна

Для скользящего окна вы должны указать размер окна и интервал сдвига, который определяет, как часто запускается новое окно. Когда интервал сдвига меньше, чем размер окна, окна перекрываются, и элементы могут быть назначены более чем одному окну. Если сдвиг больше, чем размер окна, некоторые элементы могут оказаться между окнами и, следовательно, будут отброшены.

Следующий код показывает, как сгруппировать показания датчика в скользящих окнах размером 1 ч с 15-минутным интервалом сдвига. Каждый

результат измерения будет добавлен в четыре окна. API DataStream предоставляет назначители по времени события и времени обработки, а также методы быстрого доступа; величина сдвига может быть установлена в качестве третьего параметра назначителя окна:

```
// Назначитель скользящего окна времени события.
val slidingAvgTemp = sensorData
  .keyBy(_.id)
    // Создаем окно времени событий длиной 1 час каждые 15 минут.
  .window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(15)))
  .process(new TemperatureAverager)

// Назначитель скользящего окна времени обработки.
val slidingAvgTemp = sensorData
  .keyBy(_.id)
    // Создаем окно времени обработки длиной 1 час каждые 15 минут.
  .window(SlidingProcessingTimeWindows.of(Time.hours(1), Time.minutes(15)))
  .process(new TemperatureAverager)

// Назначение скользящего окна с использованием ярлыка.
val slidingAvgTemp = sensorData
  .keyBy(_.id)
    // Ярлык для window.(SlidingEventTimeWindow.of(size, slide)).
  .timeWindow(Time.hours(1), Time(minutes(15)))
  .process(new TemperatureAverager)
```

6.3.2.3. Окна сеанса

Назначитель окна сеанса помещает элементы в неперекрывающиеся окна активности различного размера. Границы окон сеанса определяются промежутками бездействия –интервалами времени, когда не поступают записи. На рис. 6.3 показано, как элементы входных данных назначаются окнам сеанса.

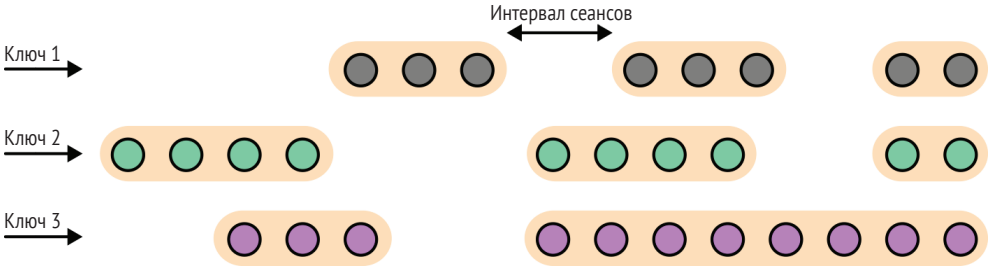

Рис. 6.3 ❖ Назначитель окна сеанса помещает элементы в окна различного размера, определяемые промежутком между сеансами

В следующих примерах показано, как сгруппировать показания датчиков в окна сеансов, где каждый сеанс определяется 15-минутным периодом бездействия:

```
// Назначитель окна сеанса по времени события.
val sessionWindows = sensorData
```

```
  .keyBy(_.id)
    // Создаем окно сеанса по времени события с разрывом 15 минут.
  .window(EventTimeSessionWindows.withGap(Time.minutes(15)))
  .process(...)

// Назначитель окна сеанса по времени обработки.
val sessionWindows = sensorData
  .keyBy(_.id)
    // Создаем окно сеанса по времени события с разрывом 15 минут.
  .window(ProcessingTimeSessionWindows.withGap(Time.minutes(15)))
  .process(...)
```

Поскольку начало и конец окна сеанса зависят от полученных элементов, назначитель окна не может сразу назначить все элементы правильному окну. Вместо этого назначитель `SessionWindows` изначально сопоставляет каждый входящий элемент с его собственным окном с меткой времени элемента в качестве времени начала и промежутком времени в сеансе в качестве размера окна. Впоследствии он объединяет все окна с перекрывающимися диапазонами.

6.3.3. Применение функций в окнах

Оконные функции определяют вычисления, которые выполняются над элементами окна. К окну можно применить два типа функций.

1. *Функции инкрементного агрегирования* применяются непосредственно, когда элемент добавляется к окну, а также удерживает и обновляет одно значение как состояние окна. Эти функции обычно занимают очень мало места и в итоге выдают агрегированное значение. `ReduceFunction` и `AggregateFunction` – это функции инкрементного агрегирования.
2. *Функции полного окна* собирают все элементы окна и перебирают список всех собранных элементов при вычислениях. Такие функции обычно требуют больше места, но допускают более сложную логику, чем функции инкрементного агрегирования. `ProcessWindowFunction` – это функция полного окна.

В этом разделе мы обсуждаем различные типы функций, которые могут применяться к окну для выполнения агрегирования или произвольных вычислений над содержимым окна. Мы также покажем, как совместно применять функции инкрементного агрегирования и функции полного окна в оконном операторе.

6.3.3.1. ReduceFunction

Функция `ReduceFunction` была представлена в разделе 5.2.2 при обсуждении выполнения агрегирования в потоках с ключами. `ReduceFunction` принимает два значения одного типа и объединяет их в одно значение того же типа. Применительно к разбитому на окна потоку `ReduceFunction` инкрементно

агрегирует элементы, назначенные окну. В окне хранится только текущий результат агрегирования – одно значение входного (и выходного) типа ReduceFunction. При получении нового элемента функция ReduceFunction вызывается с новым элементом и текущим значением, считываемым из состояния окна. Состояние окна заменяется результатом функции ReduceFunction.

Преимущества применения ReduceFunction к окну – постоянный и небольшой размер состояния для каждого окна и простой интерфейс функций. Однако приложения для ReduceFunction обычно ограничиваются простыми агрегатами, поскольку типы ввода и вывода должны быть одинаковыми.

В примере 6.10 показана лямбда-функция reduce, которая вычисляет минимальную температуру для каждого датчика каждые 15 с.

Пример 6.10 ❖ Применение лямбда-функции reduce к WindowedStream

```
val minTempPerWindow: DataStream[(String, Double)] = sensorData
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(15))
  .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

6.3.3.2. AggregateFunction

Подобно ReduceFunction, AggregateFunction также инкрементно применяется к элементам, которые назначены окну. Более того, состояние оконного оператора с AggregateFunction также состоит из одного значения.

Хотя интерфейс AggregateFunction намного более гибкий, его также сложнее реализовать по сравнению с интерфейсом ReduceFunction. В следующем коде показан интерфейс AggregateFunction:

```
public interface AggregateFunction<IN, ACC, OUT> extends Function, Serializable {

  // Создаем новый аккумулятор для запуска новой агрегации.
  ACC createAccumulator();

  // Добавляем входной элемент в аккумулятор и возвращает аккумулятор.
  ACC add(IN value, ACC accumulator);

  // Вычисляем результат из аккумулятора и возвращаем его.
  OUT getResult(ACC accumulator);

  // Объединяем два аккумулятора и возвращаем результат.
  ACC merge(ACC a, ACC b);
}
```

Интерфейс определяет тип ввода IN, тип аккумулятора ACC и тип результата OUT. В отличие от ReduceFunction промежуточный тип данных и тип вывода не зависят от типа ввода.

В примере 6.11 показано, как использовать функцию AggregateFunction для вычисления средней температуры показаний датчика для каждого окна. Аккумулятор поддерживает текущую сумму и подсчет, а метод getResult() вычисляет среднее значение.

Пример 6.11 ❖ Применение AggregateFunction к WindowedStream

```
val avgTempPerWindow: DataStream[(String, Double)] = sensorData
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(15))
.aggregate(new AvgTempFunction)

// AggregateFunction для вычисления средней температуры по датчику.
// Аккумулятор хранит сумму температур и счетчик событий.
class AvgTempFunction
    extends AggregateFunction
  [(String, Double), (String, Double, Int), (String, Double)] {

  override def createAccumulator() = {
  ("", 0.0, 0)
  }

  override def add(in: (String, Double), acc: (String, Double, Int)) = {
  (in._1, in._2 + acc._2, 1 + acc._3)
  }

  override def getResult(acc: (String, Double, Int)) = {
  (acc._1, acc._2 / acc._3)
  }

  override def merge(acc1: (String, Double, Int), acc2: (String, Double, Int)) = {
  (acc1._1, acc1._2 + acc2._2, acc1._3 + acc2._3)
  }
}
```

6.3.3.3. ProcessWindowFunction

ReduceFunction и AggregateFunction итеративно применяются к событиям, которые назначены окну. Однако иногда нам нужен доступ ко всем элементам окна для выполнения более сложных вычислений, таких как вычисление медианы значений в окне или поиск наиболее часто встречающегося значения. Для таких приложений не подходят ни ReduceFunction, ни AggregateFunction. API DataStream предлагает функцию ProcessWindowFunction для выполнения произвольных вычислений над содержимым окна.

API DataStream в Flink 1.7 имеет интерфейс WindowFunction. Впоследствии WindowFunction была заменена функцией ProcessWindowFunction и здесь не обсуждается.

В следующем коде показан интерфейс функции ProcessWindowFunction:

```
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
    extends AbstractRichFunction {

  // Обработка окна.
  void process(
    KEY key, Context ctx, Iterable<IN> vals, Collector<OUT> out) throws Exception;
```

```
  // Удаление любого оконного состояния, если окно очищено.
  public void clear(Context ctx) throws Exception {}

  // Контекст хранит метаданные окна.
  public abstract class Context implements Serializable {

    // Возвращает метаданные окна.
    public abstract W window();

    // Возвращает текущее время обработки.
    public abstract long currentProcessingTime();

    // Возвращает текущий водяной знак с привязкой к событию.
    public abstract long currentWatermark();

    // Сссылка на пооконное состояние.
    public abstract KeyedStateStore windowState();

    // Ссылка на глобальное состояние.
    public abstract KeyedStateStore globalState();

    // Отправка записи в боковой выход, определенный в OutputTag.
    public abstract <X> void output(OutputTag<X> outputTag, X value);
  }
}
```

Метод `process()` вызывается с ключом окна `Iterable` для доступа к элементам окна и `Collector` для выдачи результатов. Более того, этот метод имеет параметр `Context`, как и другие методы процесса. Объект `Context` функции `ProcessWindowFunction` предоставляет доступ к метаданным окна, текущему времени обработки и водяному знаку, хранилищам состояний для управления глобальными состояниями для каждого окна и ярлыкам, а также боковым выходам для выдачи результатов.

Мы уже обсуждали некоторые возможности объекта `Context` при представлении функций процесса, такие как доступ к текущей обработке и выходным данным во время события и боковым выходам. Однако объект `Context` функции `ProcessWindowFunction` также предлагает уникальные возможности. Метаданные окна обычно содержат информацию, которая может использоваться в качестве идентификатора для окна, например метки времени начала и окончания в случае временного окна.

Еще одна опция – глобальные состояния для каждого окна и для каждого ключа. *Глобальное состояние* (global state) для каждого ключа относится к ключевому состоянию, которое не привязано к какому-либо окну, в то время как состояние каждого окна относится к экземпляру окна, который в настоящее время оценивается. Состояние каждого окна полезно для хранения информации, которая должна совместно использоваться несколькими вызовами метода `process()` в одном окне, что может произойти из-за настройки допустимой задержки или использования настраиваемого триггера. Функция `ProcessWindowFunction`, которая использует состояние кон-

кретного окна, должна реализовать свой метод `clear()` для очистки любого состояния этого окна перед очисткой окна. Глобальное состояние можно использовать для обмена информацией между несколькими окнами с одним и тем же ключом.

В примере 6.12 поток данных датчика сгруппирован в шагающие окна продолжительностью 5 с, а для вычисления самой низкой и самой высокой температуры, встреченной в окне, используется функция `ProcessWindowFunction`. Она выдает по одной записи для каждого окна, состоящей из отметки времени начала и окончания окна, а также минимальной и максимальной температуры.

Пример 6.12 ❖ Вычисление минимальной и максимальной температуры для каждого датчика и окна с помощью функции `ProcessWindowFunction`

```
// Выводим наименьшую и наибольшую температуру каждые 5 секунд.
val minMaxTempPerWindow: DataStream[MinMaxTemp] = sensorData
  .keyBy(_.id)
  .timeWindow(Time.seconds(5))
  .process(new HighAndLowTempProcessFunction)

case class MinMaxTemp(id: String, min: Double, max:Double, endTs: Long)
/**
* Функция ProcessWindowFunction, которая вычисляет наименьшую и наибольшую температуру
* в каждом окне и передает их вместе с конечной меткой времени окна.
*/
class HighAndLowTempProcessFunction
    extends ProcessWindowFunction[SensorReading, MinMaxTemp, String, TimeWindow] {

  override def process(
      key: String,
      ctx: Context,
      vals: Iterable[SensorReading],
      out: Collector[MinMaxTemp]): Unit = {

    val temps = vals.map(_.temperature)
    val windowEnd = ctx.window.getEnd
    out.collect(MinMaxTemp(key, temps.min, temps.max, windowEnd))
  }
}
```

Внутри окно, которое обрабатывается функцией `ProcessWindowFunction`, сохраняет все назначенные события в `ListState`[1]. Собирая все события и предоставляя доступ к метаданным окна и другим функциям, `ProcessWindowFunction` может реализовать гораздо больше вариантов использования, чем `ReduceFunction` или `AggregateFunction`. Однако состояние окна, которое собирает все события, может стать значительно больше, чем состояние окна, элементы которого агрегируются постепенно.

---

[1] ListState и его характеристики быстродействия подробно обсуждаются в главе 7.

6.3.3.4. Инкрементное агрегирование и функция ProcessWindowFunction

`ProcessWindowFunction` – очень мощная оконная функция, но вы должны использовать ее с осторожностью, поскольку она обычно содержит больше данных в состоянии, чем функции с инкрементным агрегированием. Довольно часто логика, которую необходимо применить к окну, может быть выражена как инкрементное агрегирование, но для этого также требуется доступ к метаданным или состоянию окна.

Если у вас есть логика инкрементного агрегирования, но вам также нужен доступ к метаданным окна, вы можете объединить `ReduceFunction` или `AggregateFunction`, которые выполняют инкрементное агрегирование, с `ProcessWindowFunction`, которая предоставляет дополнительные возможности. Элементы, назначенные окну, будут немедленно агрегированы, и при срабатывании триггера окна агрегированный результат будет передан `ProcessWindowFunction`. Параметр `Iterable` метода `ProcessWindowFunction.process()` предоставит только одно значение – результат с инкрементным агрегированием.

В API DataStream это делается путем передачи `ProcessWindowFunction` в качестве второго параметра методам `reduce()` или `aggregate()`, как показано в следующем коде:

```
input
  .keyBy(...)
  .timeWindow(...)
  .reduce(
    incrAggregator: ReduceFunction[IN],
    function: ProcessWindowFunction[IN, OUT, K, W])

input
  .keyBy(...)
  .timeWindow(...)
  .aggregate(
    incrAggregator: AggregateFunction[IN, ACC, V],
    windowFunction: ProcessWindowFunction[V, OUT, K, W])
```

Код в примерах 6.13 и 6.14 показывает, как решить ту же задачу, что и код в примере 6.12, с комбинацией функций `ReduceFunction` и `ProcessWindowFunction`, генерирующих каждые 5 с минимальную и максимальную температуру для каждого датчика и метку времени окончания каждого окна.

Пример 6.13 ❖ Применение `ReduceFunction` для инкрементного агрегирования и `ProcessWindowFunction` для финализации результата окна

```
case class MinMaxTemp(id: String, min: Double, max:Double, endTs: Long)

val minMaxTempPerWindow2: DataStream[MinMaxTemp] = sensorData
  .map(r => (r.id, r.temperature, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(5))
  .reduce(
```

```
    // Инкрементное вычисление минимальной и максимальной температуры.
    (r1: (String, Double, Double), r2: (String, Double, Double)) => {
      (r1._1, r1._2.min(r2._2), r1._3.max(r2._3))
    },
    // Финализация результата в ProcessWindowFunction.
    new AssignWindowEndProcessFunction()
  )
```

Как вы можете видеть в примере 6.13, функции `ReduceFunction` и `ProcessWindowFunction` определены в вызове метода `reduce()`. Поскольку агрегирование выполняется функцией `ReduceFunction`, функции `ProcessWindowFunction` остается лишь добавить отметку времени окончания окна к инкрементно вычисляемому результату, как показано в примере 6.14.

Пример 6.14 ❖ Реализация функции `ProcessWindowFunction`, которая назначает временную метку конца окна инкрементально вычисляемому результату

```
class AssignWindowEndProcessFunction
  extends
  ProcessWindowFunction[(String, Double, Double), MinMaxTemp, String, TimeWindow] {

  override def process(
      key: String,
      ctx: Context,
      minMaxIt: Iterable[(String, Double, Double)],
      out: Collector[MinMaxTemp]): Unit = {

    val minMax = minMaxIt.head
    val windowEnd = ctx.window.getEnd
    out.collect(MinMaxTemp(key, minMax._2, minMax._3, windowEnd))
  }
}
```

6.3.4. Настройка оконных операторов

Оконные операторы, определенные с помощью встроенных средств назначения окон Flink, могут использоваться во многих типичных случаях. Однако по мере того, как вы начинаете писать более продвинутые потоковые приложения, вы можете столкнуться с необходимостью реализовать более сложную логику работы с окнами, например с окнами, которые выдают ранние результаты и обновляют свои результаты, если встречаются опоздавшие элементы, или с окнами, которые начинаются и заканчиваются, когда получены определенные записи.

API DataStream предоставляет интерфейсы и методы для определения пользовательских оконных операторов, позволяя вам реализовать свои собственные назначители, триггеры и вытеснители. Наряду с ранее обсуждавшимися оконными функциями эти компоненты работают вместе в оконном операторе, группируя и обрабатывая элементы в окнах.

Когда элемент поступает к оконному оператору, он передается в `WindowAssigner`. Назначитель определяет, в какие окна нужно направить элемент. Если окно еще не существует, оно создается.

Если оконный оператор сконфигурирован с функцией инкрементного агрегирования, такой как `ReduceFunction` или `AggregateFunction`, вновь добавленный элемент немедленно агрегируется, а результат сохраняется как содержимое окна. Если оконный оператор не имеет функции инкрементного агрегирования, новый элемент добавляется к списку `ListState`, который содержит все назначенные элементы.

Каждый раз, когда элемент добавляется в окно, он также передается триггеру окна. Триггер определяет (срабатывает), когда окно считается готовым к обработке и когда окно освобождается и его содержимое очищается. Триггер может принять решение на основе назначенных элементов или зарегистрированных таймеров (аналогично функции процесса) обработать или очистить содержимое своего окна в определенные моменты времени.

Что происходит при срабатывании триггера, зависит от настроенных функций оператора окна. Если оператор сконфигурирован только с функцией инкрементного агрегирования, выдается текущий результат агрегирования. Этот случай показан на рис. 6.4.

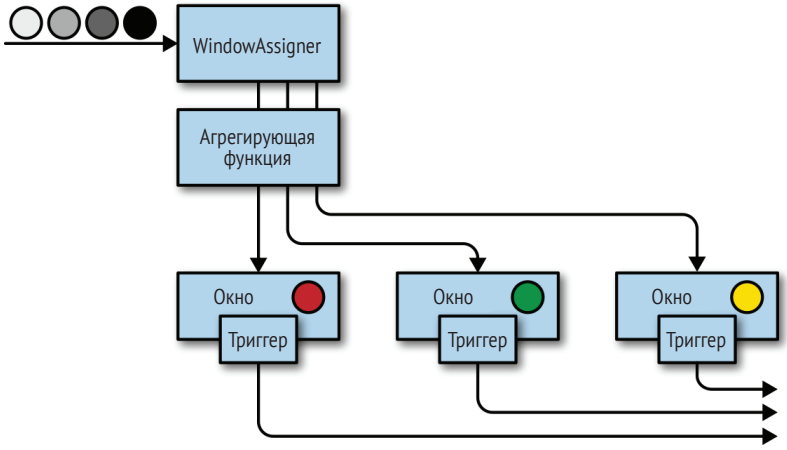

Рис. 6.4 ❖ Оператор окна с функцией инкрементного агрегирования (единственный кружок в каждом окне представляет его агрегированное состояние окна)

Если у оператора есть только функция полного окна, функция применяется ко всем элементам окна, и результат выдается, как показано на рис. 6.5.

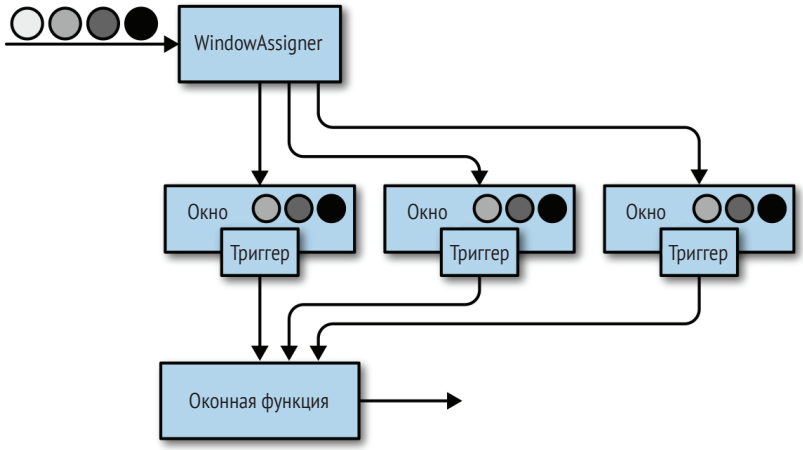

Рис. 6.5 ❖ Оператор окна с функцией полного окна
(кружки в каждом окне представляют собранные необработанные входные записи)

Наконец, если у оператора есть функция инкрементного агрегирования и функция полного окна, функция полного окна применяется к агрегированному значению, и выдается результат. На рис. 6.6 показан этот случай.

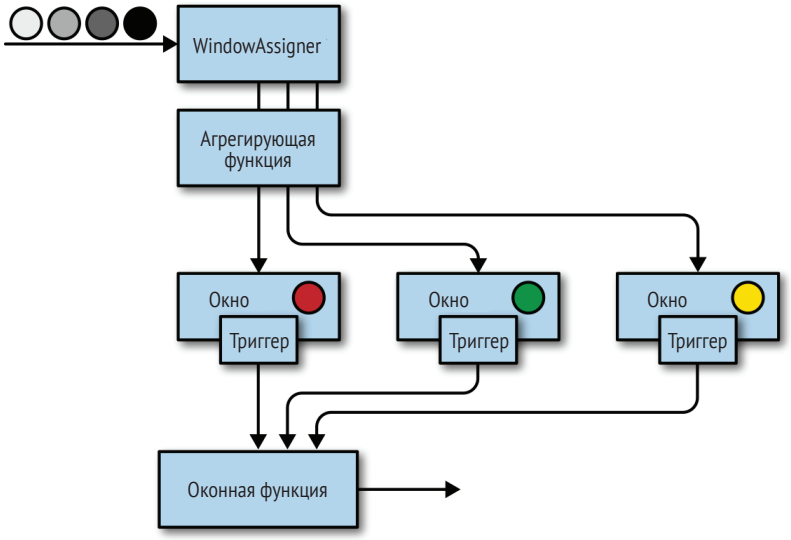

Рис. 6.6 ❖ Оператор окна с функцией инкрементального агрегирования и полного окна
(единственный кружок в каждом окне представляет его агрегированное состояние окна)

Вытеснитель (evictor) – это необязательный компонент, который может быть введен до или после вызова функции `ProcessWindowFunction`. Вытеснитель может удалить собранные элементы из содержимого окна. Поскольку он должен перебирать все элементы, его можно использовать, только если не применяется функция инкрементного агрегирования.

В следующем коде показано, как определить оконный оператор с настраиваемым триггером и вытеснителем:

```
stream
.keyBy(...)
.window(...) // Назначитель окна.
[.trigger(...)] // Опционально: определяем триггер.
[.evictor(...)] // Опционально: определяем вытеснитель.
.reduce/aggregate/process(...) // Определяем оконную функцию.
```

И если вытеснители являются необязательными компонентами, то каждый оператор окна нуждается в триггере, чтобы решить, когда обрабатывать окна. Чтобы предоставить более краткий API оконного оператора, каждый `WindowAssigner` имеет триггер по умолчанию, который используется, если нет явно определенного триггера.

Обратите внимание, что явно определенный триггер переопределяет существующий триггер, а не дополняет его – окно будет обрабатываться только на основе последнего определенного триггера.

В следующих разделах мы обсудим жизненный цикл окон и представим интерфейсы для определения настраиваемых назначителей окон, триггеров и средств выселения.

6.3.4.1. Жизненный цикл окна

Оператор окна создает и обычно также удаляет окна во время обработки входящих элементов потока. Как обсуждалось ранее, элементы назначаются окнам с помощью `WindowAssigner`, триггер решает, когда оценивать окно, а оконная функция выполняет фактическую оценку окна. В этом разделе мы обсуждаем жизненный цикл окна – когда оно создается, из какой информации оно состоит и когда удаляется.

Окно создается, когда WindowAssigner назначает ему первый элемент. Следовательно, не существует окна, которое не содержит хотя бы один элемент. Окно состоит из следующих частей состояния.

Содержимое окна

Содержимое окна содержит элементы, которые были назначены окну, или результат инкрементной агрегации в случае, если оператор окна имеет `ReduceFunction` или `AggregateFunction`.

Объект окна

`WindowAssigner` возвращает ноль, один или несколько оконных объектов. Оператор окна группирует элементы на основе возвращаемых объектов. Следовательно, объект окна содержит информацию, используемую для

различения окон друг от друга. Каждый объект окна имеет отметку времени окончания, которая определяет момент времени, после которого окно и его состояние могут быть удалены.

Таймеры триггера
Триггер может регистрировать таймеры для обратного вызова в определенные моменты времени – например, для обработки окна или очистки его содержимого. Эти таймеры обслуживаются оператором окна.

Настраиваемое состояние в триггере
Триггер может определять и использовать настраиваемое состояние для каждого окна и для каждого ключа. Это состояние полностью контролируется триггером и не поддерживается оператором окна.

Оператор окна удаляет окно, когда достигается время окончания окна, определяемое меткой времени окончания объекта окна. Произойдет ли это с семантикой времени обработки или времени события, зависит от значения, возвращаемого методом `WindowAssigner.isEventTime()`.

Когда окно удаляется, оператор окна автоматически очищает содержимое окна и аннулирует объект окна. Настраиваемое состояние триггера и зарегистрированные таймеры триггера не очищаются, потому что это состояние непрозрачно для оператора окна. Следовательно, триггер должен очистить все свое состояние в методе `Trigger.clear()`, чтобы предотвратить утечку состояния.

6.3.4.2. Назначители окон

`WindowAssigner` определяет окно назначения для каждого поступающего элемента. Элемент можно добавить в ноль, одно или несколько окон. Ниже показан интерфейс `WindowAssigner`:

```
public abstract class WindowAssigner<T, W extends Window>
    implements Serializable {

  // Возвращает коллекцию окон, которым назначен элемент.
  public abstract Collection<W> assignWindows(
    T element,
    long timestamp,
    WindowAssignerContext context);

  // Возвращает триггер WindowAssigner по умолчанию.
  public abstract Trigger<T, W> getDefaultTrigger(
    StreamExecutionEnvironment env);

  // Возвращает TypeSerializer для окон этого WindowAssigner.
  public abstract TypeSerializer<W> getWindowSerializer(
    ExecutionConfig executionConfig);

  // Сообщает, создает ли назначитель окно времени события.
  public abstract boolean isEventTime();

  // Контекст, предоставляющий доступ к текущему окну обработки.
```

```
  public abstract static class WindowAssignerContext {

    // Возвращает текущее время обработки.
    public abstract long getCurrentProcessingTime();
  }
}
```

`WindowAssigner` типизирован для типа входящих элементов и типа окон, которым эти элементы назначаются. Также необходимо предоставить триггер по умолчанию, который используется, если явный триггер не указан. Код в примере 6.15 создает настраиваемый назначитель для 30-секундных шагающих окон времени событий.

Пример 6.15 ❖ Назначитель для шагающих окон времени событий

```
/** Пользовательское окно, которое группирует события в 30-секундные окна */
class ThirtySecondsWindows
    extends WindowAssigner[Object, TimeWindow] {

  val windowSize: Long = 30 * 1000L
  override def assignWindows(
      o: Object,
      ts: Long,
      ctx: WindowAssigner.WindowAssignerContext): java.util.List[TimeWindow] = {

  // Округление в меньшую сторону каждые 30 секунд.
  val startTime = ts - (ts % windowSize)
  val endTime = startTime + windowSize
  // Выпуск соответствующего TimeWindow.
  Collections.singletonList(new TimeWindow(startTime, endTime))
  }

  override def getDefaultTrigger(
      env: environment.StreamExecutionEnvironment): Trigger[Object, TimeWindow] = {
    EventTimeTrigger.create()
  }

  override def getWindowSerializer(
      executionConfig: ExecutionConfig): TypeSerializer[TimeWindow] = {
    new TimeWindow.Serializer
  }
  override def isEventTime = true
}
```

Назначение GlobalWindows

Назначитель `GlobalWindows` отображает все элементы в одно и то же глобальное окно. Его триггер по умолчанию – `NeverTrigger`, который, как следует из названия, никогда не срабатывает. Следовательно, назначителю `GlobalWindows` требуется настраиваемый триггер и, возможно, вытеснитель для выборочного удаления элементов из состояния окна.

Метка времени окончания `GlobalWindows` имеет тип `Long.MAX_VALUE`. Следовательно, `GlobalWindows` никогда не будет полностью очищен. При использовании `KeyedStream` с нарастающим пространством ключей `Global Windows` будет хранить некоторое состояние для каждого ключа. Его следует использовать только с осторожностью.

В дополнение к интерфейсу `WindowAssigner` существует также интерфейс `MergingWindowAssigner`, расширяющий `WindowAssigner`. Интерфейс `MergingWindowAssigner` используется для оконных операторов, которым необходимо объединить существующие окна. Одним из примеров такого назначителя является назначитель `EventTimeSessionWindows`, который мы обсуждали ранее, он работает путем создания нового окна для каждого поступающего элемента и последующего объединения перекрывающихся окон.

При объединении окон необходимо убедиться, что состояние всех объединяемых окон и их триггеров также соответствующим образом объединено. Интерфейс `Trigger` имеет метод обратного вызова, который вызывается, когда происходит слияние окон, и выполняет слияние состояний, связанных с окнами. Более подробно слияние окон обсуждается в следующем разделе.

6.3.4.3. Триггеры

Триггеры определяют, когда выполняется обработка окна и выдаются его результаты. Триггер может принять решение о срабатывании в зависимости от выполнения условий, зависящих от времени или данных, таких как количество элементов или определенные наблюдаемые значения элементов. Например, триггеры по умолчанию для ранее упомянутых временных окон срабатывают, когда время обработки или водяной знак превышают отметку времени конечной границы окна.

Триггеры имеют доступ к свойствам времени и таймерам и могут работать с состоянием. Следовательно, они столь же мощны, как и функции процесса. Например, вы можете реализовать логику запуска, которая срабатывает, когда окно получает определенное количество элементов, когда в окно добавляется элемент с определенным значением или после обнаружения заданного паттерна среди добавленных элементов, например «два события одного типа в пределах 5 с». Пользовательский триггер также может применяться для вычисления и выдачи ранних результатов из окна времени событий, до того как водяной знак достигнет отметки времени окончания окна. Это обычная стратегия получения (неполных) результатов с малой задержкой, несмотря на использование консервативной стратегии водяных знаков.

Каждый раз, когда вызывается триггер, он создает `TriggerResult`, который определяет, что должно произойти с окном. `TriggerResult` может принимать одно из следующих значений.

`CONTINUE`
: Никаких действий не предпринимается.

`FIRE`
: Если оконный оператор имеет `ProcessWindowFunction`, эта функция вызывается и выдает результат. Если в окне есть только функция инкрементного агрегирования (`ReduceFunction` или `AggregateFunction`), выдается текущий результат агрегирования. Состояние окна не меняется.

`PURGE`
: Содержимое окна полностью отбрасывается, а само окно, включая все метаданные, удаляется. Кроме того, вызывается метод `ProcessWindowFunction.clear()` для очистки всего пользовательского состояния для каждого окна.

`FIRE_AND_PURGE`
: Сначала обрабатывает окно (`FIRE`), а затем удаляет все состояние и метаданные (`PURGE`).

Возможные значения `TriggerResult` позволяют реализовать сложную оконную логику. Пользовательский триггер может срабатывать несколько раз, вычисляя новые или обновленные результаты или очищая окно без выдачи результата, если выполняется определенное условие. Следующий блок кода демонстрирует API `Trigger`:

```
public abstract class Trigger<T, W extends Window> implements Serializable {

  // Вызывается для каждого элемента, добавляемого к окну.
  TriggerResult onElement(
  T element, long timestamp, W window, TriggerContext ctx);

  // Вызывается при срабатывании таймера времени обработки.
  public abstract TriggerResult onProcessingTime(
  long timestamp, W window, TriggerContext ctx);

  // Вызывается при срабатывании таймера времени события.
  public abstract TriggerResult onEventTime(
  long timestamp, W window, TriggerContext ctx);

  // Возвращает true, если этот триггер поддерживает слияние состояний триггеров.
  public boolean canMerge();

  // Вызывается, когда несколько окон объединены в одно окно
  // и необходимо объединить состояния триггеров.
  public void onMerge(W window, OnMergeContext ctx);

  // Очищает каждое состояние, которое триггер мог удерживать для данного окна.
  // Этот метод вызывается, когда очищается окно.
  public abstract void clear(W window, TriggerContext ctx);
}

// Контекстный объект, передаваемый методу Trigger, чтобы позволить ему
// регистрировать обратные вызовы и работать с состоянием.
public interface TriggerContext {

  // Возвращает текущее время обработки.
  long getCurrentProcessingTime();

  // Возвращает текущее время водяного знака.
  long getCurrentWatermark();

  // Регистрирует таймер времени обработки.
  void registerProcessingTimeTimer(long time);

  // Регистрирует таймер времени событий.
  void registerEventTimeTimer(long time);

  // Удаляет таймер времени обработки.
  void deleteProcessingTimeTimer(long time);
```

```
  // Удаляет таймер времени событий.
  void deleteEventTimeTimer(long time);

  // Запрашивает объект состояния, связанного с окном, и ключ триггера.
  <S extends State> S getPartitionedState(StateDescriptor<S, ?> stateDescriptor);
}

// Расширение TriggerContext, переданное методу Trigger.onMerge().
public interface OnMergeContext extends TriggerContext {

  // Объединение состояний соответствующих окон.
  // Эти состояния должны поддерживать объединение.
  void mergePartitionedState(StateDescriptor<S, ?> stateDescriptor);
}
```

Как видите, API `Trigger` можно использовать для реализации сложной логики, предоставляя доступ ко времени и состоянию. Особого внимания требуют два аспекта триггеров: очистка состояния и объединение триггеров.

При использовании оконного состояния в триггере необходимо убедиться, что это состояние правильно удаляется при удалении окна. В противном случае оконный оператор со временем будет накапливать все больше и больше состояний, и ваше приложение, вероятно, в какой-то момент выйдет из строя. Чтобы очистить все состояние при удалении окна, метод триггера `clear()` должен удалить все настраиваемые состояния для каждого окна и удалить все таймеры времени обработки и времени события с помощью объекта `TriggerContext`. Невозможно очистить состояние в методе обратного вызова таймера, поскольку эти методы не вызываются после удаления окна.

Если триггер применяется вместе с `MergingWindowAssigner`, он должен иметь возможность обрабатывать случай, когда два окна объединяются. В этом случае необходимо объединить любые пользовательские состояния триггеров. Метод `canMerge()` объявляет, что триггер поддерживает слияние, а метод `onMerge()` должен реализовать логику для выполнения слияния. Если триггер не поддерживает слияние, его нельзя использовать в сочетании с `MergingWindowAssigner`.

Когда триггеры объединяются, все дескрипторы настраиваемых состояний должны быть переданы методу `mergePartitionedState()` объекта `OnMergeContext`.

Обратите внимание, что объединяемые триггеры могут использовать только примитивы состояния, которые могут быть объединены автоматически, – `ListState`, `ReduceState` или `AggregatingState`.

В примере 6.16 показан триггер, который срабатывает раньше, чем наступит время окончания окна. Триггер регистрирует таймер, когда первое событие назначается окну, на одну секунду раньше текущего водяного знака. Когда таймер срабатывает, регистрируется новый таймер. Таким образом, триггер срабатывает не чаще, чем каждую секунду.

Пример 6.16 ❖ Триггер с ранним срабатыванием

```
/** Триггер с ранним срабатыванием. Триггер срабатывает не чаще, чем раз в секунду. */
class OneSecondIntervalTrigger
    extends Trigger[SensorReading, TimeWindow] {

  override def onElement(
      r: SensorReading,
      timestamp: Long,
      window: TimeWindow,
      ctx: Trigger.TriggerContext): TriggerResult = {

  // firstSeen возвращает false если еще не установлен.
  val firstSeen: ValueState[Boolean] = ctx.getPartitionedState(
    new ValueStateDescriptor[Boolean]("firstSeen", classOf[Boolean]))

  // Регистрируем начальный таймер только для первого элемента.
  if (!firstSeen.value()) {
    // Вычисляем время следующего раннего срабатывания округлением водяного знака до секунд.
    val t = ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark % 1000))
    ctx.registerEventTimeTimer(t)
    // Регистрируем таймер конца окна.
    ctx.registerEventTimeTimer(window.getEnd)
    firstSeen.update(true)
  }
  // Продолжаем. Не обрабатываем поэлементно.
  TriggerResult.CONTINUE
}

override def onEventTime(
    timestamp: Long,
    window: TimeWindow,
    ctx: Trigger.TriggerContext): TriggerResult = {
  if (timestamp == window.getEnd) {
    // Финальная обработка и очистка состояния окна.
    TriggerResult.FIRE_AND_PURGE
  } else {
    // Регистрация следующего раннего таймера.
    val t = ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark % 1000))
    if (t < window.getEnd) {
      ctx.registerEventTimeTimer(t)
    }
    // Срабатывание триггера для обработки окна.
    TriggerResult.FIRE
  }
}

override def onProcessingTime(
    timestamp: Long,
    window: TimeWindow,
    ctx: Trigger.TriggerContext): TriggerResult = {
  // Продолжаем. Мы не используем таймеры по времени обработки.
  TriggerResult.CONTINUE
}
```

```
override def clear(
    window: TimeWindow,
    ctx: Trigger.TriggerContext): Unit = {

  // Очистка состояния таймера.
    val firstSeen: ValueState[Boolean] = ctx.getPartitionedState(
      new ValueStateDescriptor[Boolean]("firstSeen", classOf[Boolean]))
    firstSeen.clear()
  }
}
```

Обратите внимание, что триггер использует настраиваемое состояние, которое очищается с помощью метода `clear()`. Поскольку мы используем простое необъединяемое `ValueState`, триггер не может быть объединен.

6.3.4.4. Вытеснители

`Evictor` (вытеснитель) – это дополнительный компонент оконного механизма Flink. Он может удалять элементы из окна до или после срабатывания оконной функции.

В примере 6.17 показан интерфейс `Evictor`.

Пример 6.17 ❖ Интерфейс `Evictor`

```
public interface Evictor<T, W extends Window> extends Serializable {
  // Избирательно удаляет элементы. Вызывается перед оконной функцией.
  void evictBefore(
    Iterable<TimestampedValue<T>> elements,
    int size,
    W window,
    EvictorContext evictorContext);

  // Избирательно удаляет элементы. Вызывается перед оконной функцией.
  void evictAfter(
    Iterable<TimestampedValue<T>> elements,
    int size,
    W window,
    EvictorContext evictorContext);

// Объект контекста, передаваемый в методы Evictor.
interface EvictorContext {

  // Возвращает текущее время обработки.
  long getCurrentProcessingTime();

  // Возвращает текущую метку времени события.
  long getCurrentWatermark();
}
```

Методы `evictBefore()` и `evictAfter()` вызываются до и после применения оконной функции к содержимому окна соответственно. Оба метода вызываются с помощью `Iterable`, который обслуживает все элементы, добавленные

в окно, количество элементов в окне (`size`), объект окна и `EvictorContext`, обеспечивающий доступ к текущему времени обработки и водяному знаку. Элементы удалены из окна вызовом метода `remove()` в `Iterator`, который можно получить из `Iterable`.

Преагрегация и вытеснение

Вытеснители перебирают список элементов в окне. Они могут применяться только в том случае, если окно собирает все добавленные события и не применяет `ReduceFunction` или `AggregateFunction` для постепенного агрегирования содержимого окна.

Вытеснители часто применяются в `GlobalWindow` для частичной очистки окна – без очистки всего состояния окна.

6.4. Объединение потоков по времени

Распространенным требованием при работе с потоками является соединение или объединение событий двух потоков. В API DataStream есть два встроенных оператора для объединения потоков по условию времени: интервальное объединение и оконное объединение. В этом разделе мы описываем оба оператора.

Если вы не можете выразить требуемую семантику объединения с помощью встроенных операторов соединения Flink, вы можете реализовать собственную логику объединения как `CoProcessFunction`, `BroadcastProcessFunction` или `KeyedBroadcastProcessFunction`.

Обратите внимание, что вы должны разработать оператор с эффективными шаблонами доступа к состоянию и эффективными стратегиями очистки состояния.

6.4.1. Интервальное объединение

Интервальное объединение объединяет события из двух потоков, которые имеют общий ключ и временные метки которых находятся на расстоянии не более заданных интервалов друг от друга.

На рис. 6.7 показано интервальное объединение двух потоков, A и B, которое объединяет событие из A с событием из B, если метка времени события B не менее чем на час раньше и не более чем на 15 мин позже, чем метка времени события A. Интервал объединения является *симметричным*, то есть событие из B соединяется со всеми событиями из A, которые происходят не более чем на 15 мин раньше и не более чем на час позже, чем событие B.

В настоящее время интервальное объединение поддерживает только время события и работает с семантикой `INNER JOIN` (события, для которых нет совпадающего события, не будут перенаправлены). Интервальное объединение определяется, как показано в примере 6.18.

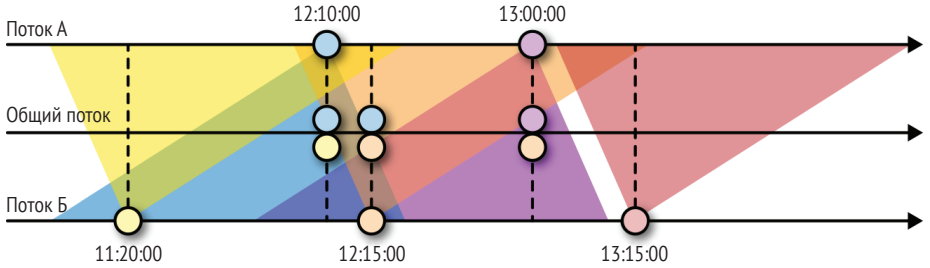

Рис. 6.7 ❖ Интервальное объединение, объединяющее два потока A и B

Пример 6.18 ❖ Использование интервального объединения

