Содержание

Предисловие ... 11

От издательства ... 13

Благодарности ... 14

Об авторах ... 15

Колофон ... 16

Глава 1. Введение в потоковую обработку с учетом состояния ... 17

1.1. Традиционные инфраструктуры данных ... 18
1.1.1. Транзакционная обработка ... 18
1.1.2. Аналитическая обработка ... 19
1.2. Обработка потоков с учетом состояния ... 21
1.2.1. Событийно-ориентированные приложения ... 23
1.2.2. Конвейеры данных ... 24
1.2.3. Потоковая аналитика ... 25
1.3. Эволюция потоковой обработки с открытым исходным кодом ... 26
1.3.1. Немного истории ... 27
1.4. Обзорное знакомство с Flink ... 29
1.4.1. Запуск вашего первого приложения Flink ... 30
1.5. Заключение ... 33

Глава 2. Основы потоковой обработки ... 34

2.1. Введение в потоковое программирование ... 34
2.1.1. Графы потока данных ... 34
2.1.2. Параллелизм данных и параллелизм задач ... 36
2.1.3. Стратегии обмена данными ... 36
2.2. Параллельная обработка потоков ... 37
2.2.1. Задержка и пропускная способность ... 37
2.2.2. Операции с потоками данных ... 40
2.3. Семантика времени ... 45
2.3.1. Что означает одна минута в потоковой обработке? ... 46
2.3.2. Время обработки ... 47
2.3.3. Время события ... 47
2.3.4. Водяные знаки ... 49
2.3.5. Время обработки по сравнению со временем события ... 50
2.4. Модели состояния и согласованности ... 50

2.4.1. Сбои заданий 52
2.4.2. Гарантии результата 53
2.5. Заключение 54

Глава 3. Архитектура Apache Flink 56

3.1. Архитектура системы 56
3.1.1. Компоненты набора Flink 57
3.1.2. Развертывание приложений 58
3.1.3. Выполнение задачи 59
3.1.4. Высокодоступная конфигурация 60
3.2. Передача данных во Flink 62
3.2.1. Кредитное управление потоком 64
3.2.2. Цепочка задач 64
3.3. Обработка на основе времени события 66
3.3.1. Метки времени 66
3.3.2. Водяные знаки 67
3.3.3. Распространение водяного знака и время события 68
3.3.4. Назначение метки времени и создание водяных знаков 70
3.4. Управление состоянием 71
3.4.1. Состояние оператора 72
3.4.2. Состояние с ключевым доступом 73
3.4.3. Бэкенд состояния 74
3.4.4. Масштабирование операторов с учетом состояния 75
3.5. Контрольные точки, точки сохранения и восстановление состояния ... 77
3.5.1. Согласованные контрольные точки 77
3.5.2. Восстановление из сохраняющей контрольной точки 78
3.5.3. Алгоритм создания контрольной точки Flink 80
3.5.4. Значение контрольных точек для производительности 85
3.5.5. Точки сохранения 85
3.6. Заключение 88

Глава 4. Настройка рабочей среды для Apache Flink 89

4.1. Необходимое ПО 89
4.2. Запуск и отладка приложений Flink в среде IDE 90
4.2.1. Импорт примеров книги в IDE 90
4.2.2. Запуск приложений Flink в среде IDE 92
4.2.3. Отладка приложений Flink в среде IDE 93
4.3. Развертывание проект Flink для сборщика Maven 94
4.4. Заключение 95

Глава 5. API DataStream (v1.7) 96

5.1. Hello, Flink! 96
5.1.1. Настройка среды выполнения 98
5.1.2. Чтение входного потока 98
5.1.3. Применение преобразований 99

5.1.4. Вывод результата ... 99
5.1.5. Выполнение ... 100
5.2. Преобразования ... 100
5.2.1. Основные преобразования ... 101
5.2.2. Преобразования KeyedStream ... 104
5.2.3. Многопоточные преобразования ... 106
5.2.4. Преобразования распределения ... 110
5.3. Настройка параллельной обработки ... 113
5.4. Типы ... 114
5.4.1. Поддерживаемые типы данных ... 115
5.4.2. Создание информации о типах для типов данных ... 117
5.4.3. Явное предоставление информации о типе ... 118
5.5. Определение ключей и полей ссылок ... 119
5.5.1. Позиции поля ... 119
5.5.2. Выражения поля ... 120
5.5.3. Ключевые селекторы ... 121
5.6. Реализация функций ... 121
5.6.1. Функциональные классы ... 122
5.6.2. Лямбда-функции ... 123
5.6.3. Расширенные функции ... 123
5.7. Добавление внешних и Flink-зависимостей ... 124
5.8. Заключение ... 125