```
input1
  .keyBy(…)
  .between(<lower-bound>, <upper-bound>) // Границы исходя из input1.
  .process(ProcessJoinFunction) // Обработка пар совпавших событий.
```

Пары объединенных событий передаются в `ProcessJoinFunction`. Нижняя и верхняя границы определяются как отрицательные и положительные интервалы времени, например как `between(Time.hour(-1), Time.minute(15))`. Нижняя и верхняя границы могут быть выбраны произвольно, если нижняя граница меньше верхней; вы можете присоединиться ко всем событиям A со всеми событиями B, метки времени которых больше, чем у события A в интервале от одного до двух часов.

При интервальном объединении необходимо буферизовать записи с одного или обоих входов. Для первого входа буферизируются все записи с метками времени, превышающими текущий водяной знак – верхнюю границу. Для второго входа буферизируются все записи с метками времени больше текущего водяного знака плюс нижняя граница. Обратите внимание, что обе границы могут быть отрицательными. Объединение на рис. 6.7 хранит все записи с отметками времени, превышающими текущий водяной знак – 15 мин от потока A, – и все записи с отметками времени больше текущего водяного знака – через час от потока B. Вы должны знать, что требования к хранению для интервального объединения могут значительно увеличиться, если время события обоих входных потоков не синхронизировано, поскольку водяной знак определяется «более медленным» потоком.

6.4.2. Оконное объединение

Как следует из названия, *оконное объединение* основано на оконном механизме Flink. Элементы обоих входных потоков назначаются общим окнам и объединяются (или группируются), когда создается окно.

В примере 6.19 показано, как определить оконное объединение.

Пример 6.19 ❖ Оконное объединение двух потоков

```
input1.join(input2)
.where(...) // Атрибут ключа для input1.
```

```
.equalTo(...) // Атрибут ключа для input2.
.window(...) // Определяем WindowAssigner.
[.trigger(...)] // Не обязательно: определяем Trigger.
[.evictor(...)] // Не обязательно: определяем Evictor.
.apply(...) // Определяем JoinFunction.
```

На рис. 6.8 показано, как работает оконное объединение API DataStream.

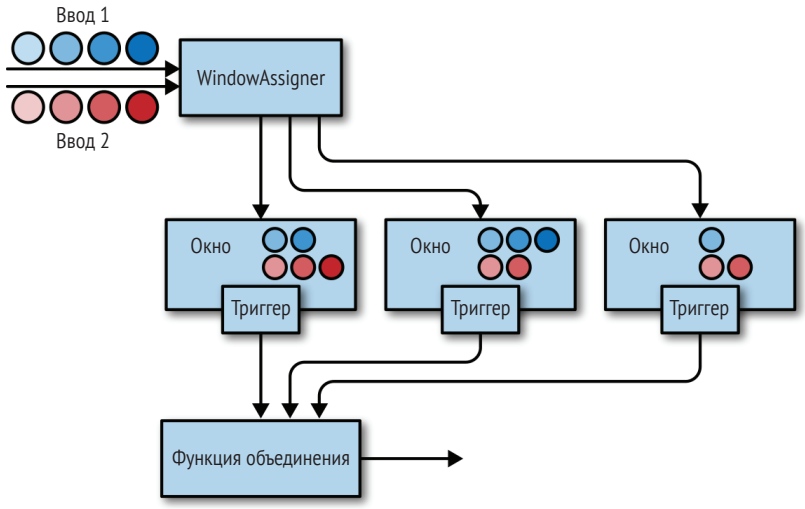

Рис. 6.8 ❖ Операция оконного объединения

Оба входных потока привязаны к своим ключевым атрибутам, и средство назначения общего окна сопоставляет события обоих потоков с общими окнами, то есть окно хранит события обоих входов. Когда срабатывает таймер окна, функция `JoinFunction` вызывается для каждой комбинации элементов из первого и второго потока входных данных – перекрестное произведение. Также можно указать настраиваемый триггер и вытеснитель. Поскольку события обоих потоков отображаются в одних и тех же окнах, триггеры и вытеснители ведут себя точно так же, как в обычных оконных операторах.

В дополнение к объединению двух потоков также можно группировать два потока в окне, начав определение оператора с `coGroup()` вместо `join()`. В целом логика такая же, но вместо вызова `JoinFunction` для каждой пары событий из обоих входов `CoGroupFunction` вызывается один раз для каждого окна с итераторами по элементам из обоих входов.

Следует отметить, что оконное объединение потоков может иметь неожиданную семантику. Например, предположим, что вы объединяете два потока с оператором объединения, для которого настроено одночасовое временное окно. Элемент первого входа не будет объединен с элементом второго входа, даже если они находятся всего в одной секунде друг от друга, но назначены двум разным окнам.

6.5. Обработка опоздавших данных

Как уже говорилось, водяные знаки могут использоваться для баланса полноты и задержки результата. Если вы не выберете очень консервативную стратегию водяных знаков, которая гарантирует, что все соответствующие записи будут включены за счет высокой задержки, вашему приложению, скорее всего, придется обрабатывать опоздавшие элементы.

Опоздавший элемент (late element) – это элемент, который поступает к оператору, когда вычисление, в которое он должен был бы внести свой вклад, уже было выполнено. В контексте оператора окна времени событие считается опоздавшим, если оно доходит до оператора, и назначитель сопоставляет его с окном, которое уже было вычислено, поскольку водяной знак оператора передал метку времени окончания окна.

API DataStream предоставляет различные варианты обработки поздних событий:

- опоздавшие события можно просто отбросить;
- опоздавшие события могут быть перенаправлены в отдельный поток;
- результаты вычислений могут быть обновлены на основе опоздавших событий, и необходимо отправить обновления.

Ниже мы подробно обсудим эти параметры и покажем, как они применяются для функций процессов и оконных операторов.

6.5.1. Отбрасывание опоздавших событий

Самый простой способ справиться с поздними событиями – просто отбросить их. Отбрасывание опоздавших событий является поведением по умолчанию для операторов окна времени события. Следовательно, опоздавший элемент не создаст новое окно.

Функция процесса может легко отфильтровать опоздавшие события, сравнивая их метки времени с текущим водяным знаком.

6.5.2. Перенаправление опоздавших событий

Опоздавшие события также можно перенаправить в другой поток данных с помощью функции бокового вывода. Затем эти события могут быть обработаны или отправлены с помощью обычной функции приемника. В зависимости от бизнес-требований далее данные могут быть интегрированы в результаты потокового приложения при помощи периодического процесса обратной загрузки.

В примере 6.20 показано, как указать оконный оператор с побочным выводом для опоздавших событий.

Пример 6.20 ❖ Определение оконного оператора с боковым выводом для опоздавших событий

```
val readings: DataStream[SensorReading] = ???

val countPer10Secs: DataStream[(String, Long, Int)] = readings
  .keyBy(_.id)
  .timeWindow(Time.seconds(10))
  // Отправляем опоздавшие данные в боковой вывод.
  .sideOutputLateData(new OutputTag[SensorReading]("late-readings"))
  // Считаем записи в окне.
  .process(new CountFunction())

// Запрашиваем опоздавшие события из бокового вывода как поток.
val lateStream: DataStream[SensorReading] = countPer10Secs
  .getSideOutput(new OutputTag[SensorReading]("late-readings"))
```

Функция процесса может идентифицировать поздние события, сравнивая метки времени событий с текущим водяным знаком и выдавая их с помощью обычного API бокового вывода. В примере 6.21 показана функция `ProcessFunction`, которая отфильтровывает опоздавшие показания датчика со своего входа и перенаправляет их в боковой выходной поток.

Пример 6.21 ❖ Функция `ProcessFunction`, которая фильтрует опоздавшие показания датчика и перенаправляет их на боковой выход

```
val readings: DataStream[SensorReading] = ???
val filteredReadings: DataStream[SensorReading] = readings
  .process(new LateReadingsFilter)

// Запрашиваем опоздавшие данные.
val lateReadings: DataStream[SensorReading] = filteredReadings
  .getSideOutput(new OutputTag[SensorReading]("late-readings"))

/** Функция ProcessFunction, которая фильтрует опоздавшие данные датчика
* и перенаправляет их в боковой вывод */
class LateReadingsFilter
    extends ProcessFunction[SensorReading, SensorReading] {

  val lateReadingsOut = new OutputTag[SensorReading]("late-readings")

  override def processElement(
      r: SensorReading,
      ctx: ProcessFunction[SensorReading, SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {

    // Сравниваем время записи с текущим водяным знаком.
    if (r.timestamp < ctx.timerService().currentWatermark()) {
       // Это опоздавшие данные => направляем их в боковой вывод.
       ctx.output(lateReadingsOut, r)
    } else {
      out.collect(r)
    }
  }
}
```

6.5.3. Обновление результатов путем включения опоздавших событий

Опоздавшие события поступают к оператору после завершения вычисления, в которое они должны были внести свой вклад. Следовательно, оператор выдает неполный или неточный результат. Вместо того чтобы отбрасывать или перенаправлять поздние события, другая стратегия состоит в повторном вычислении неполного результата и выпуске обновления. Однако есть несколько проблем, которые необходимо принять во внимание, чтобы иметь возможность повторно вычислять и обновлять результаты.

Оператор, поддерживающий пересчет и обновление выданных результатов, должен сохранять все состояния, необходимые для вычисления после того, как был выдан первый результат. Однако, поскольку обычно оператор не может сохранять полное состояние навсегда, ему необходимо в какой-то момент очистить состояние. После очистки состояния для определенного результата результат больше не может быть обновлен, а поздние события можно только отбросить или перенаправить.

Помимо сохранения состояния, нижестоящие операторы или внешние системы, которые следуют за оператором, обновляющим ранее выданные результаты, должны иметь возможность обрабатывать эти обновления. Например, оператор приемника, который записывает результаты и обновления оператора окна с ключом в хранилище «ключ–значение», мог бы сделать это, переопределив неточные результаты с последним обновлением с помощью обновления записи. В некоторых случаях может также потребоваться явно различать первый результат и обновление из-за опоздавшего события.

API оконного оператора предоставляет метод явного объявления о том, что вы ожидаете опоздавшие элементы. При использовании окон с привязкой ко времени событий вы можете указать дополнительный период времени, называемый *допустимым опозданием* (allowed lateness). Оператор окна с допустимым опозданием не удалит окно и его состояние после того, как водяной знак пройдет отметку времени окончания окна. Вместо этого оператор продолжает поддерживать полное окно в течение допустимого периода опоздания. Если опоздавший элемент прибывает в течение допустимого периода, он обрабатывается как своевременный элемент и передается триггеру. Когда водяной знак проходит отметку времени окончания окна плюс интервал опоздания, окно окончательно удаляется, а все последующие опоздавшие элементы отбрасываются.

Допустимое опоздание можно задать с помощью метода `allowedLateness()`, как показано в примере 6.22.

Пример 6.22 ❖ Определение оконного оператора с допустимым опозданием 5 с

```
val readings: DataStream[SensorReading] = ???

val countPer10Secs: DataStream[(String, Long, Int, String)] = readings
  .keyBy(_.id)
  .timeWindow(Time.seconds(10))
  // Обработка опоздавших записей в течение 5 дополнительных секунд.
```

```
  .allowedLateness(Time.seconds(5))
  // Подсчет элементов и обновление результата с учетом опоздавших данных.
  .process(new UpdatingWindowCountFunction)

/** Функция подсчета WindowProcessFunction, которая различает первые
результаты и обновления.*/
class UpdatingWindowCountFunction
    extends ProcessWindowFunction[
            SensorReading, (String, Long, Int, String), String, TimeWindow] {
  override def process(
      id: String,
      ctx: Context,
      elements: Iterable[SensorReading],
      out: Collector[(String, Long, Int, String)]): Unit = {
    // Подсчет числа элементов.
    val cnt = elements.count(_ => true)
    // Проверка, является ли эта обработка окна первой.
    val isUpdate = ctx.windowState.getState(
      new ValueStateDescriptor[Boolean]("isUpdate", Types.of[Boolean]))
    if (!isUpdate.value()) {
      // Первая обработка, выпуск первого результата.
      out.collect((id, ctx.window.getEnd, cnt, "first"))
      isUpdate.update(true)
    } else {
      // Не первая обработка, выпуск обновления.
      out.collect((id, ctx.window.getEnd, cnt, "update"))
    }
  }
}
```

Функции обработки также могут быть реализованы для поддержки опоздавших данных. Поскольку управление состоянием всегда настраивается и выполняется вручную в функциях обработки, Flink не предоставляет встроенный API для поддержки опоздавших данных. Вместо этого вы можете реализовать необходимую логику, используя стандартные блоки меток времени записи, водяных знаков и таймеров.

6.6. Заключение

В этой главе вы узнали, как реализовать потоковые приложения, которые работают в заданное время. Мы объяснили, как настроить временные характеристики потокового приложения и как назначать метки времени и водяные знаки. Вы узнали об операторах с привязкой ко времени, включая функции обработки Flink, встроенные и настраиваемые окна. Мы также обсудили семантику водяных знаков, то, как найти компромисс между полнотой результата и задержкой выдачи, а также стратегии обработки опоздавших событий.

Глава 7

Операторы и приложения с учетом состояния

Операторы с учетом состояния и пользовательские функции являются общими строительными блоками приложений потоковой обработки. Фактически для большинства нетривиальных операций необходимо запоминать записи или промежуточные результаты, потому что данные передаются в потоковом режиме и поступают постепенно[1]. Многие из встроенных в Flink операторов DataStream, источников и приемников используют состояние и буферные записи или поддерживают частичные результаты или метаданные. Например, оконный оператор собирает входные записи или результат применения `ReduceFunction` для `ProcessWindowFunction`; функция обработки `ProcessFunction` запоминает запланированные таймеры, а некоторые функции-приемники поддерживают состояние транзакций для обеспечения функциональности «ровно один раз». В дополнение к встроенным операторам и предоставляемым источникам и приемникам API DataStream предоставляет интерфейсы для регистрации, поддержки и доступа к состоянию в определяемых пользователем функциях.

Потоковая обработка с учетом состояния влияет на многие аспекты потокового процессора, такие как восстановление после сбоев и управление памятью, а также обслуживание потоковых приложений. В главах 2 и 3 обсуждались основы потоковой обработки с учетом состояния и связанные детали архитектуры Flink соответственно. В главе 9 мы поясним, как настроить Flink для надежной обработки приложений с учетом состояния. В главе 10 дается руководство по работе с приложениями с учетом состояния – получение и восстановление из точек сохранения приложений, изменение масштаба приложений и выполнение обновлений приложений.

В этой главе основное внимание уделяется реализации определяемых пользователем функций с учетом состояния и обсуждается производительность и надежность приложений с учетом состояния. В частности, мы объясняем, как определять различные типы состояний и взаимодействовать с ними в пользовательских функциях. Мы также обсуждаем аспекты быстро-

---

[1] В этом состоит отличие от пакетной обработки, когда пользовательские функции, такие как `GroupReduceFunction`, вызываются после того, как собраны все данные для обработки.

действия и способы управления размером состояния функции. Наконец, мы покажем, как настроить доступность для запроса состояния с ключом и как получить к нему доступ из внешнего приложения.

7.1. Реализация функций с сохранением состояния

В разделе 3.4 мы объяснили, что функции могут работать с двумя типами состояния: ключевое состояние и состояние оператора. Flink предоставляет несколько интерфейсов для определения функций, использующих состояние. В этом разделе мы покажем, как реализованы функции с тем и другим состоянием.

7.1.1. Объявление ключевого состояния в RuntimeContext

Пользовательские функции могут использовать ключевое состояние для хранения состояния и доступа к состоянию в контексте ключевого атрибута. Для каждого отдельного значения ключевого атрибута Flink поддерживает один экземпляр состояния. Экземпляры ключевого состояния функции распределяются по всем параллельным задачам оператора функции. Это означает, что каждый параллельный экземпляр функции отвечает за поддиапазон пространства ключей и поддерживает соответствующие экземпляры состояния. Следовательно, состояние с ключом очень похоже на распределенную карту «ключ–значение». Обратитесь к разделу 3.4 для получения более подробной информации о ключевом состоянии.

Ключевое состояние может использоваться только функциями, которые применяются к `KeyedStream`. В свою очередь `KeyedStream` создается путем вызова метода `DataStream.keyBy()`, который определяет ключ в потоке. `KeyedStream` разделяется на указанный ключ и запоминает определение ключа. Оператор, применяемый к `KeyedStream`, применяется в контексте его определения ключа.

Flink предоставляет несколько примитивов для ключевого состояния. Примитив состояния определяет структуру состояния отдельного ключа. Выбор правильного примитива состояния зависит от того, как функция взаимодействует с состоянием. Выбор также влияет на быстродействие функции, поскольку каждый бэкенд состояния предоставляет свои собственные реализации для этих примитивов. Flink поддерживает следующие примитивы состояния:

- `ValueState[T]` содержит единственное значение типа `T`. Значение можно прочитать с помощью `ValueState.value()` и обновить с помощью `ValueState.update(value: T)`;
- `ListState[T]` содержит список элементов типа `T`. Новые элементы могут быть добавлены к списку путем вызова `ListState.add(value: T)` или

ListState.addAll(value: java.util.List[T]). Доступ к элементам состояния можно получить, вызвав ListState.get(), который возвращает Iterable[T] по всем элементам состояния. Невозможно удалить отдельные элементы из ListState, но список можно обновить, вызвав ListState.update(value: java.util.List[T]). Вызов этого метода заменит существующие значения заданным списком значений;

- MapState[K, V] содержит карту ключей и значений. Примитив состояния предлагает многие методы обычной карты Java, такие как get(key: K), put(key: K, value: V), contains(key: K), remove(key: K) и итераторы поверх содержащихся записей, ключей и значений;
- ReducingState[T] предлагает те же методы, что и ListState[T] (за исключением addAll() и update()), но вместо добавления значений в список, ReducingState.add(value: T) немедленно агрегирует значение с помощью функции ReduceFunction. Итератор, возвращаемый get(), возвращает Iterable с одной записью, которая является редуцированным значением;
- AggregatingState[I, O] ведет себя аналогично ReducingState. Однако для агрегирования значений он использует более общую функцию AggregateFunction. AggregatingState.get() вычисляет окончательный результат и возвращает его как Iterable с одним элементом.

Все примитивы состояния можно очистить, вызвав State.clear().

В примере 7.1 показано, как применить FlatMapFunction с ключевым значением ValueState к потоку измерений датчика. Пример приложения выдает предупреждение, если температура, измеренная датчиком, изменяется более чем на пороговое значение с момента последнего измерения.

Пример 7.1 ❖ Применение FlatMapFunction с ключевым значением ValueState

```
val sensorData: DataStream[SensorReading] = ???
// Разделение потока по ключу, которым служит ID датчика.
val keyedData: KeyedStream[SensorReading, String] = sensorData
  .keyBy(_.id)

// Применяем к потоку с ключом функцию FlatMapFunction, которая
// сравнивает показатели температуры и выдает предупреждение.
val alerts: DataStream[(String, Double, Double)] = keyedData
  .flatMap(new TemperatureAlertFunction(1.7))
```

К KeyedStream должна применяться функция с ключевым состоянием. Но, прежде чем мы применим функцию, нам нужно указать ключ, вызвав метод keyBy() во входном потоке. Когда вызывается метод функции обработки ключевого потока, среда выполнения Flink автоматически помещает все объекты состояния функции с ключами в контекст ключа записи, которая передается при вызове функции. Следовательно, функция может получить доступ только к состоянию, принадлежащему записи, которую она в настоящее время обрабатывает.

Пример 7.2 показывает реализацию функции FlatMapFunction с ключевым ValueState, которая проверяет, изменилась ли измеренная температура больше чем на заданную пороговую величину.

Пример 7.2 ❖ Реализация FlatMapFunction с ключевым значением ValueState

```
class TemperatureAlertFunction(val threshold: Double)
    extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {

  // Объект состояния.
  private var lastTempState: ValueState[Double] = _

  override def open(parameters: Configuration): Unit = {
    // Создаем дескриптор состояния.
    val lastTempDescriptor =
      new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
    // Получаем состояние.
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
  }

  override def flatMap(
      reading: SensorReading,
      out: Collector[(String, Double, Double)]): Unit = {
    // Извлекаем последнюю температуру из состояния.
    val lastTemp = lastTempState.value()
    // Проверяем, надо ли выдавать сигнал тревоги.
    val tempDiff = (reading.temperature - lastTemp).abs
    if (tempDiff > threshold) {
      // Температура изменилась больше, чем заданное пороговое значение.
      out.collect((reading.id, reading.temperature, tempDiff))
    }
    // Обновляем состояние lastTemp.
    this.lastTempState.update(reading.temperature)
  }
}
```

Чтобы создать объект состояния, мы должны зарегистрировать StateDescriptor в среде выполнения Flink через RuntimeContext, который предоставляется RichFunction (см. раздел 5.6, где рассказано про интерфейс RichFunction). StateDescriptor специфичен для примитива состояния и включает имя состояния и типы данных состояния. Дескрипторам для ReducingState и AggregatingState также требуется объект ReduceFunction или AggregateFunction для агрегирования добавленных значений. Имя состояния привязано к оператору, так что функция может иметь более одного объекта состояния путем регистрации нескольких дескрипторов состояния. Типы данных, обрабатываемые состоянием, указаны как объекты Class или TypeInformation (см. раздел 5.4, где говорится про обработку типов Flink). Тип данных должен быть указан, потому что Flink необходимо создать подходящий сериализатор. В качестве альтернативы также можно явно указать TypeSerializer для управления записью состояния в бэкенд состояния, контрольную точку и точку сохранения[1].

Обычно объект дескриптора состояния создается в методе open() функции RichFunction. Метод open() вызывается перед вызовом любых методов обработки, таких как flatMap() в случае FlatMapFunction. Объект дескриптора состояния (lastTempState в примере 7.2) – это обычная переменная-член класса функции.

---

[1] Формат сериализации состояния является важным аспектом при обновлении приложения и обсуждается далее в этой главе.

Объект дескриптора состояния предоставляет доступ только к состоянию, которое хранится и поддерживается в бэкенде состояния. Сам дескриптор не хранит состояние.

Когда функция регистрирует `StateDescriptor`, Flink проверяет, имеются ли в хранилище состояния данные для функции и состояния с заданным именем и типом. Это может произойти, если функция с учетом состояния перезапускается для восстановления после сбоя или когда приложение запускается из точки сохранения. В обоих случаях Flink связывает вновь зарегистрированный объект дескриптора состояния с существующим состоянием. Если бэкенд состояния не содержит состояния для данного дескриптора, состояние, связанное с дескриптором, инициализируется как пустое.

API DataStream на языке Scala предлагает синтаксические сокращения для определения функций `map` и `flatMap` с одним `ValueState`. В примере 7.3 показано, как реализовать предыдущий пример с помощью ярлыка.

Пример 7.3 ❖ Ярлык API DataStream на языке Scala для FlatMap с ключевым значением ValueState

```
val alerts: DataStream[(String, Double, Double)] = keyedData
  .flatMapWithState[(String, Double, Double), Double] {
    case (in: SensorReading, None) =>
      // Предыдущая температура не определена; обновляем текущую температуру.
      (List.empty, Some(in.temperature))
    case (r: SensorReading, lastTemp: Some[Double]) =>
      // Сравниваем перепад температуры с пороговым значением.
      val tempDiff = (r.temperature - lastTemp.get).abs
      if (tempDiff > 1.7) {
        // Порог превышен; выдаем сигнал тревоги и обновляем значение температуры.
        (List((r.id, r.temperature, tempDiff)), Some(r.temperature))
      } else {
        // Порог не превышен; просто обновляем значение температуры.
      (List.empty, Some(r.temperature))
      }
  }
```

Метод `flatMapWithState()` ожидает функцию, которая принимает `Tuple2`. Первое поле кортежа содержит входную запись для `flatMap`, а второе поле содержит `Option` извлеченного состояния для ключа обработанной записи. Опция не определена, если состояние еще не инициализировано. Функция также возвращает `Tuple2`. Первое поле – это список результатов `flatMap`, а второе поле – новое значение состояния.

7.1.2. Реализация списочного состояния с помощью интерфейса ListCheckpointed

Состояние оператора управляется каждым параллельным экземпляром оператора. Все события, которые обрабатываются в одной и той же параллельной задаче оператора, имеют доступ к одному и тому же состоянию. В раз-

деле 3.4 мы говорили, что Flink поддерживает три типа состояния оператора: списочное, каталожное и широковещательное.

Функция может работать с оператором списочного состояния, реализуя интерфейс `ListCheckpointed`. Интерфейс `ListCheckpointed` не работает с дескрипторами состояния, такими как `ValueState` или `ListState`, которые зарегистрированы в бэкенде состояния. Вместо этого функции реализуют состояние оператора как обычные переменные-члены и взаимодействуют с бэкендом состояния через функции обратного вызова интерфейса `ListCheckpointed`. Интерфейс предоставляет два метода:

```
// Возвращает снимок состояния функции в виде списка.
snapshotState(checkpointId: Long, timestamp: Long): java.util.List[T]

// Восстанавливает состояние функции из предоставленного списка.
restoreState(java.util.List[T] state): Unit
```

Метод `snapshotState()` вызывается, когда Flink запускает контрольную точку функции с учетом состояния. Этот метод имеет два параметра: `checkpointId`, который представляет собой уникальный, монотонно увеличивающийся идентификатор для контрольных точек, и `timestamp`, который представляет собой время «настенных часов» системы, когда была инициирована контрольная точка. Метод должен возвращать состояние оператора в виде списка сериализуемых объектов состояния.

Метод `restoreState()` всегда вызывается, когда необходимо инициализировать состояние функции – когда задание запускается (с точки сохранения или нет) или в случае сбоя. Метод вызывается со списком объектов состояния и должен восстанавливать состояние оператора на основе этих объектов.

В примере 7.4 показано, как реализовать интерфейс `ListCheckpointed` для функции, которая подсчитывает измерения температуры, превышающие пороговое значение для каждой секции, для каждого параллельного экземпляра функции.

Пример 7.4 ❖ Функция `RichFlatMap` с оператором списочного состояния

```
class HighTempCounter(val threshold: Double)
    extends RichFlatMapFunction[SensorReading, (Int, Long)]
    with ListCheckpointed[java.lang.Long] {

  // Индекс подзадачи.
  private lazy val subtaskIdx = getRuntimeContext
    .getIndexOfThisSubtask
  // Переменная локального счетчика.
  private var highTempCnt = 0L

  override def flatMap(
      in: SensorReading,
      out: Collector[(Int, Long)]): Unit = {
    if (in.temperature > threshold) {
      // Инкремент счетчика, если превышен порог.
      highTempCnt += 1
      // Выпуск обновления с индексом подзадачи и счетчиком.
```

```
      out.collect((subtaskIdx, highTempCnt))
    }
  }

  override def restoreState(
      state: util.List[java.lang.Long]): Unit = {
    highTempCnt = 0
    // Восстановление состояния добавлением всех значений в список.
    for (cnt <- state.asScala) {
      highTempCnt += cnt
    }
  }

  override def snapshotState(
      chkpntId: Long,
      ts: Long): java.util.List[java.lang.Long] = {
    // Снимок состояния как снимок со счетчиком.
    java.util.Collections.singletonList(highTempCnt)
  }
}
```

Функция в приведенном выше примере подсчитывает для каждого параллельного экземпляра, сколько измерений температуры превысило установленный порог. Функция использует состояние оператора и имеет единственную переменную состояния для каждого экземпляра параллельного оператора, для которого устанавливается контрольная точка и который восстанавливается с помощью методов интерфейса `ListCheckpointed`. Обратите внимание, что интерфейс `ListCheckpointed` реализован на Java и ожидает `java.util.List` вместо собственного списка Scala.

Глядя на этот пример, вы можете задаться вопросом, почему состояние оператора обрабатывается как список объектов состояния. Как обсуждалось в разделе 3.4.4, списочная структура поддерживает изменение параллелизма функций с учетом состояния оператора. Чтобы увеличить или уменьшить параллелизм функции с учетом состояния оператора, это состояние необходимо перераспределить на большее или меньшее количество экземпляров задачи. Это требует разделения или слияния объектов состояния. Поскольку логика разделения и слияния состояния настраивается для каждой функции с учетом состояния, это не может быть выполнено автоматически для произвольных типов состояния.

Предоставляя список объектов состояния, функции с учетом состояния оператора могут реализовать эту логику с помощью методов `snapshotState()` и `restoreState()`. Метод `snapshotState()` разделяет состояние оператора на несколько частей, а метод `restoreState()` собирает состояние оператора, возможно, из нескольких частей. Когда состояние функции восстанавливается, части состояния распределяются между всеми параллельными экземплярами функции и передаются методу `restoreState()`. Если параллельных подзадач больше, чем объектов состояния, некоторые подзадачи запускаются без состояния, а метод `restoreState()` вызывается с пустым списком.

Снова посмотрев на функцию `HighTempCounter` в примере 7.4, мы увидим, что каждый параллельный экземпляр оператора представляет свое состоя-

ние в виде списка с одной записью. Если мы увеличим параллелизм этого оператора, некоторые из новых подзадач будут инициализированы с пустым состоянием и начнут отсчет с нуля. Чтобы добиться лучшего поведения распределения состояний при масштабировании функции `HighTempCounter`, мы можем реализовать метод `snapshotState()`, чтобы он разбивал свой счетчик на несколько частичных счетчиков, как показано в примере 7.5.

Пример 7.5 ❖ Разделение списочного состояния для лучшего распределения во время масштабирования

```
override def snapshotState(
    chkpntId: Long,
    ts: Long): java.util.List[java.lang.Long] = {
  // Делим счетчик на 10 частичных подсчетчиков.
  val div = highTempCnt / 10
  val mod = (highTempCnt % 10).toInt
  // Возвращаем счетчик из 10 частей.
  (List.fill(mod)(new java.lang.Long(div + 1)) ++
    List.fill(10 - mod)(new java.lang.Long(div))).asJava
}
```

Интерфейс ListCheckpointed использует сериализацию Java

Интерфейс `ListCheckpointed` использует сериализатор Java для сериализации и десериализации списка объектов состояния. Это может быть проблемой, если вам нужно обновить приложение, потому что сериализация Java не позволяет переносить или настраивать пользовательский сериализатор. Реализуйте `CheckpointedFunction` вместо интерфейса `ListCheckpointed`, если вам нужно обеспечить возможность переноса состояния оператора функции.

7.1.3. Использование широковещательного состояния

Распространенным требованием для потоковых приложений является распределение одной и той же информации на все параллельные экземпляры функции и поддержание ее как восстанавливаемого состояния. Примером может служить поток правил и поток событий, к которым применяются правила. Функция, применяющая правила, принимает два входных потока, поток событий и поток правил. Она запоминает правила в состоянии оператора, чтобы применить их ко всем событиям потока событий. Поскольку каждый параллельный экземпляр функции должен содержать все правила в своем состоянии оператора, поток правил должен транслироваться, чтобы гарантировать, что каждый экземпляр функции получает все правила.

Во Flink такое состояние называется широковещательным. Широковещательное состояние можно комбинировать с обычным `DataStream` или `KeyedStream`. В примере 7.6 показано, как реализовать приложение для предупреждения об изменении температуры с пороговыми значениями, которые можно динамически настраивать через широковещательный поток.

Пример 7.6 ❖ Подключение широковещательного потока и ключевого потока событий

```
val sensorData: DataStream[SensorReading] = ???
val thresholds: DataStream[ThresholdUpdate] = ???
val keyedSensorData: KeyedStream[SensorReading, String] = sensorData.keyBy(_.id)

// Дескриптор широковещательного состояния.
val broadcastStateDescriptor =
  new MapStateDescriptor[String, Double](
    "thresholds", classOf[String], classOf[Double])

val broadcastThresholds: BroadcastStream[ThresholdUpdate] = thresholds
  .broadcast(broadcastStateDescriptor)

// Соединяем ключевой поток данных и поток правил.
val alerts: DataStream[(String, Double, Double)] = keyedSensorData
  .connect(broadcastThresholds)
  .process(new UpdatableTemperatureAlertFunction())
```

Функция с широковещательным состоянием применяется к двум потокам в три этапа.

1. Создаем `BroadcastStream`, вызывая `DataStream.broadcast()` и предоставляя один или несколько объектов `MapStateDescriptor`. Каждый дескриптор определяет отдельное состояние широковещательной передачи функции, которое позже применяется к `BroadcastStream`.
2. Соединяем `BroadcastStream` с `DataStream` или `KeyedStream`. `BroadcastStream` необходимо указать в качестве аргумента в методе `connect()`.
3. Применяем функцию к подключенным потокам. В зависимости от того, является ли другой поток ключевым или нет, может применяться функция `KeyedBroadcastProcessFunction` или `BroadcastProcessFunction`.

В примере 7.7 показана реализация функции `KeyedBroadcastProcessFunction`, которая поддерживает динамическую настройку пороговых значений датчиков во время выполнения.

Пример 7.7 ❖ Реализация функции `KeyedBroadcastProcessFunction`

```
class UpdatableTemperatureAlertFunction()
    extends KeyedBroadcastProcessFunction
      [String, SensorReading, ThresholdUpdate, (String, Double, Double)] {

  // Дескриптор широковещательного состояния.
  private lazy val thresholdStateDescriptor =
    new MapStateDescriptor[String, Double](
      "thresholds", classOf[String], classOf[Double])

  // Переменная ключевого состояния.
  private var lastTempState: ValueState[Double] = _

  override def open(parameters: Configuration): Unit = {
    // Создаем дескриптор ключевого состояния.
    val lastTempDescriptor = new ValueStateDescriptor[Double](
```

```
      "lastTemp", classOf[Double])
    // Получаем ключевое состояние.
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
  }

  override def processBroadcastElement(
      update: ThresholdUpdate,
      ctx: KeyedBroadcastProcessFunction
        [String, SensorReading, ThresholdUpdate, (String, Double, Double)]#Context,
      out: Collector[(String, Double, Double)]): Unit = {
    // Переменная широковещательного ключевого состояния.
    val thresholds = ctx.getBroadcastState(thresholdStateDescriptor)

    if (update.threshold != 0.0d) {
      // Настраиваем новое пороговое значение для датчика.
      thresholds.put(update.id, update.threshold)
    } else {
    // Удаляем пороговое значение датчика.
    thresholds.remove(update.id)
  }
}

override def processElement(
    reading: SensorReading,
    readOnlyCtx: KeyedBroadcastProcessFunction
      [String, SensorReading, ThresholdUpdate,
      (String, Double, Double)]#ReadOnlyContext,
    out: Collector[(String, Double, Double)]): Unit = {
   // Получаем широковещательное состояние только для чтения.
   val thresholds = readOnlyCtx.getBroadcastState(thresholdStateDescriptor)
   // Проверяем, установлен ли порог.
   if (thresholds.contains(reading.id)) {
     // Получаем порог для датчика.
     val sensorThreshold: Double = thresholds.get(reading.id)
     // Извлекаем последнюю температуру из состояния.
     val lastTemp = lastTempState.value()
     // Проверяем, нужно ли передавать сигнал тревоги.
     val tempDiff = (reading.temperature - lastTemp).abs
     if (tempDiff > sensorThreshold) {
       // Температура изменилась больше чем на пороговое значение.
       out.collect((reading.id, reading.temperature, tempDiff))
     }
    }
    // Обновляем состояние lastTemp.
    this.lastTempState.update(reading.temperature)
  }
}
```

`BroadcastProcessFunction` и `KeyedBroadcastProcessFunction` отличаются от обычной `CoProcessFunction`, потому что методы обработки элементов несимметричны. Методы `processElement()` и `processBroadcastElement()` вызываются с разными объектами контекста. Оба объекта контекста предлагают метод `getBroadcastState(MapStateDescriptor)`, который обеспечивает доступ к дескрип-

тору состояния широковещательной передачи. Однако дескриптор состояния широковещания, который возвращается в методе `processElement()`, обеспечивает доступ только для чтения к состоянию широковещания. Это механизм безопасности, гарантирующий, что состояние широковещательной передачи содержит одинаковую информацию во всех параллельных экземплярах. Кроме того, оба объекта контекста также предоставляют доступ к отметке времени события, текущему водяному знаку, текущему времени обработки и боковым выводам, аналогично объектам контекста других функций процесса.

Функции `BroadcastProcessFunction` и `KeyedBroadcastProcess` также отличаются друг от друга. `BroadcastProcessFunction` не предоставляет службу таймера для регистрации таймеров и, следовательно, не предлагает метод `onTimer()`. Обратите внимание, что вам не следует обращаться к ключевому состоянию из метода `processBroadcastElement()` функции `KeyedBroadcastProcessFunction`. Поскольку широковещательный ввод не указывает ключ, бэкенд состояния не может получить доступ к значению с ключом и вызовет исключение. Вместо этого контекст метода `KeyedBroadcastProcessFunction.processBroadcastElement()` предоставляет метод `applyToKeyedState(StateDescriptor, KeyedStateFunction)` для применения `KeyedStateFunction` к значению каждого ключа в ключевом состоянии, на которое ссылается `StateDescriptor`.

Трансляция событий может происходить не в детерминированном порядке

Порядок, в котором транслируемые события поступают в разные параллельные задачи оператора состояния вещания, может быть разным, если оператор, который передает транслируемые сообщения, работает с параллелизмом больше 1.

Следовательно, вы должны либо убедиться, что значение состояния широковещательной передачи не зависит от порядка, в котором принимаются широковещательные сообщения, либо убедиться, что параллелизм оператора широковещательной передачи установлен на 1.

7.1.4. Использование интерфейса CheckpointedFunction

Интерфейс `CheckpointedFunction` – это интерфейс самого низкого уровня для определения функций с учетом состояния. Он предоставляет перехватчики для регистрации и поддержания ключевого состояния и состояния оператора и является единственным интерфейсом, который предоставляет доступ к каталожному состоянию оператора – состоянию, которое полностью реплицируется в случае восстановления или перезапуска точки сохранения[1].

Интерфейс `CheckpointedFunction` определяет два метода, `initializeState()` и `snapshotState()`, которые работают аналогично методам интерфейса `ListCheckpointed` для списочного состояния операторов. Метод `initializeState()` вызывается при создании параллельного экземпляра `CheckpointedFunction`. Это происходит при запуске приложения или при перезапуске задачи из-за сбоя. Метод вызывается с объектом `FunctionInitializationContext`, который

[1] См. главу 3 для получения подробной информации о том, как распределяется каталожное состояние.

обеспечивает доступ к `OperatorStateStore` и объекту `KeyedStateStore`. Хранилища состояний отвечают за регистрацию состояния функции во время выполнения Flink и возвращают объекты состояния, такие как `ValueState`, `ListState` или `BroadcastState`. Каждое состояние регистрируется под именем, которое должно быть уникальным для функции. Когда функция регистрирует состояние, хранилище состояний пытается его инициализировать, проверяя, хранит ли бэкенд состояние для функции, зарегистрированной под данным именем. Если задача перезапущена из-за сбоя или из точки сохранения, состояние будет инициализировано из сохраненных данных. Если приложение не запускается из контрольной точки или точки сохранения, состояние изначально будет пустым.

Метод `snapshotState()` вызывается непосредственно перед взятием контрольной точки и получает объект `FunctionSnapshotContext` в качестве параметра. `FunctionSnapshotContext` дает доступ к уникальному идентификатору контрольной точки и метке времени, когда JobManager инициирует контрольную точку. Цель метода `snapshotState()` – обеспечить обновление всех объектов состояния до того, как будет выполнена контрольная точка. Более того, в сочетании с интерфейсом `CheckpointListener` метод `snapshotState()` может использоваться для последовательной записи данных во внешние хранилища данных путем синхронизации с контрольными точками Flink.

В примере 7.8 показано, как интерфейс `CheckpointedFunction` используется для создания функции с ключевым состоянием, которая подсчитывает для каждого ключа и экземпляра оператора, сколько показаний датчика превышает указанный порог.

Пример 7.8 ❖ Функция, реализующая интерфейс `CheckpointedFunction`

```
class HighTempCounter(val threshold: Double)
    extends FlatMapFunction[SensorReading, (String, Long, Long)]
    with CheckpointedFunction {

  // Локальная переменная для оператора счетчика высокой температуры.
  var opHighTempCnt: Long = 0
  var keyedCntState: ValueState[Long] = _
  var opCntState: ListState[Long] = _

  override def flatMap(
      v: SensorReading,
      out: Collector[(String, Long, Long)]): Unit = {
    // Проверяем, высокая ли температура.
    if (v.temperature > threshold) {
      // Обновляем локальный оператор счетчика высокой температуры.
      opHighTempCnt += 1
      // Обновляем счетчик высокой температуры с ключом.
      val keyHighTempCnt = keyedCntState.value() + 1
      keyedCntState.update(keyHighTempCnt)
      // Выпускаем новый счетчик.
      out.collect((v.id, keyHighTempCnt, opHighTempCnt))
    }
  }
```

```
  override def initializeState(initContext: FunctionInitializationContext): Unit = {
    // Инициализируем ключевое состояние.
    val keyCntDescriptor = new ValueStateDescriptor[Long]("keyedCnt", classOf[Long])
    keyedCntState = initContext.getKeyedStateStore.getState(keyCntDescriptor)
    // Инициализируем состояние оператора.
    val opCntDescriptor = new ListStateDescriptor[Long]("opCnt", classOf[Long])
    opCntState = initContext.getOperatorStateStore.getListState(opCntDescriptor)
    // Инициализируем локальные переменные состояния.
    opHighTempCnt = opCntState.get().asScala.sum
  }

  override def snapshotState(
      snapshotContext: FunctionSnapshotContext): Unit = {
    // Обновляем состояние оператора локальным значением.
    opCntState.clear()
    opCntState.add(opHighTempCnt)
  }
}
```

7.1.5. Получение уведомлений о пройденных контрольных точках

Частая синхронизация – основная причина ограничений быстродействия в распределенных системах. Архитектура Flink направлена на уменьшение количества точек синхронизации. Контрольные точки реализуются на основе барьеров, которые проходят вместе с данными и, следовательно, позволяют избежать глобальной синхронизации между всеми операторами приложения.

Благодаря своему механизму контрольных точек Flink может достичь очень хорошей производительности. Однако обратная сторона медали состоит в том, что состояние приложения никогда не является согласованным, за исключением логических моментов времени, когда устанавливается контрольная точка. Для некоторых операторов может быть важно знать, пройдена контрольная точка или нет. Например, функции приемника, которые стремятся записать данные во внешние системы с гарантией «ровно один раз», должны выдавать только те записи, которые были получены до успешной контрольной точки, чтобы гарантировать, что полученные данные не будут заново пересчитаны в случае сбоя.

Как обсуждалось в разделе 3.5, контрольная точка является успешной только в том случае, если все задачи оператора успешно записали свои состояния в хранилище контрольных точек. Следовательно, только JobManager может определить, успешна контрольная точка или нет. Операторы, которые должны быть уведомлены о выполненных контрольных точках, могут реализовать интерфейс `CheckpointListener`. Этот интерфейс предоставляет метод `notifyCheckpointComplete(long chkpntId)`, который может быть вызван, когда JobManager регистрирует контрольную точку как завершенную – когда все операторы успешно скопировали свое состояние в удаленное хранилище.

Обратите внимание, что Flink не гарантирует, что метод `notifyCheckpointComplete()` вызывается для каждой завершенной контрольной точки. Возможно, задача пропустила уведомление. Это нужно учитывать при реализации интерфейса.

7.2. Включение восстановления после сбоя для приложений с учетом состояния

Предполагается, что потоковые приложения работают непрерывно и должны восстанавливаться после сбоев, таких как отказы машин или процессов. Большинство потоковых приложений требует, чтобы сбои не влияли на правильность вычисленных результатов.

В разделе 3.5 мы рассказали про механизм Flink для создания согласованных контрольных точек приложения с учетом состояния, моментального снимка состояния всех встроенных и определяемых пользователем функций с учетом состояния в момент времени, когда все операторы обрабатывали все события до определенной позиции во входных потоках приложения. Чтобы обеспечить отказоустойчивость приложения, JobManager инициирует контрольные точки через регулярные промежутки времени.

Приложениям необходимо явно включить механизм периодического создания контрольной точки через `StreamExecutionEnvironment`, как показано в примере 7.9.

Пример 7.9 ❖ Включение контрольной точки для приложения

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// Устанавливаем интервал контрольных точек 10 секунд (10000 миллисекунд).
env.enableCheckpointing(10000L)
```

Интервал контрольных точек – важный параметр, который влияет на накладные расходы механизма контрольных точек во время регулярной обработки и время, необходимое для восстановления после сбоя. Более короткий интервал контрольной точки приводит к увеличению накладных расходов при регулярной обработке, но может обеспечить более быстрое восстановление, поскольку требуется повторная обработка меньшего количества данных.

Flink предоставляет возможность точной настройки поведения контрольных точек, таких как выбор гарантий согласованности («ровно один раз» или «хотя бы один раз»), количество одновременных контрольных точек и тайм-аут для отмены длительных контрольных точек, а также несколько параметров, предназначенных для бэкенда. Мы обсуждаем эти варианты более подробно в разделе 10.3.

7.3. Обеспечение работоспособности приложений с учетом состояния

Повторное вычисление состояний приложения, которое работало несколько недель, может обойтись очень дорого или вообще оказаться невозможным. В то же время необходимо поддерживать долгоработающие приложения. Необходимо исправлять ошибки, настраивать, добавлять или удалять функциональные возможности или настраивать параллелизм оператора с учетом более высоких или низких скоростей передачи данных. Следовательно, важно, чтобы состояние приложения можно было перенести в новую версию приложения или перераспределить для большего или меньшего числа операторских задач.

Flink имеет точки сохранения для поддержки приложений и их состояний. Однако для этого требуется, чтобы все операторы с учетом состояния в начальной версии приложения указывали два параметра, чтобы обеспечить надлежащее обслуживание приложения в будущем. Эти параметры представляют собой уникальный идентификатор оператора и максимальный параллелизм (для операторов с ключевым состоянием). Далее мы опишем, как установить эти параметры.

Уникальные идентификаторы оператора и максимальный параллелизм встроены в точки сохранения

Уникальный идентификатор и максимальный параллелизм операторов встроены в точку сохранения и не могут быть изменены. Невозможно запустить приложение из ранее взятой точки сохранения, если были изменены идентификаторы или максимальный параллелизм операторов.

После изменения идентификаторов операторов или максимального параллелизма вы не сможете запустить приложение из точки сохранения, а должны будете запускать его с нуля без какой-либо инициализации состояния.

7.3.1. Указание уникальных идентификаторов оператора

Для каждого оператора приложения должны быть указаны уникальные идентификаторы. Идентификаторы записываются в точку сохранения как метаданные с фактическими данными о состоянии оператора. Когда приложение запускается из точки сохранения, идентификаторы используются для отображения состояния в точке сохранения на соответствующий оператор запущенного приложения. Состояние точки сохранения может быть передано оператору запущенного приложения только при совпадении их идентификаторов.

Если вы не зададите явно уникальные идентификаторы операторам вашего приложения с учетом состояния, вы столкнетесь со значительными ограничениями, когда придется развивать приложение. Мы обсуждали важность уникальных идентификаторов операторов и отображение состояния точки сохранения более подробно в разделе 3.5.5.

Мы настоятельно рекомендуем назначать уникальные идентификаторы каждому оператору приложения. Вы можете установить идентификатор с помощью метода `uid()`, как показано в примере 7.10.

Пример 7.10 ❖ Установка уникального идентификатора оператора

```
val alerts: DataStream[(String, Double, Double)] = keyedSensorData
  .flatMap(new TemperatureAlertFunction(1.1))
  .uid("TempAlert")
```

7.3.2. Определение максимального параллелизма операторов ключевого состояния

Параметр максимального параллелизма оператора определяет количество групп ключей, на которые разбивается ключевое состояние оператора. Количество групп ключей ограничивает максимальное количество параллельных задач, до которых можно масштабировать ключевое состояние. В разделе 3.4.4 мы рассказывали о группах ключей и о том, как масштабируется состояние с ключом. Максимальный параллелизм может быть установлен для всех операторов приложения с помощью `StreamExecutionEnvironment` или для каждого оператора с помощью метода `setMaxParallelism()`, как показано в примере 7.11.

Пример 7.11 ❖ Назначение максимального параллелизма операторов

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// Устанавливаем максимальный параллелизм для этого приложения.
env.setMaxParallelism(512)

val alerts: DataStream[(String, Double, Double)] = keyedSensorData
.flatMap(new TemperatureAlertFunction(1.1))
// Устанавливаем максимальный параллелизм для этого оператора
// и переписываем значение.
.setMaxParallelism(1024)
```

Максимальный параллелизм оператора по умолчанию зависит от параллелизма оператора в первой версии приложения:

- если параллелизм меньше или равен 128, максимальный параллелизм равен 128;
- если параллелизм оператора больше 128, максимальный параллелизм вычисляется как минимум из `nextPowerOfTwo(parallelism + (parallelism/2))` и `2^15`.

7.4. Производительность и надежность приложений с учетом состояния

То, как операторы взаимодействуют с состоянием, влияет на надежность и производительность приложения. Есть несколько аспектов, которые влияют на поведение приложения, такие как выбор архитектуры для хранения состояния, которая локально поддерживает состояние и выполняет контрольные точки, конфигурация алгоритма контрольных точек и размер состояния приложения. В этом разделе мы обсуждаем аспекты, которые необходимо принимать во внимание, чтобы гарантировать надежное выполнение и стабильную производительность долгоработающих приложений.

7.4.1. Выбор бэкенда состояния

В разделе 3.4.3 мы говорили, что Flink хранит состояние приложения в бэкенде состояния. Бэкенд состояния отвечает за сохранение локального состояния каждого экземпляра задачи и отправку его в удаленном хранилище при установке контрольной точки. Поскольку локальное состояние может поддерживаться и проверяться по-разному, бэкенды состояния являются подключаемыми – два приложения могут использовать разные реализации внутреннего состояния для хранения своих состояний. Выбор бэкенда для хранения состояния влияет на надежность и быстродействие приложения. Каждый бэкенд состояния предоставляет реализации для различных примитивов состояния, таких как `ValueState`, `ListState` и `MapState`.

В настоящее время Flink предлагает три бэкенда состояния: `MemoryStateBackend`, `FsStateBackend` и `RocksDBStateBackend`:

- `MemoryStateBackend` сохраняет состояние как обычные объекты в куче процесса JVM TaskManager. Например, `MapState` поддерживается объектом Java `HashMap`. Хотя этот подход обеспечивает очень низкие задержки для чтения или записи состояния, он влияет на надежность приложения. Если состояние экземпляра задачи становится слишком большим, JVM и все экземпляры задач, запущенные на нем, могут быть уничтожены из-за `OutOfMemoryError`. Более того, этот подход может страдать от пауз при сборке мусора, поскольку он помещает в кучу много долгоживущих объектов. Когда контрольная точка установлена, `MemoryStateBackend` отправляет состояние в JobManager, который сохраняет его в своей динамической памяти. Следовательно, общее состояние приложения должно умещаться в памяти JobManager. Поскольку его память изменчива, состояние теряется в случае сбоя JobManager. Из-за этих ограничений `MemoryStateBackend` рекомендуется только для целей разработки и отладки;
- `FsStateBackend` сохраняет локальное состояние в куче JVM TaskManager, как и `MemoryStateBackend`. Однако вместо того, чтобы содержать состояние в энергозависимой памяти JobManager, `FsStateBackend` записывает

состояние в удаленную и постоянную файловую систему. Следовательно, `FsStateBackend` обеспечивает скорость за счет локальной памяти и отказоустойчивость в случае сбоев. Однако он ограничен размером памяти TaskManager и может страдать от пауз при сборке мусора;

- `RocksDBStateBackend` сохраняет все состояние в локальных экземплярах RocksDB. RocksDB – это встроенное хранилище ключей и значений, которое сохраняет данные на локальном диске. Для чтения и записи данных из RocksDB и в RocksDB их необходимо, соответственно, десериализовать и сериализовать. `RocksDBStateBackend` также указывает состояние удаленной и постоянной файловой системы. Поскольку он записывает данные на диск и поддерживает инкрементные контрольные точки (подробнее об этом см. раздел 3.5), `RocksDBStateBackend` – хороший выбор для приложений с очень большим состоянием. Пользователи сообщали о приложениях с размером состояния в несколько терабайт, использующих `RocksDBStateBackend`. Однако чтение и запись данных на диск и накладные расходы на десериализацию/сериализацию объектов приводят к снижению производительности чтения и записи по сравнению с сохранением состояния в куче.

Поскольку `StateBackend` является общедоступным интерфейсом, также возможно реализовать собственный бэкенд состояния. В примере 7.12 показано, как настроить бэкенд состояния (здесь `RocksDBStateBackend`) для приложения и всех его функций с учетом состояния.

Пример 7.12 ❖ Настройка `RocksDBStateBackend` для приложения

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val checkpointPath: String = ???
// Настройка пути сохранения контрольных точек в удаленной файловой системе.
val backend = new RocksDBStateBackend(checkpointPath)
// Настройка бэкенда состояния.
env.setStateBackend(backend)
```

В разделе 10.3 мы обсудим, как использовать и настраивать бэкенды состояния в вашем приложении.

7.4.2. Выбор примитива состояния

Производительность оператора с учетом состояния (встроенного или определяемого пользователем) зависит от нескольких аспектов, включая типы данных состояния, бэкенд состояния приложения и выбранные примитивы состояния.

Для бэкендов состояния, которые десериализуют/сериализуют объекты состояния при чтении или записи, таких как `RocksDBStateBackend`, выбор примитива состояния (`ValueState`, `ListState` или `MapState`) может оказать большое влияние на производительность приложения. Например, `ValueState` полностью десериализуется при доступе к нему и сериализуется при обновлении.

Реализация RocksDBStateBackend в ListState десериализует все записи списка перед созданием Iterable для чтения значений. Однако добавление одного значения в ListState – добавление его в конец списка – является дешевой операцией, поскольку сериализуется только добавленное значение. MapState в RocksDBStateBackend позволяет читать и записывать значения для каждого ключа – де-/сериализуются только те ключи и значения, которые считываются или записываются. При итерации по набору записей MapState сериализованные записи предварительно выбираются из RocksDB и десериализуются только при фактическом доступе к ключу или значению.

Например, с RocksDBStateBackend более эффективно использовать MapState [X, Y] вместо ValueState[HashMap[X, Y]]. ListState[X] имеет преимущество перед ValueState[List[X]], если элементы часто добавляются к списку, а к элементам списка реже обращаются.

Еще одна хорошая практика – обновлять состояние только один раз за вызов функции. Поскольку контрольные точки синхронизируются с вызовами функций, множественные обновления состояния не дают никаких преимуществ, но могут вызвать дополнительные издержки сериализации при обновлении состояния несколько раз за один вызов функции.

7.4.3. Предотвращение утечки состояния

Продолжительность непрерывной работы потоковых приложений часто измеряется месяцами или годами. Если состояние приложения постоянно увеличивается, в какой-то момент оно станет слишком большим и вынудит приложение завершить работу, если не будут предприняты действия по масштабированию приложения для выделения дополнительных ресурсов. Чтобы предотвратить увеличение потребления ресурсов приложением с течением времени, важно контролировать размер состояния оператора. Поскольку обработка состояния напрямую влияет на семантику оператора, Flink не может автоматически очищать состояние и свободное хранилище. Вместо этого все операторы, использующие состояние, должны контролировать размер своего состояния и следить за тем, чтобы оно не увеличивалось бесконечно.

Распространенной причиной роста состояния является размещение ключевого состояния в постоянно растущем пространстве значений ключа. В этом сценарии функция с учетом состояния получает записи с ключами, которые активны только в течение определенного периода времени и никогда не принимаются после этого. Типичным примером является поток событий кликов мышью, в котором у кликов есть атрибут идентификатора сеанса, срок действия которого истекает через некоторое время. В таком случае функция с ключевым состоянием будет накапливать состояния для нарастающего количества ключей. По мере роста ключевого пространства ключи с истекшим сроком действия становятся устаревшими и бесполезными. Решение этой проблемы – удалить состояния, ключи которых утратили актуальность. Однако функция с ключевым состоянием может получить доступ к состоянию только в том случае, если она получила запись с соответствующим ключом. Во многих случаях функция не знает, будет ли текущая запись последней для

ключа. Следовательно, она не сможет отбросить ключевое состояние, потому что теоретически для этого ключа может поступить новая запись.

Эта проблема существует не только для настраиваемых функций с учетом состояния, но и для некоторых встроенных операторов API DataStream. Например, вычисление выполняемых агрегатов в `KeyedStream` либо с помощью встроенных функций агрегирования, таких как `min`, `max`, `sum`, `minBy` или `maxBy`, либо с помощью настраиваемой функции `ReduceFunction` или `AggregateFunction` сохраняет состояние для каждого ключа и никогда не отбрасывает его. Следовательно, эти функции нужно использовать только в том случае, если ключевые значения взяты из постоянной и ограниченной области. Другими примерами являются окна с триггерами на основе подсчета, которые обрабатывают и очищают свое состояние при получении определенного количества записей. На окна с триггерами, основанными на времени (время обработки и время события), это не влияет, поскольку они запускают и очищают свое состояние в зависимости от времени.

Это означает, что вы должны принимать во внимание требования к приложению и свойства его входных данных, такие как пространство ключей, при разработке и реализации операторов с учетом состояния. Если вашему приложению требуется ключевое состояние для растущего пространства ключей, оно должно гарантировать своевременную очистку устаревших ключевых состояний. Это может быть сделано путем регистрации таймеров на определенный момент времени в будущем[1]. Подобно состоянию, таймеры регистрируются в контексте текущего активного ключа. По срабатыванию таймера вызывается метод обратного вызова и загружается контекст ключа таймера. Следовательно, метод обратного вызова имеет полный доступ к ключевому состоянию, а также может его очистить. Функции, которые предлагают поддержку регистрации таймеров, – это интерфейс `Trigger` для окон и функция обработки. Оба были рассмотрены в главе 6.

В примере 7.13 показана функция `KeyedProcessFunction`, которая сравнивает два последующих измерения температуры и выдает предупреждение, если разница превышает определенный порог. Это тот же вариант использования, что и в предыдущем примере ключевого состояния, но функция `KeyedProcessFunction` также очищает состояние для ключей (то есть идентификаторов датчиков), которые не предоставили никаких новых измерений температуры в течение одного часа по времени события.

Пример 7.13 ❖ Функция `KeyedProcessFunction` с учетом состояния, которая очищает свое состояние по таймеру

```
class SelfCleaningTemperatureAlertFunction(val threshold: Double)
    extends KeyedProcessFunction[String, SensorReading, (String, Double, Double)] {

  // Ключевое состояние, сохраненное для последней температуры.
  private var lastTempState: ValueState[Double] = _
  // Ключевое состояние, сохраненное для последнего зарегистрированного таймера.
  private var lastTimerState: ValueState[Long] = _
```

[1] Таймеры могут основываться на времени события или времени обработки.

```
  override def open(parameters: Configuration): Unit = {
    // Регистрация состояния для последней температуры.
    val lastTempDesc = new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
    // Регистрация состояния для последнего таймера.
    val lastTimerDesc = new ValueStateDescriptor[Long]("lastTimer", classOf[Long])
    lastTimerState = getRuntimeContext.getState(timestampDescriptor)
  }

  override def processElement(
      reading: SensorReading,
      ctx: KeyedProcessFunction
        [String, SensorReading, (String, Double, Double)]#Context,
      out: Collector[(String, Double, Double)]): Unit = {

    // Вычисление метки времени нового таймера как метка времени записи плюс 1 час.
    val newTimer = ctx.timestamp() + (3600 * 1000)
    // Получение метки времени текущего таймера.
    val curTimer = lastTimerState.value()
    // Удаление предыдущего таймера и регистрация нового таймера.
    ctx.timerService().deleteEventTimeTimer(curTimer)
    ctx.timerService().registerEventTimeTimer(newTimer)
    // Обновление состояния таймера с его меткой времени.
    lastTimerState.update(newTimer)

    // Извлечение последней температуры из состояния.
    val lastTemp = lastTempState.value()
    // Проверка, надо ли выдать предупреждение.
    val tempDiff = (reading.temperature - lastTemp).abs
    if (tempDiff > threshold) {
      // Температура возросла больше, чем на пороговое значение.
      out.collect((reading.id, reading.temperature, tempDiff))
    }

    // Обновляем состояние lastTemp.
    this.lastTempState.update(reading.temperature)
  }

  override def onTimer(
      timestamp: Long,
      ctx: KeyedProcessFunction
        [String, SensorReading, (String, Double, Double)]#OnTimerContext,
      out: Collector[(String, Double, Double)]): Unit = {

    // Очищаем состояние с данным ключом.
    lastTempState.clear()
    lastTimerState.clear()
  }
}
```

Механизм очистки состояния, реализованный вышеупомянутой функцией `KeyedProcessFunction`, работает следующим образом. Для каждого события ввода вызывается метод `processElement()`. Перед сравнением измерений тем-

пературы и обновлением последней температуры метод обновляет таймер очистки, удаляя предыдущий таймер и регистрируя новый. Время очистки вычисляется путем добавления одного часа к метке времени текущей записи. Чтобы иметь возможность удалить текущий зарегистрированный таймер, его временная метка сохраняется в дополнительном `ValueState[Long]`, называемом `lastTimerState`. После этого метод сравнивает температуры, возможно, выдает предупреждение и обновляет свое состояние.

Поскольку наша функция `KeyedProcessFunction` всегда обновляет зарегистрированный таймер, удаляя текущий таймер и регистрируя новый, для каждого ключа регистрируется только один таймер. Как только этот таймер срабатывает, вызывается метод `onTimer()`. Метод очищает все состояния, связанные с ключом, последней температурой и последним состоянием таймера.

7.5. Развитие приложений с учетом состояния

Часто бывает необходимо исправить ошибку или доработать бизнес-логику долгоработающего потокового приложения, учитывающего состояние. Следовательно, работающее приложение необходимо заменить на обновленную версию, как правило, без потери состояния приложения.

Flink поддерживает такие обновления, выбирая точку сохранения работающего приложения, останавливая его и запуская новую версию приложения из точки сохранения[1]. Однако обновление приложения с сохранением его состояния возможно только при определенных изменениях приложения – исходное приложение и его новая версия должны быть совместимы с точкой сохранения. Далее мы расскажем, как можно развивать приложения, сохраняя совместимость с точками сохранения.

В разделе 3.5.5 мы говорили, что к каждому состоянию в точке сохранения можно обращаться с помощью составного идентификатора, состоящего из уникального идентификатора оператора и имени состояния, объявленного дескриптором состояния.

При разработке приложений не забывайте о развитии

Важно понимать, что первоначальная архитектура приложения определяет, можно ли его изменить позже совместимым с точкой сохранения способом. Многие изменения будут невозможны, если исходная версия не была разработана с учетом обновлений. Назначение операторам уникальных идентификаторов обязательно для большинства изменяемых приложений.

Когда приложение запускается из точки сохранения, операторы запущенного приложения инициализируются путем поиска соответствующих состояний из точки сохранения с использованием идентификаторов операторов и имен состояний. С точки зрения совместимости с точкой сохранения это означает, что приложение можно развивать тремя способами.

---

[1] В главе 10 объясняется, как создавать точки сохранения запущенных приложений и как запускать новое приложение из существующей точки сохранения.

1. Обновление или расширение логики приложения без изменения или удаления существующего состояния. Обновление включает добавление в приложение операторов с учетом состояния или без него.
2. Удаление состояния из приложения.
3. Изменение состояния существующего оператора путем изменения примитива состояния или типа данных состояния.

В следующих разделах мы обсудим эти три случая.

7.5.1. Обновление приложения без изменения существующего состояния

Если приложение обновляется без удаления или изменения существующего состояния, оно всегда совместимо с точкой сохранения и может быть запущено из точки сохранения более ранней версии.

Если вы добавляете новый оператор с учетом состояния в приложение или новое состояние в существующий оператор, состояние будет инициализировано как пустое при запуске приложения из точки сохранения.

Изменение типа входных данных встроенных операторов с учетом состояния

Обратите внимание, что изменение типа входных данных встроенных операторов с учетом состояния, таких как агрегирование окон, объединение с привязкой ко времени или асинхронные функции, часто изменяет тип их внутреннего состояния. Следовательно, такие изменения несовместимы с точками сохранения, даже если они выглядят незначительными.

7.5.2. Удаление состояния из приложения

Вместо добавления новых состояний в приложение вы также можете доработать приложение, удалив состояние – либо путем полного удаления оператора, учитывающего состояние, либо просто состояния из функции. Когда новая версия приложения запускается из точки сохранения предыдущей версии, точка сохранения содержит состояние, которое не может быть сопоставлено с перезапущенным приложением. То же самое происходит и в случае изменения уникального идентификатора оператора или имени состояния.

По умолчанию Flink не запускает приложения, которые не восстанавливают все состояния, содержащиеся в точке сохранения, чтобы избежать потери состояния в этой точке. Однако можно отключить эту проверку безопасности, как описано в разделе 10.1. Следовательно, нетрудно обновить приложение, удалив операторы, учитывающие состояние или состояние из существующего оператора.

7.5.3. Изменение состояния оператора

Хотя добавить или удалить состояние из приложения довольно просто и это не влияет на совместимость точек сохранения, изменить состояние сущест-

вующего оператора немного труднее. Состояние можно изменить двумя способами:

1) изменяя тип данных состояния, например изменяя `ValueState[Int]` на `ValueState[Double]`;
2) путем изменения типа примитива состояния, например путем изменения `ValueState[List[String]]` на `ListState[String]`.

Изменение типа данных состояния возможно в нескольких особых случаях. Однако в настоящее время Flink не поддерживает изменение примитива (или структуры) состояния. Сообщество предложило несколько идей для решения этой задачи, например автономный инструмент для преобразования точек сохранения. Однако, начиная с Flink 1.7, такого инструмента не существует. Далее мы сосредоточимся на изменении типа данных состояния.

Чтобы понять проблему изменения типа данных состояния, мы должны понять, как данные состояния представлены в точке сохранения. Точка сохранения состоит в основном из сериализованных данных состояния. Сериализаторы, преобразующие JVM-объекты состояния в байты, генерируются и настраиваются системой типов Flink. Это преобразование основано на типе данных состояния. Например, если у вас есть `ValueState[String]`, система типов Flink генерирует `StringSerializer` для преобразования объектов `String` в байты. Сериализатор также используется для преобразования необработанных байтов обратно в объекты JVM. В зависимости от того, хранит ли бэкенд состояния сериализованные данные (например, `RocksDBStateBackend`) или как объекты в куче (например, `FSStateBackend`), это происходит, когда состояние считывается функцией или когда приложение перезапускается из точки сохранения.

Поскольку система типов Flink генерирует сериализаторы в зависимости от типа данных состояния, сериализаторы, вероятно, изменятся при изменении типа данных состояния. Например, если вы изменили `ValueState[String]` на `ValueState[Double]`, Flink создаст `DoubleSerializer` для доступа к состоянию. Неудивительно, что использование `DoubleSerializer` для десериализации двоичных данных, сгенерированных путем сериализации `String` с помощью `StringSerializer`, завершится ошибкой. Следовательно, изменение типа данных состояния поддерживается только в очень редких случаях.

В Flink 1.7 изменение типа данных состояния поддерживается, если тип данных был определен как тип Apache Avro и если новый тип данных также является типом Avro, который произошел от исходного типа в соответствии с правилами эволюции схемы Avro. Система типов Flink автоматически генерирует сериализаторы, которые могут читать предыдущие версии типа данных.

Изменение состояния и миграция – важная тема в сообществе Flink, которой уделяется много внимания. Вы можете рассчитывать на улучшенную поддержку этих сценариев в будущих версиях Apache Flink. Несмотря на все эти усилия, мы рекомендуем всегда дважды проверять, можно ли развивать приложение в соответствии с вашим планом, прежде чем запускать его в производство.

7.6. Запрашиваемое состояние

Многим приложениям потоковой обработки необходимо делиться своими результатами с другими приложениями. Распространенным подходом является запись результатов в базу данных или хранилище значений ключей и получение другими приложениями результатов из этого хранилища данных. Такая архитектура подразумевает, что необходимо настроить и поддерживать отдельную систему хранения, что может потребовать больших усилий, особенно если это тоже должна быть распределенная система.

Apache Flink поддерживает так называемое *запрашиваемое состояние* (queryable state) для ситуаций, в которых обычно требуется внешние хранилища данных для обмена данными. Во Flink любое ключевое состояние может быть представлено внешним приложениям как состояние, доступное для запроса, и действует как хранилище значений ключей только для чтения. Приложение потоковой обработки с учетом состояния обрабатывает события как обычно, а также сохраняет и обновляет свои промежуточные или конечные результаты в запрашиваемом состоянии. Внешние приложения могут запрашивать ключевое состояние во время работы потокового приложения.

Обратите внимание, что поддерживаются только запросы по одиночному ключу. Невозможно запросить диапазоны ключей или даже выполнить более сложные запросы.

Запрашиваемое состояние не перекрывает все варианты использования, требующие внешнего хранилища данных. Например, запрашиваемое состояние доступно только во время работы приложения. Оно недоступно во время перезапуска приложения из-за ошибки, для изменения масштаба приложения или для его миграции в другой кластер. Однако оно значительно упрощает реализацию многих приложений, таких как информационные панели реального времени или другие приложения для мониторинга.

Далее мы обсуждаем архитектуру службы запрашиваемых состояний Flink и объясняем, как внешние приложения могут запрашивать, а потоковые приложения в свою очередь передавать запрашиваемое состояние.

7.6.1. Архитектура и обслуживание запросов состояния

Служба запрашиваемых состояний Flink состоит из трех процессов:

1) `QueryableStateClient` используется внешним приложением для отправки запросов и получения результатов;
2) `QueryableStateClientProxy` принимает и обслуживает клиентские запросы. Каждый TaskManager запускает клиентский прокси-сервер. Поскольку состояние с ключом распределяется по всем параллельным экземплярам оператора, прокси-сервер должен идентифицировать TaskManager, который поддерживает состояние запрошенного ключа. Эта информация запрашивается у JobManager, который управляет на-

значением группы ключей, и кешируется после получения[1]. Клиентский прокси извлекает состояние из бэкенда состояний соответствующего TaskManager и передает результат клиенту;

3) `QueryableStateServer` обслуживает запросы клиентского прокси. Каждый TaskManager запускает бэкенд состояний, который извлекает состояние запрошенного ключа из локальной части бэкенда состояния и возвращает его запрашивающему клиентскому прокси.

На рис. 7.1 показана архитектура службы запрашиваемых состояний.

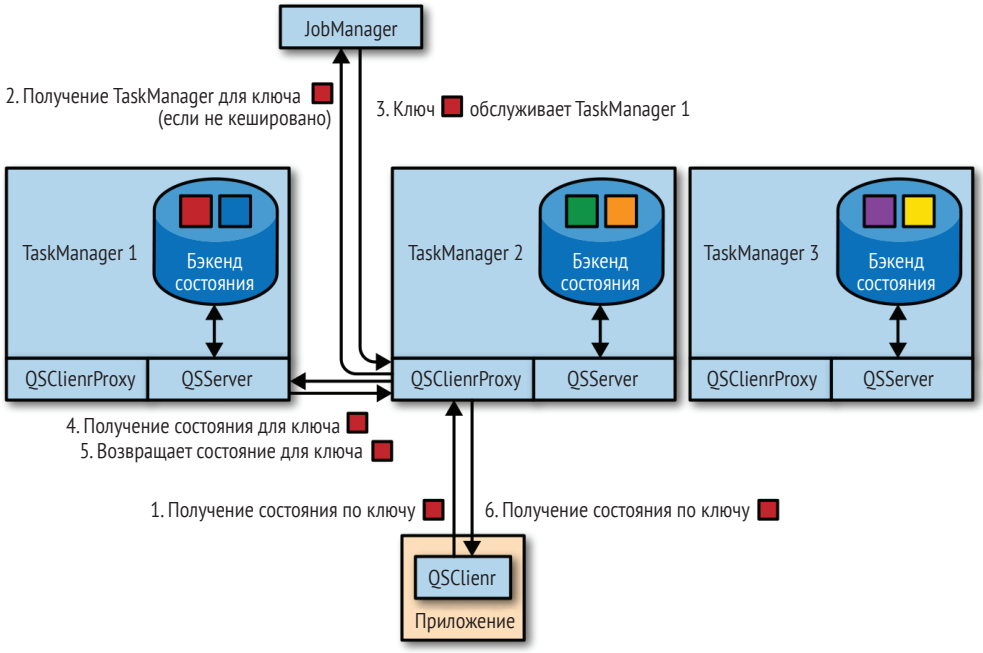

Рис. 7.1 ❖ Архитектура службы запрашиваемых состояний Flink

Чтобы включить службу запрашиваемого состояния в настройке Flink – для запуска клиентского прокси и серверных потоков в процессах TaskManager, – вам необходимо добавить JAR-файл **`flink-queryable-state-runtime`** в путь к классам процесса TaskManager. Это делается путем копирования его из папки **`./opt`** вашей установки в папку **`./lib`**. Когда JAR-файл находится в пути к классам, запрошенные потоки состояния запускаются автоматически и могут обслуживать запросы клиента состояния. При правильной настройке вы найдете следующее сообщение журнала в журналах TaskManager:

```
Started the Queryable State Proxy Server @ …
```

Порты, используемые прокси-сервером и сервером клиента, а также дополнительные параметры можно настроить в файле **`./conf/flink-conf.yaml`**.

[1] Группы ключей обсуждаются в главе 3.

7.6.2. Отображение состояния запроса

Реализовать потоковое приложение с запрашиваемым состоянием просто. Все, что вам нужно сделать, – это определить функцию с ключевым состоянием и сделать состояние доступным для запроса, вызвав метод `setQueryable(String)` в `StateDescriptor` перед получением дескриптора состояния. В примере 7.14 показано, как сделать `lastTempState` доступным для запроса, чтобы проиллюстрировать использование состояния с ключом.

Пример 7.14 ❖ Настройка ключевого состояния для обработки запросов

```
override def open(parameters: Configuration): Unit = {

  // Создаем дескриптор состояния.
  val lastTempDescriptor =
    new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
  // Включаем обработку запросов и задаем внешний идентификатор.
  lastTempDescriptor.setQueryable("lastTemperature")
  // Получаем обработчик состояния.
  lastTempState = getRuntimeContext
    .getState[Double](lastTempDescriptor)
}
```

Внешний идентификатор, который передается с методом `setQueryable()`, может быть выбран произвольно и используется только для настройки клиента запрашиваемого состояния.

В дополнение к универсальному способу включения запросов для любого типа ключевого состояния Flink также предлагает ярлыки для определения приемников потока, которые хранят события потока в запрашиваемом состоянии. В примере 7.15 показано, как использовать приемник запрашиваемого состояния.

Пример 7.15 ❖ Запись DataStream в приемник запрашиваемого состояния

```
val tenSecsMaxTemps: DataStream[(String, Double)] = sensorData
// Отображение на ID датчика и температуру.
.map(r => (r.id, r.temperature))
// Вычисляем каждые 10 секунд максимальную температуру каждого датчика.
.keyBy(_._1)
.timeWindow(Time.seconds(10))
.max(1)

// Сохранаяем максимальную температуру за последние 10 секунд
// для каждого датчика в запрашиваемом состоянии.
tenSecsMaxTemps
  // Ключ по ID датчика.
  .keyBy(_._1)
  .asQueryableState("maxTemperature")
```

Метод `asQueryableState()` добавляет в поток приемник запрашиваемого состояния. Тип запрашиваемого состояния – `ValueState`, который содержит

значения типа входного потока – наш пример (`String, Double`). Для каждой полученной записи запрашиваемый приемник состояния вставляет запись в `ValueState`, так что последнее событие для каждого ключа всегда сохраняется.

Приложение с функцией, имеющей запрашиваемое состояние, выполняется так же, как и любое другое приложение. Вам нужно только убедиться, что диспетчеры задач настроены для запуска своих служб запрашиваемого состояния, как обсуждалось в предыдущем разделе.

7.6.3. Запрос состояния из внешних приложений

Любое приложение на основе JVM может запросить состояние работающего приложения Flink с помощью `QueryableStateClient`. Этот класс предоставляется зависимостью `flink-queryable-state-client-java`, которую вы можете добавить в свой проект следующим образом:

```
<dependency>
  <groupid>org.apache.flink</groupid>
  <artifactid>flink-queryable-state-client-java_2.12</artifactid>
  <version>1.7.1</version>
</dependency>
```

`QueryableStateClient` инициализируется именем хоста любого TaskManager и портом, который прослушивает прокси-сервер запрашиваемого состояния. По умолчанию клиентский прокси прослушивает порт 9067, но порт можно настроить в файле **`./conf/flink-conf.yaml`**:

```
val client: QueryableStateClient =
  new QueryableStateClient(tmHostname, proxyPort)
```

После получения клиента состояния вы можете запросить состояние приложения, вызвав метод `getKvState()`. Метод принимает несколько параметров, таких как `JobID` запущенного приложения, идентификатор состояния, ключ, для которого должно быть выбрано состояние, `TypeInformation` для ключа и `StateDescriptor` запрашиваемого состояния. `JobID` можно получить через REST API, веб-интерфейс или файлы журнала. Метод `getKvState()` возвращает `CompletableFuture[S]`, где `S` – это тип состояния (например, `ValueState[_]` или `MapState[_, _]`). Следовательно, клиент может отправлять несколько асинхронных запросов и ждать их результатов. В примере 7.16 показана простая консольная панель управления, которая запрашивает состояние приложения, показанного в предыдущем разделе.

Пример 7.16 ❖ Простое приложение панели управления, которое запрашивает состояние приложения Flink

```
object TemperatureDashboard {

  // Считаем, что консоль и приложение запущены на одной локальной машине.
  val proxyHost = "127.0.0.1"
  val proxyPort = 9069
```

```
// jobId запущенного QueryableStateJob
// может быть найдено в журнале или в списке задач в веб-интерфейсе.
val jobId = "d2447b1a5e0d952c372064c886d2220a"

// Количество датчиков в запросе.
val numSensors = 5
// Частота запросов состояния.
val refreshInterval = 10000

def main(args: Array[String]): Unit = {
  // Настраиваем приложение-клиент с хостом и портом запрашиваемого прокси-сервера.
  val client = new QueryableStateClient(proxyHost, proxyPort)

  val futures = new Array[
    CompletableFuture[ValueState[(String, Double)]]](numSensors)
  val results = new Array[Double](numSensors)

  // Печатаем заголовок таблицы консольного вывода.
  val header =
    (for (i <- 0 until numSensors) yield "sensor_" + (i + 1))
     .mkString("\t| ")
  println(header)

  // Бесконечный цикл.
  while (true) {
    // Отправка асинхронных запросов.
    for (i <- 0 until numSensors) {
      futures(i) = queryState("sensor_" + (i + 1), client)
    }
    // Ожидание результата.
    for (i <- 0 until numSensors) {
      results(i) = futures(i).get().value()._2
    }
    // Печать результата.
    val line = results.map(t => f"$t%1.3f").mkString("\t| ")
    println(line)

    // Ожидание отправки следующего запроса.
     Thread.sleep(refreshInterval)
  }
  client.shutdownAndWait()
}

def queryState(
  key: String,
  client: QueryableStateClient)
: CompletableFuture[ValueState[(String, Double)]] = {

client
  .getKvState[String, ValueState[(String, Double)], (String, Double)](
    JobID.fromHexString(jobId),
    "maxTemperature",
    key,
    Types.STRING,
```

```
      new ValueStateDescriptor[(String, Double)](
        "", // Имя состояния здесь не имеет значения.
        Types.TUPLE[(String, Double)]))
  }
}
```

Чтобы запустить код примера, вы должны сначала запустить потоковое приложение с запрашиваемым состоянием. После запуска найдите `JobID` в файле журнала или в веб-интерфейсе; пропишите этот `JobID` в коде панели управления и также запустите его. После этого панель управления начнет запрашивать состояние запущенного потокового приложения.

7.7. Заключение

Практически каждое нетривиальное потоковое приложение использует состояние. API DataStream предоставляет мощные, но простые в использовании инструменты для доступа и обработки состояния оператора. Он предлагает различные типы примитивов состояния и поддерживает подключаемые бэкенды состояния. Хотя разработчики обладают большой свободой во взаимодействии с состоянием, среда выполнения Flink управляет терабайтами состояния и обеспечивает семантику «ровно один раз» в случае сбоя. Комбинация вычислений на основе времени, описанных в главе 6, и масштабируемого управления состоянием дает разработчикам возможность реализовать сложные потоковые приложения. Запрашиваемое состояние – это простая в использовании функция, которая может избавить вас от необходимости настраивать и поддерживать базу данных или хранилище ключей и значений для передачи результатов потокового приложения внешним приложениям.

Глава 8

Чтение и запись при работе с внешними системами

Данные могут храниться во множестве различных систем, таких как файловые системы, хранилища объектов, системы реляционных баз данных, хранилища пар «ключ–значение», поисковые индексы, журналы событий, очереди сообщений и т. д. Каждый класс систем был разработан под строго определенные цели и отлично справляется с задачей. Поэтому современные инфраструктуры часто включают в себя несколько различных систем хранения. Прежде чем добавлять в этот салат новый компонент, следует задать очевидный вопрос: «Насколько хорошо он будет сочетаться с другими компонентами в моей инфраструктуре?»

Внедрение системы обработки данных, такой как Apache Flink, требует тщательного обсуждения, поскольку она не содержит собственную подсистему хранения, а полагается на внешние системы. Следовательно, для процессоров данных, таких как Flink, важно обеспечить хорошую библиотеку соединителей для чтения и записи данных во внешние системы, а также API для реализации настраиваемых соединителей. Однако для потокового процессора, который хочет предоставить значимые гарантии целостности данных в случае сбоя, недостаточно просто иметь возможность читать или записывать данные во внешние хранилища данных.

В этой главе мы обсудим, как соединители источника и приемника влияют на гарантии согласованности и сохранности данных потоковых приложений Flink, и представим самые популярные соединители Flink для чтения и записи данных. Вы узнаете, как реализовать настраиваемые соединители источника и приемника, а также функции, которые отправляют асинхронные запросы на чтение или запись во внешние хранилища данных.

8.1. Гарантии согласованности приложений

В разделе 3.5 вы узнали, что механизм контрольных точек и восстановления Flink периодически устанавливает согласованные контрольные точки состо-

яния приложения. В случае сбоя состояние приложения восстанавливается с последней завершенной контрольной точки, и обработка продолжается. Однако возможности сбросить состояние приложения до согласованного значения недостаточно для достижения удовлетворительных гарантий обработки для приложения. Соединители источника и приемника приложения должны быть интегрированы с механизмом контрольных точек и восстановления Flink и обеспечивать определенные свойства, чтобы иметь возможность давать значимые гарантии.

Чтобы обеспечить согласованность состояния «ровно один раз»[1], каждый соединитель источника приложения должен иметь возможность устанавливать свои позиции чтения в положение, ранее отмеченное контрольной точкой. При установке контрольной точки оператор источника сохраняет свои позиции чтения и восстанавливает эти позиции во время восстановления. Примерами соединителей источника, поддерживающих контрольную точку позиций чтения, являются файловые источники, которые хранят смещения чтения в байтовом потоке файла, или источник Kafka, который хранит смещения чтения в разделах тем, используемых им. Если приложение принимает данные из соединителя, который не может сохранять и сбрасывать позицию чтения, оно может пострадать от потери данных в случае сбоя и предоставить гарантии лишь «не более одного раза».

Комбинация механизма контрольной точки и восстановления Flink и сбрасываемых соединителей источника гарантирует, что приложение не потеряет данные. Однако приложение может выдать результаты дважды, потому что все результаты, которые были созданы после последней успешной контрольной точки (той, к которой приложение возвращается в случае восстановления), будут отправлены снова. Следовательно, сбрасываемых источников и механизма восстановления Flink недостаточно для обеспечения гарантии сквозной согласованности «ровно один раз», даже если состояние приложения согласовано «ровно один раз».

Приложение, целью которого является предоставление сквозной гарантии «ровно один раз», требует специальных соединителей приемника. Есть два метода, которые соединители приемника могут применять в разных ситуациях для достижения гарантии «ровно один раз»: *идемпотентная* запись и *транзакционная* запись.

8.1.1. Идемпотентные записи

Идемпотентная операция (idempotent operation) может выполняться несколько раз, но приведет только к одному изменению. Например, многократная вставка одной и той же пары «ключ–значение» в хеш-карту является идемпотентной операцией, потому что первая операция вставки добавляет значение для ключа в карту, и все последующие вставки не изменят карту, поскольку она уже содержит пару «ключ–значение». С другой стороны,

---

[1] Согласованность состояния «ровно один раз» – это условие для обеспечения *сквозной* согласованности «ровно один раз», но это не одно и то же.

операция добавления не является идемпотентной операцией, потому что добавление элемента несколько раз приводит к нескольким добавлениям. Операции идемпотентной записи интересны для потоковых приложений, поскольку их можно выполнять несколько раз без изменения результатов. Следовательно, они могут до некоторой степени смягчить эффект повторения результатов, присущий механизму контрольных точек Flink.

Следует отметить, что приложение, которое полагается на идемпотентные приемники для достижения ровно однократных результатов, должно гарантировать, что оно переопределяет ранее записанные результаты во время воспроизведения. Например, приложение с приемником, выполняющим вставку-обновление в хранилище «ключ–значение», должно гарантировать, что оно детерминированно вычисляет ключи, которые используются для обновления. Более того, приложения, считывающие данные из системы приемника, могут получить неожиданные результаты во время восстановления приложения. Когда начинается воспроизведение, уже выданные результаты могут быть отменены более ранними результатами. Следовательно, приложение, которое потребляет выходные данные восстанавливающегося приложения, может испытать скачок во времени, например считывать меньший счетчик, чем раньше. Кроме того, общий результат потокового приложения будет в несогласованном состоянии во время воспроизведения, потому что некоторые результаты будут отменены, а другие – нет. После того как воспроизведение завершится и приложение преодолеет точку, в которой оно ранее потерпело сбой, результат снова будет согласованным.

8.1.2. Транзакционные записи

Второй подход к достижению сквозной согласованности «ровно один раз» основан на транзакционной записи. Идея здесь состоит в том, чтобы записывать во внешнюю систему приемника только те результаты, которые были вычислены до последней успешной контрольной точки. Такое поведение гарантирует сквозное выполнение ровно один раз, потому что в случае сбоя приложение сбрасывается до последней контрольной точки, и после этой контрольной точки в систему приемника не поступало никаких результатов. За счет записи данных только после того, как контрольная точка завершена, транзакционный подход не страдает от несогласованности воспроизведения идемпотентных записей. Однако это увеличивает задержку, потому что результаты становятся видимыми только после завершения контрольной точки.

Flink предоставляет два строительных блока для реализации транзакционных соединителей приемника – *журнал с упреждающей записью* (write ahead log, WAL) и *двухфазную фиксацию транзакций* (two phase commit, 2PC). Приемник WAL записывает все данные результатов в состояние приложения и отправляет их системе приемника после получения уведомления о завершении контрольной точки. Поскольку приемник буферизует записи в бэкенде состояния, приемник WAL может использоваться с любой системой приемника. Однако он не может предоставить безукоризненные гарантии «ровно один

раз»[1], увеличивает размер состояния приложения, и системе-приемнику приходится иметь дело с нестандартным шаблоном записи.