Глава 6. Операторы на основе времени и оконные операторы ... 126

6.1. Настройка показателей времени ... 126
6.1.1. Назначение меток времени и создание водяных знаков ... 128
6.1.2. Водяные знаки, задержка и полнота ... 132
6.2. Функции процесса ... 133
6.2.1. TimerService и таймеры ... 134
6.2.2. Передача потоков на боковые выходы ... 136
6.2.3. CoProcessFunction ... 137
6.3. Оконные операторы ... 139
6.3.1. Определение оконных операторов ... 139
6.3.2. Встроенные средства назначения окон ... 140
6.3.3. Применение функций в окнах ... 144
6.3.4. Настройка оконных операторов ... 150
6.4. Объединение потоков по времени ... 161
6.4.1. Интервальное объединение ... 161
6.4.2. Оконное объединение ... 162
6.5. Обработка опоздавших данных ... 164
6.5.1. Отбрасывание опоздавших событий ... 164
6.5.2. Перенаправление опоздавших событий ... 164
6.5.3. Обновление результатов путем включения опоздавших событий ... 166
6.6. Заключение ... 167

Глава 7. Операторы и приложения с учетом состояния ... 168
7.1. Реализация функций с сохранением состояния ... 169
7.1.1. Объявление ключевого состояния в RuntimeContext ... 169
7.1.2. Реализация списочного состояния с помощью интерфейса ListCheckpointed ... 172
7.1.3. Использование широковещательного состояния ... 175
7.1.4. Использование интерфейса CheckpointedFunction ... 178
7.1.5. Получение уведомлений о пройденных контрольных точках ... 180
7.2. Включение восстановления после сбоя для приложений с учетом состояния ... 181
7.3. Обеспечение работоспособности приложений с учетом состояния ... 182
7.3.1. Указание уникальных идентификаторов оператора ... 182
7.3.2. Определение максимального параллелизма операторов ключевого состояния ... 183
7.4. Производительность и надежность приложений с учетом состояния ... 184
7.4.1. Выбор бэкенда состояния ... 184
7.4.2. Выбор примитива состояния ... 185
7.4.3. Предотвращение утечки состояния ... 186
7.5. Развитие приложений с учетом состояния ... 189
7.5.1. Обновление приложения без изменения существующего состояния ... 190
7.5.2. Удаление состояния из приложения ... 190
7.5.3. Изменение состояния оператора ... 190
7.6. Запрашиваемое состояние ... 192
7.6.1. Архитектура и обслуживание запросов состояния ... 192
7.6.2. Отображение состояния запроса ... 194
7.6.3. Запрос состояния из внешних приложений ... 195
7.7. Заключение ... 197

Глава 8. Чтение и запись при работе с внешними системами ... 198
8.1. Гарантии согласованности приложений ... 198
8.1.1. Идемпотентные записи ... 199
8.1.2. Транзакционные записи ... 200
8.2. Соединители Apache Flink ... 201
8.2.1. Соединитель источника Apache Kafka ... 202
8.2.2. Соединитель приемника Apache Kafka ... 205
8.2.3. Соединитель файлового источника ... 209
8.2.4. Соединитель файлового приемника ... 211
8.2.5. Соединитель приемника Apache Cassandra ... 213
8.3. Реализация пользовательской исходной функции ... 216
8.3.1. Сбрасываемые функции источника ... 218
8.3.2. Функции источника, метки времени и водяные знаки ... 219
8.4. Реализация пользовательской функции приемника ... 220

8.4.1. Идемпотентные соединители приемника 222
8.4.2. Соединители транзакционных приемников........ 223
8.5. Асинхронный доступ к внешним системам........ 230
8.6. Заключение 233