Напротив, приемнику 2PC требуется система приемника, которая предлагает поддержку транзакций или предоставляет модули для эмуляции транзакций. Для каждой контрольной точки приемник запускает транзакцию и добавляет все полученные записи к транзакции, записывая их в систему приемника, но не фиксирует ее. Когда он получает уведомление о завершении контрольной точки, он фиксирует транзакцию и материализует записанные результаты. Механизм основан на способности приемника после восстановления зафиксировать транзакцию, которая была открыта до завершения контрольной точки.

Протокол 2PC совмещен с существующим механизмом контрольных точек Flink. Барьеры контрольной точки – это уведомления для начала новой транзакции; уведомления всех операторов об успехе их отдельной контрольной точки – это их голоса за фиксацию, а сообщения JobManager, которые уведомляют об успехе контрольной точки, – это инструкции для фиксации транзакций. В отличие от приемников WAL приемники 2PC могут обеспечить вывод «ровно один раз» в зависимости от потоковой системы и ее реализации. Более того, приемник 2PC непрерывно вносит записи в систему по сравнению с пиковой записью приемника WAL.

В табл. 8.1 показаны гарантии сквозной согласованности для различных типов соединителей источника и приемника, которые могут быть достигнуты в лучшем случае; в зависимости от реализации приемника фактическая согласованность может быть хуже.

Таблица 8.1. Гарантия сквозной согласованности для различных комбинаций источников и приемников

	Несбрасываемый источник	**Сбрасываемый источник**
Любой приемник	Не более одного раза	Хотя бы один раз
Идемпотентный приемник	Не более одного раза	Ровно один раз (временное нарушение целостности во время восстановления)
Приемник WAL	Не более одного раза	Хотя бы один раз
Приемник 2PC	Не более одного раза	Ровно один раз

8.2. Соединители Apache Flink

Apache Flink предоставляет соединители для чтения и записи данных в различные системы хранения. Очереди сообщений и журналы событий, такие как Apache Kafka, Kinesis или RabbitMQ, являются общими источниками входящих потоков данных. В средах с преобладанием пакетной обработки потоки данных также часто загружаются путем мониторинга каталога файловой системы и чтения файлов по мере их появления.

---

[1] Мы обсуждаем гарантии согласованности приемника WAL более подробно в разделе 8.4.2.

На стороне приемника потоки данных часто создаются в очередях сообщений, чтобы сделать события доступными для последующих потоковых приложений, записываются в файловые системы для архивирования или предоставления данных для автономной аналитики или пакетных приложений или вставляются в хранилища пар «ключ–значение» или в реляционную базу данных, такую как Cassandra, ElasticSearch или MySQL, чтобы сделать данные доступными для поиска и запросов или для обслуживания приложений панели мониторинга.

К сожалению, для большинства этих систем хранения нет стандартных интерфейсов, кроме JDBC для реляционных СУБД. Вместо этого каждая система имеет собственную библиотеку соединителей с собственным протоколом. Как следствие, системы обработки, такие как Flink, должны поддерживать несколько специальных соединителей, чтобы иметь возможность считывать события и записывать события в наиболее распространенные очереди сообщений, журналы событий, файловые системы, хранилища пар «ключ–значение» и системы баз данных.

Flink предоставляет соединители для Apache Kafka, Kinesis, RabbitMQ, Apache Nifi, различных файловых систем, Cassandra, ElasticSearch и JDBC. Кроме того, проект Apache Bahir предоставляет дополнительные коннекторы Flink для ActiveMQ, Akka, Flume, Netty и Redis.

Чтобы использовать один из этих соединителей в вашем приложении, необходимо добавить его зависимость в файл сборки вашего проекта. Мы объяснили, как добавить зависимости соединителя в разделе 5.7.

В следующем разделе мы обсудим соединители для Apache Kafka, файловых источников и приемников, а также Apache Cassandra. Это наиболее широко используемые соединители, также они представляют важные типы систем источников и приемников. Вы можете найти дополнительную информацию о других соединителях в документации для Apache Flink или Apache Bahir.

8.2.1. Соединитель источника Apache Kafka

Apache Kafka – это распределенная потоковая платформа. Ее ядро – это распределенная система обмена сообщениями «публикация–подписка», которая широко применяется для приема и распределения потоков событий. Мы кратко объясним основные концепции Kafka, прежде чем углубиться в детали коннектора Kafka Flink.

Kafka организует потоки событий как так называемые *темы*. Тема – это журнал событий, который гарантирует, что события читаются в том же порядке, в котором они были записаны. Чтобы масштабировать запись и чтение из темы, ее можно разделить на разделы, которые распределены по кластеру. Гарантия упорядочения ограничена разделом – Kafka не предоставляет гарантий упорядочивания при чтении из разных разделов. Позиция чтения в разделе Kafka называется *смещением* (offset).

Flink предоставляет соединители источника для всех распространенных версий Kafka. Начиная с Kafka 0.11, API клиентской библиотеки постоянно развивается с добавлением новых функций. Например, в Kafka 0.10 добав-

лена поддержка меток времени записи. Начиная с версии 1.0, API оставался стабильным. Flink предоставляет универсальный соединитель Kafka, который работает для всех версий Kafka, начиная с 0.11. Flink также имеет специальные соединители для версий Kafka 0.8, 0.9, 0.10 и 0.11. В оставшейся части этого раздела мы сосредоточимся на универсальном соединителе, а для соединителей, зависящих от версии, мы отсылаем вас к документации Flink.

Зависимость для универсального коннектора Flink Kafka добавляется в проект Maven, как показано ниже:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.12</artifactId>
    <version>1.7.1</version>
</dependency>
```

Соединитель Flink для Kafka принимает потоки событий параллельно. Каждая параллельная задача источника может считывать данные из одного или нескольких разделов. Задача отслеживает для каждого раздела его текущее смещение чтения и включает его в данные контрольной точки. При восстановлении после сбоя смещения восстанавливаются, и исходный экземпляр продолжает чтение со смещения контрольной точки. Соединитель Flink для Kafka не полагается на собственный механизм отслеживания смещения Kafka, который основан на так называемых группах потребителей. На рис. 8.1 показано назначение разделов экземплярам источника.

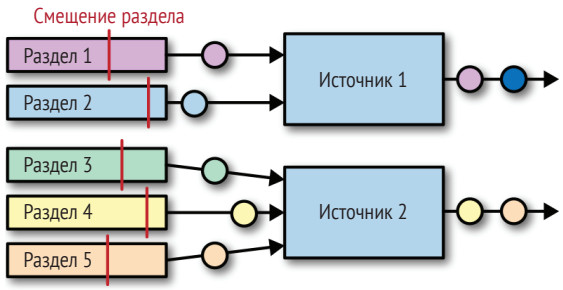

Рис. 8.1 ❖ Чтение смещения разделов темы Kafka

Соединитель источника Kafka создается, как показано в примере 8.1.

Пример 8.1 ❖ Создание источника Flink-Kafka

```
val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "test")

val stream: DataStream[String] = env.addSource(
  new FlinkKafkaConsumer[String](
    "topic",
    new SimpleStringSchema(),
    properties))
```

Конструктор принимает три аргумента. Первый аргумент определяет темы для чтения. Это может быть отдельная тема, список тем или регулярное выражение, которое охватывает все темы для чтения. При чтении из нескольких тем соединитель Kafka обрабатывает все разделы всех тем одинаково и мультиплексирует их события в один поток.

Второй аргумент – это `DeserializationSchema` или `KeyedDeserializationSchema`. Сообщения Kafka хранятся в виде необработанных байтовых сообщений и должны быть десериализованы в объекты Java или Scala. `SimpleStringSchema`, который используется в примере 8.1, представляет собой встроенный интерфейс `DeserializationSchema`, который просто десериализует байтовый массив в строку. Кроме того, Flink предоставляет реализации для кодировок Apache Avro и текстовых JSON. `DeserializationSchema` и `KeyedDeserializationSchema` – это общедоступные интерфейсы, поэтому вы всегда можете реализовать настраиваемую логику десериализации.

Третий параметр – это объект `Properties` для настройки клиента Kafka, который используется внутри для подключения к Kafka и чтения из него. Минимальная конфигурация свойств состоит из двух записей: `"bootstrap.servers"` и `"group.id"`. Обратитесь к документации Kafka для ознакомления с дополнительными параметрами конфигурации.

Чтобы извлечь метки времени событий и создать водяные знаки, вы можете предоставить `AssignerWithPeriodicWatermark` или `AssignerWithPunctuatedWatermark` потребителю Kafka, вызвав назначитель `FlinkKafkaConsumer.assignTimestampsAndWatermark()`[1]. Назначитель применяется к каждому разделу, чтобы использовать гарантии упорядочения по разделам, а исходный экземпляр объединяет водяные знаки раздела в соответствии с протоколом распространения водяных знаков (см. раздел 3.3.3).

Обратите внимание, что водяные знаки исходного экземпляра не могут работать, если раздел становится неактивным и не предоставляет сообщения. Как следствие, один неактивный раздел вызывает остановку всего приложения, потому что водяные знаки приложения не работают.

Начиная с версии 0.10.0, Kafka поддерживает метки времени сообщений. При чтении из Kafka версии 0.10 или более поздней потребитель автоматически извлекает метку времени сообщения как метку времени события, если приложение работает в режиме времени события. В этом случае вам попрежнему необходимо сгенерировать водяные знаки и следует применить `AssignerWithPeriodicWatermark` или `AssignerWithPunctuatedWatermark`, который пересылает ранее назначенную метку времени Kafka.

Есть еще несколько заслуживающих внимания опций конфигурации. Вы можете настроить начальную позицию, с которой будут считываться разделы темы. Допустимые варианты:

- последняя позиция чтения, известная Kafka для группы потребителей, которая была настроена с помощью параметра `group.id`. Это поведение по умолчанию:

[1] См. главу 6 для получения подробной информации об интерфейсах назначителя временных меток.

```
FlinkKafkaConsumer.setStartFromGroupOffsets()
```

- самое раннее смещение каждого отдельного раздела:

```
FlinkKafkaConsumer.setStartFromEarliest()
```

- последнее смещение каждого отдельного раздела:

```
FlinkKafkaConsumer.setStartFromLatest()
```

- все записи с меткой времени больше заданной (требуется Kafka 0.10.x или новее):

```
FlinkKafkaConsumer.setStartFromTimestamp (long)
```

- определенные позиции чтения для всех разделов, предоставляемые объектом `Map`:

```
FlinkKafkaConsumer.setStartFromSpecificOffsets(Map)
```

Обратите внимание, что эта конфигурация влияет только на начальные позиции чтения. В случае восстановления или при запуске из точки сохранения приложение начнет чтение со смещений, сохраненных в контрольной точке или точке сохранения.

Потребитель связки Flink–Kafka может быть настроен на автоматическое обнаружение новых тем, соответствующих регулярному выражению, или новых разделов, которые были добавлены в тему. Эти функции отключены по умолчанию и могут быть включены путем добавления параметра `flink.partition-discovery.interval-millis` с неотрицательным значением к объекту `Properties`.

8.2.2. Соединитель приемника Apache Kafka

Flink предоставляет соединители для приемников для всех версий Kafka, начиная с 0.8. Благодаря Kafka 0.11 API клиентской библиотеки был усовершенствован – туда добавлены новые функции, такие как поддержка меток времени записи в Kafka 0.10 и транзакционная запись в Kafka 0.11. Начиная с версии 1.0, API оставался стабильным. Flink предоставляет универсальный коннектор Kafka, который работает для всех версий Kafka, начиная с 0.11. Flink также имеет соединители для конкретных версий Kafka 0.8, 0.9, 0.10 и 0.11. В оставшейся части этого раздела мы сосредоточимся на универсальном соединителе и отсылаем вас к документации Flink, чтобы узнать о соединителях для конкретных версий. Зависимость для универсального соединителя Kafka Flink добавляется в проект Maven, как показано ниже:

```
<dependency>
   <groupId>org.apache.flink</groupId>
   <artifactId>flink-connector-kafka_2.12</artifactId>
   <version>1.7.1</version>
</dependency>
```

Приемник Kafka добавляется в приложение DataStream, как показано в примере 8.2.

Пример 8.2 ❖ Создание приемника Flink-Kafka

```
val stream: DataStream[String] = ...

val myProducer = new FlinkKafkaProducer[String](
  "localhost:9092", // Список брокеров.
  "topic", // Тема назначения.
  new SimpleStringSchema) // Схема сериализации.

stream.addSink(myProducer)
```

Конструктор, используемый в примере 8.2, получает три параметра. Первый параметр – это разделенная запятыми строка адресов брокера Kafka. Второй – это имя темы, в которую записываются данные, а последний – это `SerializationSchema`, которая преобразует входные типы приемника (`String` в примере 8.2) в байтовый массив. `SerializationSchema` – это аналог `DeserializationSchema`, который мы обсуждали в разделе источников Kafka.

`FlinkKafkaProducer` предоставляет больше конструкторов с различными комбинациями аргументов, а именно:

- подобно соединителю источника Kafka, вы можете передать объект `Properties` для предоставления настраиваемых параметров внутреннему клиенту Kafka. При использовании свойств список брокеров должен быть указан как свойство `bootstrap.servers`. Ознакомьтесь с документацией Kafka для получения полного списка параметров;
- вы можете указать `FlinkKafkaPartitioner`, чтобы контролировать, как записи отображаются на разделы Kafka. Мы обсудим эту функцию более подробно позже в этом разделе;
- вместо использования `SerializationSchema` для преобразования записей в байтовые массивы вы также можете указать `KeyedSerializationSchema`, который сериализует запись в два байтовых массива – один для ключа, а другой для значения сообщения Kafka. Более того, `KeyedSerializationSchema` также предоставляет больше функций, специфичных для Kafka, таких как переопределение целевой темы для записи в несколько тем.

8.2.2.1. Гарантия «хотя бы один раз» для приемника Kafka

Гарантия согласованности приемника Kafka зависит от его конфигурации. На приемник Kafka предоставляется гарантия «хотя бы один раз» при следующих условиях:

- механизм контрольной точки Flink включен, и все источники приложения могут быть сброшены;
- соединитель приемника выдает исключение, если запись не удалась, что приводит к сбою и восстановлению приложения. Это поведение по умолчанию. Внутренний клиент Kafka может быть настроен на повторные попытки записи до объявления их неудачными, если для свойства

retries установлено значение больше нуля (по умолчанию). Вы также можете настроить приемник на регистрацию только сбоев записи, вызвав setLogFailuresOnly(true) для объекта приемника. Обратите внимание, что это аннулирует любые гарантии вывода приложения;

- соединитель приемника ожидает, пока Kafka подтвердит записи «на лету», прежде чем завершить свою контрольную точку. Это поведение по умолчанию. Вызвав метод setFlushOnCheckpoint(false) для объекта-приемника, вы можете отключить это ожидание. Однако это также отключит любые гарантии вывода.

8.2.2.2. Гарантия «ровно один раз» для приемника Kafka

В Kafka 0.11 появилась поддержка транзакционной записи. Благодаря этой функции приемник Kafka в среде Flink также может обеспечивать гарантированный вывод «ровно один раз», при условии, что приемник и Kafka правильно настроены. Опять же, приложение Flink должно поддерживать контрольные точки и использовать данные из сбрасываемых источников. Кроме того, FlinkKafkaProducer предоставляет конструктор с параметром Semantic, который контролирует гарантии согласованности, предоставляемые приемником. Возможные значения согласованности:

- Semantic.NONE, который не дает никаких гарантий, – записи могут быть потеряны или записаны несколько раз;
- Semantic.AT_LEAST_ONCE, который гарантирует, что запись не будет потеряна, но может быть дублирована. Это значение по умолчанию;
- Semantic.EXACTLY_ONCE, который основан на механизме транзакций Kafka для записи ровно один раз.

Есть несколько вещей, которые следует учитывать при запуске приложения Flink с приемником Kafka, работающим в режиме «ровно один раз», и это помогает примерно понять, как Kafka обрабатывает транзакции. Вкратце транзакции Kafka работают, добавляя все сообщения в журнал раздела и помечая сообщения открытых транзакций как незафиксированные. Как только транзакция зафиксирована, маркеры меняются на зафиксированные. Для потребителя, который читает из темы, можно настроить уровень изоляции (через свойство isolation.level), чтобы объявить, может ли он читать незафиксированные сообщения (read_uncommitted по умолчанию) или нет (read_committed). Если потребителю установлен уровень read_committed, он прекращает чтение из раздела, когда обнаруживает незафиксированное сообщение, и возобновляет работу, когда сообщение зафиксировано. Следовательно, открытые транзакции могут блокировать чтение раздела потребителями и вызывать значительные задержки. Kafka защищает от этого, отклоняя и закрывая транзакции по истечении интервала тайм-аута, который настраивается с помощью свойства transaction.timeout.ms.

В контексте приемника Kafka это важно, потому что просрочка транзакций, например из-за длительных циклов восстановления, приводит к потере данных. Итак, очень важно правильно настроить свойство тайм-аута транз-

акции. По умолчанию приемник Kafka устанавливает для параметра `transaction.timeout.ms` значение одного часа, следовательно, вам вероятно потребуется настроить свойство `transaction.max.timeout.ms` вашего экземпляра Kafka, которое по умолчанию установлено на 15 мин. Более того, видимость зафиксированных сообщений зависит от интервала между контрольными точками приложения Flink. Обратитесь к документации Flink, чтобы узнать о некоторых других важных ситуациях при использовании согласованности «ровно один раз».

Проверьте конфигурацию вашего кластера Kafka

Конфигурация кластера Kafka по умолчанию может привести к потере данных даже после подтверждения записи. Вам следует внимательно проверить конфигурацию вашего экземпляра Kafka, обращая особое внимание на следующие параметры:

- `acks`;
- `log.flush.interval.messages`;
- `log.flush.interval.ms`;
- `log.flush.*`.

Мы отсылаем вас к документации Kafka за подробной информацией о параметрах конфигурации и рекомендациями по настройке подходящей конфигурации.

8.2.2.3. Пользовательское разбиение на разделы и запись меток времени сообщений

При написании сообщений в тему Kafka задача приемника Kafka может выбирать, в какой раздел темы писать. В некоторых конструкторах приемника Kafka можно определить `FlinkKafkaPartitioner`. Если не указано иное, разделитель по умолчанию сопоставляет каждую задачу приемника с одним разделом Kafka – все записи, созданные одной задачей приемника, записываются в один и тот же раздел, и один раздел может содержать записи нескольких задач приемника, если задач больше, чем разделов. Если количество разделов превышает количество подзадач, конфигурация по умолчанию приводит к появлению пустых разделов, что может вызвать проблемы для приложений Flink, использующих раздел в режиме времени события.

Используя настраиваемый `FlinkKafkaPartitioner`, вы можете управлять пересылкой записей в разделы темы. Например, вы можете создать разделитель на основе ключевого атрибута записей или циклический разделитель для равномерного распределения. Существует также возможность делегировать разделение Kafka на основе ключа сообщения. Для этого нужно использовать `KeyedSerializationSchema`, чтобы извлечь ключи сообщений и назначить параметру `FlinkKafkaPartitioner` нулевое значение, чтобы отключить разделитель по умолчанию.

Наконец, приемник Kafka может быть настроен для записи меток времени событий, начиная с Kafka 0.10. Запись метки времени события в Kafka включается путем вызова метода `setWriteTimestampToKafka(true)` для объекта-приемника.

8.2.3. Соединитель файлового источника

Файловые системы обычно используются для экономичного хранения больших объемов данных. В архитектурах больших данных они часто служат источником и приемником данных для приложений пакетной обработки. В сочетании с расширенными форматами файлов, такими как Apache Parquet или Apache ORC, файловые системы могут эффективно обслуживать механизмы аналитических запросов, такие как Apache Hive, Apache Impala или Presto. Поэтому файловые системы обычно используются для «соединения» потоковых и пакетных приложений.

Apache Flink имеет сбрасываемый соединитель источника для получения данных из файлов в виде потоков. Файловый источник является частью модуля `flink-streaming-java`. Следовательно, вам не нужно добавлять какие-либо другие зависимости для использования этой функции. Flink поддерживает различные типы файловых систем, такие как локальные файловые системы (включая локально смонтированные общие ресурсы NFS или SAN, Hadoop HDFS, Amazon S3 и OpenStack Swift FS). Обратитесь к разделу 9.4 чтобы узнать, как настроить файловые системы в Flink. В примере 8.3 показано, как принимать поток, считывая текстовые файлы построчно.

Пример 8.3 ❖ Создание файлового источника

```
val lineReader = new TextInputFormat(null)

val lineStream: DataStream[String] = env.readFile[String](
  lineReader,               // FileInputFormat.
  "hdfs:///path/to/my/data", // Путь для чтения.
  FileProcessingMode
    .PROCESS_CONTINUOUSLY,   // Режим обработки.
  30000L)
```

Аргументы метода `StreamExecutionEnvironment.readFile()`:

- `FileInputFormat`, который отвечает за чтение содержимого файлов. Мы обсудим детали этого интерфейса ниже в этом разделе. Нулевой параметр `TextInputFormat` в примере 8.3 определяет путь, который задается отдельно;
- путь, который следует прочитать. Если путь относится к файлу, читается отдельный файл. Если путь ведет к каталогу, `FileInputFormat` сканирует каталог на наличие файлов для чтения;
- режим, в котором следует читать путь. Режим может быть `PROCESS_ONCE` или `PROCESS_CONTINUOUSLY`. В режиме `PROCESS_ONCE` путь чтения сканируется один раз при запуске задания, и считываются все соответствующие файлы. В `PROCESS_CONTINUOUSLY` путь периодически сканируется (после начального сканирования), а новые и измененные файлы постоянно читаются;
- интервал в миллисекундах, в течение которого путь периодически сканируется. Параметр игнорируется в режиме `PROCESS_ONCE`.

`FileInputFormat` – это специальный `InputFormat` для чтения файлов из файловой системы[1]. `FileInputFormat` считывает файлы в два этапа. Сначала он сканирует путь к файловой системе и создает так называемые *разделения ввода* для всех совпадающих файлов. Разделение ввода определяет диапазон в файле, обычно через начальное смещение и длину. После разделения большого файла на несколько частей эти части можно распределить между несколькими задачами чтения для параллельного чтения файла. В зависимости от кодировки файла может потребоваться создание только одного раздела для чтения файла целиком. Второй шаг `FileInputFormat` – получить разделение ввода, прочитать диапазон файлов, который определяется разделением, и вернуть все соответствующие записи.

`FileInputFormat`, используемый в приложении DataStream, также должен реализовывать интерфейс `CheckpointableInputFormat`, который определяет методы для контрольной точки и сбрасывает текущую позицию чтения `InputFormat` в разделенном файле. Соединитель файлового источника обеспечивает лишь гарантию «хотя бы один раз» при включении контрольной точки, если `FileInputFormat` не реализует интерфейс `CheckpointableInputFormat`, потому что формат ввода начнет чтение с начала разделения, которое было обработано, когда была установлена последняя завершенная контрольная точка.

В версии 1.7 Flink предоставляет несколько классов, которые расширяют `FileInputFormat` и реализуют `CheckpointableInputFormat`. `TextInputFormat` считывает текстовые файлы построчно (разделенные символами новой строки), подклассы `CsvInputFormat` читают файлы со значениями, разделенными запятыми, а `AvroInputFormat` читает файлы с записями в кодировке Avro.

В режиме `PROCESS_CONTINUOUSLY` соединитель файлового источника определяет новые файлы на основе их отметки времени модификации. Это означает, что файл полностью повторно обрабатывается, если изменилась метка времени его модификации. Сюда входят модификации, связанные с добавлением записей. Следовательно, распространенный метод непрерывного приема файлов – записать их во временный каталог и атомарно переместить в отслеживаемый каталог после их завершения. Когда файл полностью загружен и контрольная точка завершена, его можно удалить из каталога. Мониторинг загруженных файлов путем отслеживания метки времени изменения также имеет значение, если вы читаете из распределенных файловых хранилищ с фиксацией списка операций, таких как S3. Поскольку файлы могут отображаться не в порядке их отметок времени модификации, они могут игнорироваться соединителем файлового источника.

Обратите внимание, что в режиме `PROCESS_ONCE` контрольные точки не выставляются после сканирования пути файловой системы и создания всех разделений.

Если вы хотите использовать соединитель файлового источника в приложении с привязкой к времени событий, вы должны знать, что создание водяных знаков может оказаться сложной задачей, поскольку входные разделы генерируются в одном процессе и циклически распределяются среди

---

[1] `InputFormat` – это интерфейс Flink для определения источников данных в DataSet API.

всех параллельных считывателей, которые обрабатывают их в порядке метки времени модификации файла. Чтобы создать удовлетворительные водяные знаки, вам необходимо определить наименьшую метку времени записи, включающуюся в разделение, которое позже обрабатывается задачей.

8.2.4. Соединитель файлового приемника

Запись потока в файлы является обычным требованием, например, для подготовки данных с низкой задержкой для автономного анализа «на лету». Поскольку большинство приложений могут читать файлы только после того, как они завершены, а потоковые приложения работают в течение длительного времени, соединители потоковых приемников обычно разбивают свой вывод на несколько файлов. Более того, записи обычно организуются в так называемые *корзины* (bucket), чтобы приложения-потребители могли лучше следить за тем, какие данные следует читать.

Как и соединитель файлового источника, соединитель Flink `StreamingFileSink` содержится в модуле `flink-streaming-java`. Следовательно, вам не нужно добавлять зависимость к вашему файлу сборки, чтобы использовать его.

`StreamingFileSink` обеспечивает сквозные гарантии «ровно один раз» для приложения при условии, что приложение настроено на использование контрольных точек «ровно один раз» и все его источники сбрасываются в случае сбоя. Мы обсудим механизм восстановления более подробно дальше в этом разделе. Пример 8.4 показывает, как создать `StreamingFileSink` с минимальной конфигурацией и добавить его в поток.

Пример 8.4 ❖ Создание `StreamingFileSink` в режиме построчного кодирования

```
val input: DataStream[String] = …
val sink: StreamingFileSink[String] = StreamingFileSink
  .forRowFormat(
    new Path("/base/path"),
    new SimpleStringEncoder[String]("UTF-8"))
  .build()
input.addSink(sink)
```

Когда `StreamingFileSink` получает запись, она назначается корзине. Корзина – это подкаталог базового пути, который настраивается с помощью построителя `StreamingFileSink` – `"/base/path"` в примере 8.4.

Корзину выбирает `BucketAssigner`, который является публичным интерфейсом и возвращает для каждой записи `BucketId`, определяющий каталог, в который будет помещена запись. `BucketAssigner` можно настроить в построителе с помощью метода `withBucketAssigner()`. Если `BucketAssigner` не указан явно, используется `DateTimeBucketAssigner`, который назначает записи почасовым сегментам на основе времени обработки, когда они записываются.

Каждый каталог корзины содержит несколько файлов частей, которые одновременно записываются несколькими параллельными экземплярами `StreamingFileSink`. Более того, каждый параллельный экземпляр разбивает

свой вывод на несколько файлов частей. Путь к файлу части имеет следующий формат:

```
[base-path]/[bucket-path]/part-[task-idx]-[id]
```

Например, для базового пути `"/johndoe/demo"` и префикса части `"part"` путь `"/johndoe/demo/2018-07-22-17/part-4-8"` указывает на восемь файлов, которые были записаны пятой (отсчет с 0) задачей приемника в корзину `"2018-07-22-17"`, – корзина от 17:00 часов 22 июля 2018 г.

Идентификаторы зафиксированных файлов могут не быть последовательными

Непоследовательность идентификаторов файлов – последнего числа в имени зафиксированного файла – не указывает на потерю данных. `StreamingFileSink` просто увеличивает идентификаторы файлов. При удалении ожидающих файлов он не использует их идентификаторы повторно.

`RollingPolicy` определяет, когда задача создает новый файл части. Вы можете настроить `RollingPolicy` с помощью метода `withRollingPolicy()` в построителе. По умолчанию `StreamingFileSink` использует `DefaultRollingPolicy`, который настроен на завершение файлов частей, когда они превышают 128 Мб или старше 60 с. Вы также можете настроить интервал бездействия, по истечении которого файл части будет завершен.

`StreamingFileSink` поддерживает два режима записи данных в файлы частей: построчное кодирование и массовое кодирование. В режиме построчного кодирования каждая запись индивидуально кодируется и добавляется к файлу части. При массовом кодировании записи собираются и записываются партиями. Apache Parquet, который организует и сжимает записи в формате столбцов, представляет собой формат файла, который требует массового кодирования.

В примере 8.4 создается `StreamingFileSink` с кодировкой строк при помощи `Encoder`, который вносит отдельные записи в файл части. В примере 8.4 мы используем `SimpleStringEncoder`, вызывающий метод `toString()` записи и записывающий строковое представление записи в файл. `Encoder` – это простой интерфейс с одним легко реализуемым методом.

`StreamingFileSink` с массовым кодированием создается, как показано в примере 8.5.

Пример 8.5 ❖ Создание `StreamingFileSink` в режиме массового кодирования

```
val input: DataStream[String] = …
val sink: StreamingFileSink[String] = StreamingFileSink
  .forBulkFormat(
    new Path("/base/path"),
    ParquetAvroWriters.forSpecificRecord(classOf[AvroPojo]))
    .build()
input.addSink(sink)
```

Для `StreamingFileSink` в режиме массового кодирования требуется `BulkWriter.Factory`. В примере 8.5 мы используем Parquet для файлов Avro. Обратите внимание, что Parquet реализован в модуле `flink-parquet`, который необхо-

димо добавить в качестве зависимости. `BulkWriter.Factory` – это интерфейс, который можно реализовать для пользовательских форматов файлов, таких как Apache Orc.

`StreamingFileSink` в режиме массового кодирования не может выбрать `RollingPolicy`. Форматы массового кодирования можно комбинировать только с `OnCheckpointRollingPolicy`, который завершает файлы частей на каждой контрольной точке.

`StreamingFileSink` обеспечивает гарантию вывода «ровно один раз». Приемник достигает этого с помощью протокола фиксации, который перемещает файлы через разные этапы – в обработке, в ожидании и завершении – и основан на механизме контрольных точек Flink. Пока приемник записывает данные в файл, этот файл находится в состоянии обработки. Когда `RollingPolicy` решает свернуть файл, он закрывается и переводится в состояние ожидания путем его переименования. Ожидающие файлы переводятся в завершенное состояние (снова путем переименования), когда завершается следующая контрольная точка.

Ожидающие файлы, возможно, никогда не будут зафиксированы

В некоторых ситуациях ожидающий файл никогда не фиксируется. `StreamingFileSink` гарантирует, что это не приведет к потере данных. Однако эти файлы не очищаются автоматически.

Перед тем как вручную удалить ожидающий файл, вам необходимо проверить, задерживается ли он или вот-вот будет зафиксирован. Как только вы найдете зафиксированный файл с тем же индексом задачи и более высоким идентификатором, вы можете безопасно удалить ожидающий файл.

В случае сбоя задача приемника должна сбросить свой текущий исполняемый файл на его смещение записи в последней успешной контрольной точке. Это делается путем закрытия текущего незавершенного файла и удаления недопустимой части в конце файла, например с помощью операции усечения файловой системы.

StreamingFileSink требует включения контрольной точки

`StreamingFileSink` никогда не будет перемещать файлы из ожидающего в завершенное состояние, если приложение не использует контрольную точку.

8.2.5. Соединитель приемника Apache Cassandra

Apache Cassandra – это популярная масштабируемая и высокодоступная система управления колоночными (столбцовыми) базами данных. Cassandra моделирует наборы данных как таблицы строк, которые состоят из нескольких типизированных столбцов. Один или несколько столбцов должны быть определены как (составные) первичные ключи. Каждую строку можно однозначно идентифицировать по ее первичному ключу. Среди других API Cassandra поддерживает язык запросов Cassandra Query Language (CQL), похожий на SQL язык для чтения и записи, а также создания, изменения и уда-

ления объектов базы данных, таких как пространства ключей и таблицы.

Flink предоставляет соединитель для записи потоков данных в Cassandra. Модель данных Cassandra основана на первичных ключах, и все записи в Cassandra происходят с семантикой *upsert* (обновление/вставка). В сочетании с механизмом контрольной точки «ровно один раз», сбрасываемыми источниками и детерминированной логикой приложения upsert-записи в конечном итоге обеспечивают выход «ровно один раз». Выходные данные являются согласованными с гарантией «ровно один раз» только в конечном итоге, потому что во время восстановления результаты сбрасываются до предыдущей версии, а это означает, что потребители могут читать более старые результаты, чем полученные недавно. Кроме того, версии значений для нескольких ключей могут быть несинхронизированы.

Чтобы предотвратить временные несоответствия во время восстановления и обеспечить гарантию вывода «ровно один раз» для приложений с недетерминированной логикой приложения, соединитель Flink Cassandra может быть настроен для использования режима WAL. Мы обсудим более подробно режим WAL ниже в этом разделе. Следующий код показывает зависимость, которую необходимо добавить в файл сборки вашего приложения, чтобы использовать соединитель приемника Cassandra:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-cassandra_2.12</artifactId>
  <version>1.7.1</version>
</dependency>
```

Чтобы проиллюстрировать использование соединителя приемника Cassandra, мы возьмем простой пример таблицы Cassandra, которая содержит данные о показаниях датчиков и состоит из двух столбцов: `sensorId` и `temperature`. Операторы CQL в примере 8.6 создают пространство ключей `example` и таблицу `sensors` в этом пространстве ключей.

Пример 8.6 ❖ Определение примера таблицы Cassandra

```
CREATE KEYSPACE IF NOT EXISTS example
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'};

CREATE TABLE IF NOT EXISTS example.sensors (
  sensorId VARCHAR,
  temperature FLOAT,
  PRIMARY KEY(sensorId)
);
```

Flink предоставляет различные реализации приемника для записи потоков данных разных типов в Cassandra. Кортежи Java Flink и тип `Row`, а также встроенные в Scala кортежи и case-классы обрабатываются иначе, чем типы POJO, определяемые пользователем. Обсудим оба случая отдельно. В примере 8.7 показано, как создать приемник, который записывает поток данных кортежей, case-классов или строк в таблицу Cassandra. В этом примере `DataStream[(String, Float)]` записывается в таблицу `sensors`.

Пример 8.7 ❖ Создание приемника Cassandra для кортежей

```
val readings: DataStream[(String, Float)] = ???

val sinkBuilder: CassandraSinkBuilder[(String, Float)] =
  CassandraSink.addSink(readings)
sinkBuilder
  .setHost("localhost")
  .setQuery(
    "INSERT INTO example.sensors(sensorId, temperature) VALUES (?, ?);")
  .build()
```

Приемники Cassandra создаются и настраиваются с помощью построителя, который получается путем вызова метода `CassandraSink.addSink()` с объектом `DataStream`, который должен быть сгенерирован. Метод возвращает правильный построитель для типа данных `DataStream`. В примере 8.7 он возвращает построитель приемника Cassandra, который обрабатывает кортежи Scala.

Построители приемников Cassandra для кортежей, case-классов и строк требуют спецификации запроса CQL INSERT[1]. Запрос настраивается с помощью метода `CassandraSinkBuilder.setQuery()`. Во время выполнения приемник регистрирует запрос как подготовленный оператор и преобразует поля кортежей, case-классов или строк в параметры для подготовленного оператора. Поля сопоставляются с параметрами в зависимости от их положения; первое значение преобразуется в первый параметр и т. д.

Поскольку поля POJO не имеют естественного порядка, их нужно обрабатывать по-другому. В примере 8.8 показано, как настроить приемник Cassandra для POJO типа `SensorReading`.

Пример 8.8 ❖ Создание приемника Cassandra для POJO

```
val readings: DataStream[SensorReading] = ???

CassandraSink.addSink(readings)
  .setHost("localhost")
  .build()
```

Как вы можете видеть в примере 8.8, мы не указываем запрос INSERT. Вместо этого POJO передаются в Cassandra Object Mapper, который автоматически сопоставляет поля POJO с полями таблицы Cassandra. Для того чтобы это работало, класс POJO и его поля должны быть снабжены аннотациями Cassandra и предоставлять сеттеры и геттеры для всех полей, как показано в примере 8.9. Flink требует конструктор по умолчанию, как упоминалось в разделе 5.4.1 при обсуждении поддерживаемых типов данных.

Пример 8.9 ❖ Класс POJO с аннотациями Cassandra Object Mapper

```
@Table(keyspace = "example", name = "sensors")
class SensorReadings(
  @Column(name = "sensorId") var id: String,
```

[1] В отличие от операторов SQL INSERT операторы CQL INSERT ведут себя как запросы upsert – они заменяют существующие строки с тем же первичным ключом.

```
@Column(name = "temperature") var temp: Float) {

def this() = {
    this("", 0.0)
}
  def setId(id: String): Unit = this.id = id
  def getId: String = id
  def setTemp(temp: Float): Unit = this.temp = temp
  def getTemp: Float = temp
}
```

В дополнение к параметрам конфигурации, показанным в примерах 8.7 и 8.8, построитель приемника Cassandra предоставляет еще несколько методов для настройки соединителя приемника:

- `setClusterBuilder(ClusterBuilder)`: `ClusterBuilder` создает `Cluster`, который управляет подключением к Cassandra. Среди прочего, он может настроить имена хостов и порты одной или нескольких точек контакта; определить политики балансировки нагрузки, повторных попыток и повторного подключения; и предоставить учетные данные для доступа;
- `setHost(String, [Int])`: этот метод представляет собой ярлык для простого построителя кластеров, настроенного с использованием имени хоста и порта единственной точки контакта. Если порт не настроен, по умолчанию используется порт 9042;
- `setQuery(String)`: указывает запрос CQL INSERT для записи кортежей, case-классов или строк в Cassandra. Запрос не должен быть настроен на выдачу POJO;
- `setMapperOptions(MapperOptions)`: предоставляет параметры для Object Mapper Cassandra, такие как конфигурации для согласованности, времени жизни (TTL) и обработки пустых полей. Параметры игнорируются, если приемник выдает кортежи, case-классы или строки;
- `enableWriteAheadLog([CheckpointCommitter])`: позволяет WAL предоставлять гарантии вывода «ровно один раз» в случае недетерминированной логики приложения. `CheckpointCommitter` используется для хранения информации о выполненных контрольных точках во внешнем хранилище данных. Если `CheckpointCommitter` не настроен, информация записывается в конкретную таблицу Cassandra.

Соединитель приемника Cassandra с WAL реализован на основе оператора Flink `GenericWriteAheadSink`. В разделе 8.4.2 более подробно описано, как работает этот оператор, включая роль `CheckpointCommitter`, и какие гарантии согласованности он обеспечивает.

8.3. Реализация пользовательской исходной функции

API DataStream предоставляет два интерфейса для реализации исходных соединителей вместе с соответствующими абстрактными классами `RichFunction`:

- `SourceFunction` и `RichSourceFunction` могут использоваться для определения непараллельных соединителей источника – источников, которые запускаются с одной задачей;
- `ParallelSourceFunction` и `RichParallelSourceFunction` могут использоваться для определения соединителей источника, которые работают с несколькими параллельными экземплярами задач.

За исключением непараллельного и параллельного, оба интерфейса идентичны. Как и в случае с богатыми вариантами функций обработки[1], подклассы `RichSourceFunction` и `RichParallelSourceFunction` могут переопределять методы `open()` и `close()` и получать доступ к `RuntimeContext`, который, помимо прочего, предоставляет количество экземпляров параллельных задач и индекс текущего экземпляра.

`SourceFunction` и `ParallelSourceFunction` определяют два метода:

1) `void run(SourceContext<T> ctx);`
2) `void cancel()`.

Метод `run()` выполняет фактическую работу по чтению или получению записей и загрузке их в приложение Flink. В зависимости от системы, из которой получены данные, данные могут быть отправлены или извлечены. Метод `run()` вызывается Flink один раз и запускается в выделенном исходном потоке, обычно считывая или получая данные и создавая записи в бесконечном цикле (бесконечный поток). Задача может быть явно отменена в какой-то момент времени или завершена в случае конечного потока, когда ввод полностью израсходован.

Метод `cancel()` вызывается Flink, когда приложение отменяется и закрывается. Чтобы выполнить плавное завершение работы, метод `run()`, который выполняется в отдельном потоке, должен завершаться, как только вызывается метод `cancel()`. В примере 8.10 показана простая функция источника, которая считает от 0 до `Long.MaxValue`.

Пример 8.10 ❖ `SourceFunction`, которая считает до Long.MaxValue

```
class CountSource extends SourceFunction[Long] {
  var isRunning: Boolean = true

  override def run(ctx: SourceFunction.SourceContext[Long]) = {

    var cnt: Long = -1
    while (isRunning && cnt < Long.MaxValue) {
      cnt += 1
      ctx.collect(cnt)
    }
  }

  override def cancel() = isRunning = false
}
```

[1] Богатые функции обсуждались в главе 5.

8.3.1. Сбрасываемые функции источника

Ранее в этой главе мы говорили, что Flink может обеспечить удовлетворительные гарантии согласованности только для приложений, использующих соединители источника, которые могут воспроизводить свои выходные данные. Функция источника может воспроизвести свой вывод, если внешняя система, предоставляющая данные, предоставляет API для получения и сброса смещения позиции чтения. Примерами таких систем являются файловые системы, которые предоставляют смещение файлового потока и метод поиска для перемещения файлового потока в определенную позицию, или Apache Kafka, который предоставляет смещения для каждого раздела темы и может устанавливать позицию чтения раздела. Контрпример – это соединитель источника, который считывает данные из сетевого сокета, немедленно отбрасывающего доставленные данные.

Функция источника, которая поддерживает воспроизведение вывода, должна быть интегрирована с механизмом контрольной точки Flink и сохранять все текущие позиции чтения при установке контрольной точки. Когда приложение запускается из точки сохранения или восстанавливается после сбоя, смещения чтения извлекаются из последней контрольной точки или точки сохранения. Если приложение запускается без существующего состояния, смещения чтения должны быть установлены на значение по умолчанию. Сбрасываемая исходная функция должна реализовывать интерфейс `CheckpointedFunction` и сохранять смещения чтения и всю связанную метаинформацию, такую как пути к файлам или идентификатор раздела, в списочном состоянии операторов или каталожном состоянии в зависимости от того, как смещения должны распределяться на параллельные экземпляры задачи в случае масштабируемого приложения (см. раздел 3.4.4 для получения подробной информации о распределении списочного состояния операторов и каталожного состояния).

Кроме того, важно убедиться, что метод `SourceFunction.run()`, который выполняется в отдельном потоке, не увеличивает смещение чтения и не передает данные, пока выполняется контрольная точка; другими словами, пока вызывается метод `CheckpointedFunction.snapshotState()`. Это делается путем защиты кода в `run()`, который продвигает позицию чтения и генерирует записи в блоке. В свою очередь блок синхронизируется с объектом блокировки, получаемым из метода `SourceContext.getCheckpointLock()`. В примере 8.11 производится сбрасывание `CountSource` из примера 8.10.