Глава 9. Настройка Flink для потоковых приложений........ 234

9.1. Режимы развертывания........ 234
9.1.1. Автономный кластер 234
9.1.2. Docker........ 236
9.1.3. Apache Hadoop YARN........ 238
9.1.4. Kubernetes........ 241
9.2. Режим высокой доступности........ 245
9.2.1. Высокая доступность в автономном режиме........ 246
9.2.2. Высокодоступная конфигурация YARN 247
9.2.3. Высокодоступная конфигурация Kubernetes 248
9.3. Интеграция с компонентами Hadoop........ 249
9.4. Конфигурация файловой системы........ 250
9.5. Конфигурация системы 252
9.5.1. Java и загрузка классов 252
9.5.2. Процессор 253
9.5.3. Основная память и сетевые буферы........ 253
9.5.4. Дисковое хранилище 255
9.5.5. Контрольные точки и бэкенды состояния........ 256
9.5.6. Безопасность 256
9.6. Заключение 257

Глава 10. Работа с Flink и потоковыми приложениями 258

10.1. Запуск и управление потоковыми приложениями 258
10.1.1. Точки сохранения........ 259
10.1.2. Управление приложениями с помощью клиента командной строки........ 260
10.1.3. Управление приложениями с помощью REST API........ 265
10.1.4. Объединение и развертывание приложений в контейнерах........ 270
10.2. Управление планированием задач 273
10.2.1. Управление цепочкой задач........ 273
10.2.2. Определение групп совместного использования слотов 274
10.3. Настройка контрольных точек и восстановления 276
10.3.1. Настройка контрольных точек........ 276
10.3.2. Настройка бэкендов состояния........ 279
10.3.3. Настройка восстановления........ 281
10.4. Мониторинг кластеров и приложений Flink 283
10.4.1. Веб-интерфейс Flink 283
10.4.2. Система метрик........ 285
10.4.3. Мониторинг задержки........ 290
10.5. Настройка журналирования........ 291
10.6. Заключение........ 292

Глава 11. Что дальше? 293
11.1. Остальная часть экосистемы Flink 293
11.1.1. API DataSet для пакетной обработки 293
11.1.2. Table API и SQL для реляционного анализа 294
11.1.3. FlinkCEP для обработки сложных событий и сопоставления с образцом 294
11.1.4. Gelly для обработки графов 295
11.2. Присоединяйтесь к сообществу Flink 295

Предметный указатель 296

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.

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.

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.

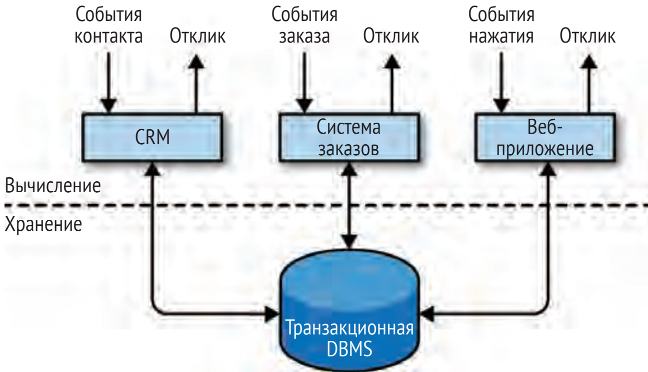

Рис. 1.1 ❖ Традиционная структура транзакционных приложений, хранящих данные в удаленной системе баз данных

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 могут 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.

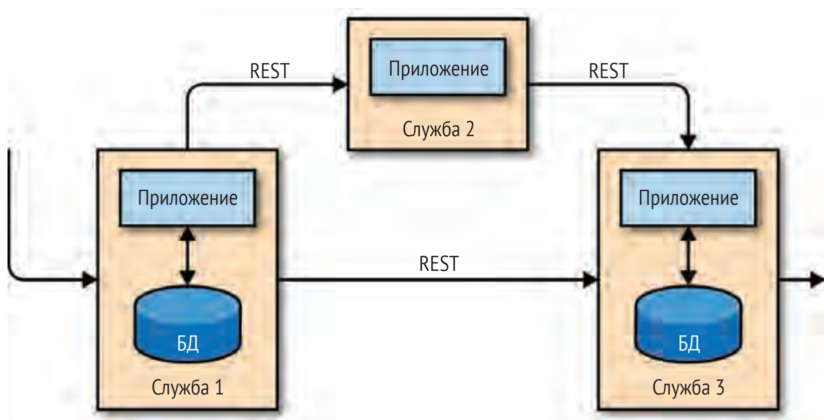

Рис. 1.2 ❖ Архитектура микросервисов

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.