Пример 8.11 ❖ Сбрасываемая функция `SourceFunction`

```
class ResettableCountSource
    extends SourceFunction[Long] with CheckpointedFunction {

  var isRunning: Boolean = true
  var cnt: Long = _
  var offsetState: ListState[Long] = _

  override def run(ctx: SourceFunction.SourceContext[Long]) = {
```

```
    while (isRunning && cnt < Long.MaxValue) {
      // Синхронизация выдачи данных и контрольной точки.
      ctx.getCheckpointLock.synchronized {
        cnt += 1
        ctx.collect(cnt)
      }
    }
  }

  override def cancel() = isRunning = false

  override def snapshotState(snapshotCtx: FunctionSnapshotContext): Unit = {
    // Удаление предыдущего указателя.
    offsetState.clear()
    // Добавление текущего указателя.
    offsetState.add(cnt)
  }

  override def initializeState(
      initCtx: FunctionInitializationContext): Unit = {

    val desc = new ListStateDescriptor[Long]("offset", classOf[Long])
    offsetState = initCtx.getOperatorStateStore.getListState(desc)
    // Инициализация переменной счетчика.
    val it = offsetState.get()
    cnt = if (null == it || !it.iterator().hasNext) {
      -1L
    } else {
      it.iterator().next()
    }
  }
}
```

8.3.2. Функции источника, метки времени и водяные знаки

Другой важный аспект функций источника – это метки времени и водяные знаки. Как указано в разделах 3.3 и 6.1.1, API DataStream предоставляет два способа назначения меток времени и создания водяных знаков. Метки времени и водяные знаки могут быть назначены и сгенерированы с помощью специального `TimestampAssigner` (подробности в разделе 6.1.1) или назначены и сгенерированы функцией источника.

Функция источника назначает метки времени и отправляет водяные знаки через свой текстовый объект `SourceCon.SourceContext`, который предоставляет следующие методы:

- `def collectWithTimestamp(T record, long timestamp): Unit`;
- `def emitWatermark(Watermark watermark): Unit`.

Первый метод генерирует запись со связанной с ней меткой времени, а второй – водяной знак предоставляемого типа.

Помимо устранения необходимости в дополнительном операторе, назначение меток времени и создание водяных знаков в исходной функции может пригодиться, если один параллельный экземпляр функции источника потребляет записи из нескольких потоковых разделов, таких как разделы темы Kafka. Обычно внешние системы, такие как Kafka, гарантируют порядок сообщений только внутри потокового раздела. Учитывая случай оператора функции источника, который работает с параллелизмом 2 и считывает данные из темы Kafka с шестью разделами, каждый параллельный экземпляр функции источника будет читать записи из трех разделов темы Kafka. Следовательно, каждый экземпляр функции источника мультиплексирует записи трех потоковых разделов, чтобы передать их. Мультиплексирование записей, скорее всего, вносит дополнительную неупорядоченность по отношению к меткам времени событий, так что нисходящий назначитель меток времени может создавать больше опоздавших записей, чем ожидалось.

Чтобы избежать такого поведения, функция источника может генерировать водяные знаки для каждого раздела потока независимо и всегда выпускать наименьший водяной знак своих разделов в качестве окончательного водяного знака. Таким образом, она может гарантировать, что гарантии порядка используются для каждого раздела и не генерируются ненужные опоздавшие записи.

Еще одна проблема, с которой приходится иметь дело функциям источника, – это экземпляры, которые бездействуют и больше не отправляют данные. Это весьма актуальная проблема, поскольку она может помешать всему приложению обновлять свои водяные знаки и, следовательно, привести к зависанию приложения. Поскольку водяные знаки должны управляться данными, генератор водяных знаков (либо интегрированный в функцию источника, либо в назначитель меток времени) не будет генерировать новые водяные знаки, если он не получает входные записи. Если вы посмотрите, как Flink распространяет и обновляет водяные знаки (см. раздел 3.3.3), вы можете увидеть, что один оператор, который не обновляет водяные знаки, может остановить все водяные знаки приложения, если приложение включает в себя операцию перемешивания (`keyBy()`, `rebalance()` и т. д.).

Flink содержит механизм, который позволяет избежать таких ситуаций, отмечая функции источника как временно неактивные. В режиме ожидания механизм распространения водяных знаков Flink игнорирует неиспользуемый раздел потока. Источник автоматически становится активным, как только он снова начинает отправлять записи. Функция источника может решить, когда пометить себя как неактивную, и делает это путем вызова метода `SourceContext.markAsTempoporaryIdle()`.

8.4. Реализация пользовательской функции приемника

В API DataStream любой оператор или функция может отправлять данные во внешнюю систему или приложение. `DataStream` необязательно должен в ко-

нечном итоге приходить в оператор приемника. Например, вы можете реализовать `FlatMapFunction`, которая передает каждую входящую запись через вызов HTTP POST, а не через свой `Collector`. Тем не менее API DataStream предоставляет специальный интерфейс `SinkFunction` и соответствующий абстрактный класс `RichSinkFunction`[1]. Интерфейс `SinkFunction` предоставляет единственный метод:

```
void invoke (значение IN, контекст ctx)
```

Объект `Context` функции `SinkFunction` обеспечивает доступ к текущему времени обработки, текущему водяному знаку (то есть текущему времени события в приемнике) и метке времени записи.

В примере 8.12 показана простая функция `SinkFunction`, которая записывает показания датчика в сокет. Обратите внимание, что перед запуском программы вам необходимо запустить процесс, который прослушивает сокет. В противном случае программа завершится с ошибкой `ConnectException`, потому что не удалось открыть соединение с сокетом. Выполните команду **`nc -l localhost 9191`** в Linux, чтобы прослушивать localhost:9191.

Пример 8.12 ❖ Простая функция `SinkFunction`, которая записывает данные в сокет

```
val readings: DataStream[SensorReading] = ???

// Записываем показания датчика в сокет.
readings.addSink(new SimpleSocketSink("localhost", 9191))
  // Устанавливаем параллелизм 1, т.к. только один поток может писать в сокет.
  .setParallelism(1)

// -----

class SimpleSocketSink(val host: String, val port: Int)
    extends RichSinkFunction[SensorReading] {

  var socket: Socket = _
  var writer: PrintStream = _

  override def open(config: Configuration): Unit = {
    // Открываем сокет и записыватель.
    socket = new Socket(InetAddress.getByName(host), port)
    writer = new PrintStream(socket.getOutputStream)
  }

  override def invoke(
      value: SensorReading,
      ctx: SinkFunction.Context[_]): Unit = {
    // Записываем данные датчика в сокет.
    writer.println(value.toString)
```

[1] Обычно используется интерфейс `RichSinkFunction`, поскольку функциям приемника обычно требуется установить соединение с внешней системой в методе `RichFunction.open()` (см. главу 5 для получения подробной информации об интерфейсе `RichFunction`).

```
    writer.flush()
  }

  override def close(): Unit = {
    // Закрываем записыватель и сокет.
    writer.close()
    socket.close()
  }
}
```

Как уже говорилось, гарантии сквозной согласованности приложения зависят от свойств его соединителей приемников. Для достижения сквозной семантики «ровно один раз» приложению требуются идемпотентные или транзакционные соединители приемника. Функция `SinkFunction` в примере 8.12 не выполняет идемпотентную запись и не поддерживает транзакционную запись. Из-за того, что сокет имеет свойство «только добавление», невозможно выполнять идемпотентную запись. Поскольку сокет не имеет встроенной поддержки транзакций, транзакционная запись может выполняться только с использованием стандартного приемника WAL Flink. В следующих разделах вы узнаете, как реализовать идемпотентные или транзакционные соединители приемников.

8.4.1. Идемпотентные соединители приемника

Для многих приложений интерфейса `SinkFunction` достаточно для реализации идемпотентного соединителя приемника. Это возможно, если выполняются следующие два условия:

1) данные результата имеют *детерминированный* (составной) ключ, по которому могут выполняться идемпотентные обновления. Для приложения, которое вычисляет среднюю температуру на датчик в минуту, детерминированным составным ключом может служить сочетание идентификатора датчика и метки времени для каждой минуты. Детерминированные ключи важны для обеспечения правильной перезаписи всех операций записи в случае восстановления;
2) внешняя система, например реляционная база данных или хранилище значений ключей, поддерживает обновления для каждого ключа.

В примере 8.13 показано, как реализовать и использовать идемпотентную функцию `SinkFunction`, которая выполняет запись в базу данных JDBC, в данном случае во встроенную базу данных Apache Derby.

Пример 8.13 ❖ Идемпотентная функция `SinkFunction`, которая записывает в базу данных JDBC

```
val readings: DataStream[SensorReading] = ???

// Запись показаний датчика в таблицу Derby.
readings.addSink(new DerbyUpsertSink)

// -----
```

```
class DerbyUpsertSink extends RichSinkFunction[SensorReading] {
  var conn: Connection = _
  var insertStmt: PreparedStatement = _
  var updateStmt: PreparedStatement = _

  override def open(parameters: Configuration): Unit = {
    // Соединение со встроенной таблицей Derby локальной памяти.
    conn = DriverManager.getConnection(
       "jdbc:derby:memory:flinkExample",
       new Properties())
    // Подготовка операторов вставки и обновления.
    insertStmt = conn.prepareStatement(
      "INSERT INTO Temperatures (sensor, temp) VALUES (?, ?)")
    updateStmt = conn.prepareStatement(
      "UPDATE Temperatures SET temp = ? WHERE sensor = ?")
  }

  override def invoke(r: SensorReading, context: Context[_]): Unit = {
    // Настройка параметров и выполнение оператора обновления.
    updateStmt.setDouble(1, r.temperature)
    updateStmt.setString(2, r.id)
    updateStmt.execute()
    // Выполнение оператора вставки, если оператор обновления не обновил ни одной строки.
    if (updateStmt.getUpdateCount == 0) {
      // Настройка параметров оператора вставки.
      insertStmt.setString(1, r.id)
      insertStmt.setDouble(2, r.temperature)
      // Выполнение оператора вставки.
      insertStmt.execute()
    }
  }

  override def close(): Unit = {
    insertStmt.close()
    updateStmt.close()
    conn.close()
  }
}
```

Поскольку Apache Derby не предоставляет встроенного оператора `UPSERT`, приемник из примера выше выполняет запись по типу `UPSERT`, сначала пытаясь обновить строку и вставляя новую строку, если строки с данным ключом не существует. Соединитель приемника Cassandra следует тому же подходу, когда WAL не включен.

8.4.2. Соединители транзакционных приемников

Когда идемпотентный соединитель приемника не подходит из-за характеристик вывода приложения, свойств требуемой системы приемника или из-за более строгих требований согласованности, альтернативой могут служить соединители транзакционного приемника. Как мы говорили ранее, соеди-

нители транзакционных приемников должны быть интегрированы с механизмом контрольных точек Flink, поскольку они могут отправлять данные во внешнюю систему только после успешного завершения контрольной точки.

Чтобы упростить реализацию транзакционных приемников, API DataStream предоставляет два шаблона, которые можно расширить для реализации пользовательских операторов приемников. Оба шаблона реализуют интерфейс `CheckpointListener` для получения уведомлений от JobManager о завершенных контрольных точках (подробности об интерфейсе см. в разделе 7.1.5):

- `GenericWriteAheadSink` собирает все исходящие записи для каждой контрольной точки и сохраняет их в состоянии оператора задачи приемника. Состояние хранится в контрольной точке и восстанавливается в случае сбоя. Когда задача получает уведомление о завершении контрольной точки, она передает записи о завершенных контрольных точках во внешнюю систему. Этот интерфейс реализует соединитель приемника Cassandra с включенным WAL;
- `TwoPhaseCommitSinkFunction` использует транзакционные функции внешней системы приемника. Для каждой контрольной точки он запускает новую транзакцию и отправляет все последующие записи в систему приемника в контексте текущей транзакции. Приемник фиксирует транзакцию, когда получает уведомление о завершении соответствующей контрольной точки.

Далее мы описываем оба интерфейса и гарантии их согласованности.

8.4.2.1. GenericWriteAheadSink

`GenericWriteAheadSink` упрощает реализацию операторов приемника с улучшенными качествами согласованности. Оператор интегрирован с механизмом контрольных точек Flink и стремится отправлять каждую запись во внешнюю систему ровно один раз. Однако вы должны знать, что существуют сценарии сбоя, в которых приемник журнала упреждающей записи создает записи более одного раза. Следовательно, `GenericWriteAheadSink` не предоставляет абсолютную гарантию «ровно один раз», а предоставляет только гарантию «хотя бы один раз». Мы обсудим эти сценарии более подробно дальше в этой главе.

`GenericWriteAheadSink` добавляет все полученные записи в журнал упреждающей записи, который сегментирован по контрольным точкам. Каждый раз, когда оператор приемника получает барьер контрольной точки, он запускает новый раздел, и все последующие записи добавляются в новый раздел. WAL сохраняется в контрольную точку как состояние оператора. Поскольку журнал будет восстановлен в случае сбоя, записи не будут потеряны.

Когда `GenericWriteAheadSink` получает уведомление о завершенной контрольной точке, он отправляет все записи, которые хранятся в WAL в сегменте, соответствующем успешной контрольной точке. В зависимости от конкретной реализации оператора приемника записи могут быть записаны в любое хранилище или систему сообщений. Когда все записи были успешно отправлены, соответствующая контрольная точка должна быть внутренне зафиксирована.

Контрольная точка фиксируется в два этапа. Во-первых, приемник постоянно хранит информацию о том, что контрольная точка была зафиксирована, а во-вторых, он удаляет записи из WAL. Информацию о фиксации невозможно сохранить в состоянии приложения Flink, потому что оно не является постоянным и будет сброшено в случае сбоя. Вместо этого `GenericWriteAheadSink` использует подключаемый компонент `CheckpointCommitter` для хранения и поиска информации о зафиксированных контрольных точках во внешнем постоянном хранилище. Например, соединитель приемника Cassandra по умолчанию использует `CheckpointCommitter`, который записывает информацию о контрольных точках в Cassandra.

Благодаря встроенной логике `GenericWriteAheadSink` нетрудно реализовать приемник, использующий WAL. Операторам, расширяющим `GenericWriteAheadSink`, необходимо предоставить три параметра конструктора:

- `CheckpointCommitter`, как обсуждалось ранее;
- `TypeSerializer` для сериализации входных записей;
- идентификатор задания, который передается в `CheckpointCommitter` для различения информации о фиксации при перезапуске приложения.

Более того, оператор упреждающей записи должен реализовать единственный метод:

```
boolean sendValues(Iterable<IN> values, long chkpntId, long timestamp)
```

`GenericWriteAheadSink` вызывает метод `sendValues()` для передачи записей завершенной контрольной точки во внешнюю систему хранения. Метод получает `Iterable` по всем записям контрольной точки, ID контрольной точки и метку времени, когда контрольная точка была установлена. Метод должен возвращать `true`, если все записи обработаны успешно, и `false`, если запись не удалась.

В примере 8.14 показана реализация приемника с упреждающей записью, который записывает в стандартный вывод. Он использует `FileCheckpointCommitter`, который мы здесь не обсуждаем. Вы можете найти его реализацию в репозитории, содержащем примеры книги.

Обратите внимание, что `GenericWriteAheadSink` не реализует интерфейс функции приема. Таким образом, приемники, расширяющие `GenericWriteAheadSink`, не могут быть добавлены с помощью `DataStream.addSink()`, но присоединяются с помощью метода `DataStream.transform()`.

Пример 8.14 ❖ Приемник WAL, который отправляет данные в стандартный вывод

```
val readings: DataStream[SensorReading] = ???

// Запись показаний датчика в стандартный вывод через журнал опережающей записи.
readings.transform(
  "WriteAheadSink", new SocketWriteAheadSink)

// -----

class StdOutWriteAheadSink extends GenericWriteAheadSink[SensorReading](
```

```
    // CheckpointCommitter, который фиксирует контрольные точки
    // в локальной файловой системе.
    new FileCheckpointCommitter(System.getProperty("java.io.tmpdir")),
    // Сериализатор записей.
    createTypeInformation[SensorReading]
      .createSerializer(new ExecutionConfig),
    // Случайный идентификатор JobID, применяемый в CheckpointCommitter.
    UUID.randomUUID.toString) {

  override def sendValues(
      readings: Iterable[SensorReading],
      checkpointId: Long,
      timestamp: Long): Boolean = {

  for (r <- readings.asScala) {
    // Запись данных в стандартный вывод.
    println(r)
    }
    true
  }
}
```

Репозиторий примеров содержит приложение, которое выходит из строя и регулярно восстанавливается, чтобы продемонстрировать поведение `StdOutWriteAheadSink` и обычного приемника `DataStream.print()` в случае сбоев.

Как упоминалось ранее, `GenericWriteAheadSink` не может предоставить абсолютную гарантию «ровно один раз». Есть два случая сбоя, которые могут привести к повторной отправке записей:

1) программа терпит сбой, пока задача выполняет метод `sendValues()`. Если внешняя система-приемник не дает гарантию *атомарной записи* – или все, или ни одной, – может случиться так, что некоторые данные могли быть записаны, а другие – нет. Поскольку контрольная точка еще не была зафиксирована, приемник снова запишет все записи во время восстановления;
2) все записи внесены правильно, и метод `sendValues()` возвращает `true`; однако программа завершается ошибкой до вызова `CheckpointCommitter` или `CheckpointCommitter` не может зафиксировать контрольную точку. Во время восстановления все записи еще не зафиксированных контрольных точек будут отправлены заново.

Обратите внимание, что эти сценарии сбоя не влияют на гарантии использования соединителя приемника Cassandra «ровно один раз», поскольку он выполняет запись UPSERT. Соединитель приемника Cassandra выигрывает от WAL, потому что он защищает от недетерминированных ключей и предотвращает несогласованные записи в Cassandra.

8.4.2.2. TwoPhaseCommitSinkFunction

Flink предоставляет интерфейс `TwoPhaseCommitSinkFunction`, чтобы упростить реализацию функций приемника, которые обеспечивают сквозные гарантии «ровно один раз». Однако то, предоставляет ли функция приемника 2PC

такие гарантии или нет, зависит от деталей реализации. Мы начинаем обсуждение этого интерфейса с вопроса: «Не слишком ли дорого обходится протокол 2PC?»

Короче говоря, 2PC – дорогостоящий подход к обеспечению согласованности в распределенной системе. Однако в контексте Flink протокол запускается только один раз для каждой контрольной точки. Более того, протокол `TwoPhaseCommitSinkFunction` совмещен с обычным механизмом контрольных точек Flink и добавляет лишь небольшие накладные расходы. Функция `TwoPhaseCommitSinkFunction` работает очень похоже на приемник WAL, но не собирает записи о состоянии приложения Flink; напротив, она записывает их в открытой транзакции во внешнюю систему приемника.

`TwoPhaseCommitSinkFunction` реализует следующий протокол. Прежде чем задача-приемник выдаст свою первую запись, она запускает транзакцию во внешней системе-приемнике. Все полученные впоследствии записи обрабатываются в контексте транзакции. Фаза голосования протокола 2PC начинается, когда JobManager инициирует контрольную точку и вводит барьеры в источники приложения. Когда оператор получает барьер, он проверяет его состояние и отправляет сообщение подтверждения в JobManager, как только это будет сделано. Когда задача-приемник получает барьер, она сохраняет свое состояние, подготавливает текущую транзакцию к фиксации и подтверждает контрольную точку в JobManager. Сообщения подтверждения для JobManager аналогичны голосованию за фиксацию протокола 2PC. Задача приемника на этот момент еще не должна зафиксировать транзакцию, потому что нет уверенности, что все задачи завершили свои контрольные точки. Задача приемника также запускает новую транзакцию для всех записей, поступающих до следующего барьера контрольной точки.

Когда JobManager получает уведомления об успешных контрольных точках от всех экземпляров задач, он отправляет уведомление о завершении контрольных точек всем заинтересованным задачам. Это уведомление соответствует команде фиксации протокола 2PC. Когда задача-приемник получает уведомление, она фиксирует все открытые транзакции предыдущих контрольных точек[1]. После того как задача-приемник подтверждает свою контрольную точку, она должна иметь возможность зафиксировать соответствующую транзакцию даже в случае сбоя. Если транзакция не может быть зафиксирована, приемник теряет данные. Итерация протокола 2PC завершается успешно, когда все задачи-приемники фиксируют свои транзакции.

Обобщим требования к системе внешнего приемника:

- система внешнего приемника должна обеспечивать поддержку транзакций, или приемник должен иметь возможность имитировать транзакции во внешней системе. Следовательно, приемник должен иметь возможность записи в систему приемника, но записанные данные не должны быть видимыми до того, как они будут зафиксированы;
- транзакция должна быть открытой и принимать записи в течение интервала контрольной точки;

[1] Задаче может потребоваться совершить несколько транзакций, если сообщение подтверждения потеряно.

- транзакция должна дождаться подтверждения, пока не будет получено уведомление о завершении контрольной точки. В случае цикла восстановления это может занять некоторое время. Если система-приемник закрывает транзакцию (например, с тайм-аутом), незавершенные данные будут потеряны;
- приемник должен иметь возможность восстанавливать транзакцию после сбоя процесса. Некоторые системы-приемники предоставляют идентификатор транзакции, который можно использовать для фиксации или отмены открытой транзакции;
- фиксация транзакции должна быть идемпотентной операцией – приемник или внешняя система должны иметь возможность заметить, что транзакция уже зафиксирована, или повторная фиксация не должна иметь никакого эффекта.

Возможно, вам будет легче понять протокол и требования системы приемника после конкретного примера. В примере 8.15 показана функция `TwoPhaseCommitSinkFunction`, которая выполняет запись в файловую систему с гарантией «ровно один раз». По сути, это упрощенная версия `BucketingFileSink`, о которой говорилось ранее.

Пример 8.15 ❖ Транзакционный приемник, записывающий данные в файлы

```
class TransactionalFileSink(val targetPath: String, val tempPath: String)
    extends TwoPhaseCommitSinkFunction[(String, Double), String, Void](
      createTypeInformation[String].createSerializer(new ExecutionConfig),
      createTypeInformation[Void].createSerializer(new ExecutionConfig)) {

  var transactionWriter: BufferedWriter = _
  /** Создаем временный файл для транзакции, в которую будем помещать записи.
    */
  override def beginTransaction(): String = {
    // Путь к файлу транзакции включает текущее время и индекс задачи.
    val timeNow = LocalDateTime.now(ZoneId.of("UTC"))
      .format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
    val taskIdx = this.getRuntimeContext.getIndexOfThisSubtask
    val transactionFile = s"$timeNow-$taskIdx"

    // Создаем файл транзакции и записыватель.
    val tFilePath = Paths.get(s"$tempPath/$transactionFile")
    Files.createFile(tFilePath)
    this.transactionWriter = Files.newBufferedWriter(tFilePath)
    println(s"Creating Transaction File: $tFilePath")
    // Возвращаем имя файла транзакции для последующего использования.
    transactionFile
  }

  /** Запись данных в текущий файл транзакции */
  override def invoke(
      transaction: String,
      value: (String, Double),
      context: Context[_]): Unit = {
    transactionWriter.write(value.toString)
```

```
    transactionWriter.write('\n')
  }

  /** Выгрузка и закрытие текущего файла транзакции */
  override def preCommit(transaction: String): Unit = {
    transactionWriter.flush()
    transactionWriter.close()
  }

  /** Фиксируем транзакцию путем переноса предварительного файла транзакции
    * в целевой каталог.
    */
  override def commit(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    // Проверяем наличие файла, чтобы гарантировать идемпотентность фиксации.
    if (Files.exists(tFilePath)) {
      val cFilePath = Paths.get(s"$targetPath/$transaction")
      Files.move(tFilePath, cFilePath)
    }
  }

  /** Прерываем транзакцию удалением файла транзакции */
  override def abort(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    if (Files.exists(tFilePath)) {
      Files.delete(tFilePath)
    }
  }
}
```

`TwoPhaseCommitSinkFunction[IN, TXN, CONTEXT]` имеет три параметра типа:

1) `IN` определяет тип входных записей. В примере 8.15 это `Tuple2` с полями типа `String` и `Double`;
2) `TXN` определяет идентификатор транзакции, который может использоваться для идентификации и восстановления транзакции после сбоя. В примере 8.15 это строка, содержащая имя файла транзакции;
3) `CONTEXT` определяет необязательный настраиваемый контекст. `TransactionalFileSink` в примере 8.15 не нуждается в контексте, поэтому он устанавливает тип как `Void`.

Конструктору `TwoPhaseCommitSinkFunction` требуются два `TypeSerializer` – один для типа `TXN`, а другой для типа `CONTEXT`.

Наконец, `TwoPhaseCommitSinkFunction` определяет пять функций, которые необходимо реализовать:

1) `beginTransaction(): TXN` запускает новую транзакцию и возвращает идентификатор транзакции. `TransactionalFileSink` в примере 8.15 создает новый файл транзакции и возвращает его имя в качестве идентификатора;
2) `invoke(txn: TXN, value: IN, context: Context[_])`: модуль записывает значение в текущую транзакцию. Приемник в примере 8.15 добавляет значение в виде строки в файл транзакции;

3) `preCommit(txn: TXN)`: модуль предварительно фиксирует транзакцию. Предварительно зафиксированная транзакция может не получать дальнейшие записи. Наша реализация в примере 8.15 выгружает и закрывает файл транзакции;
4) `commit(txn: TXN)`: модуль фиксирует транзакцию. Эта операция должна быть идемпотентной – записи не должны записываться в систему вывода дважды, даже если этот метод вызывается дважды. В примере 8.15 мы проверяем, существует ли еще файл транзакции, и перемещаем его в целевой каталог, если это так;
5) `abort(txn: TXN)`: модуль прерывает транзакцию. Этот метод также может вызываться дважды для транзакции. Наш `TransactionalFileSink` в примере 8.15 проверяет, существует ли еще файл транзакции, и удаляет его, если это так.

Как видите, реализация интерфейса не слишком сложна. Однако гарантии сложности и согласованности реализации зависят, среди прочего, от функций и возможностей системы приемника. Например, производитель Kafka Flink реализует интерфейс `TwoPhaseCommitSinkFunction`. Как упоминалось ранее, соединитель может потерять данные, если транзакция откатывается из-за тайм-аута[1]. Следовательно, он не предлагает окончательных гарантий «ровно один раз», даже если он реализует интерфейс `TwoPhaseCommitSinkFunction`.

8.5. Асинхронный доступ к внешним системам

Помимо приема или передачи потоков данных, еще одним распространенным вариантом использования, который требует взаимодействия с внешней системой хранения, является *обогащение потока данных* путем поиска информации в удаленной базе данных. Примером может служить хорошо известный эталонный тест потоковой обработки Yahoo!, основанный на потоке кликов по рекламе, которые необходимо дополнить подробностями о соответствующей кампании, размещенными в хранилище пар «ключ–значение».

Простой подход для таких случаев использования – реализовать функцию `MapFunction`, которая запрашивает в хранилище данных каждую обработанную запись, ожидает, пока запрос вернет результат, обогащает запись и выдает результат. Хотя этот подход легко реализовать, он страдает серьезной проблемой: каждый запрос к внешнему хранилищу данных добавляет значительную задержку (запрос/ответ состоит из двух сетевых сообщений), и `MapFunction` тратит бо̀льшую часть своего времени на ожидание результатов запроса.

Для уменьшения задержки удаленных вызовов ввода/вывода Apache Flink предоставляет асинхронный механизм `AsyncFunction`, который одновременно

[1] См. подробности в разделе 8.2.2.

отправляет несколько запросов и асинхронно обрабатывает их результаты. Функцию можно настроить для сохранения порядка записей (запросы могут возвращаться в порядке, отличном от порядка, в котором они были отправлены) или возвращать результаты в порядке поступления результатов запроса для дальнейшего уменьшения задержки. Эта функция также должным образом интегрирована с механизмом контрольных точек Flink: входные записи, которые в настоящее время ожидают ответа, помечаются контрольными точками, а запросы повторяются в случае восстановления. Более того, `AsyncFunction` правильно работает с обработкой по времени события, потому что она гарантирует, что записи не «перескочат» водяные знаки, даже если разрешены неупорядоченные результаты.

Чтобы воспользоваться преимуществами `AsyncFunction`, внешняя система должна предоставить клиента, который поддерживает асинхронные вызовы, что характерно для многих систем. Если система предоставляет только синхронного клиента, вы можете создавать потоки для отправки запросов и их обработки. Интерфейс `AsyncFunction` показан ниже:

```
trait AsyncFunction[IN, OUT] extends Function {
  def asyncInvoke(input: IN, resultFuture: ResultFuture[OUT]): Unit
}
```

Параметры типов функции определяют ее типы ввода и вывода. Метод `asyncInvoke()` вызывается для каждой входной записи с двумя параметрами. Первый параметр – это входная запись, а второй – объект обратного вызова, возвращающий результат функции или исключение. В примере 8.16 мы показываем, как применять `AsyncFunction` к DataStream.

Пример 8.16 ❖ Применение `AsyncFunction`

```
val readings: DataStream[SensorReading] = ???

val sensorLocations: DataStream[(String, String)] = AsyncDataStream
  .orderedWait(
    readings,
    new DerbyAsyncFunction,
    5, TimeUnit.SECONDS, // Запрос тайм-аута спустя 5 с.
    100)                 // Минимум 100 конкурентных запросов.
```

Асинхронный оператор, применяющий `AsyncFunction`, настраивается с помощью объекта `AsyncDataStream`[1], который предоставляет два статических метода: `orderedWait()` и `unorderedWait()`. Оба метода перегружены для разных комбинаций параметров. Метод `orderedWait()` применяет асинхронный оператор, который выдает результаты в порядке входных записей, в то время как оператор метода `unorderedWait()` только обеспечивает выравнивание водяных знаков и барьеров контрольных точек. Дополнительные параметры указывают, когда истечет время ожидания асинхронного вызова записи и сколько одновременных запросов нужно запустить. В примере 8.17 по-

[1] API Java предоставляет класс AsyncDataStream с соответствующими статическими методами.

казана функция DerbyAsyncFunction, которая запрашивает встроенную базу данных Derby через интерфейс JDBC.

Пример 8.17 ❖ Функция AsyncFunction, которая обращается к Derby через JDBC

```
class DerbyAsyncFunction
    extends AsyncFunction[SensorReading, (String, String)] {

  // Получение контекста выполнения, применяемого в запросах.
  private lazy val cachingPoolExecCtx =
    ExecutionContext.fromExecutor(Executors.newCachedThreadPool())
  // Прямой контекст выполнения для передачи будущего результата объекту обратного вызова.
  private lazy val directExecCtx =
    ExecutionContext.fromExecutor(
      org.apache.flink.runtime.concurrent.Executors.directExecutor())

  /**
    * Выпонение запроса JDBC в потоке и обработка результата
    * с асинхронным обратным вызовом.
    */
  override def asyncInvoke(
      reading: SensorReading,
      resultFuture: ResultFuture[(String, String)]): Unit = {
  val sensor = reading.id
  // Получение раздела таблицы Derby как Future.
  val room: Future[String] = Future {
     // Новое соединение и оператор для каждой записи.
     // Примечание: это НЕ лучшая практика!
     // Подключения и подготовленные операторы следует кешировать.
     val conn = DriverManager
       .getConnection(
          "jdbc:derby:memory:flinkExample",
          new Properties())
      val query = conn.createStatement()

      // Отправляем запрос и ждем результат; это синхронный вызов.
      val result = query.executeQuery(
        s"SELECT room FROM SensorLocations WHERE sensor = '$sensor'")

      // Получаем room (если есть).
      val room = if (result.next()) {
        result.getString(1)
      } else {
        "UNKNOWN ROOM"
      }

      // Закрываем результат, оператор и соединение.
      result.close()
      query.close()
      conn.close()
      // Возвращаем room.
      room
    }(cachingPoolExecCtx)
```

```
    // Применяем к результату обратный вызов.
    room.onComplete {
      case Success(r) => resultFuture.complete(Seq((sensor, r)))
      case Failure(e) => resultFuture.completeExceptionally(e)
    }(directExecCtx)
  }
}
```

Метод `asyncInvoke()` функции `DerbyAsyncFunction` в примере 8.17 обертывает блокирующий запрос JDBC в `Future`, который выполняется через `CachedThreadPool`. Чтобы пример был кратким, мы создаем новое соединение JDBC для каждой записи, что, конечно, довольно неэффективно, и этого следует избегать. `Future[String]` содержит результат запроса JDBC.

Наконец, мы применяем обратный вызов `onComplete()` к `Future` и передаем результат (или возможное исключение) обработчику `ResultFuture`. В отличие от JDBC-запроса `Future` обратный вызов `onComplete()` обрабатывает `DirectExecutor`, поскольку передача результата в `ResultFuture` – это легкая операция, не требующая выделенного потока. Обратите внимание, что все операции выполняются неблокирующим образом.

Важно отметить, что экземпляр `AsyncFunction` последовательно вызывается для каждой из входных записей – экземпляр функции не вызывается в многопоточном режиме. Следовательно, метод `asyncInvoke()` должен быстро вернуться, запустив асинхронный запрос и обработав результат с помощью обратного вызова, который пересылает результат в `ResultFuture`. Следует избегать использования следующих распространенных неудачных приемов:

- отправка запроса, блокирующего метод `asyncInvoke()`;
- отправка асинхронного запроса, но ожидание внутри метода `asyncInvoke()` завершения запроса.

8.6. Заключение

В этой главе вы узнали, как приложения Flink DataStream могут считывать данные и записывать данные во внешние системы, а также познакомились с требованиями к приложению для достижения различных гарантий сквозной согласованности. Мы представили наиболее часто используемые встроенные соединители источника и приемника Flink, которые также служат представителями различных типов систем хранения, таких как очереди сообщений, файловые системы и хранилища пар «ключ–значение».

Затем мы показали вам, как реализовать настраиваемые соединители источника и приемника, включая соединители приемника WAL и 2PC, предоставив подробные примеры. Наконец, вы узнали о функции Flink `AsyncFunction`, которая может значительно улучшить быстродействие взаимодействия с внешними системами за счет асинхронного выполнения и обработки запросов.

Глава 9

Настройка Flink для потоковых приложений

Сегодняшние инфраструктуры данных разнообразны. Структуры распределенной обработки данных, такие как Apache Flink, должны быть настроены для взаимодействия с несколькими компонентами, такими как менеджеры ресурсов, файловые системы и службы распределенной координации.

В этой главе мы обсудим различные способы развертывания кластеров Flink и способы их безопасной настройки и обеспечения высокой доступности. Мы расскажем про настройки Flink для различных версий Hadoop и файловых систем и обсудим наиболее важные параметры конфигурации главного и рабочих процессов Flink. Прочитав эту главу, вы узнаете, как установить и настроить кластер Flink.

9.1. Режимы развертывания

Flink можно развернуть в различных средах, таких как локальная машина, кластер без операционной системы, кластер Hadoop YARN или кластер Kubernetes. В разделе 3.1.1 мы представили различные компоненты системы Flink: JobManager, TaskManager, ResourceManager и Dispatcher. Далее мы объясним, как настроить и запустить Flink в различных средах, включая автономные кластеры, Docker, Apache Hadoop YARN и Kubernetes, и какое сочетание компонентов Flink применяется в каждом варианте.

9.1.1. Автономный кластер

Автономный кластер Flink состоит как минимум из одного главного процесса и как минимум одного процесса TaskManager, которые выполняются на одной или нескольких машинах. Все процессы выполняются как обычные процессы Java JVM. На рис. 9.1 показан автономный вариант Flink.

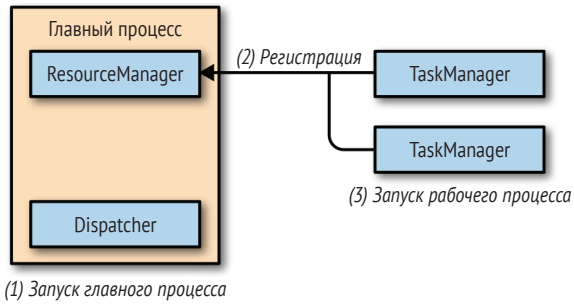

Рис. 9.1 ❖ Запуск автономного кластера Flink

Главный процесс запускает Dispatcher и ResourceManager в отдельных потоках. После запуска TaskManager регистрируется в ResourceManager. На рис. 9.2 показано, как задание передается в автономный кластер.

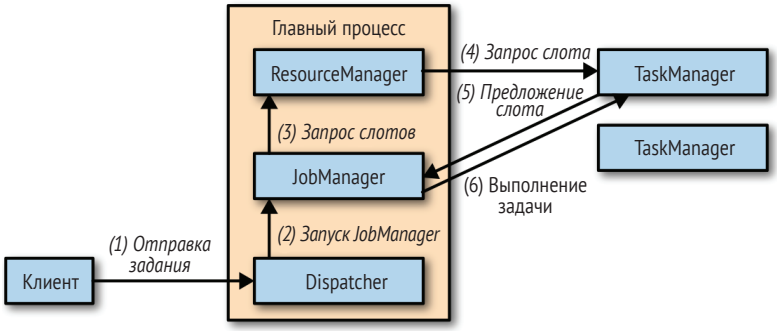

Рис. 9.2 ❖ Отправка приложения в автономный кластер Flink

Клиент отправляет задание диспетчеру Dispatcher, который изнутри запускает поток JobManager и предоставляет JobGraph для выполнения. JobManager запрашивает необходимые слоты обработки у ResourceManager и развертывает задание для выполнения после получения запрошенных слотов.

В автономном варианте главный процесс и рабочие процессы (воркеры) не перезапускаются автоматически в случае сбоя. Задание может восстановиться после сбоя воркера, если доступно достаточное количество слотов обработки. Это можно обеспечить, запустив один или нескольких резервных воркеров. Восстановление задания после сбоя главного процесса требует настройки режима высокой доступности, как описано далее в этой главе.

Чтобы настроить автономный кластер Flink, загрузите двоичный дистрибутив с веб-сайта Apache Flink и извлеките tar-архив с помощью команды:

```
tar xfz ./flink-1.7.1-bin-scala_2.12.tgz
```

Извлеченный каталог содержит папку **./bin** со сценариями bash[1] для запуска и остановки процессов Flink. Сценарий **./bin/start-cluster.sh** запускает главный процесс на локальном компьютере и один или несколько диспетчеров задач на локальном или удаленном компьютере.

Flink предварительно настроен для запуска локальной установки и запуска одного мастера и одного TaskManager на локальном компьютере. Сценарии запуска должны иметь возможность запускать процесс Java. Если путь к исполняемому файлу **java** не находится в PATH, базовую папку установки Java можно указать, экспортировав переменную среды JAVA_HOME или установив параметр env.java.home в файле **./conf/flink-conf.yaml**. Локальный кластер Flink запускается путем вызова **./bin/start-cluster.sh**. Вы можете посетить веб-интерфейс Flink по адресу **http://localhost:8081** и проверить количество подключенных диспетчеров задач и доступных слотов.

Чтобы запустить распределенный кластер Flink, который работает на нескольких машинах, вам необходимо настроить конфигурацию по умолчанию и выполнить еще несколько шагов:

- имена хостов (или IP-адреса) всех машин, на которых должны работать диспетчеры задач, должны быть перечислены в файле **./conf/slaves**;
- сценарий **start-cluster.sh** требует конфигурации SSH без пароля на всех машинах, чтобы иметь возможность запускать процессы TaskManager;
- дистрибутивный общий ресурс Flink должен находиться на всех машинах по одному и тому же пути. Распространенный подход – смонтировать общий сетевой каталог с дистрибутивом Flink на каждой машине;
- имя хоста (или IP-адрес) машины, на которой запущен главный процесс, необходимо настроить в файле **./conf/flink-conf.yaml** с ключом конфигурации jobmanager.rpc.address.

После того как все настроено, вы можете запустить кластер Flink, выполнив сценарий **./bin/start-cluster.sh**. Сценарий запустит локальный JobManager и один TaskManager для каждой записи в файле подчиненных устройств. Вы можете проверить, был ли запущен главный процесс и все ли TaskManager были успешно зарегистрированы, обратившись к веб-интерфейсу на компьютере, на котором запущен главный процесс. Локальный или распределенный автономный кластер останавливается вызовом скрипта **./bin/stop-cluster.sh**.

9.1.2. Docker

Docker – популярная платформа, применяемая для упаковки и запуска приложений в контейнерах. Контейнеры Docker запускаются ядром операционной системы хост-машины и поэтому более легкие, чем виртуальные машины.

[1] Чтобы запустить Flink в Windows, вы можете использовать готовый сценарий bat или обычные сценарии bash в подсистеме Windows для Linux (WSL) или Cygwin. Все скрипты работают только для локальных настроек.

Более того, они изолированы и общаются только через строго определенные каналы. Контейнер запускается из образа, который определяет программное обеспечение в контейнере.

Члены сообщества Flink настраивают и создают образы Docker для Apache Flink и загружают их в Docker Hub, общедоступный репозиторий образов Docker[1]. В репозитории хранятся образы Docker для самых последних версий Flink.

Запуск Flink в Docker – это простой способ настроить кластер Flink на вашем локальном компьютере. Для локальной установки Docker вам необходимо запустить два типа контейнеров: главный контейнер, который запускает Dispatcher и ResourceManager, и один или несколько рабочих контейнеров, которые запускают TaskManager. Контейнеры работают вместе как автономный кластер (см. раздел 9.1.1). После запуска TaskManager регистрируется в ResourceManager. Когда задание отправляется диспетчеру Dispatcher, он порождает поток JobManager, запрашивающий слоты обработки у ResourceManager. ResourceManager в ответ назначает процессы TaskManager для JobManager, который развертывает задание после того, как будут доступны все необходимые ресурсы.

Главный и рабочий контейнеры запускаются из одного образа Docker с разными параметрами, как показано в примере 9.1.

Пример 9.1 ❖ Запуск главного контейнера и рабочего контейнера в Docker

```
// Запуск главного процесса.
docker run -d --name flink-jobmanager \
  -e JOB_MANAGER_RPC_ADDRESS=jobmanager \
  -p 8081:8081 flink:1.7 jobmanager

// Запуск рабочего процесса (предоставьте имена для запуска более одного TaskManager).
docker run -d --name flink-taskmanager-1 \
  --link flink-jobmanager:jobmanager \
  -e JOB_MANAGER_RPC_ADDRESS=jobmanager flink:1.7 taskmanager
```

Docker загрузит запрошенный образ и его зависимости из Docker Hub и запустит контейнеры с Flink. Внутреннее имя хоста Docker для JobManager передается в контейнеры через переменную `JOB_MANAGER_RPC_ADDRESS`, которая используется в точке входа контейнера для настройки конфигурации Flink.

Параметр `-p 8081:8081` первой команды сопоставляет порт 8081 главного контейнера с портом 8081 хост-компьютера, чтобы сделать веб-интерфейс доступным с хоста. Вы можете получить доступ к веб-интерфейсу, открыв в браузере **http://localhost:8081**. Веб-интерфейс можно использовать для загрузки файлов JAR приложения и запуска приложения. Порт также предоставляет REST API Flink. Следовательно, вы также можете отправлять приложения с помощью клиента CLI Flink по адресу **`./bin/flink`**, управлять запущенными приложениями или запрашивать информацию о кластере или запущенных приложениях.

---

[1] Образы Flink Docker не являются частью официального выпуска Apache Flink.

Обратите внимание, что в настоящее время невозможно передать пользовательскую конфигурацию в образы Docker. Вам необходимо создать свой собственный образ Docker, если вы хотите настроить некоторые параметры. Скрипты сборки доступных образов Docker для Flink – хорошая отправная точка для создания настроенных образов.

Вместо того чтобы вручную запускать два (или более) контейнера, вы также можете создать сценарий конфигурации Docker Compose, который автоматически запускает и настраивает кластер Flink, работающий в контейнерах Docker и, возможно, других службах, таких как ZooKeeper и Kafka. Мы не будем вдаваться в подробности этого режима, но, среди прочего, конфигурация Docker Compose должна указывать конфигурацию сети, чтобы процессы Flink, выполняемые в изолированных контейнерах, могли взаимодействовать друг с другом. За подробностями обращайтесь к документации Apache Flink.