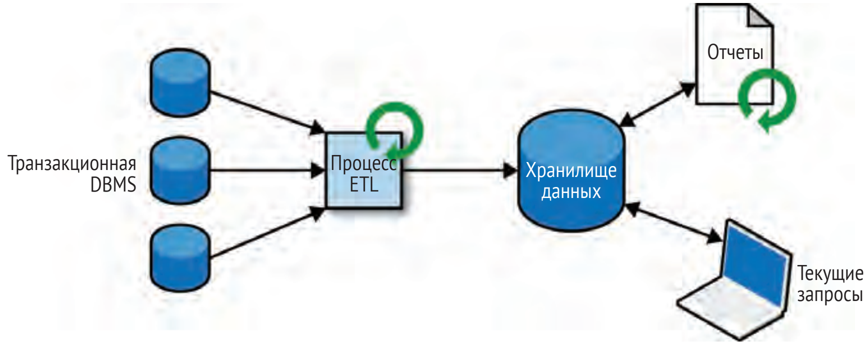

Рис. 1.3 ❖ Традиционная архитектура хранилища данных для аналитической обработки

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.

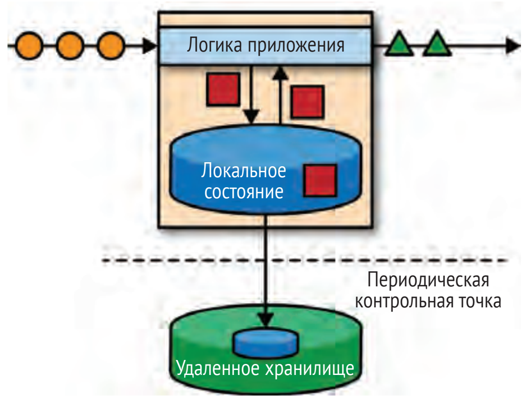

Рис. 1.4 ❖ Приложение для потоковой обработки с учетом состояния

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.

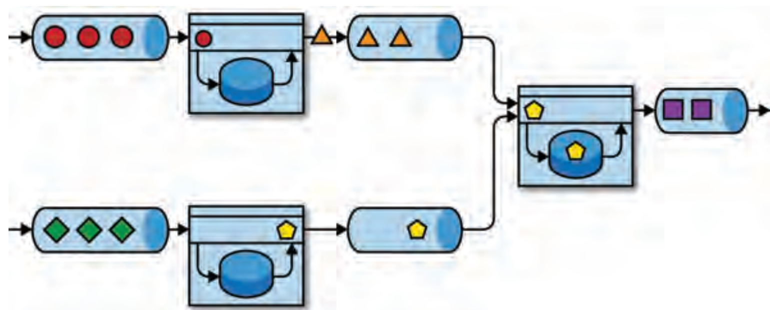

Рис. 1.5 ❖ Архитектура событийного приложения

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.

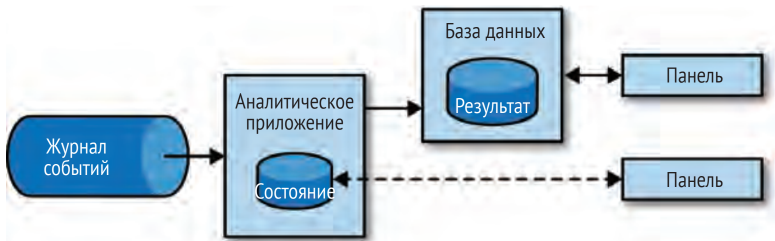

Рис. 1.6 ❖ Приложение потоковой аналитики

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.