9.1.3. Apache Hadoop YARN

YARN – это компонент диспетчера ресурсов Apache Hadoop. Он управляет вычислительными ресурсами кластерной среды – ЦП и памятью машин кластера – и предоставляет их приложениям, запрашивающим ресурсы. YARN предоставляет ресурсы в виде контейнеров[1], которые распределены в кластере и в которых приложения запускают свои процессы. Благодаря своему происхождению из экосистемы Hadoop YARN обычно используется в средах обработки данных.

Flink может работать на YARN в двух режимах: режиме задания и режиме сеанса. В режиме задания кластер Flink запускается на ограниченное время для выполнения одного задания. После завершения задания кластер Flink останавливается и все ресурсы освобождаются. На рис. 9.3 показано, как задание Flink отправляется в кластер YARN.

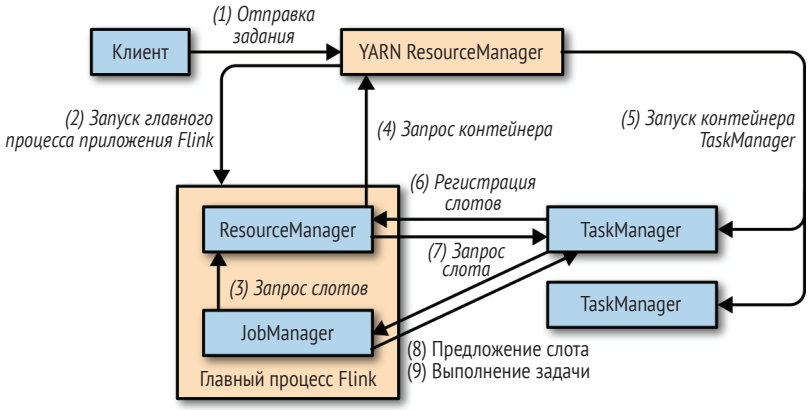

Рис. 9.3 ❖ Запуск кластера Flink на YARN в режиме задания

[1] Обратите внимание, что концепция контейнера в YARN отличается от контейнера в Docker.

Когда клиент отправляет задание на выполнение, он подключается к YARN ResourceManager, чтобы запустить новый главный процесс приложения YARN, который состоит из потока JobManager и ResourceManager. JobManager запрашивает необходимые слоты у ResourceManager для выполнения задания Flink. Затем ResourceManager Flink запрашивает контейнеры у ResourceManager YARN и запускает процессы TaskManager. После запуска TaskManager регистрирует свои слоты в ResourceManager Flink, который предоставляет их JobManager. Наконец, JobManager отправляет задачи задания в TaskManager для выполнения.

В режиме сеанса запускается долгоработающий кластер Flink, который может запускать несколько заданий и должен быть остановлен вручную. При запуске в режиме сеанса Flink подключается к ResourceManager YARN, чтобы запустить главный процесс приложения, который запускает поток Dispatcher и поток Flink ResourceManager. На рис. 9.4 показана схема сеанса Flink YARN в режиме ожидания.

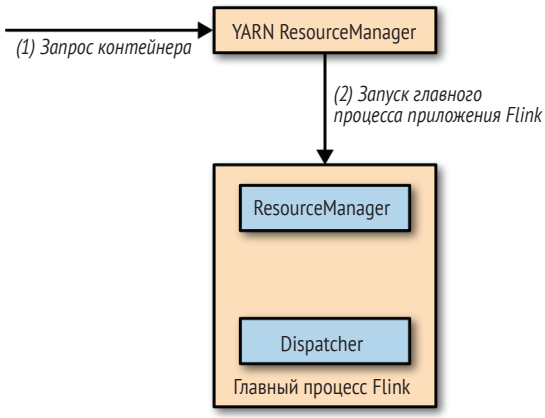

Рис. 9.4 ❖ Запуск кластера Flink на YARN в режиме сеанса

Когда задание отправлено на выполнение, Dispatcher запускает поток JobManager, который запрашивает слоты у ResourceManager Flink. Если слотов недостаточно, ResourceManager Flink запрашивает дополнительные контейнеры у ResourceManager YARN для запуска процессов TaskManager, которые регистрируются в Flink ResourceManager. Когда доступно достаточно слотов, ResourceManager Flink назначает их JobManager и запускает выполнение задания. На рис. 9.5 показано, как выполняется задание в режиме сеанса YARN.

Для обоих вариантов – режима задания и сеанса – отказавшие диспетчеры задач будут автоматически перезапущены силами ResourceManager Flink. В файле конфигурации `./conf/flink-conf.yaml` есть несколько параметров, которые вы можете использовать для управления поведением Flink при восстановлении в YARN. Например, вы можете настроить максимальное количество отказавших контейнеров до завершения работы приложения.

Для восстановления после сбоев главного процесса необходимо настроить высокодоступную конфигурацию, как описано в следующем разделе.

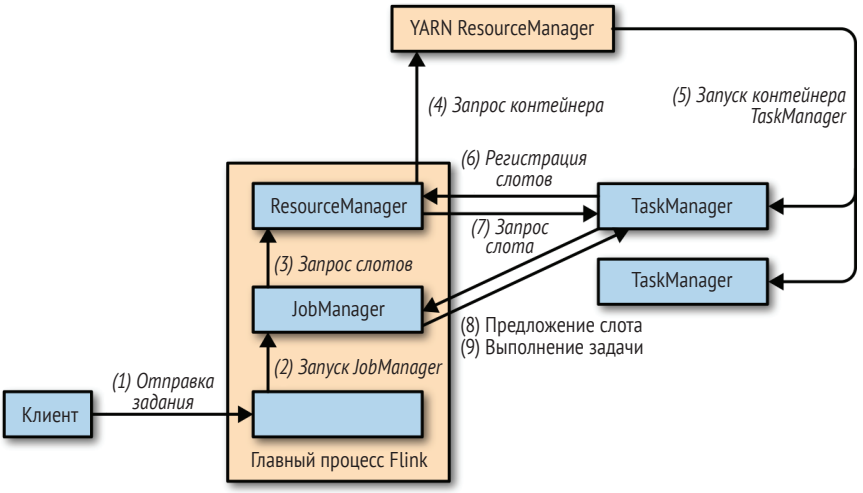

Рис. 9.5 ❖ Отправка задания в кластер сеанса Flink YARN

Независимо от того, запускаете ли вы Flink в режиме задания или сеанса в YARN, он должен иметь доступ к зависимостям Hadoop в правильной версии и путь к конфигурации Hadoop. Раздел 9.3 подробно описывает необходимую конфигурацию.

Итак, вы правильно настроили рабочую конфигурацию YARN и HDFS. Теперь задание Flink может быть отправлено для выполнения в YARN посредством клиента командной строки Flink с помощью следующей команды:

```
./bin/flink run -m yarn-cluster ./path/to/job.jar
```

Параметр `-m` определяет хост, на который отправляется задание. Если присутствует ключевое слово `yarn-cluster`, клиент отправляет задание в кластер YARN, как это определено конфигурацией Hadoop. Клиент командной строки Flink поддерживает множество других параметров, таких как возможность управления памятью контейнеров TaskManager. Обратитесь к документации, чтобы узнать о доступных параметрах. Веб-интерфейс запущенного кластера Flink обслуживается главным процессом, запущенным на каком-либо узле в кластере YARN. Вы можете получить к нему доступ через веб-интерфейс YARN, который содержит ссылку на странице обзора приложения в разделе **Tracking URL: ApplicationMaster**.

Сеанс Flink в YARN запускается с помощью сценария `./bin/yarn-session.sh`, который также использует различные параметры для управления размером контейнеров, именем приложения YARN или для предоставления динамических свойств. По умолчанию сценарий печатает информацию о подключении кластера сеанса и ожидает ввод. Сеанс останавливается, и все ресурсы освобождаются, когда сценарий прекращает работу. Также можно запустить

сеанс YARN в автономном режиме, используя флаг **-d**. Отсоединенный сеанс Flink можно завершить с помощью служебных программ YARN.

После запуска сеанса Flink в YARN вы можете отправлять задания в сеанс с помощью команды **./bin/flink run ./path/to/job.jar**.

Обратите внимание, что вам не нужно предоставлять информацию о соединении, так как Flink запомнил детали соединения сеанса Flink, запущенного на YARN. Как и в режиме работы, веб-интерфейс Flink связан с обзором приложений веб-интерфейса YARN.

9.1.4. Kubernetes

Kubernetes – это платформа с открытым исходным кодом, которая позволяет пользователям развертывать и масштабировать контейнерные приложения в распределенной среде. Имея кластер Kubernetes и приложение, упакованное в образ контейнера, вы можете создать развертывание приложения, которое сообщает Kubernetes, сколько экземпляров приложения нужно запустить. Kubernetes будет запускать запрошенное количество контейнеров в любом месте на своих ресурсах и перезапускать их в случае сбоя. Kubernetes также может позаботиться об открытии сетевых портов для внутренней и внешней связи и может предоставлять сервисы для обнаружения процессов и балансировки нагрузки. Kubernetes работает локально, в облачных средах или в гибридной инфраструктуре.

Развертывание фреймворков и приложений для обработки данных на Kubernetes стало очень популярным. Apache Flink также можно развернуть на Kubernetes. Прежде чем углубляться в детали того, как настроить Flink на Kubernetes, нам нужно кратко объяснить несколько терминов Kubernetes:

- *кокон*[1] (pod) – это контейнер, который запускается и управляется Kubernetes[2];
- *развертывание* (deployment) определяет заданное количество запускаемых модулей или контейнеров. Kubernetes гарантирует, что запрошенное количество модулей постоянно работает, и автоматически перезапускает отказавшие модули. Развертывания можно увеличивать или уменьшать.

Kubernetes может запускать кокон в любом месте своего кластера. Когда модуль перезапускается после сбоя или при изменении масштаба развертывания IP-адрес может измениться. Очевидно, что это проблема, если коконы должны взаимодействовать друг с другом. Kubernetes предоставляет сервисы для решения этой проблемы. *Сервис* определяет политику доступа к определенной группе модулей. Он заботится об обновлении маршрутизации, когда модуль запускается на другом узле в кластере.

---

[1] К сожалению, в русском языке почему-то прижилась калька «под», хотя английское слово pod переводится как «кокон, стручок», что звучит ничуть не хуже, чем «контейнер» Docker, и хорошо отражает суть изолированного запуска. – *Прим. перев.*

[2] Kubernetes также поддерживает коконы, состоящие из нескольких тесно связанных контейнеров.

Запуск Kubernetes на локальной машине

Kubernetes предназначен для кластерных операций. Однако проект Kubernetes предоставляет Minikube – среду для запуска одноузлового кластера Kubernetes локально на одной машине для тестирования или повседневной разработки. Мы рекомендуем настроить Minikube, если вы хотите попробовать запустить Flink на Kubernetes и у вас нет кластера Kubernetes под рукой.

Чтобы успешно запускать приложения в кластере Flink, развернутом на Minikube, перед развертыванием Flink необходимо выполнить следующую команду:

```
minikube ssh 'sudo ip link set docker0 promisc on'
```

Настройка Flink для Kubernetes определяется двумя развертываниями: одно для модуля, на котором выполняется главный процесс, а другое – для модулей рабочего процесса. Также существует служба, которая предоставляет порты основного модуля рабочим модулям. Два типа модулей – главный и рабочий – ведут себя так же, как процессы автономного развертывания или развертывания Docker, которые мы описали ранее. Конфигурация развертывания главного модуля показана в примере 9.2.

Пример 9.2 ❖ Развертывание Kubernetes для главного модуля Flink

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: flink-master
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: flink
        component: master
    spec:
containers:
- name: master
  image: flink:1.7
  args:
  - jobmanager
  ports:
  - containerPort: 6123
    name: rpc
  - containerPort: 6124
    name: blob
  - containerPort: 6125
    name: query
  - containerPort: 8081
    name: ui
  env:
  - name: JOB_MANAGER_RPC_ADDRESS
    value: flink-master
```

Это развертывание указывает, что должен быть запущен один главный контейнер (`replicas: 1`). Главный контейнер запускается из образа Docker

Flink 1.7 (`image: flink:1.7`) с аргументом, который запускает главный процесс (`args: - jobmanager`). Кроме того, развертывание указывает, какие порты контейнера открывать для RPC, диспетчер больших двоичных объектов (для обмена большими файлами), запрашиваемый сервер состояний, а также веб-интерфейс и интерфейс REST. В примере 9.3 показано развертывание рабочих модулей.

Пример 9.3 ❖ Развертывание Kubernetes для двух рабочих модулей Flink

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: flink-worker
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: flink
        component: worker
    spec:
      containers:
      - name: worker
        image: flink:1.7
        args:
        - taskmanager
        ports:
        - containerPort: 6121
        name: data
        - containerPort: 6122
        name: rpc
        - containerPort: 6125
        name: query
      env:
      - name: JOB_MANAGER_RPC_ADDRESS
        value: flink-master
```

Развертывание рабочего модуля выглядит почти идентично развертыванию главного модуля с некоторыми отличиями. Прежде всего в развертывании рабочего модуля указаны две реплики, что означает запуск двух рабочих контейнеров. Рабочие контейнеры основаны на одном и том же образе Flink Docker, но запускаются с другим аргументом (`args: -taskmanager`). Более того, развертывание также открывает несколько портов и передает имя службы развертывания главного модуля Flink, чтобы рабочие модули могли получить доступ к нему. Определение службы, которое раскрывает главный процесс и делает его доступным для рабочих контейнеров, показано в примере 9.4.

Пример 9.4 ❖ Служба Kubernetes для главного модуля Flink

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: flink-master
spec:
  ports:
  - name: rpc
    port: 6123
  - name: blob
    port: 6124
  - name: query
    port: 6125
  - name: ui
    port: 8081
  selector:
    app: flink
    component: master
```

Вы можете создать развертывание Flink для Kubernetes, сохранив каждое определение в отдельном файле, например **`master-deployment.yaml`**, **`worker-deployment.yaml`** или **`master-service.yaml`**. Файлы также находятся в нашем репозитории. Когда у вас есть файлы определений, вы можете зарегистрировать их в Kubernetes с помощью команды **`kubectl`**:

```
kubectl create -f master-deployment.yaml
kubectl create -f worker-deployment.yaml
kubectl create -f master-service.yaml
```

При выполнении этих команд Kubernetes начинает развертывание запрошенных контейнеров. Вы можете показать статус всех развертываний, выполнив следующую команду:

```
kubectl get deployments
```

Когда вы создаете развертывания в первый раз, потребуется некоторое время, прежде чем будет загружен образ контейнера Flink. Когда все коконы будут запущены, у вас будет кластер Flink, работающий на Kubernetes. Однако с данной конфигурацией Kubernetes не экспортирует порт во внешние среды. Следовательно, вы не можете получить доступ к главному контейнеру для отправки приложения или доступа к веб-интерфейсу. Сначала вам нужно указать Kubernetes создать перенаправление портов из главного контейнера на ваш локальный компьютер. Это делается с помощью следующей команды:

```
kubectl port-forward deployment/flink-master 8081:8081
```

Когда работает переадресация портов, вы можете получить доступ к веб-интерфейсу по адресу **http://localhost: 8081**.

Теперь вы можете загружать и отправлять задания в кластер Flink, работающий на Kubernetes. Кроме того, вы можете отправлять приложения с помощью клиента Flink CLI (**`./bin/flink`**) и получать доступ к интерфейсу REST, чтобы запрашивать информацию о кластере Flink или управлять запущенными приложениями.

Когда рабочий кокон выходит из строя, Kubernetes автоматически перезапускает его, и приложение будет восстановлено (при условии, что контроль-

ная точка была активирована и правильно настроена). Для восстановления после сбоя главного кокона вам необходимо настроить высокодоступную конфигурацию.

Вы можете выключить кластер Flink, работающий в Kubernetes, выполнив следующие команды:

```
kubectl delete -f master-deployment.yaml
kubectl delete -f worker-deployment.yaml
kubectl delete -f master-service.yaml
```

С помощью образов Flink Docker, которые мы использовали в этом разделе, вы не сможете создать пользовательские конфигурации развертываний Flink. Вам нужно будет создать собственные образы Docker с настроенной конфигурацией. Сценарий сборки для предоставляемого образа – хорошая отправная точка для создания пользовательского образа.

9.2. Режим высокой доступности

Большинство потоковых приложений идеально работает непрерывно с минимальным временем простоя. Следовательно, многие приложения должны иметь возможность автоматически восстанавливаться после сбоя любого процесса, вовлеченного в выполнение. В то время как сбои рабочих процессов обрабатываются ResourceManager, обработка сбоев компонента JobManager требует работы в режиме *высокой доступности* (highly available, HA).

JobManager Flink хранит метаданные о приложении и его выполнении, такие как файл JAR приложения, JobGraph и указатели на завершенные контрольные точки. Эта информация должна быть восстановлена в случае отказа главного процесса. Режим высокой доступности Flink основан на Apache ZooKeeper, службе для распределенной координации и согласованного хранения, а также на постоянном удаленном хранилище, таком как HDFS, NFS или S3. JobManager сохраняет все соответствующие данные в постоянном хранилище и записывает указатель на информацию – путь хранения – в ZooKeeper. В случае сбоя новый JobManager ищет указатель в ZooKeeper и загружает метаданные из постоянного хранилища. Мы рассказали о принципе работы и внутреннем устройстве режима высокой доступности Flink в разделе 3.1.4. В этом разделе мы настроим режим HA для различных вариантов развертывания.

Для установки Flink HA требуется работающий кластер Apache ZooKeeper и постоянное удаленное хранилище, такое как HDFS, NFS или S3. Чтобы помочь пользователям быстро запустить кластер ZooKeeper в целях тестирования, Flink предоставляет вспомогательный сценарий для начальной загрузки. Во-первых, вам необходимо настроить хосты и порты всех процессов ZooKeeper, участвующих в кластере, настроив файл `./conf/zoo.cfg`. Как только это будет сделано, вы можете вызвать `./bin/start-zookeeper-quorum.sh`, чтобы запустить процесс ZooKeeper на каждом настроенном узле.

Не используйте start-zookeeper-quorum.sh для производственных сред

Вам не следует использовать скрипт Flink ZooKeeper для производственных сред. Вместо этого тщательно настройте и разверните кластер ZooKeeper самостоятельно.

Режим Flink HA настраивается в файле **`./conf/flink-conf.yaml`** путем установки параметров, как показано в примере 9.5.

Пример 9.5 ❖ Настройка кластера Flink HA

```
# НЕОБХОДИМО: включить режим HA через ZooKeeper.
high-availability: zookeeper

# НЕОБХОДИМО: предоставить список всех серверов ZooKeeper , занятых в кворуме.
high-availability.zookeeper.quorum: address1:2181[,...],addressX:2181

# НЕОБХОДИМО: указать расположение метаданных задания в удаленном хранилище.
high-availability.storageDir: hdfs:///flink/recovery

# РЕКОМЕНДУЕТСЯ: задать базовый путь для всех кластеров Flink в ZooKeeper.
# Это изолирует Flink от других платформ, использующих кластер ZooKeeper.
high-availability.zookeeper.path.root: /flink
```

9.2.1. Высокая доступность в автономном режиме

Автономное развертывание Flink не зависит от поставщика ресурсов, такого как YARN или Kubernetes. Все процессы запускаются вручную, и нет компонента, который отслеживает эти процессы и перезапускает их в случае сбоя. Следовательно, для автономного кластера Flink требуются резервные процессы Dispatcher и TaskManager, которые могут взять на себя работу отказавших процессов.

Помимо запуска резервных процессов TaskManager, автономное развертывание не требует дополнительной настройки для восстановления после сбоев диспетчера задач. Все запущенные процессы TaskManager регистрируются в активном ResourceManager. Приложение может восстановиться после сбоя TaskManager, пока достаточно слотов обработки находится в режиме ожидания, чтобы компенсировать потерянный TaskManager. ResourceManager выделяет ранее бездействующие слоты обработки, и приложение перезапускается.

В случае высокодоступной конфигурации все процессы Dispatcher регистрируются в ZooKeeper, который в свою очередь выбирает процесс-лидер, ответственный за выполнение приложений. Когда приложение отправлено на выполнение, ответственный Dispatcher запускает поток JobManager, который сохраняет свои метаданные в настроенном постоянном хранилище и указатель в ZooKeeper, как обсуждалось ранее. Если главный процесс, запускающий активный Dispatcher и JobManager, выходит из строя, ZooKeeper выбирает новый Dispatcher в качестве лидера. Лидирующий Dispatcher восстанавливает отказавшее приложение, запустив новый поток JobManager, который ищет указатель метаданных в ZooKeeper и загружает метаданные из постоянного хранилища.

В дополнение к ранее описанной конфигурации для автономного высокодоступного варианта требуются следующие изменения конфигурации. В **./conf/flink-conf.yaml** вам нужно установить идентификатор кластера для каждого работающего кластера. Это необходимо, если несколько кластеров Flink полагаются на один и тот же экземпляр ZooKeeper для восстановления после сбоя:

```
# РЕКОМЕНДУЕТСЯ: задать базовый путь для кластера Flink в ZooKeeper.
# Это изолирует кластеры Flink друг от друга.
# Для получения метаданных отказавшего кластера требуется идентификатор кластера.
high-availability.cluster-id: /cluster-1
```

Если у вас работает кворум ZooKeeper и правильно настроен Flink, вы можете использовать обычный сценарий **./bin/start-cluster.sh** для запуска автономного кластера высокой доступности, добавив дополнительные имена хостов и порты в файл **./conf/masters**.

9.2.2. Высокодоступная конфигурация YARN

YARN – это менеджер ресурсов и контейнеров кластера. По умолчанию он автоматически перезапускает отказавший главный контейнер и контейнеры TaskManager. Следовательно, для достижения высокой доступности вам не нужно запускать резервные процессы в конфигурации YARN.

Главный процесс Flink запускается как YARN ApplicationMaster[1]. YARN автоматически перезапускает отказавший ApplicationMaster, но отслеживает и ограничивает количество перезапусков, чтобы предотвратить бесконечные циклы восстановления. Вам необходимо настроить максимальное количество перезапусков ApplicationManager в файле конфигурации YARN **yarn-site.xml**, как показано ниже:

```
<property>
  <name>yarn.resourcemanager.am.max-attempts</name>
  <value>4</value>
  <description>
    Максимальное количество попыток выполнения мастера приложения.
    Значение по умолчанию - 2, т.е. приложение перезапускается не более одного раза.
  </description>
</property>
```

Кроме того, вам необходимо настроить файл конфигурации Flink **./conf/flink-conf.yaml** и настроить количество попыток перезапуска приложения:

```
# Перезапустить приложение не более 3 раз (+ первоначальный запуск).
# Должно быть меньше или равно настроенному максимальному количеству попыток.
yarn.application-attempts: 4
```

[1] ApplicationMaster – это главный процесс приложения YARN.

YARN подсчитывает только количество перезапусков из-за сбоев приложения – перезапуски из-за вытеснения, сбоев оборудования или перезагрузки не учитываются при подсчете количества попыток перезапуска приложения. Если вы запускаете Hadoop YARN версии 2.6 или новее, Flink автоматически настраивает интервал действия неудачной попытки. Этот параметр указывает, что приложение полностью отменяется только в том случае, если оно превышает количество попыток перезапуска в пределах заданного интервала, то есть попытки, предшествующие этому интервалу, не учитываются. Flink настраивает интервал на то же значение, что и параметр `akka.ask.timeout` в **`./conf/flink-conf.yaml`**, со значением по умолчанию 10 с.

Имея работающий кластер ZooKeeper и правильно настроенные настройки YARN и Flink, вы можете запустить кластер Flink в режиме задания или режиме сеанса, как если бы режим высокой доступности не был включен – с помощью команд **`./bin/flink run -m yarn-cluster`** и **`./bin/yarn-session.sh`**.

Обратите внимание, что вы должны настроить разные идентификаторы кластера для всех кластеров сеанса Flink, которые подключаются к одному кластеру ZooKeeper. При запуске кластера Flink в режиме задания идентификатор кластера автоматически устанавливается равным идентификатору запущенного приложения и поэтому является уникальным.

9.2.3. Высокодоступная конфигурация Kubernetes

При запуске Flink в Kubernetes с главным и рабочим развертыванием, как описано в разделе 9.1.4, Kubernetes автоматически перезапускает отказавшие контейнеры, чтобы обеспечить правильное количество коконов. Этого достаточно для восстановления после сбоев рабочих процессов, которые обрабатываются ResourceManager. Однако восстановление после сбоев главного процесса требует дополнительной настройки, как мы говорили ранее.

Чтобы включить режим высокой доступности Flink, вам необходимо настроить конфигурацию Flink и предоставить такую информацию, как имена хостов узлов кворума ZooKeeper, путь к постоянному хранилищу и идентификатор кластера для Flink. Все эти параметры необходимо добавить в файл конфигурации Flink (**`./conf/flink-conf.yaml`**).

Пользовательская конфигурация в образах Flink

К сожалению, образ Flink Docker, который мы использовали ранее в примерах Docker и Kubernetes, не поддерживает установку пользовательских параметров конфигурации. Следовательно, образ нельзя использовать для настройки высокодоступного кластера Flink в Kubernetes. Вместо этого вам нужно создать собственный образ, который содержит необходимые параметры непосредственно в коде либо будет достаточно гибким, чтобы динамически настраивать конфигурацию с помощью параметров или переменных среды. Стандартные образы Flink Docker – хорошая отправная точка для настройки ваших собственных образов Flink.

9.3. Интеграция с компонентами Hadoop

Apache Flink можно легко интегрировать с Hadoop YARN и HDFS и другими компонентами экосистемы Hadoop, такими как HBase. Во всех этих случаях Flink требует наличия зависимостей Hadoop.

Есть три способа предоставить Flink зависимости от Hadoop.

1. Используйте двоичный дистрибутив Flink, созданный для конкретной версии Hadoop. Flink предоставляет сборки для наиболее часто используемых базовых версий Hadoop.
2. Соберите Flink для конкретной версии Hadoop. Это полезно, если ни один из двоичных дистрибутивов Flink не работает с версией Hadoop, развернутой в вашей среде; например, если вы используете доработанную версию Hadoop или версию Hadoop от дистрибьютора, такого как Cloudera, Hortonworks или MapR.
   Чтобы собрать Flink для конкретной версии Hadoop, вам понадобится исходный код Flink, который можно получить, загрузив исходный дистрибутив с веб-сайта или клонировав ветку стабильного выпуска из репозитория Git проекта, Java JDK не ниже версии 8, и Apache Maven 3.2. Войдите в базовую папку с исходным кодом Flink и выполните одну из следующих команд:

   ```
   // Сборка Flink для заданной версии Hadoop.
   mvn clean install -DskipTests -Dhadoop.version=2.6.1

   // Сборка Flink для версии Hadoop от дистрибьютора.
   mvn clean install -DskipTests -Pvendor-repos \
   -Dhadoop.version=2.6.1-cdh5.0.0
   ```

   Завершенная сборка находится в папке **`./build-target`**.
3. Используйте дистрибутив Flink без Hadoop и вручную настройте путь к классам для зависимостей Hadoop. Этот подход полезен, если ни одна из предоставленных сборок не подходит для вашей конфигурации. Путь к классам зависимостей Hadoop должен быть объявлен в переменной окружения `HADOOP_CLASSPATH`. Если переменная не настроена, вы можете автоматически настроить ее с помощью следующей команды: **`export HADOOP_CLASSPATH = `hadoop classpath``**.
   Параметр `classpath` команды `hadoop` выводит настроенный путь к классам.

Помимо настройки зависимостей Hadoop, вам необходимо указать расположение каталога конфигурации Hadoop. Это нужно сделать путем экспорта переменной среды `HADOOP_CONF_DIR` (предпочтительно) или `HADOOP_CONF_PATH`. Как только Flink получит конфигурацию Hadoop, он сможет подключиться к ResourceManager YARN и HDFS.

9.4. Конфигурация файловой системы

Apache Flink использует файловые системы для различных задач. Приложения могут считывать свой ввод из файлов и записывать свои результаты в файлы (см. раздел 8.2.3), контрольные точки приложений и метаданные сохраняются в удаленных файловых системах для восстановления (см. раздел 3.5), а некоторые внутренние компоненты используют файловые системы для распределения данных по задачам, например, файлы JAR приложений.

Flink поддерживает широкий спектр файловых систем. Поскольку Flink является распределенной системой и выполняет процессы в кластерной или облачной среде, файловые системы обычно должны быть доступны глобально. По этой причине обычно используемыми файловыми системами являются Hadoop HDFS, S3 и NFS.

Подобно другим системам обработки данных, Flink смотрит на схему URI пути, чтобы идентифицировать файловую систему, к которой относится путь. Например, путь **file:///home/user/data.txt** указывает на файл в локальной файловой системе, а **hdfs:/// namenode:50010/home/user/data.txt** – на файл в указанном кластере HDFS.

Файловая система представлена в Flink реализацией класса `org.apache.flink.core.fs.FileSystem`. Класс `FileSystem` реализует операции файловой системы, такие как чтение из файлов и запись в них, создание каталогов или файлов и перечисление содержимого каталога. Процесс Flink (JobManager или TaskManager) создает один объект `FileSystem` для каждой настроенной файловой системы и совместно использует его для всех локальных задач, чтобы гарантировать соблюдение настроенных ограничений, таких как ограничения на количество открытых подключений.

Flink предоставляет следующие реализации для наиболее часто используемых файловых систем.

Локальная файловая система

Flink имеет встроенную поддержку локальных файловых систем, включая локально смонтированные сетевые файловые системы, такие как NFS или SAN, и не требует дополнительной настройки. На локальные файловые системы ссылается схема URI **file:**//.

Hadoop HDFS

Коннектор Flink для HDFS всегда находится в пути к классам Flink. Однако для работы требуются зависимости Hadoop от пути к классам. Раздел 9.3 объясняет, как обеспечить загрузку зависимостей Hadoop. Пути HDFS имеют префикс схемы **hdfs:**//.

Amazon S3

Flink предоставляет два альтернативных соединителя файловой системы для подключения к S3, основанные на Apache Hadoop и Presto. Оба соединителя полностью автономны и не имеют никаких зависимостей. Чтобы

установить любой из этих соединителей, переместите соответствующий файл JAR из папки **./opt** в папку **./lib**. Документация Flink предоставляет более подробную информацию о конфигурации файловых систем S3. Пути S3 указываются по схеме **s3:**//.

OpenStack Swift FS

Flink предоставляет соединитель для Swift FS, основанного на Apache Hadoop. Соединитель полностью автономен и не имеет никаких зависимостей. Он устанавливается путем перемещения файла JAR быстрого соединения из **./opt** в папку **./lib**. Пути Swift FS идентифицируются по схеме **swift:**//.

В случае работы с файловыми системами, для которых отсутствует специальный коннектор, Flink может делегировать полномочия соединителю файловой системы Hadoop, если он правильно настроен. Вот почему Flink поддерживает все файловые системы HCFS.

Flink предоставляет несколько параметров конфигурации в **./conf/flink-conf.yaml**, чтобы указать файловую систему по умолчанию и ограничить количество подключений к файловой системе. Вы можете указать схему файловой системы по умолчанию (`fs.default-scheme`), которая автоматически добавляется в качестве префикса, если путь не предоставляет схему. Если вы, например, укажете префикс `fs.default-scheme: hdfs://nnode1:9000`, путь **/result** будет расширен до **hdfs://nnode1:9000/result**.

Вы можете ограничить количество соединений, которые читают (вводят) и записывают (выводят) данные. Конфигурация может быть определена для схемы URI. Соответствующие ключи конфигурации:

```
fs.<scheme>.limit.total: (число, 0 / -1 означает отсутствие ограничения)
fs.<scheme>.limit.input: (число, 0 / -1 означает отсутствие ограничения)
fs.<scheme>.limit.output: (число, 0 / -1 означает отсутствие ограничений)
fs.<scheme>.limit.timeout: (миллисекунды, 0 означает бесконечность)
fs.<scheme>.limit.stream-timeout: (миллисекунды, 0 означает бесконечность)
```

Количество подключений отслеживается для каждого процесса TaskManager и полномочий пути – **hdfs://nnode1:50010** и **hdfs://nnode2:50010** отслеживаются отдельно. Пределы подключений можно настроить по отдельности для входных и выходных подключений или как общее количество подключений. Когда файловая система достигает предела количества подключений и пытается открыть новое подключение, она блокируется и ожидает закрытия другого подключения. Параметры тайм-аута определяют, как долго ждать, пока запрос на соединение не завершится неудачно (`fs.<scheme>.limit.timeout`), и как долго ждать, пока не будет закрыто простаивающее соединение (`fs.<scheme>.limit.stream-timeout`).

Вы также можете предоставить настраиваемый соединитель файловой системы. Ознакомьтесь с документацией Flink, чтобы узнать, как реализовать и зарегистрировать пользовательскую файловую систему.

9.5. Конфигурация системы

Apache Flink предлагает множество параметров для настройки его поведения и улучшения быстродействия. Все параметры можно определить в файле **`./conf/flink-conf.yaml`**, который организован как плоский файл YAML, состоящий из пар «ключ–значение». Файл конфигурации считывается различными компонентами, такими как стартовые сценарии, главный и рабочий процессы JVM и клиент CLI. Например, сценарии запуска, такие как **`./bin/start-cluster.sh`**, анализируют файл конфигурации для извлечения параметров JVM и настроек размера кучи, а клиент CLI (**`./bin/flink`**) извлекает информацию о подключении для доступа к мастер-процессу. Изменения в файле конфигурации не вступят в силу до перезапуска Flink.

Для облегчения запуска сразу после установки Flink предварительно настроен на локальную работу. Вам придется настроить конфигурацию для успешного запуска Flink в распределенных средах. В этом разделе мы обсуждаем различные аспекты, которые обычно необходимо настроить при настройке кластера Flink. Мы отсылаем вас к официальной документации за исчерпывающим списком и подробным описанием всех параметров.

9.5.1. Java и загрузка классов

По умолчанию Flink запускает процессы JVM, используя исполняемый файл Java, связанный с помощью переменной среды `PATH`. Если Java отсутствует в `PATH` или, если вы хотите использовать другую версию Java, вы можете указать корневую папку установки Java с помощью переменной среды `JAVA_HOME` или ключа `env.java.home` в файле конфигурации. Процессы JVM Flink можно запускать с настраиваемыми параметрами Java – например, для точной настройки сборщика мусора или для включения удаленной отладки с помощью ключей `env.java.opts`, `env.java.opts.jobmanager` и `env.java.opts.taskmanager`.

При выполнении заданий с внешними зависимостями нередко встречаются проблемы с загрузкой классов. Чтобы запустить приложение Flink, все классы в файле JAR приложения должны быть загружены загрузчиком классов. Flink регистрирует классы каждого задания в отдельном загрузчике классов пользовательского кода, чтобы гарантировать, что зависимости задания не связаны с зависимостями времени выполнения Flink или зависимостями других заданий. Загрузчики классов пользовательского кода удаляются после завершения соответствующего задания. Загрузчик системных классов Flink загружает все файлы JAR в папку **`./lib`**, а загрузчики классов пользовательского кода являются производными от системного загрузчика классов.

По умолчанию Flink сначала ищет классы пользовательского кода в дочернем (пользовательском) загрузчике классов, а затем в родительском (системном) загрузчике классов, чтобы предотвратить конфликты версий в случае, если задание использует ту же зависимость, что и Flink. Однако вы также можете инвертировать порядок поиска с помощью конфигурационного ключа `classloader.resolve-order`.

Обратите внимание, что некоторые классы всегда разрешаются первыми в родительском загрузчике классов (`classloader.parent-first-patterns.default`). Вы можете расширить список, предоставив белый список шаблонов имен классов, которые сначала разрешаются из родительского загрузчика классов (`classloader.parent-first-patterns.additional`).

9.5.2. Процессор

Flink не занимается активным ограничением количества потребляемых ресурсов ЦП. Однако он использует слоты обработки (см. подробности в разделе 3.1.3) для управления количеством задач, которые могут быть назначены рабочему процессу (TaskManager). TaskManager предоставляет определенное количество слотов, которые зарегистрированы в ResourceManager и управляются им. JobManager запрашивает один или несколько слотов для выполнения приложения. Каждый слот может обрабатывать один фрагмент приложения, одну параллельную задачу каждого оператора приложения. Следовательно, JobManager должен получить как минимум количество слотов, равное максимальному параллелизму операторов приложения[1]. Задачи выполняются как потоки в рабочем процессе (TaskManager) и занимают столько ресурсов ЦП, сколько им необходимо.

Количество слотов, предлагаемых TaskManager, контролируется ключом `taskmanager.numberOfTaskSlots` в файле конфигурации. По умолчанию – один слот на каждый диспетчер задач. Количество слотов обычно нужно настраивать только для автономных конфигураций, поскольку запуск Flink в диспетчере ресурсов кластера (YARN, Kubernetes, Mesos) позволяет легко запускать несколько диспетчеров задач (каждый с одним слотом) на вычислительный узел.

9.5.3. Основная память и сетевые буферы

Главный и рабочий процессы Flink имеют разные требования к памяти. Главный процесс в основном управляет вычислительными ресурсами (ResourceManager) и координирует выполнение приложений (JobManager), в то время как рабочий процесс берет на себя тяжелую работу и обрабатывает потенциально большие объемы данных.

Обычно главный процесс занимает умеренный объем памяти. По умолчанию он запускается с динамической памятью JVM объемом 1 Гб. Если главному процессу необходимо управлять несколькими приложениями или приложением с большим количеством операторов, вам может потребоваться увеличить размер кучи JVM с помощью ключа конфигурации `jobmanager.heap.size`.

Настройка памяти рабочего процесса немного сложнее, потому что есть несколько компонентов, связанных с разными типами памяти. Самый важ-

[1] Можно назначить операторы разным группам с разделением слотов и, таким образом, назначить их задачи отдельным слотам.

ный параметр – это размер кучи JVM, который задается ключом `taskmanager.heap.size`. Память кучи используется для всех объектов, включая среду выполнения TaskManager, операторы и функции приложения, а также актуальные рабочие данные. Состояние приложения, которое использует бэкенд состояния в памяти или файловой системе, также сохраняется в JVM. Обратите внимание, что одна задача потенциально может потреблять всю память кучи JVM, на которой она выполняется. Flink не гарантирует наличие и не предоставляет динамическую память для каждой задачи или слота. Конфигурации с одним слотом для каждого TaskManager обеспечивают лучшую изоляцию ресурсов и могут предотвратить влияние некорректно работающего приложения на несвязанные приложения. Если вы запускаете приложения с множеством зависимостей, объем памяти JVM вне кучи также может значительно увеличиться, поскольку в ней хранятся все классы TaskManager и пользовательского кода.

Помимо JVM, существуют два других основных потребителя памяти, сетевой стек Flink и RocksDB, когда она используется в качестве бэкенда состояния. Сетевой стек Flink основан на библиотеке Netty, которая выделяет сетевые буферы из собственной памяти (вне кучи). Flink требуется достаточное количество сетевых буферов, чтобы иметь возможность отправлять записи от одного рабочего процесса к другому. Количество буферов зависит от общего количества сетевых соединений между задачами оператора. Для двух операторов, связанных раздельным или широковещательным соединителем, количество сетевых буферов определяется произведением параллелизма передающего и принимающего операторов. Для приложений с несколькими шагами разделения эта квадратичная зависимость может быстро привести к поглощению значительного объема памяти, необходимой для передачи по сети.

Конфигурация Flink по умолчанию подходит только для распределенного развертывания в небольшом масштабе и требует корректировки для более масштабных систем. Если количество буферов не настроено должным образом, отправка задания завершится ошибкой `java.io.IOException: Insufficient number of network buffers` (недостаточное количество сетевых буферов). В этом случае вам следует выделить больше памяти для сетевого стека.

Объем памяти, выделяемой для сетевых буферов, настраивается с помощью ключа `taskmanager.network.memory.fraction`, который определяет долю размера кучи JVM, выделенную для сетевых буферов. По умолчанию используется 10 % размера кучи JVM. Поскольку буферы выделяются как память вне кучи, размер кучи JVM уменьшается на эту величину. Ключ конфигурации `taskmanager.memory.segment-size` определяет размер сетевого буфера, который по умолчанию составляет 32 Кб. Уменьшение размера сетевого буфера увеличивает количество буферов, но может снизить эффективность сетевого стека. Вы также можете указать минимальный (`taskmanager.network.memory.min`) и максимальный (`taskmanager.network.memory.max`) объем памяти, который используется для сетевых буферов (по умолчанию 64 Мб и 1 Гб соответственно), чтобы установить абсолютный пределы для относительного значения конфигурации.

RocksDB – еще один потребитель памяти, который необходимо учитывать при настройке памяти рабочего процесса. К сожалению, выяснить потре-

бление памяти RocksDB непросто, потому что оно зависит от количества ключевых состояний в приложении. Flink создает отдельный (встроенный) экземпляр RocksDB для каждой задачи оператора с ключом. В каждом экземпляре каждое отдельное состояние оператора хранится в отдельном семействе столбцов (или таблице). В конфигурации по умолчанию для каждого семейства столбцов требуется от 200 до 240 Мб памяти вне кучи. Вы можете настроить RocksDB и улучшить быстродействие с помощью множества параметров.

При настройке параметров памяти TaskManager вы должны определить размер кучи JVM, чтобы оставалось достаточно памяти вне кучи JVM (классы и метаданные) и RocksDB, если она настроена как бэкенд состояния. Сетевая память автоматически вычитается из настроенного размера кучи JVM. Имейте в виду, что некоторые диспетчеры ресурсов, такие как YARN, немедленно уничтожают контейнер, если он превышает выделенный бюджет памяти.

9.5.4. Дисковое хранилище

Рабочий процесс Flink по нескольким причинам хранит данные в локальной файловой системе, включая получение JAR-файлов приложения, запись файлов журнала и сохранение состояния приложения, если настроен бэкенд состояния RocksDB. С помощью конфигурационного ключа `io.tmp.dirs` вы можете указать один или несколько каталогов (разделенных двоеточиями), которые используются для хранения данных в локальной файловой системе. По умолчанию данные записываются во временный каталог по умолчанию, как это определено системным свойством Java `java.io.tmpdir` или `/tmp` в Linux и MacOS. Параметр `io.tmp.dirs` используется как значение по умолчанию для локального пути хранения большинства компонентов Flink. Однако эти пути также можно настроить индивидуально.

Убедитесь, что временные каталоги не очищаются автоматически

Некоторые дистрибутивы Linux периодически очищают временный каталог /tmp. Обязательно отключите это поведение или настройте другой каталог, если вы планируете постоянно запускать приложения Flink. В противном случае восстановление задания может остаться без метаданных, которые хранились во временном каталоге, и завершиться ошибкой.

Ключ `blob.storage.directory` настраивает каталог локального хранилища на сервере больших двоичных объектов, который используется для обмена файлами большого размера, такими как файлы JAR приложения. Ключ `env.log.dir` настраивает каталог, в который TaskManager записывает свои файлы журнала (по умолчанию каталог `./log` в настройке Flink). Наконец, бэкенд состояния RocksDB хранит состояние приложения в локальной файловой системе. Каталог настраивается с помощью ключа `state.backend.rocksdb.localdir`. Если каталог хранилища не настроен явно, RocksDB использует значение параметра `io.tmp.dirs`.

9.5.5. Контрольные точки и бэкенды состояния

Flink предлагает несколько опций для настройки того, как бэкенды состояния создают контрольные точки. Все параметры могут быть явно указаны в коде приложения, как описано в разделе 10.3. Однако вы также можете указать через файл конфигурации Flink настройки по умолчанию для кластера Flink, которые применяются, если не объявлены параметры для конкретного задания.

Важный выбор, влияющий на быстродействие приложения, – это бэкенд, который поддерживает его состояние. Вы можете определить бэкенд состояния по умолчанию для кластера с помощью ключа `state.backend`. Кроме того, вы можете включить асинхронную контрольную точку (`state.backend.async`) и инкрементную контрольную точку (`state.backend.incremental`). Некоторые бэкенды не поддерживают все параметры и могут игнорировать их. Вы также можете настроить корневые каталоги в удаленном хранилище, в которые записываются контрольные точки (`state.checkpoints.dir`) и точки сохранения (`state.savepoints.dir`).

Некоторые параметры контрольных точек зависят от конкретного бэкенда. Для бэкенда состояния RocksDB вы можете определить один или несколько путей, по которым RocksDB хранит свои локальные файлы (`state.backend.rocksdb.localdir`), и будет ли состояние таймера храниться в куче (по умолчанию) или в RocksDB (`state.backend.rocksdb.timer-service.factory`).

Наконец, вы можете включить и настроить локальное восстановление для кластера Flink по умолчанию[1]. Чтобы включить локальное восстановление, установите для параметра `state.backend.local-recovery` значение `true`. Также можно указать место хранения локальной копии состояния (`taskmanager.state.local.root-dirs`).

9.5.6. Безопасность

Платформы обработки данных являются чувствительными компонентами ИТ-инфраструктуры компании и должны быть защищены от несанкционированного использования и доступа к данным. Apache Flink поддерживает аутентификацию Kerberos и может быть настроен для шифрования всего сетевого обмена данными с помощью SSL.

Flink поддерживает интеграцию Kerberos с Hadoop и его компонентами (YARN, HDFS, HBase), ZooKeeper и Kafka. Вы можете включить и настроить поддержку Kerberos для каждой службы отдельно. Flink поддерживает два режима аутентификации – keytabs и токены делегирования Hadoop. Keytabs являются предпочтительным подходом, потому что токены истекают через некоторое время, что может вызвать проблемы для долгоработающих приложений потоковой обработки. Обратите внимание, что учетные данные привязаны к кластеру Flink, а не к работающему заданию; все приложения, работающие

[1] См. «Настройка восстановления» для получения подробной информации об этой функции.

в одном кластере, используют один и тот же токен аутентификации. Если вам нужно работать с другими учетными данными, вам следует запустить новый кластер. Обратитесь к документации Flink для получения подробных инструкций по включению и настройке аутентификации Kerberos.

Flink поддерживает взаимную аутентификацию коммуницирующих партнеров и шифрование внутреннего и внешнего сетевого обмена с помощью SSL. Для внутренней связи (вызовы RPC, передача данных и связь службы BLOB-объектов для распространения библиотек или других артефактов) все процессы Flink (Dispatcher, ResourceManager, JobManager и TaskManager) выполняют взаимную аутентификацию – отправители и получатели проверяют друг друга с помощью сертификата SSL. Сертификат действует как общий секрет и может быть встроен в контейнеры или прикреплен к настройке YARN.

Весь внешний обмен со службами Flink – отправка и управление приложениями и доступ к интерфейсу REST – происходит через конечные точки REST/HTTP[1]. Вы также можете включить шифрование SSL для этих подключений. Также можно включить взаимную аутентификацию. Однако рекомендуемый подход – установка и настройка выделенной прокси-службы, которая контролирует доступ к конечной точке REST. Причина в том, что прокси-сервисы предлагают больше вариантов аутентификации и конфигурации, чем Flink. Шифрование и аутентификация для связи с запрашиваемым состоянием пока не поддерживаются.

По умолчанию проверка подлинности и шифрование SSL отключены. Поскольку для установки требуется несколько шагов, таких как создание сертификатов, настройка TrustStores и KeyStores, а также настройка наборов шифров, мы отсылаем вас к официальной документации Flink. Документация также включает инструкции и советы для различных сред, таких как автономные кластеры, Kubernetes и YARN.

9.6. Заключение

В этой главе мы обсудили конфигурацию Flink в различных средах, включая настройку параметров высокой доступности. Мы объяснили, как включить поддержку различных файловых систем и как интегрировать их с Hadoop и его компонентами. Наконец, обсудили наиболее важные параметры конфигурации. Мы не предоставили исчерпывающего руководства по настройке; вместо этого отсылаем вас к официальной документации Apache Flink для получения полного списка и подробных описаний всех параметров конфигурации.

[1] В главе 10 обсуждается отправка заданий и интерфейс REST.

Глава 10

Работа с Flink и потоковыми приложениями

Потоковые приложения работают длительное время, а их рабочие нагрузки зачастую непредсказуемы. Задачи потоковой передачи нередко выполняются непрерывно в течение нескольких месяцев, поэтому их операционные потребности сильно отличаются от потребностей краткосрочных пакетных заданий. Рассмотрим сценарий, в котором вы обнаруживаете ошибку в развернутом приложении. Если ваше приложение является пакетным заданием, вы можете легко исправить ошибку в автономном режиме, а затем повторно развернуть новый код приложения после завершения текущего экземпляра задания. Но что, если ваша работа связана с длительной потоковой обработкой? Как выполнить реконфигурацию с минимальными усилиями, гарантируя при этом правильность результатов?

Если вы используете Flink, вам не о чем беспокоиться. Flink выполнит за вас всю тяжелую работу, чтобы вы могли легко контролировать и перенастраивать свои задания с минимальными усилиями, сохраняя при этом семантику состояния «ровно один раз». В этой главе мы представляем инструменты, которые Flink предлагает для работы и поддержки постоянно действующих потоковых приложений. Мы покажем вам, как собирать метрики и отслеживать ваши приложения, а также как сохранять согласованность результатов, когда вы хотите обновить код приложения или настроить ресурсы вашего приложения.

10.1. Запуск и управление потоковыми приложениями

Как и следовало ожидать, обслуживание потоковых приложений является более сложной задачей, чем обслуживание пакетных приложений. В то время как потоковые приложения сохраняют состояние и работают непрерывно,

пакетные приложения выполняются периодически. Перенастройку, масштабирование или обновление пакетного приложения можно выполнять между выполнениями, что намного проще, чем обновление приложения, которое непрерывно принимает, обрабатывает и отправляет данные.

Однако Apache Flink имеет множество функций, которые значительно упрощают обслуживание потоковых приложений. Большинство этих функций основано на точках сохранения[1]. Flink предоставляет следующие интерфейсы для мониторинга и управления своими главными и рабочими процессами и приложениями:

- клиент командной строки – это инструмент, используемый для отправки приложений и управления ими;
- REST API – это базовый интерфейс, который используется клиентом командной строки и веб-интерфейсом. Доступ к нему могут получить пользователи и сценарии, и он обеспечивает доступ ко всем системным показателям и показателям приложений, а также к конечным точкам для отправки приложений и управления ими;
- Web UI – это веб-интерфейс, который предоставляет подробную информацию и показатели о кластере Flink и запущенных приложениях. Он также предлагает базовые функции для отправки приложений и управления ими. Веб-интерфейс описан в разделе 10.4.1.

В этом разделе мы объясняем практические аспекты точек сохранения и обсуждаем, как запускать, останавливать, приостанавливать и возобновлять, масштабировать и обновлять потоковые приложения с отслеживанием состояния с помощью клиента командной строки и REST API Flink.

10.1.1. Точки сохранения

Точка сохранения в основном идентична контрольной точке – это согласованный и полный снимок состояния приложения. Однако жизненные циклы контрольных точек и точек сохранения различаются. Контрольные точки создаются автоматически, загружаются в случае сбоя и автоматически удаляются Flink по истечении срока давности (в зависимости от конфигурации приложения). Более того, контрольные точки автоматически удаляются при отмене приложения, если сохранение контрольных точек не задано в явном виде. Напротив, точки сохранения должны запускаться вручную пользователем или внешней службой и никогда не удаляются автоматически.

Точка сохранения – это каталог в постоянном хранилище данных. Он состоит из подкаталога, в котором хранятся файлы данных, содержащие состояния всех задач, и двоичного файла метаданных, который включает абсолютные пути ко всем файлам данных. Поскольку пути в файле метаданных являются абсолютными, перемещение точки сохранения на другой путь сделает ее непригодной для использования. Вот структура точки сохранения:

[1] См. главу 3, чтобы узнать о точках сохранения и о том, что с ними можно делать.

```
# Путь к корневому каталогу.
/savepoints/

# Путь к отдельной точке сохранения.
/savepoints/savepoint-:shortjobid-:savepointid/

# Двоичные метаданные точки сохранения.
/savepoints/savepoint-:shortjobid-:savepointid/_metadata

# Сохраненное состояние оператора.
/savepoints/savepoint-:shortjobid-:savepointid/:xxx
```

10.1.2. Управление приложениями с помощью клиента командной строки

Клиент командной строки Flink предоставляет возможности запуска, остановки и управления приложениями Flink. Он считывает свою конфигурацию из файла **`./conf/flink-conf.yaml`** (см. раздел 9.5). Вы можете вызвать его из корневого каталога установки Flink с помощью команды **`./bin/flink`**.

При запуске без дополнительных параметров клиент выводит справочное сообщение.

Клиент командной строки в Windows

Клиент командной строки основан на сценарии bash. Поэтому он не работает с командной строкой Windows. Сценарий **`./bin/flink.bat`** для командной строки Windows предоставляет лишь очень ограниченную функциональность. Если вы пользователь Windows, мы рекомендуем использовать обычный клиент командной строки и запускать его на WSL или Cygwin.

10.1.2.1. Запуск приложения

Вы можете запустить приложение с помощью следующей команды для клиента командной строки:

```
./bin/flink run ~/myApp.jar
```

Приведенная выше команда запускает приложение из метода `main()` класса, на который имеется ссылка в свойстве `program-class` файла **`META-INF/MANIFEST.MF`**, без передачи каких-либо аргументов приложению. Клиент отправляет файл JAR главному процессу, который распространяет его на рабочие узлы.

Вы можете передать аргументы методу `main()` приложения, добавив их в конце команды:

```
./bin/flink run ~/myApp.jar my-arg1 my-arg2 my-arg3
```

По умолчанию клиент не закрывается после отправки приложения, а ожидает его завершения. Вы можете отправить приложение в автономном режиме с флагом `-d`, как показано здесь:

```
./bin/flink run -d ~/myApp.jar
```

Вместо того чтобы ждать завершения приложения, клиент возвращается в терминал и выводит JobID отправленного задания. JobID используется для указания задания при создании точки сохранения, отмене или изменении масштаба приложения. Вы можете указать параллелизм приложения по умолчанию с помощью флага `-p`:

```
./bin/flink run -p 16 ~/myApp.jar
```

Приведенная выше команда устанавливает параллелизм среды выполнения по умолчанию равным 16. Параллелизм среды выполнения по умолчанию перезаписывается всеми настройками, явно указанными в исходном коде приложения, – параллелизмом, который определяется путем вызова `setParallelism()` в `StreamExecutionEnvironment` или оператором, который имеет приоритет над значением по умолчанию.

Если в файле манифеста JAR-файла вашего приложения не указан класс записи, вы можете указать этот класс с помощью параметра `-c`:

```
./bin/flink run -c my.app.MainClass ~/myApp.jar
```

Клиент попытается запустить статический метод `main()` класса `my.app.MainClass`.

По умолчанию клиент отправляет приложение мастеру Flink, указанному в файле **`./conf/flink-conf.yaml`** (см. раздел 9.5). Вы можете подать заявку в конкретный главный процесс, используя флаг `-m`:

```
./bin/flink run -m myMasterHost:9876 ~/myApp.jar
```

Эта команда отправляет приложение главному процессу, который работает на хосте `myMasterHost`, порт `9876`.

Обратите внимание, что состояние приложения будет пустым, если вы запустите его в первый раз или не предоставите точку сохранения или контрольную точку для инициализации состояния. В этом случае некоторые операторы с сохранением состояния запускают специальную логику для инициализации своего состояния. Например, источнику Kafka необходимо выбрать смещения разделов, из которых он потребляет тему, если восстановленные позиции чтения недоступны.

10.1.2.2. Список запущенных приложений

Для любого действия, которое вы хотите применить к работающему заданию, вам необходимо предоставить идентификатор JobID, указывающий на приложение. Идентификатор задания можно получить из веб-интерфейса, REST API или с помощью клиента командной строки. Клиент распечатывает список всех запущенных заданий, включая их JobID, когда вы запускаете следующую команду:

```
./bin/flink list -r
Waiting for response...
------------------ Running/Restarting Jobs -------------------
```

```
17.10.2018 21:13:14 : bc0b2ad61ecd4a615d92ce25390f61ad :
Socket Window WordCount (RUNNING)
--------------------------------------------------------------
```

В этом примере JobID – `bc0b2ad61ecd4a615d92ce25390f61ad`.

10.1.2.3. Создание и сброс точки сохранения

Точку сохранения для работающего приложения можно установить при помощи клиента командной строки следующим образом:

```
./bin/flink savepoint <jobId> [savepointPath]
```

Команда запускает создание точки сохранения для задания с указанным JobID. Если вы явно укажете путь к точке сохранения, она будет сохранена в указанном каталоге. В противном случае используется каталог точки сохранения по умолчанию, настроенный в файле **`flink-conf.yaml`**.

Чтобы активировать точку сохранения для задания `bc0b2ad61ecd4a615d92ce25390f61ad` и сохранить ее в каталоге **`hdfs:///xxx:50070/savepoints`**, мы вызываем клиент командной строки:

```
./bin/flink savepoint bc0b2ad61ecd4a615d92ce25390f61ad \
hdfs:///xxx:50070/savepoints
Triggering savepoint for job bc0b2ad61ecd4a615d92ce25390f61ad.
Waiting for response...
Savepoint completed.
Path: hdfs:///xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5ccef8
You can resume your program from this savepoint with the run command.
```

Точки сохранения могут занимать значительный объем места и не удаляются автоматически. Их необходимо удалить вручную, чтобы освободить занимаемое хранилище. Точка сохранения удаляется командой:

```
./bin/flink savepoint -d <savepointPath>
```

Чтобы удалить точку сохранения, сработавшую ранее, вызовите команду:

```
./bin/flink savepoint -d \
hdfs:///xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5ccef8
Disposing savepoint 'hdfs:///xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5ccef8'.
Waiting for response...
Savepoint 'hdfs:///xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5ccef8' disposed.
```

Удаление точки сохранения

Вы не должны удалять точку сохранения до завершения другой контрольной точки или точки сохранения. Поскольку точки сохранения обрабатываются системой аналогично обычным контрольным точкам, операторы также получают уведомления о завершении контрольных точек, включая точки сохранения, и выполняют соответствующие действия. Например, приемники транзакций фиксируют изменения во внешних системах после завершения точки сохранения. Чтобы гарантировать вывод «ровно один раз», Flink должен восстановиться с последней завершенной контрольной точки или точки сохранения.

Восстановление после сбоя завершится ошибкой, если Flink попытается восстановиться из удаленной точки сохранения. После завершения новой контрольной точки (или точки сохранения) вы можете безопасно удалить предыдущую точку сохранения.

10.1.2.4. Отмена приложения

Работу приложения можно прекратить двумя способами: с точкой сохранения или без нее. Чтобы отменить запущенное приложение, не создавая точку сохранения, выполните следующую команду:

```
./bin/flink cancel <jobId>
```

Чтобы получить точку сохранения перед отменой работающего приложения, добавьте флаг `-s` к команде отмены:

```
./bin/flink cancel -s [savepointPath] <jobId>
```

Если вы не укажете путь к точке сохранения, будет использоваться каталог точки сохранения по умолчанию, настроенный в файле **`./conf/flink-conf.yaml`** (см. раздел 9.5). Команда не выполняется, если папка точки сохранения не указана явно в команде и не доступна из конфигурации. Например, чтобы отменить приложение с JobID `bc0b2ad61ecd4a615d92ce25390f61ad` и сохранить точку сохранения в **hdfs:///xxx:50070/savepoints**, выполните команду:

```
./bin/flink cancel -s \
hdfs:///xxx:50070/savepoints d5fdaff43022954f5f02fcd8f25ef855
Cancelling job bc0b2ad61ecd4a615d92ce25390f61ad
with savepoint to hdfs:///xxx:50070/savepoints.
Cancelled job bc0b2ad61ecd4a615d92ce25390f61ad.
Savepoint stored in hdfs:///xxx:50070/savepoints/savepoint-bc0b2a-d08de07fbb10.
```

При отмене приложения возможен сбой

Обратите внимание, что задание будет продолжать выполняться, если не удалось создать точку сохранения. Вам нужно будет сделать еще одну попытку отменить задание.

10.1.2.5. Запуск приложения из точки сохранения

Запустить приложение из точки сохранения довольно просто. Все, что вам нужно сделать, – это запустить приложение с помощью команды **`run`** и дополнительно указать путь к точке сохранения с параметром `-s`:

```
./bin/flink run -s <savepointPath> [options] <jobJar> [arguments]
```

Когда задание запускается, Flink сопоставляет отдельные снимки состояния точки сохранения со всеми состояниями запущенного приложения. Это сопоставление выполняется в два этапа. Сначала Flink сравнивает уникальные идентификаторы операторов точки сохранения и операторов приложения, а затем сопоставляет для каждого оператора идентификаторы состояния точки сохранения и приложения (подробности см. в разделе 3.5.5).

Вы должны определить уникальные идентификаторы оператора

Если вы не назначаете своим операторам уникальные идентификаторы с помощью метода `uid()`, Flink назначает идентификаторы по умолчанию, представляющие собой хеш-значения, которые зависят от типа оператора и всех предыдущих операторов. Поскольку невозможно изменить идентификаторы в точке сохранения, у вас будет меньше возможностей для обновления и развития вашего приложения, если вы не назначите вручную идентификаторы операторов с помощью `uid()`.

Как уже упоминалось, приложение может быть запущено из точки сохранения, только если оно совместимо с точкой сохранения. Немодифицированное приложение всегда можно перезапустить с его точки сохранения. Однако, если перезапущенное приложение не идентично приложению, из которого была взята точка сохранения, следует рассмотреть три случая:

1) если вы *добавили новое состояние* в приложение или изменили уникальный идентификатор оператора с учетом состояния, Flink не найдет соответствующий снимок состояния в точке сохранения. В этом случае новое состояние инициализируется как пустое;
2) если вы *удалили состояние* из приложения или изменили уникальный идентификатор оператора с учетом состояния, в точке сохранения есть состояние, которое не может быть сопоставлено с приложением. В этом случае Flink не запускает приложение, чтобы не потерять состояние в точке сохранения. Вы можете отключить эту проверку безопасности, добавив параметр `-n` к команде запуска;
3) если вы *изменили состояние* в приложении – изменили примитив состояния или изменили тип данных состояния – приложение не запустится. Это означает, что вы не сможете легко развивать тип данных состояния в своем приложении, если только вы не спроектируете свое приложение с учетом эволюции состояния с самого начала. Сообщество Flink в настоящее время работает над улучшением поддержки эволюции состояния. (см. раздел 7.5.3).

10.1.2.6. Увеличение и уменьшение масштаба приложения

Вы можете без труда уменьшить или увеличить параллелизм приложения. Вам нужно создать точку сохранения, отменить приложение и перезапустить его с настроенным параллелизмом из точки сохранения. Состояние приложения автоматически перераспределяется между большим или меньшим количеством параллельных операторских задач. В разделе 3.4.4 подробно рассказано о том, как масштабируются различные типы состояния оператора и ключевые состояния. Однако следует учитывать несколько моментов.

Если вам требуются результаты «ровно один раз», вы должны создать точку сохранения и остановить приложение с помощью встроенной команды **`savepoint-and-cancel`**. Это предотвращает завершение другой контрольной точки после точки сохранения, что приведет к срабатыванию приемников ровно один раз для передачи данных после точки сохранения.

Как говорилось в разделе 5.3, параллелизм приложения и его операторов можно указать по-разному. По умолчанию операторы работают с паралле-

лизмом их среды выполнения `StreamExecutionEnvironment`. Параллелизм по умолчанию можно указать при запуске приложения (например, с помощью параметра `-p` в клиенте командной строки). Если вы реализуете приложение так, что параллелизм его операторов зависит от параллелизма среды по умолчанию, вы можете просто масштабировать приложение, запустив его из того же файла JAR и указав новый параллелизм. Однако, если вы жестко запрограммировали параллелизм в `StreamExecutionEnvironment` или в некоторых операторах, вам может потребоваться скорректировать исходный код, перекомпилировать и переупаковать приложение перед его отправкой на выполнение.

Если параллелизм вашего приложения зависит от параллелизма среды по умолчанию, Flink предоставляет атомарную команду изменения масштаба, которая берет точку сохранения, останавливает приложение и перезапускает его с новым параллелизмом по умолчанию:

```
./bin/flink modify <jobId> -p <newParallelism>
```

Чтобы изменить масштаб приложения с jobId `bc0b2ad61ecd4a615d92ce25390f61ad` до параллелизма 16, выполните команду:

```
./bin/flink modify bc0b2ad61ecd4a615d92ce25390f61ad -p 16
Modify job bc0b2ad61ecd4a615d92ce25390f61ad.
Rescaled job bc0b2ad61ecd4a615d92ce25390f61ad. Its new parallelism is 16.
```

Как сказано в разделе 3.4.4, Flink распределяет ключевое состояние с детализацией по так называемым группам ключей. Следовательно, количество таких групп оператора с отслеживанием состояния определяет его максимальный параллелизм. Количество групп ключей настраивается для каждого оператора с помощью метода `setMaxParallelism()` (см. раздел 7.3.2).

10.1.3. Управление приложениями с помощью REST API

REST API доступен напрямую пользователям или скриптам и предоставляет информацию о кластере Flink и его приложениях, включая показатели, а также конечные точки для отправки и управления приложениями. Flink обслуживает REST API и веб-интерфейс с одного и того же веб-сервера, который работает как часть процесса Dispatcher. По умолчанию оба компонента доступны на порту 8081. Вы можете настроить другой порт в файле **`./conf/flink-conf.yaml`** с помощью ключа конфигурации `rest.port`. Значение `-1` отключает REST API и веб-интерфейс.

Распространенным инструментом командной строки для взаимодействия с REST API является `curl`. Типичная команда REST `curl` выглядит так:

```
curl -X <HTTP-Method> [-d <parameters>] http://hostname:port/v1/<REST-point>
```

Здесь `v1` указывает версию REST API. Flink 1.7 предоставляет первую версию (v1) API. Предположим, что вы запускаете локальную версию Flink, ко-

торая предоставляет свой REST API на порт 8081. Следующая команда отправляет запрос GET в конечную точку REST **/overview**:

```
curl -X GET http://localhost:8081/v1/overview
```

Команда возвращает некоторую базовую информацию о кластере, такую как версия Flink, количество диспетчеров задач, слотов и запущенных, завершенных, отмененных или неудачных заданий:

```
{
 "taskmanagers":2,
 "slots-total":8,
 "slots-available":6,
 "jobs-running":1,
 "jobs-finished":2,
 "jobs-cancelled":1,
 "jobs-failed":0,
 "flink-version":"1.7.1",
 "flink-commit":"89eafb4"
}
```

Далее мы перечисляем и кратко описываем наиболее важные вызовы REST. Обратитесь к официальной документации Apache Flink для получения полного списка поддерживаемых вызовов. Раздел 10.1.2 предоставляет более подробную информацию о некоторых операциях, таких как обновление или масштабирование приложения.

10.1.3.1. Управление и мониторинг кластера Flink

REST API предоставляет конечные точки для запроса информации о работающем кластере и для завершения его работы. В табл. 10.1, 10.2 и 10.3 показаны запросы REST для получения информации о кластере Flink, например количество слотов задач, запущенные и завершенные задания, конфигурация JobManager или список всех подключенных процессов TaskManager.

Таблица 10.1. Запрос REST для получения базовой информации о кластере

Запрос	GET /overview
Ответ	Базовая информация о кластере, как показано выше

Таблица 10.2. Запрос REST для получения конфигурации JobManager

Запрос	GET /jobmanager/config
Ответ	Возвращает конфигурацию JobManager, как описано в ./conf/flink-conf.yaml

Таблица 10.3. Запрос REST для вывода списка всех подключенных диспетчеров задач

Запрос	GET /taskmanagers
Ответ	Возвращает список всех процессов TaskManagwer, включая их идентификаторы и основную информацию, такую как статистика памяти и порты подключения

В табл. 10.4 показан запрос REST для перечисления всех показателей, собранных для JobManager.

Таблица 10.4. Запрос REST для вывода списка доступных метрик JobManager

Запрос	GET /jobmanager/metrics
Ответ	Возвращает список доступных метрик JobManager

Чтобы получить одну или несколько метрик JobManager, добавьте в запрос параметр `get` с перечислением нужных метрик:

```
curl -X GET http://hostname:port/v1/jobmanager/metrics?get=metric1,metric2
```

В табл. 10.5 показан запрос REST для перечисления всех метрик, собранных для процессов TaskManager.

Таблица 10.5. Запрос REST для вывода списка доступных метрик TaskManager

Запрос	GET /taskmanagers/<tmId>/metrics
Параметры	tmId: идентификатор связанного процесса TaskManager
Ответ	Возвращает список метрик для выбранного процесса TaskManager

Чтобы получить одну или несколько метрик TaskManager, добавьте в запрос параметр `get` со всеми запрошенными метриками:

```
curl -X GET http://hostname:port/v1/taskmanagers/<tmId>/metrics?get=metric1
```

Вы также можете завершить работу кластера с помощью вызова REST, который показан в табл. 10.6.

Таблица 10.6. Запрос REST на выключение кластера

Запрос	GET /jobmanager/metrics
Действие	Завершает работу кластера Flink. Обратите внимание, что в автономном режиме будет завершен только главный процесс, а рабочие процессы продолжат работу

10.1.3.2. Управление и мониторинг приложений Flink

REST API также можно использовать для управления и мониторинга приложений Flink. Чтобы запустить приложение, сначала необходимо загрузить файл JAR приложения в кластер. В табл. 10.7, 10.8 и 10.9 показаны конечные точки REST для управления этими файлами JAR.

Таблица 10.7. Запрос REST для загрузки файла JAR

Запрос	POST /jars/upload
Параметры	Файл должен быть отправлен как составные данные
Действие	Загрузка JAR-файла в кластер
Ответ	Место хранения загруженного JAR-файла

Команда `curl` для загрузки файла JAR:

```
curl -X POST -H "Expect:" -F "jarfile=@path/to/flink-job.jar" \
http://hostname:port/v1/jars/upload
```

Таблица 10.8. Запрос REST для вывода списка всех загруженных файлов JAR

Запрос	GET /jars
Ответ	Список всех загруженных файлов JAR. Список включает внутренний идентификатор файла JAR, его исходное имя и время, когда он был загружен

Таблица 10.9. Запрос REST на удаление файла JAR

Запрос	DELETE /jars/<jarId>
Параметры	jarId: идентификатор JAR-файла, предоставленный командой вывода списка JAR-файлов
Действие	Удаление JAR-файла, на который ссылается идентификатор

Приложение запускается из загруженного файла JAR с помощью вызова REST, показанного в табл. 10.10.

Таблица 10.10. Запрос REST для запуска приложения

Запрос	DELETE /jars/<jarId>
Параметры	jarId: идентификатор JAR-файла, из которого запускается приложение. Вы можете передать дополнительные параметры, такие как аргументы задания, класс входа, параллелизм по умолчанию, путь к точке сохранения и флаг allow-nonrestored-state в виде объекта JSON
Действие	Запускает приложение, определенное файлом JAR (и классом записи) с предоставленными параметрами. Если указан путь к точке сохранения, состояние приложения инициализируется из точки сохранения
Ответ	Идентификатор задания запущенного приложения

Команда `curl` для запуска приложения с параллелизмом по умолчанию 4:

```
curl -d '{"parallelism":"4"}' -X POST \
http://localhost:8081/v1/jars/43e844ef-382f-45c3-aa2f-00549acd961e_App.jar/run
```

Таблицы 10.11, 10.12 и 10.13 показывают, как управлять запущенными приложениями с помощью REST API.

Таблица 10.11. Запрос REST для вывода списка всех приложений

Запрос	GET /jobs
Ответ	Перечисляет идентификаторы заданий всех запущенных приложений и идентификаторы заданий последних неудачных, отмененных и завершенных приложений

Таблица 10.12. Запрос REST для отображения деталей приложения

Запрос	GET /jobs/<jobId>
Ответ	Базовая статистика, такая как имя приложения, время начала (и время окончания), а также информация о выполненных задачах, включая количество принятых и отправленных записей и байтов

Таблица 10.13. Запрос REST для отмены приложения

Запрос	PATCH /jobs/<jobId>
Параметры	jobId: идентификатор задания, предоставленный командой вывода списка приложений
Действие	Прекращение работы приложения

REST API также предоставляет более подробную информацию о следующих аспектах приложения:

- плане выполнения операторов приложения;
- конфигурации приложения;
- собранных метриках приложения на разных уровнях детализации;
- метриках контрольных точек;
- показателях противодавления;
- исключениях, которые привели к сбою приложения.

Ознакомьтесь с официальной документацией, чтобы узнать, как получить доступ к этой информации.

Вы также можете создать точку сохранения работающего приложения с помощью вызова REST, который показан в табл. 10.14.

Таблица 10.14. Запрос REST для создания точки сохранения приложения

Запрос	POST /jobs/<jobId>/savepoints
Параметры	jobId: идентификатор задания, предоставленный командой вывода списка приложений
Действие	Создает точку сохранения приложения
Ответ	Идентификатор запроса, чтобы проверить, сработал ли триггер точки сохранения

Команда `curl` для запуска точки сохранения без отмены:

```
curl -d '{"target-directory":"file:///savepoints", "cancel-job":"false"}'\
-X POST http://localhost:8081/v1/jobs/e99cdb41b422631c8ee2218caa6af1cc/savepoints
{"request-id":"ebde90836b8b9dc2da90e9e7655f4179"}
```

При отмене приложения с помощью точки сохранения возможен сбой

Запрос на отмену приложения будет успешным только в том случае, если была успешно создана точка сохранения. Приложение продолжит работу, если команда точки сохранения не удалась.

Чтобы проверить, был ли успешным запрос с идентификатором `ebde90836b8b9dc2da90e9e7655f4179`, и получить путь к точке сохранения, выполните команду:

```
curl -X GET http://localhost:8081/v1/jobs/e99cdb41b422631c8ee2218caa6af1cc/\
savepoints/ebde90836b8b9dc2da90e9e7655f4179
{"status":{"id":"COMPLETED"}
"operation":{"location":"file:///savepoints/savepoint-e99cdb-34410597dec0"}}
```

Для удаления точки сохранения используйте вызов REST, показанный в табл. 10.15.

Таблица 10.15. Запрос REST на удаление точки сохранения

Запрос	POST /savepoint-disposal
Параметры	Путь к удаляемой точке сохранения должен быть предоставлен в качестве параметра в объекте JSON
Действие	Удаляет точку сохранения
Ответ	Идентификатор запроса для проверки, успешно ли была удалена точка сохранения

Чтобы удалить точку сохранения с помощью `curl`, запустите:

```
curl -d '{"savepoint-path":"file:///savepoints/savepoint-e99cdb-34410597"}'\
-X POST http://localhost:8081/v1/savepoint-disposal
{"request-id":"217a4ffe935ceac2c281bdded76729d6"}
```

В табл. 10.16 показан вызов REST для изменения масштаба приложения.

Таблица 10.16. Запрос REST для изменения масштаба приложения

Запрос	PATCH /jobs/<jobID>/rescaling
Параметры	Идентификатор задания, предоставленный командой получения списка приложений. Кроме того, вам необходимо предоставить новый параллелизм приложения в качестве параметра URL
Действие	Принимает точку сохранения, отменяет приложение и перезапускает его с новым параллелизмом по умолчанию из точки сохранения
Ответ	Идентификатор запроса, чтобы проверить, был ли запрос изменения масштаба успешным или нет

Чтобы изменить масштаб приложения с помощью `curl` до нового параллелизма по умолчанию, равного 16 потокам, выполните команду:

```
curl -X PATCH
http://localhost:8081/v1/jobs/129ced9aacf1618ebca0ba81a4b222c6/rescaling\
?parallelism=16
{"request-id":"39584c2f742c3594776653f27833e3eb"}
```


Приложение может не масштабироваться

Приложение продолжит работу с исходным параллелизмом, если точка сохранения не создана. Вы можете проверить статус запроса на изменение масштаба, используя идентификатор запроса.

10.1.4. Объединение и развертывание приложений в контейнерах

До сих пор мы говорили о том, как запустить приложение на работающем кластере Flink. Мы называем это развертыванием в режиме фреймворка.

В разделе 3.1.2 мы кратко упомянули альтернативу – режим библиотеки, который не требует наличия запущенного кластера Flink для отправки задания.

В режиме библиотеки приложение объединяется в образ Docker, который также содержит необходимые двоичные файлы Flink. Образ можно запустить двумя способами – как контейнер JobMaster или как контейнер TaskManager. Когда образ развертывается как JobMaster, контейнер запускает главный процесс Flink, который немедленно выбирает связанное приложение для запуска. Контейнер TaskManager регистрируется в JobMaster и предлагает свои слоты обработки. Как только становится доступно достаточно слотов, контейнер JobMaster развертывает приложение для выполнения.

Библиотечный стиль запуска приложений Flink напоминает развертывание микросервисов в контейнерной среде. При развертывании на платформе оркестровки контейнеров, такой как Kubernetes, платформа перезапускает отказавшие контейнеры. В этом разделе мы описываем, как создать образ Docker для конкретного задания и как развернуть связанное приложение библиотечного типа на Kubernetes.

10.1.4.1. Создание образа Flink Docker для конкретного задания

Apache Flink предоставляет сценарий для создания образов Flink Docker для конкретных заданий. Сценарий включен в набор исходного кода и размещен в репозитории Git. Он не является частью двоичных дистрибутивов Flink.

Вы можете либо загрузить и распаковать исходный код Flink, либо клонировать репозиторий Git. Начиная с базовой папки дистрибутива, скрипт находится по адресу **`./flink-container/docker/build.sh`**.

Сценарий сборки создает и регистрирует новый образ Docker, основанный на образе Java Alpine, минимальном базовом образе, предоставляющем Java-машину. Для скрипта требуются следующие параметры:

- путь к архиву Flink;
- путь к JAR-файлу приложения;
- название нового изображения.

Чтобы создать образ с Flink 1.7.1, который содержит примеры приложений из этой книги, выполните следующий сценарий:

```
cd ./flink-container/docker
./build.sh \
    --from-archive <path-to-Flink-1.7.1-archive> \
    --job-jar <path-to-example-apps-JAR-file> \
    --image-name flink-book-apps
```

Если вы запустите команду Docker `images` после завершения скрипта сборки, вы должны увидеть новый образ Docker под названием `flink-book-apps`.

Каталог **`./flink-container/docker`** также содержит файл **`docker-compose.yml`** для развертывания приложения Flink с помощью `docker-compose`.

Если вы запустите следующую команду, на одном главном и трех рабочих контейнерах в Docker будет развернут пример приложения из раздела 1.4:

```
FLINK_DOCKER_IMAGE_NAME=flink-book-jobs \
FLINK_JOB=io.github.streamingwithflink.chapter1.AverageSensorReadings \
DEFAULT_PARALLELISM=3 \
docker-compose up -d
```

Вы можете отслеживать приложение и управлять им при помощи веб-интерфейса, доступного по адресу **http://localhost:8081**.

10.1.4.2. Запуск образа Docker для определенного задания в Kubernetes

Запуск образа Docker для определенного задания в Kubernetes очень похож на запуск кластера Flink в Kubernetes, как описано в разделе 9.1.4. В принципе, вам нужно только настроить файлы YAML, которые описывают ваши развертывания, чтобы использовать образ, содержащий код задания, и настроить его для автоматического запуска задания при запуске контейнера.

Flink предоставляет шаблоны для файлов YAML, представленных в дистрибутиве исходного кода или размещенных в репозитории Git проекта. Начиная с базового каталога, шаблоны находятся в **`./flink-container/kubernetes`**.

В каталоге находятся два файла шаблона:

1) `job-cluster-job.yaml.template` настраивает главный контейнер как задание Kubernetes;
2) `task-manager-deployment.yaml.template` настраивает рабочий контейнер как развертывание Kubernetes.

Оба файла шаблона содержат заполнители, которые необходимо заменить фактическими значениями:

- `${FLINK_IMAGE_NAME}`: имя образа для конкретного задания;
- `${FLINK_JOB}`: основной класс запускаемого задания;
- `${FLINK_JOB_PARALLELISM}`: степень параллелизма для задания. Этот параметр также определяет количество запущенных рабочих контейнеров.

Как видите, это те же параметры, что мы использовали при развертывании образа для конкретного задания с помощью `docker-compose`. В каталоге также находится YAML-файл **`job-cluster-service.yaml`**, который определяет службу Kubernetes. После того как вы скопировали файлы шаблонов и настроили необходимые значения, можете развернуть приложение в Kubernetes, как и раньше, с помощью `kubectl`:

```
kubectl create -f job-cluster-service.yaml
kubectl create -f job-cluster-job.yaml
kubectl create -f task-manager-deployment.yaml
```

Запуск образов для определенных заданий на Minikube

Запуск образа для определенного задания в кластере Minikube требует на несколько шагов больше, чем описано в разделе 9.1.4. Проблема в том, что Minikube пытается получить пользовательский образ из общедоступного реестра образов Docker, а не из локального реестра Docker вашей машины.

Однако вы можете настроить Docker для развертывания его образов в собственном реестре Minikube, выполнив следующую команду:

```
eval $(minikube docker-env)
```

Все образы, которые вы создаете впоследствии в этой оболочке, развертываются в реестре образов Minikube. Minikube должен быть запущен.
Более того, вам необходимо присвоить параметру `ImagePullPolicy` в файлах YAML значение `Never`, чтобы гарантировать, что Minikube извлекает изображение из собственного реестра.

После запуска контейнеров для определенных заданий вы можете рассматривать кластер как обычный кластер Flink, как описано в разделе 9.1.4.

10.2. Управление планированием задач

Приложения Flink выполняются параллельно за счет распараллеливания операторов на задачи и распределения этих задач по рабочим процессам в кластере. Как и во многих других распределенных системах, быстродействие приложения Flink во многом зависит от того, как запланированы задачи. Быстродействие приложения может в значительной степени зависеть от рабочего процесса, которому назначена задача, от расположенных рядом с этой задачей других задач и количества задач, назначенных рабочему процессу.

В разделе 3.1.3 мы описали, как Flink назначает задачи слотам и как он использует цепочку задач для снижения затрат на локальный обмен данными. В этом разделе мы обсудим, как вы можете настроить поведение по умолчанию и управлять цепочкой задач и назначением задач по слотам для повышения быстродействия ваших приложений.

10.2.1. Управление цепочкой задач

Цепочка задач объединяет параллельные задачи двух или более операторов в одну задачу, которая выполняется одним потоком. Объединенные задачи обмениваются записями посредством вызовов методов и, таким образом, практически не требуют затрат на связь. Поскольку цепочка задач улучшает быстродействие большинства приложений, она включена по умолчанию в Flink.

Однако некоторые приложения не получают выгоду от цепочки задач. Одна из причин – иногда лучше разорвать цепочку вычислительно дорогостоящих функций, чтобы выполнять их на разных слотах обработки. Вы можете полностью отключить цепочку задач для приложения через `StreamExecutionEnvironment`:

```
StreamExecutionEnvironment.disableOperatorChaining()
```

Помимо отключения цепочки для всего приложения, вы также можете управлять цепочкой отдельных операторов. Чтобы отключить цепочку для

определенного оператора, вы можете вызвать его метод `disableChaining()`. Это предотвратит привязку задач оператора к предыдущим и последующим задачам (пример 10.1).

Пример 10.1 ❖ Отключение цепочки задач для оператора

```
val input: DataStream[X] = ...
val result: DataStream[Y] = input
  .filter(new Filter1())
  .map(new Map1())
  // Отключение цепочки для Map2.
  .map(new Map2()).disableChaining()
  .filter(new Filter2())
```

Код в примере 10.1 приводит к трем задачам – связанной задаче для `Filter1` и `Map1`, отдельной задаче для `Map2` и задаче для `Filter2`, которую нельзя связывать с `Map2`.

Также можно начать новую цепочку с помощью оператора, вызвав его метод `startNewChain()` (пример 10.2). Задачи оператора не будут привязаны к предыдущим задачам, но будут привязаны к последующим задачам, если выполнены требования для объединения.

Пример 10.2 ❖ Начать новую цепочку задач с оператором

```
val input: DataStream[X] = ...
val result: DataStream[Y] = input
  .filter(new Filter1())
  .map(new Map1())
  // Запуск новой цепочки для Map2 и Filter.
  .map(new Map2()).startNewChain()
  .filter(new Filter2())
```

В примере 10.2 созданы две связанные задачи: одна задача для `Filter1` и `Map1`, а другая задача для `Map2` и `Filter2`. Обратите внимание, что новая связанная задача начинается с оператора, для которого вызывается метод `startNewChain()` – в нашем примере это `Map2`.

10.2.2. Определение групп совместного использования слотов

Стратегия планирования задач Flink по умолчанию назначает полный фрагмент программы – до одной задачи каждого оператора приложения в один слот обработки[1]. В зависимости от сложности приложения и вычислительных затрат операторов эта стратегия по умолчанию может перегружать слот обработки. Механизм Flink для ручного управления назначением задач слотам – это группы с разделением слотов.

[1] Поведение расписания по умолчанию было объяснено в главе 3.

Каждый оператор является членом группы совместного использования слотов. Все задачи операторов, входящих в одну группу, обрабатываются одними и теми же слотами. В группе с совместным использованием слотов задачи назначаются слотам, как описано в разделе 3.1.3, – каждый слот обрабатывает до одной задачи каждого оператора, который является членом группы. Следовательно, группе с совместным использованием слотов требуется столько слотов обработки, сколько составляет максимальный параллелизм ее операторов. Задачи операторов, находящихся в разных группах, не выполняются одними и теми же слотами.

По умолчанию каждый оператор входит в группу разделения слотов `default`. Для каждого оператора вы можете явно указать его группу разделения слотов с помощью метода `slotSharingGroup(String)`. Оператор наследует группу совместного использования слота своих операторов ввода, если все они являются членами одной группы. Если операторы ввода находятся в разных группах, оператор находится в группе `default`. В примере 10.3 показано, как указать группы с разделением слотов в приложении Flink DataStream.

Пример 10.3 ❖ Управление планированием задач с помощью групп с разделением слотов

```
// Слотовая группа "зеленая" ("green").
val a: DataStream[A] = env.createInput(...)
  .slotSharingGroup("green")
  .setParallelism(4)
val b: DataStream[B] = a.map(...)
  // Слотовая группа "green" унаследована от a.
  .setParallelism(4)

// Слотовая группа "желтая" ("yellow").
val c: DataStream[C] = env.createInput(...)
  .slotSharingGroup("yellow")
  .setParallelism(2)

// Слотовая группа "синяя" ("blue").
val d: DataStream[D] = b.connect(c.broadcast(...)).process(...)
  .slotSharingGroup("blue")
  .setParallelism(4)
val e = d.addSink()
  // Слотовая группа "blue" унаследована от d.
  .setParallelism(2)
```

Приложение в примере 10.3 состоит из пяти операторов, двух источников, двух промежуточных операторов и оператора приемника. Операторы распределены по трем группам с разделением слотов: зеленой, желтой и синей. На рис. 10.1 показан JobGraph приложения и то, как его задачи отображаются в слоты обработки.

Приложению требуется 10 слотов обработки. Для синих и зеленых групп с совместным использованием слотов требуется по четыре слота для каждой из-за максимального параллелизма назначенных им операторов. Для желтой группы с разделением слотов требуется два слота.

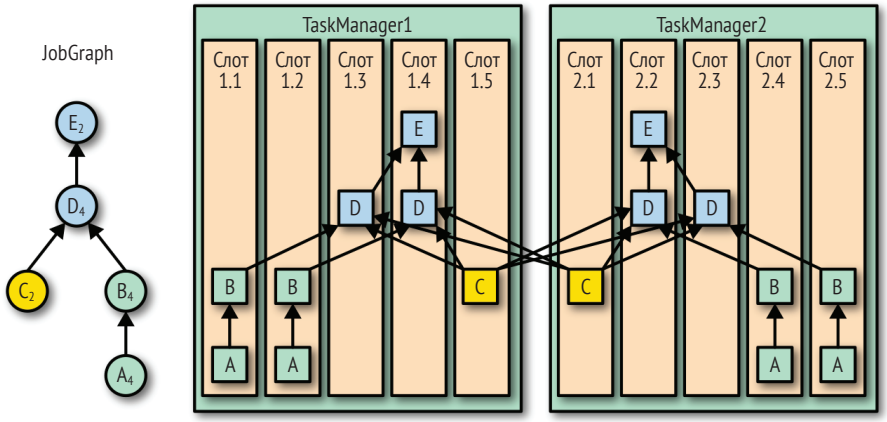

Рис. 10.1 ❖ Управление планированием задач с помощью групп с разделением слотов

10.3. Настройка контрольных точек и восстановления

Приложение Flink, работающее в режиме высокой доступности, периодически создает контрольную точку своего состояния. Создание контрольных точек может быть дорогостоящей операцией, поскольку объем данных, которые необходимо скопировать в постоянное хранилище, иногда бывает довольно большим. Увеличение интервала контрольных точек снижает накладные расходы на отказоустойчивость при обычной работе. Однако это также увеличивает объем данных, которые задание должно обработать при восстановлении после сбоя, прежде чем оно догонит хвостовую часть потока.

Flink предоставляет несколько параметров для настройки контрольных точек и бэкендов состояния. Настройка этих параметров важна для обеспечения надежной и бесперебойной работы потоковых приложений в производственной среде. Например, сокращение накладных расходов на каждую контрольную точку может способствовать более высокой частоте контрольных точек, что приводит к более быстрым циклам восстановления. В этом разделе мы описываем параметры, используемые для управления установкой контрольных точек и восстановлением приложений.

10.3.1. Настройка контрольных точек

Когда вы включаете контрольные точки для приложения, вы должны указать *интервал контрольных точек* – интервал, через который JobManager будет инициировать контрольные точки в источниках приложения.

Контрольные точки включены в StreamExecutionEnvironment:

```
val env: StreamExecutionEnvironment = ???