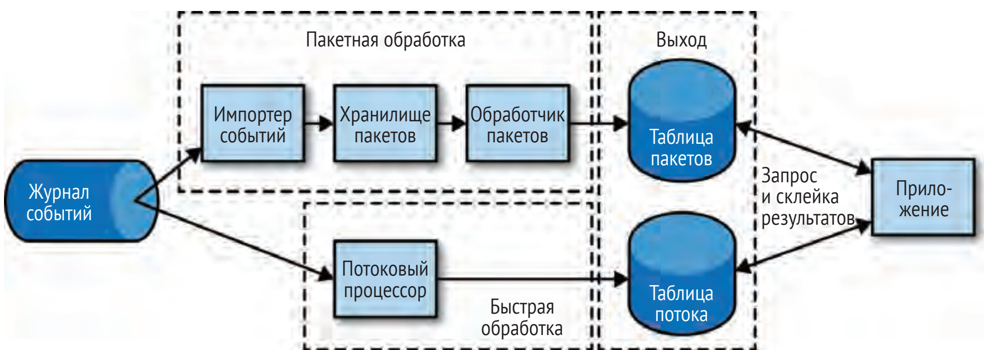

Рис. 1.7 ❖ Лямбда-архитектура

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).

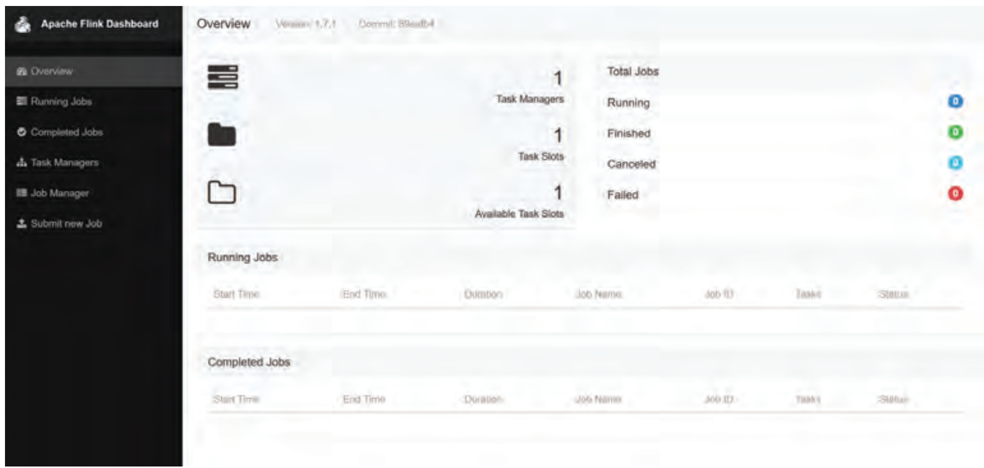

Рис. 1.8 ❖ Снимок экрана веб-панели управления Apache Flink

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.

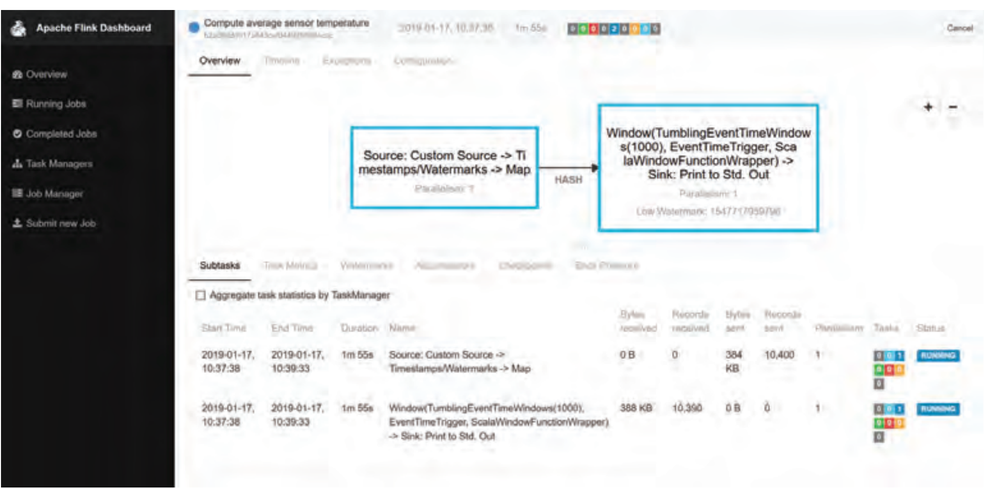

Рис. 1.9 ❖ Снимок экрана веб-панели Apache Flink, показывающий выполняемую задачу

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.