// Разрешаем установку контрольных точек с интервалом 10 секунд.
env.enableCheckpointing(10000);
```

Дополнительные параметры для настройки поведения контрольных точек доступны в CheckpointConfig, который можно получить из StreamExecutionEnvironment:

```
// Получаем CheckpointConfig из StreamExecutionEnvironment.
val cpConfig: CheckpointConfig = env.getCheckpointConfig
```

По умолчанию Flink создает контрольные точки, чтобы гарантировать согласованность состояния «ровно один раз». Однако его также можно настроить для предоставления гарантии «хотя бы один раз»:

```
// Установить режим "хотя бы один раз".
cpConfig.setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE);
```

В зависимости от характеристик приложения, размера его состояния, а также бэкенда состояния и его конфигурации запись контрольной точки может занять до нескольких минут. Более того, размер состояния может со временем меняться, возможно, из-за изменения длительности окон. Поэтому завершение контрольной точки нередко занимает больше времени, чем заданный интервал между точками. По умолчанию Flink позволяет одновременно выполнять только одну контрольную точку, чтобы механизм контрольных точек не забирал слишком много ресурсов у штатного процесса обработки. Если – в соответствии с настроенным интервалом контрольной точки – пришло время запустить контрольную точку, но есть незавершенная контрольная точка, вторая контрольная точка будет приостановлена до завершения первой контрольной точки.

Если для многих или всех контрольных точек требуется больше времени, чем интервал между контрольными точками, такое поведение может быть неоптимальным по двум причинам. Во-первых, это означает, что регулярная обработка данных приложением всегда будет конкурировать за ресурсы с контрольными точками. Следовательно, обработка потоковых данных замедляется и может не успевать за поступающими данными. Во-вторых, контрольная точка может быть отложена, потому что нам нужно дождаться завершения другой контрольной точки, что приведет к более длительному наверстыванию во время восстановления. Flink располагает параметрами настройки для решения этих ситуаций.

Чтобы приложение работало без помех, вы можете настроить минимальную паузу между контрольными точками. Если вы настроите минимальную паузу на 30 с, то новая контрольная точка не будет запущена в течение первых 30 с после завершения контрольной точки. Это также означает, что эффективный интервал между контрольными точками составляет не менее 30 с и одновременно выполняется не более одной контрольной точки.

```
// Работать не менее 30 секунд без контрольных точек.
cpConfig.setMinPauseBetweenCheckpoints(30000);
```

В определенных ситуациях может потребоваться уверенность, что контрольные точки создаются строго с заданной периодичностью, даже если создание контрольной точки превышает интервал между точками. Одним из примеров может быть ситуация, когда контрольные точки занимают много времени на создание, но не потребляют много ресурсов; например, если обращения ко внешним системам происходят с большой задержкой. На этот случай предусмотрена настройка максимального количества одновременных контрольных точек.

```
// Разрешить выполнение не более трех контрольных точек одновременно.
cpConfig.setMaxConcurrentCheckpoints(3);
```

Точки сохранения создаются одновременно с контрольными точками. Flink не задерживает явно запущенные точки сохранения из-за операций с контрольными точками. Точка сохранения будет запускаться всегда независимо от того, сколько контрольных точек выполняется.

Чтобы ограничить длительность создания контрольной точки, вы можете настроить интервал тайм-аута, по истечении которого контрольная точка отменяется. По умолчанию контрольные точки отменяются через 10 мин.

```
// Контрольные точки должны быть завершены в течение 5 минут, иначе они будут отменены.
cpConfig.setCheckpointTimeout(300000);
```

Наконец, вы также можете настроить, что произойдет в случае сбоя контрольной точки. По умолчанию сбой контрольной точки вызывает исключение, которое приводит к перезапуску приложения. Вы можете отключить это поведение и позволить приложению продолжить работу после ошибки контрольной точки.

```
// Не перезапускать задание из-за ошибки контрольной точки.
cpConfig.setFailOnCheckpointingErrors(false);
```

10.3.1.1. Сжатие контрольной точки

Flink поддерживает сжатые контрольные точки и точки сохранения. До версии Flink 1.7 единственным поддерживаемым алгоритмом сжатия был Snappy. Вы можете включить сжатие контрольных точек и точек сохранения следующим образом:

```
val env: StreamExecutionEnvironment = ???

// Включить сжатие контрольной точки.
env.getConfig.setUseSnapshotCompression(true)
```

Обратите внимание, что сжатие контрольных точек не поддерживается для инкрементных контрольных точек RocksDB.

10.3.1.2. Сохранение контрольных точек после остановки приложения

Назначение контрольных точек – восстановить приложение после сбоя. Следовательно, они не создаются, когда задание перестает выполняться из-за сбоя или явной отмены. Однако вы также можете включить функцию, называемую *внешними контрольными точками*, чтобы сохранять контрольные точки после остановки приложения.

```
// Включение внешних контрольных точек.
cpConfig.enableExternalizedCheckpoints(
  ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION)
```

Есть два варианта внешних контрольных точек:

1) `RETAIN_ON_CANCELLATION` сохраняет контрольную точку как после полного отказа приложения, так и при отмене приложения;
2) `DELETE_ON_CANCELLATION` сохраняет контрольную точку только после полного отказа приложения. Если приложение отменено, внешняя контрольная точка удаляется.

Внешние контрольные точки не заменяют точки сохранения. Они используют формат хранения, зависящий от бэкенда состояния, и не поддерживают изменение масштаба. Следовательно, их достаточно для перезапуска приложения после сбоя, но они обеспечивают меньшую гибкость, чем точки сохранения. Как только приложение снова запустится, вы можете записать точку сохранения.

10.3.2. Настройка бэкендов состояния

Бэкенд состояния приложения отвечает за поддержание локального состояния, выполнение контрольных точек и точек сохранения, а также восстановление состояния приложения после сбоя. Следовательно, выбор и конфигурация бэкенда состояния приложения во многом определяет быстродействие механизма контрольных точек. Различные бэкенды состояния подробно описаны в разделе 7.4.1.

По умолчанию в приложении используется бэкенд состояния `MemoryStateBackend`. Поскольку он хранит все состояние в памяти, а контрольные точки полностью хранятся в энергозависимом хранилище кучи JobManager с ограниченным размером JVM, не рекомендуется использовать его для производственных сред. Тем не менее он хорошо подходит для локальной разработки приложений Flink. Раздел 9.5.5 описывает настройку бэкенда состояния по умолчанию для кластера Flink.

Вы также можете явно выбрать бэкенд состояния приложения:

```
val env: StreamExecutionEnvironment = ???

// Создаем и настраиваем бэкенд состояния по вашему выбору.
val stateBackend: StateBackend = ???
// Назначаем бэкенд состояния.
env.setStateBackend(stateBackend)
```

Различные бэкенды состояния можно запустить с минимальными настройками, как показано ниже. `MemoryStateBackend` вообще не требует никаких параметров. Однако есть конструкторы, которые принимают параметры для включения или отключения асинхронной контрольной точки (включено по умолчанию) и ограничивают размер состояния (по умолчанию 5 Мб):

```
// Создаем MemoryStateBackend.
val memBackend = new MemoryStateBackend()
```

`FsStateBackend` требует только путь к месту хранения контрольных точек. Также есть варианты конструктора для включения или отключения асинхронной контрольной точки (по умолчанию включено):

```
// Создаем FsStateBackend, который указывает на папку /tmp/ckp.
val fsBackend = new FsStateBackend("file:///tmp/ckp", true)
```

`RocksDBStateBackend` требует только путь к месту хранения контрольных точек и принимает необязательный параметр для включения инкрементных контрольных точек (по умолчанию отключен). `RocksDBStateBackend` всегда записывает контрольные точки асинхронно:

```
// Создаем RocksDBStateBackend, который записывает инкрементные контрольные точки
// в папку /tmp/ckp.
val RockBackend = new RocksDBStateBackend ("file:///tmp/ckp", true)
```

В разделе 9.5.5 мы обсудили параметры конфигурации для бэкендов состояния. Конечно, вы также можете настроить бэкенд состояния в своем приложении, переопределив значения по умолчанию или конфигурацию всего кластера. Для этого вам необходимо создать новый объект бэкенда, передав объект `Configuration` в бэкенд состояния, см. раздел 9.5.5, где приведено описание доступных параметров конфигурации:

```
// Все встроенные бэкенды Flink настраиваются.
val backend: ConfigurableStateBackend = ???

// Создаем конфигурацию и устанавливаем значения параметров.
val sbConfig = new Configuration()
sbConfig.setBoolean("state.backend.async", true)
sbConfig.setString("state.savepoints.dir", "file:///tmp/svp")

// Создаем настроенную копию бэкенда.
val configureBackend = backend.configure(sbConfig)
```

Поскольку RocksDB является внешним компонентом, он содержит собственный набор параметров настройки, который также можно сконфигурировать под ваше приложение. По умолчанию RocksDB оптимизирован для хранения на SSD и не обеспечивает большое быстродействие, если состояние хранится на механических жестких дисках. Flink предоставляет несколько предустановленных настроек для повышения быстродействия при работе с обычным оборудованием. Описание доступных настроек приведено в документации Flink. Вы можете применить предустановленные параметры к `RocksDBStateBackend` следующим образом:

```
val backend: RocksDBStateBackend = ???

// Установить предустановленные параметры для хранилища на механических дисках.
backend.setPredefinedOptions (PredefinedOptions.SPINNING_DISK_OPTIMIZED)
```

10.3.3. Настройка восстановления

Когда приложение с контрольной точкой выходит из строя, оно перезапускается, вызывает свои задачи, восстанавливает их состояния, включая смещения чтения исходных задач, и продолжает обработку. Сразу после перезапуска приложение находится в стадии наверстывания упущенного. Поскольку исходные задачи приложения были сброшены на более раннюю позицию ввода, ему приходится обрабатывать данные, которые обрабатывались до сбоя, а также данные, накопленные во время простоя приложения.

Чтобы иметь возможность догнать поток – достичь его хвоста, – приложение должно обрабатывать накопленные данные быстрее, чем поступают новые данные. Пока приложение догоняет поток, увеличивается задержка обработки – интервал времени между моментом доступности ввода и моментом фактической обработки.

Следовательно, приложение нуждается в дополнительных свободных ресурсах для фазы наверстывания после перезапуска приложения, чтобы успешно возобновить свою обычную обработку. Это означает, что приложение не должно занимать 100 % доступных ресурсов во время обычной обработки. Чем больше свободных ресурсов доступно для восстановления, тем быстрее завершается фаза наверстывания и тем быстрее время ожидания обработки возвращается к норме.

Помимо соображений, касающихся ресурсов для восстановления, мы обсудим еще две темы, связанные с восстановлением: *стратегии перезапуска* и *локальное восстановление*.

10.3.3.1. Стратегии перезапуска

В зависимости от сбоя, который привел к отказу приложения, сразу после перезапуска приложение может снова остановиться из-за того же сбоя. Типичный пример – недопустимые или поврежденные входные данные, которые приложение не может обработать. В такой ситуации приложение попадет в бесконечный цикл восстановления, потребляя много ресурсов, без возможности когда-либо вернуться к обычной обработке. Flink предлагает три стратегии перезапуска для решения этой проблемы:

1) *с фиксированной задержкой* – перезапускает приложение фиксированное количество раз и ждет заданное время перед попыткой перезапуска;
2) *по частоте отказов* – перезапускает приложение, пока не превышается настраиваемая частота отказов. Частота отказов определяется как максимальное количество отказов в пределах заданного интервала времени. Например, вы можете настроить перезапуск приложения, если оно не дает сбоев более трех раз за последние десять минут;

3) *без перезапуска* – не перезапускает приложение, а немедленно завершает его работу.

Стратегия перезапуска приложения настраивается через `StreamExecutionEnvironment`, как показано в примере 10.4.

Пример 10.4 ❖ Настройка стратегии перезапуска приложения

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setRestartStrategy(
  RestartStrategies.fixedDelayRestart (
    5,                            // Количество попыток перезапуска.
    Time.of (30, TimeUnit.SECONDS) // Задержка между попытками.
))
```

Стратегия перезапуска по умолчанию представляет собой перезапуск с фиксированной задержкой с количеством попыток, указанном в `Integer.MAX_VALUE` и 10-секундной задержкой.

10.3.3.2. Локальное восстановление

Бэкенды состояния Flink (кроме `MemoryStateBackend`) хранят контрольные точки в удаленной файловой системе. Это гарантирует, во-первых, что состояние сохраняется и является постоянным, а во-вторых, его можно перераспределить в случае потери рабочего узла или масштабирования приложения. Однако чтение состояния из удаленного хранилища во время восстановления не очень эффективно. Более того, в большинстве случаев после восстановления приложение удается запустить на тех же рабочих машинах, которые применялись до сбоя.

Flink поддерживает функцию, называемую *локальным восстановлением*, чтобы значительно ускорить восстановление, если приложение можно перезапустить на тех же машинах. Если этот параметр включен, бэкенды состояния также сохраняют копию данных контрольной точки на локальном диске своего рабочего узла в дополнение к записи данных в удаленную систему хранения. Когда приложение перезапускается, Flink первым делом пытается запланировать прежние задачи для прежних рабочих узлов. Если это удается, задачи сначала пытаются загрузить данные контрольной точки с локального диска. В случае возникновения каких-либо проблем они обращаются к удаленному хранилищу.

Локальное восстановление реализовано так, что критерием надежности восстановления является наличие копии состояния в удаленной системе. Задача подтверждает контрольную точку, только если удаленная запись прошла успешно.

Благодаря этому контрольная точка не выйдет из строя из-за потери копии в локальном хранилище. Поскольку данные контрольной точки записываются дважды, локальное восстановление увеличивает накладные расходы на создание контрольной точки.

Локальное восстановление можно включить и настроить для кластера в файле **`flink-conf.yaml`** или для каждого приложения, включив следующее в конфигурацию бэкенда состояния:

- `state.backend.local-recovery`: этот флаг включает или отключает локальное восстановление. По умолчанию локальное восстановление отключено;
- `taskmanager.state.local.root-dirs`: этот параметр указывает один или несколько локальных путей, по которым хранятся копии локального состояния.

Локальное восстановление относится только к состоянию с ключом, которое всегда разбито на разделы и обычно составляет большую часть размера состояния. Состояние оператора не будет храниться локально, и его необходимо получить из удаленной системы хранения. Однако обычно оно намного меньше, чем состояние с ключом. Более того, локальное восстановление не поддерживается `MemoryStateBackend`, который в любом случае не поддерживает большое состояние.

10.4. Мониторинг кластеров и приложений Flink

Мониторинг вашего потокового задания необходим для уверенности в его нормальной работе и раннего выявления потенциальных симптомов неправильной конфигурации, недостаточного выделения ресурсов или неожиданного поведения. Это особенно важно, когда задание потоковой обработки является частью более крупного конвейера обработки данных или управляемой событиями службы в приложении, ориентированном на пользователя. Вероятно, вы захотите как можно точнее отслеживать его быстродействие и убедиться, что оно соответствует определенным целевым показателям задержки, пропускной способности, использования ресурсов и т. д.

Flink собирает набор заданных метрик во время выполнения, а также предоставляет структуру, которая позволяет вам определять и отслеживать собственные метрики.

10.4.1. Веб-интерфейс Flink

Самый простой способ получить обзор вашего кластера Flink, а также взглянуть на то, что происходит внутри задания, – использовать веб-интерфейс Flink. Вы можете получить доступ к панели управления, перейдя по адресу **http://<jobmanager-hostname>:8081**.

На главном экране вы увидите обзор конфигурации вашего кластера, включая количество диспетчеров задач, количество настроенных и доступных слотов задач, а также запущенные и завершенные задания. На рис. 10.2 показан пример главного экрана приборной панели. Меню слева содержит ссылки на более подробную информацию о заданиях и параметрах конфигурации, а также позволяет отправлять задания путем загрузки JAR.

Если вы щелкнете по запущенному заданию, вы сможете быстро просмотреть статистику по выполняемой задаче или подзадаче, как показано на

рис. 10.3. Вы можете проверить продолжительность работы, обмен в байтах и записях, и, если хотите, агрегировать их для каждого TaskManager.

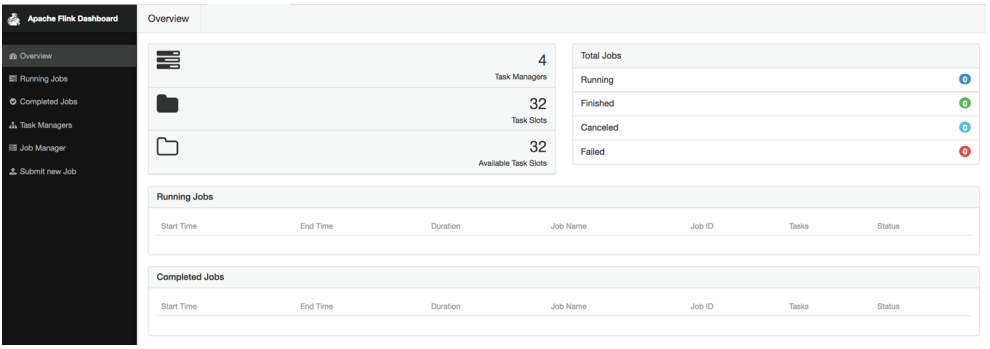

Рис. 10.2 ❖ Главный экран веб-интерфейса Apache Flink

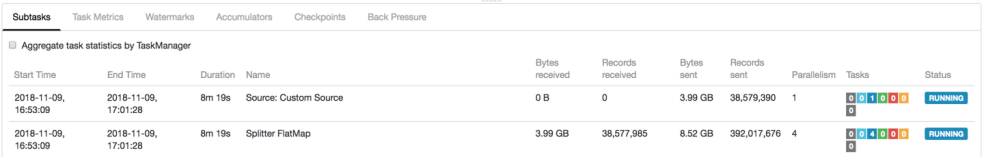

Рис. 10.3 ❖ Статистика по запущенному заданию

Если вы щелкнете по вкладке **Task Metrics** (метрики задачи), вы можете выбрать другие метрики из раскрывающегося меню, как показано на рис. 10.4. Метрики представляют более детальную статистику о ваших задачах, такую как использование буфера, водяные знаки и скорость ввода/вывода.

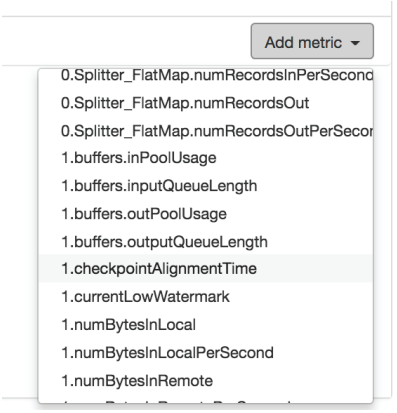

Рис. 10.4 ❖ Выбор метрик для построения графика

На рис. 10.5 показано, как выбранные метрики отображаются в виде постоянно обновляемых графиков.

Рис. 10.5 ❖ Графики выбранных метрик в реальном времени

На вкладке **Checkpoints** (рис. 10.3) отображается статистика о предыдущих и текущих контрольных точках. На вкладке **Overview** (Обзор) вы можете увидеть, сколько контрольных точек было запущено, сколько из них выполняется, выполнено успешно или провалено. Если вы нажмете на вкладку **History** (История), вы сможете получить более подробную информацию, такую как статус, время срабатывания, размер состояния и количество байтов, буферизованных во время фазы выравнивания контрольной точки. Обзор сводки объединяет статистику контрольных точек и предоставляет минимальные, максимальные и средние значения по всем завершенным контрольным точкам. Наконец, в разделе **Configuration** (Конфигурация) вы можете проверить свойства конфигурации контрольных точек, такие как интервал и установленные значения тайм-аута.

Точно так же на вкладке **Back Pressure** (Противодавление) отображается статистика противодавления для каждого оператора и подзадачи. Если щелкнуть строку, запускается отбор измерений противодавления, и вы увидите сообщение **Sampling in progress...** (Выполняется отбор измерений) в течение примерно пяти секунд. По завершении отбора измерений вы увидите статус противодавления во втором столбце. Задачи, испытывающие высокое противодавление, будут помечены значком **HIGH**; в противном случае вы должны увидеть сообщение **OK** зеленого цвета.

10.4.2. Система метрик

При запуске системы обработки данных, такой как Flink, в производственной среде важно отслеживать ее поведение, чтобы иметь возможность обнаруживать снижение быстродействия и диагностировать причину этого. Flink по умолчанию собирает несколько метрик системы и приложений. Метрики собираются для каждого оператора и процесса TaskManager или JobManager. Здесь мы описываем некоторые из наиболее часто используемых метрик и отсылаем вас к документации Flink для получения полного списка доступных метрик.

Категории метрик включают использование ЦП, занятую память, количество активных потоков, статистику сборки мусора, сетевые метрики, такие как количество помещенных в очередь буферов ввода/вывода, общие метрики кластера, такие как количество запущенных заданий или доступные ресурсы, метрики заданий, включая время выполнения, количество повторных попыток и информацию о контрольных точках, статистику ввода/вывода, включая количество обменов записями локально и удаленно, информацию о водяных знаках, метрики для конкретных соединителей и т. д.

10.4.2.1. Регистрация и использование метрик

Чтобы зарегистрировать метрики, вы должны получить `MetricGroup`, вызвав `getMetrics()` в `RuntimeContext`, как показано в примере 10.5.

Пример 10.5 ❖ Регистрация и использование показателей в FilterFunction

```
class PositiveFilter extends RichFilterFunction[Int] {

  @transient private var counter: Counter = _

  override def open(parameters: Configuration): Unit = {
    counter = getRuntimeContext
      .getMetricGroup
      .counter("droppedElements")
    }

    override def filter(value: Int): Boolean = {
    if (value > 0) {
      true
    }
    else {
      counter.inc()
      false
    }
  }
}
```

10.4.2.2. Группы метрик

Метрики Flink регистрируются и доступны через интерфейс `MetricGroup`, который предоставляет способы создания вложенных именованных иерархий метрик и методы для регистрации следующих типов показателей.

Counter (счетчик)
: Метрика `org.apache.flink.metrics.Counter` получает показания счетчика и предоставляет методы для инкремента и декремента. Вы можете зарегистрировать метрику счетчика, используя метод `counter(String name, Counter counter)` в `MetricGroup`.

Gauge (индикатор)
: Метрика `Gauge` вычисляет значение любого типа в определенный момент времени. Чтобы использовать `Gauge`, вы реализуете интерфейс `org.apache.`

flink.metrics.Gauge и регистрируете его с помощью метода gauge(String name, Gauge gauge) в MetricGroup. Код в примере 10.6 показывает реализацию метрики WatermarkGauge, которая показывает текущий водяной знак.

Пример 10.6 ❖ Реализация метрики WatermarkGauge, которая показывает текущий водяной знак

```
public class WatermarkGauge implements Gauge<Long> {
  private long currentWatermark = Long.MIN_VALUE;

  public void setCurrentWatermark(long watermark) {
    this.currentWatermark = watermark;
    }

  @Override
  public Long getValue() {
    return currentWatermark;
  }
}
```

Метрики представлены в виде строк

Репортеры показателей превратят значение Gauge в строку, поэтому убедитесь, что вы предоставили соответствующую реализацию toString(), если она не предоставляется используемым типом.

Histogram (гистограмма)

Вы можете использовать гистограмму для представления распределения числовых данных. Гистограмма Flink специально реализована для создания отчетов о показателях длинных значений. Интерфейс org.apache.flink.metrics.Histogram позволяет собирать значения, получать текущее количество собранных значений и создавать статистику, такую как минимальное, максимальное, стандартное отклонение и среднее значение, для значений, которые встречались до текущего момента.

Помимо создания вашей собственной реализации гистограммы, Flink также позволяет использовать гистограмму DropWizard после добавления следующих зависимостей:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-metrics-dropwizard</artifactId>
  <version>flink-version</version>
</dependency>
```

Затем вы можете зарегистрировать гистограмму DropWizard в своей программе Flink, используя DropwizardHistogramWrapper, как показано в примере 10.7.

Пример 10.7 ❖ Использование DropwizardHistogramWrapper

```
// Создаем и регистрируем гистограмму.
DropwizardHistogramWrapper histogramWrapper =
```

```
    new DropwizardHistogramWrapper(
      new com.codahale.metrics.Histogram(new SlidingWindowReservoir(500)))
metricGroup.histogram("myHistogram", histogramWrapper)

// Обновляем гистограмму.
histogramWrapper.update(value)
```

Meter (измеритель)

Вы можете использовать метрику Meter для измерения скорости (в событиях в секунду), с которой происходят определенные события. Интерфейс `org.apache.flink.metrics.Meter` предоставляет методы для пометки возникновения одного или нескольких событий, получения текущей частоты событий в секунду и получения текущего количества событий, отмеченных измерителем.

Как и в случае с гистограммами, вы можете использовать измерители DropWizard, добавив зависимость `flink-metrics-dropwizard` в ваш файл **`pom.xml`** и обернув измеритель в класс `DropwizardMeterWrapper`.

10.4.2.3. Метрики области и форматирования

Метрики Flink относятся к области действия, которая может быть либо областью системы для метрик, предоставляемых системой, либо областью пользователя для настраиваемых, определяемых пользователем метрик. На метрики ссылается уникальный идентификатор, который может содержать до трех частей:

1) имя, которое пользователь указывает при регистрации метрики;
2) необязательная область пользователя;
3) область системы.

Например, имя `"myCounter"`, пользовательская область `"MyMetrics"` и системная область `"localhost.taskmanager.512"` вместе дают нам идентификатор `"localhost.taskmanager.512.MyMetrics.myCounter"`. Вы можете изменить значение разделителя по умолчанию `"."`, задав параметр конфигурации `metrics.scope.delimiter`.

Область системы определяет, к какому компоненту системы относится метрика и какую контекстную информацию она должна включать. Метрики могут быть привязаны к JobManager, TaskManager, заданию, оператору или задаче. Вы можете настроить, какую контекстную информацию должна содержать метрика, установив соответствующие параметры метрики в файле **`flink-conf.yaml`**. Мы перечисляем некоторые из этих параметров конфигурации и их значения по умолчанию в табл. 10.17.

Ключи конфигурации содержат постоянные строки, такие как `taskmanager`, и переменные, показанные в угловых скобках. Последние будут заменены во время выполнения фактическими значениями. Например, область по умолчанию для метрик `TaskManager` может создать область `localhost.taskmanager.512`, где `localhost` и `512` – значения параметров. В табл. 10.18 показаны все переменные, доступные для настройки областей метрик.

Таблица 10.17. Параметры конфигурации области системы и их значения по умолчанию

Область	Ключ конфигурации	Значение по умолчанию
JobManager	metrics.scope.jm	<host>.jobmanager
JobManager и задание	metrics.scope.jm.job	<host>.jobmanager.<job_name>
TaskManager	metrics.scope.tm	<host>.taskmanager.<tm_id>
TaskManager и задание	metrics.scope.tm.job	<host>.taskmanager.<tm_id>.<job_name>
Задача	metrics.scope.task	<host>.taskmanager.<tm_id>.<job_name>.<task_name>.<subtask_index>
Оператор	metrics.scope.operator	<host>.taskmanager.<tm_id>.<job_name>.<operator_name>.<subtask_index>

Таблица 10.18. Доступные переменные для настройки форматирования областей метрик

Область	Допустимые значение
JobManager:	<host>
TaskManager:	<host>, <tm_id>
Job:	<job_id>, <job_name>
Task:	<task_id>, <task_name>, <task_attempt_id>, <task_attempt_num>, <subtask_index>
Operator:	<operator_id>, <operator_name>, <subtask_index>

Идентификаторы области действия для задания должны быть уникальными

Если несколько копий одного и того же задания выполняются одновременно, метрики могут быть неточными из-за конфликтов строк. Чтобы избежать такого риска, вы должны убедиться, что идентификаторы области действия для каждого задания уникальны. С этим легко справиться, включив в идентификатор `<job_id>`.

Вы также можете определить пользовательскую область для метрик, вызвав метод `addGroup()` объекта `MetricGroup`, как показано в примере 10.8.

Пример 10.8 ❖ Определение области действия пользователя «MyMetrics»

```
counter = getRuntimeContext
  .getMetricGroup
  .addGroup ("MyMetrics")
  .counter ("myCounter")
```

10.4.2.4. Доступ к метрикам

Теперь, когда вы узнали, как регистрировать, определять и группировать метрики, вам может быть интересно, как получить к ним доступ из внешних систем. В конце концов, вы, вероятно, собираете метрики, потому что хотите создать информационную панель, отображающую параметры системы в реальном времени, или передать результаты измерений в другое приложе-

ние. Вы можете предоставлять метрики внешним серверным модулям через *репóртеры* (reporter), и Flink предоставляет реализацию для некоторых из них (см. табл. 10.19).

Таблица 10.19. Список доступных репортеров метрик

Репортер	Реализация
JMX	org.apache.flink.metrics.jmx.JMXReporter
Graphite	org.apache.flink.metrics.graphite.GraphiteReporter
Prometheus	org.apache.flink.metrics.prometheus.PrometheusReporter
PrometheusPushGateway	org.apache.flink.metrics.prometheus. PrometheusPushGatewayReporter
StatsD	org.apache.flink.metrics.statsd.StatsDReporter
Datadog	org.apache.flink.metrics.datadog.DatadogHttpReporter
Slf4j	org.apache.flink.metrics.slf4j.Slf4jReporter

Если вы хотите использовать бэкенд метрик, который не включен в приведенный выше список, вы также можете определить своей собственный репòртер, реализовав интерфейс `org.apache.flink.metrics.reporter.MetricReporter`.

Репортеры нужно настроить в файле **`flink-conf.yaml`**. Добавление следующих строк в вашу конфигурацию определит JMX-репортер `my_reporter`, который прослушивает порты 9020-9040:

```
metrics.reporters: my_reporter
Metrics.reporter.my_jmx_reporter.class: org.apache.flink.metrics.jmx.JMXReporter
metrics.reporter.my_jmx_reporter.port: 9020-9040
```

Обратитесь к документации Flink для получения полного списка параметров конфигурации для каждого поддерживаемого репортера.

10.4.3. Мониторинг задержки

Задержка, вероятно, является одной из первых метрик, которую вы захотите отслеживать для оценки быстродействия вашего потокового задания. В то же время это также одна из самых сложных метрик для определения в распределенном потоковом движке с богатой семантикой, таком как Flink. В разделе 2.2.1 мы определили задержку в широком смысле как время, необходимое для обработки события. Вы можете себе представить, насколько проблематичной может стать на практике точная реализация этого определения, если мы попытаемся отследить задержку каждого события в высокоскоростной потоковой передаче со сложным потоком данных. Учитывая, что оконные операторы еще больше усложняют отслеживание задержки, если событие влияет на несколько окон, нужно ли нам сообщать о задержке первого вызова или нам нужно ждать, пока мы оценим все окна, которым может принадлежать событие? А что, если окно запускается несколько раз?

Flink следует простому подходу с низкими накладными расходами, чтобы обеспечить полезное значение метрики задержки. Вместо того чтобы пытаться строго измерить задержку для каждого события, он аппроксимирует задержку, периодически генерируя специальную запись в источниках и позволяя пользователям отслеживать, сколько времени требуется, чтобы эта запись прибыла в приемники. Эта специальная запись называется *маркером задержки* (latency marker), и она содержит метку времени, указывающую, когда она была отправлена.

Чтобы включить отслеживание задержки, вам необходимо настроить, как часто маркеры задержки отправляются из источников. Вы можете сделать это, задав значение `latencyTrackingInterval` в `ExecutionConfig`, как показано здесь:

```
env.getConfig.setLatencyTrackingInterval(500L)
```

Интервал указывается в миллисекундах. Получив маркер задержки, все операторы, кроме приемников, направляют его вниз по потоку. Маркеры задержки используют те же каналы и очереди потока данных, что и обычные записи потока, поэтому их отслеживаемая задержка отражает время *ожидания* обработки для записей. Однако они не измеряют время, необходимое для *обработки* записей.

Операторы хранят статистику задержки в индикаторе задержки, который содержит минимальные, максимальные и средние значения, а также значения 50, 95 и 99 процентилей. Операторы приемника хранят статистику по маркерам задержки, полученным для каждого экземпляра параллельного источника; таким образом, проверка маркера задержки в приемниках может использоваться, чтобы приблизительно определить, сколько времени требуется записи для прохождения потока данных. Если вы хотите настроить обработку маркера задержки в операторах, вы можете переопределить метод `processLatencyMarker()` и получить соответствующую информацию с помощью методов `getMarkedTime()`, `getVertexId()` и `getSubTaskIndex()` из `LatencyMarker`.

Остерегайтесь перекоса часов

Если вы не используете службу автоматической синхронизации часов, такую как NTP, часы ваших компьютеров могут страдать от перекоса часов. В этом случае оценка отслеживания задержки не будет надежной, поскольку ее текущая реализация предполагает синхронизированные часы.

10.5. Настройка журналирования

Журналирование или ведение журнала – еще один важный инструмент для отладки и понимания поведения ваших приложений. По умолчанию Flink использует абстракцию журналирования SLF4J вместе с фреймворком журналирования **`log4j`**.

В примере 10.9 показана функция `MapFunction`, которая регистрирует каждое преобразование входной записи.

Пример 10.9 ❖ Использование журналирования в `MapFunction`

```
import org.apache.flink.api.common.functions.MapFunction
import org.slf4j.LoggerFactory
import org.slf4j.Logger

class MyMapFunction extends MapFunction[Int, String] {

  Logger LOG = LoggerFactory.getLogger(MyMapFunction.class)

  override def map(value: Int): String = {
    LOG.info("Converting value {} to string.", value)
    value.toString
  }
}
```

Чтобы изменить свойства логгеров **`log4j`**, измените файл **`log4j.properties`** в папке **`conf/`**. Например, в следующей строке для корневого уровня ведения журнала устанавливается значение «предупреждение»:

```
log4j.rootLogger=WARN
```

Чтобы установить собственное имя файла и местоположение этого файла, передайте в JVM параметр `-Dlog4j.configuration=`. Flink также предоставляет файл **`log4j-cli.properties`**, используемый клиентом командной строки, и файл **`log4j-yarn-session.properties`**, используемый клиентом командной строки при запуске сеанса YARN.

Альтернативой **`log4j`** является **`logback`**, и Flink также предоставляет файлы конфигурации по умолчанию для этого бэкенда. Чтобы использовать **`logback`** вместо **`log4j`**, вам нужно удалить **`log4j`** из папки **`lib/`**. Мы отсылаем вас к документации Flink и руководству по **`logback`** для получения подробной информации о том, как установить и настроить бэкенд журналирования.

10.6. Заключение

В этой главе мы обсудили запуск, управление и отслеживание приложений Flink в производственной среде. Мы рассказали про компонент Flink, который собирает и предоставляет метрики системы и приложений, как настроить систему ведения журнала, а также как запускать, останавливать, возобновлять и масштабировать приложения с помощью клиента командной строки и REST API.

Глава 11

Что дальше?

Позади остался долгий путь, и вы добрались до конца этой книги! Но ваше путешествие по Flink только началось, и в этой главе пойдет речь о путях, которые вы можете выбрать. Мы кратко расскажем о дополнительных возможностях Flink, не рассмотренных в этой книге, и дадим вам несколько ссылок на дополнительные ресурсы Flink. Вокруг Flink существует активное сообщество, и мы рекомендуем вам начать общаться с другими пользователями, вносить свой вклад в сообщество и узнать, какие компании создают продукты и решения с помощью Flink, чтобы найти источники идей для своей работы.

11.1. Остальная часть экосистемы Flink

Хотя в этой книге особое внимание уделяется потоковой обработке, на самом деле Flink представляет собой универсальную платформу распределенной обработки данных, которую можно использовать и для других типов анализа данных. Кроме того, Flink предлагает доменные библиотеки и API для реляционных запросов, а также для обработки сложных событий (CEP) и графов.

11.1.1. API DataSet для пакетной обработки

Flink – это полноценный пакетный процессор, который можно использовать для реализации сценариев использования, требующих однократных или периодических запросов к ограниченным входным данным. Программы DataSet строятся как серия преобразований, как и программы DataStream, с той разницей, что DataSet – это ограниченный набор данных. API DataSet предоставляет операторы для выполнения фильтрации, сопоставления, выбора, объединения и группировки, а также соединители для чтения и записи наборов данных из внешних систем, и во внешние системы, такие как файловые системы и базы данных. Используя API DataSet, вы также можете определять итерационные программы Flink, которые выполняют функцию цикла для фиксированного количества шагов или до тех пор, пока не будет выполнен критерий сходимости.

Пакетные задания внутренне представлены как программы потока данных и выполняются по тому же принципу, что и задания потоковой обработки. В настоящее время два API используют разные среды выполнения и не могут быть смешаны. Однако сообщество Flink уже работает над их объединением, и предоставление единого API для анализа ограниченных и неограниченных потоков данных в одной программе является приоритетом в будущей дорожной карте Flink.

11.1.2. Table API и SQL для реляционного анализа

Несмотря на то что базовые API DataStream и DataSet разделены, вы можете реализовать унифицированную потоковую и пакетную аналитику Flink, используя его высокоуровневые реляционные API-интерфейсы: Table API и SQL.

Table API – это API на *языке интегрированных запросов* (LINQ) для Scala и Java. Запросы могут без изменений выполняться как для пакетного, так и для потокового анализа. LINQ предлагает общие операторы для написания реляционных запросов, включая выбор, проекцию, агрегацию и объединение, а также поддерживает IDE для автозаполнения и проверки синтаксиса.

Flink SQL следует стандарту ANSI SQL и использует Apache Calcite для синтаксического анализа и оптимизации запросов. Flink предоставляет унифицированный синтаксис и семантику для пакетных и потоковых запросов. Благодаря обширной поддержке пользовательских функций SQL может охватывать самые разные варианты использования. Вы можете встраивать SQL-запросы в обычные программы Flink DataSet и DataStream или напрямую отправлять SQL-запросы в кластер Flink с помощью клиента командной строки для SQL. Клиент командной строки позволяет извлекать и визуализировать результаты запросов в командной строке, что делает его отличным инструментом для опробования и отладки запросов Flink SQL или выполнения исследовательских запросов к потоковым или пакетным данным. Кроме того, вы можете использовать клиент командной строки для отправки отдельных запросов, которые напрямую записывают свои результаты во внешние системы хранения.

11.1.3. FlinkCEP для обработки сложных событий и сопоставления с образцом

FlinkCEP – это высокоуровневый API и библиотека для обнаружения сложных шаблонов событий. Он реализован поверх API DataStream и позволяет вам указывать шаблоны, которые вы хотите обнаружить в своем потоке. Обычные примеры использования CEP включают финансовые приложения, обнаружение мошенничества, мониторинг и оповещение в сложных системах, а также обнаружение сетевых вторжений или подозрительного поведения пользователей.

11.1.4. Gelly для обработки графов

Gelly – это API и библиотека обработки графов Flink. Он основан на DataSet API и поддержке Flink для эффективных пакетных итераций. Gelly предоставляет высокоуровневые программные абстракции как в Java, так и в Scala для выполнения преобразований графов, агрегирования и итеративной обработки. Он также включает набор общих графовых алгоритмов, готовых к использованию.

Высокоуровневые API и интерфейсы Flink хорошо интегрированы друг с другом и с API DataStream и DataSet, так что вы можете легко смешивать их и переключаться между библиотеками и API в одной программе. Например, вы можете извлекать шаблоны из DataStream с помощью библиотеки CEP, а затем использовать SQL для анализа извлеченных шаблонов, или вы можете использовать API таблиц для фильтрации и проецирования таблиц в графы перед их анализом с помощью алгоритма из библиотеки Gelly.

11.2. Присоединяйтесь к сообществу Flink

Apache Flink имеет непрерывно растущее гостеприимное сообщество, в которое входят участники и пользователи со всего мира. Вот несколько ресурсов, которые вы можете использовать, чтобы задавать вопросы, посещать мероприятия, связанные с Flink, и знакомиться с конкретными примерами использования Flink:

Подписка на рассылки

- user@flink.apache.org: поддержка пользователей и вопросы;
- dev@flink.apache.org: разработка, релизы и обсуждения в сообществе;
- community@flink.apache.org: новости сообщества и встречи.

Блоги

- https://flink.apache.org/blog;
- https://www.ververica.com/blog.

Встречи и конференции

- https://flink-forward.org;
- https://www.meetup.com/topics/apache-flink.

Мы надеемся, что, прочитав на эту книгу, вы лучше поймете всю мощь и возможности Apache Flink. Мы призываем вас стать активной частью его сообщества.

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Фабиан Уэске, Василики Калаври

Потоковая обработка данных с Apache Flink

Главный редактор *Мовчан Д. А.*
Заместитель главного редактора *Сенченкова Е. А.*
dmkpress@gmail.com
Перевод *Яценков В. С.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 24,21. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**