

O'REILLY®

Создание событийно-управляемых микросервисов

Масштабирование использования организационных данных



bhv®

Адам Беллемар

Building Event-Driven Microservices

Leveraging Organizational Data at Scale

Adam Bellemare

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Адам Беллемар

Создание событийно-управляемых микросервисов

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.273
ББК 32.973.26
Б43

Беллемар А.

Б43 Создание событийно-управляемых микросервисов: Пер. с англ. — СПб.: БХВ-Петербург, 2022. — 320 с.: ил.

ISBN 978-5-9775-6757-2

Книга описывает методы создания событийно-управляемых микросервисов для обработки больших объемов данных и предлагает шаблоны приложений, использующих подобную архитектуру. Рассказано о роли микросервисов в поддержке событийно-управляемых проектов, представлены примеры практических реализаций подобных архитектур как силами сотрудников организации, так и с привлечением сторонних специалистов. Подробно описаны инструменты, необходимые для разработки экосистемы микросервисов. Приведены способы решения возникающих проблем, даны рекомендации по налаживанию взаимодействия команд и отдельных сотрудников в процессе создания событийно-управляемых микросервисных систем.

Для системных архитекторов, разработчиков и ИТ-специалистов

УДК 004.273
ББК 32.973.26

Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Перевод с английского	<i>Андрея Логунова</i>
Научный редактор	<i>Тимур Напреев</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Оформление обложки	<i>Карина Соловьевой</i>

© 2021 BHV

Authorized Russian translation of the English edition of *Building Event-Driven Microservices* ISBN 9781492057895

© 2020 Adam Bellemare

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Авторизованный перевод с английского языка на русский издания *Building Event-Driven Microservices*
ISBN 9781492057895

© 2020 Adam Bellemare.

Перевод опубликован и продается с разрешения компании- правообладателя O'Reilly Media, Inc.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-1-492-05789-5 (англ.)
ISBN 978-5-9775-6757-2 (рус.)

© Adam Bellemare, 2020
© Перевод на русский язык, оформление.
ООО "БХВ-Петербург", ООО "БХВ", 2021

Оглавление

https://t.me/it_boooks

Предисловие	15
Условные обозначения, принятые в книге	16
Благодарности.....	16
Предисловие научного редактора к русскому изданию.....	19
Глава 1. Почему именно событийно-управляемые микросервисы?	21
Что такое событийно-управляемые микросервисы?	22
Введение в предметно-ориентированное проектирование и ограниченные контексты	23
Использование моделей предметной области и ограниченных контекстов	24
Привязка ограниченных контекстов к бизнес-требованиям	25
Структуры обмена информацией.....	26
Структуры обмена бизнес-информацией	27
Структуры обмена технической информацией	27
Структуры обмена данными.....	28
Закон Конвея и структуры обмена информацией	29
Структуры обмена информацией в традиционных вычислениях	30
Вариант 1: создать новый сервис	30
Вариант 2: добавить новое бизнес-требование в существующий сервис	31
Плюсы и минусы каждого варианта	31
Командный сценарий: продолжение	32
Конфликтующие обстоятельства	33
Событийно-управляемые коммуникационные структуры.....	33
События являются основой обмена информацией.....	34
Потоки событий обеспечивают единственный источник истины	34
Потребители занимаются своим моделированием и выполняют свои запросы.....	34
Обмен данными улучшается по всей организации	35
Доступные данные поддерживают изменения в обмене информацией	35
Асинхронные событийно-управляемые микросервисы	35
Пример команды, использующей событийно-управляемые микросервисы	36
Синхронные микросервисы	37
Недостатки синхронных микросервисов.....	38
Связь «точка-точка»	38
Зависимое масштабирование	38
Обработка отказов сервисов	39
Управление версиями API и зависимостями..	39
Доступ к данным с привязкой к реализации	39
Распределенные монолиты	39
Тестирование.....	39

Преимущества синхронных микросервисов	39
Резюме	40
Глава 2. Основы событийно-управляемых микросервисов 41	
Построение топологий	41
Топология микросервисов	41
Бизнес-топология	42
Содержимое события	43
Структура события	43
Событие без ключа	44
Сущностное событие	44
Событие с ключом	44
Материализация состояния из сущностных событий	45
Определения и схемы данных событий	47
Принцип единственного источника событий	47
Взаимодействие микросервисов с помощью брокера событий	47
Хранение и обработка событий	48
Дополнительные факторы, которые следует учитывать	49
Брокеры событий или брокеры сообщений?	51
Потребление из неизменяемого журнала	52
Потребление в качестве потока событий	52
Потребление в форме очереди	53
Обеспечение единственного источника истины	53
Масштабное управление микросервисами	54
Размещение микросервисов в контейнерах	54
Размещение микросервисов на виртуальных машинах	54
Управление контейнерами и виртуальными машинами	55
Уплата налога на микросервисы	56
Резюме	56
Глава 3. Обмен информацией и контракты на передачу данных 57	
Событийно-управляемые контракты на передачу данных	57
Использование явных схем в качестве контрактов	58
Комментарии к определению схемы	58
Полнофункциональное развитие схемы	59
Поддержка генератора кода	60
Разрушительные изменения схемы	61
Учет критических изменений схемы для событий	62
Выбор формата события	63
Дизайн событий	64
Говорите правду, всю правду и ничего, кроме правды	64
Используйте одно-единственное определение события в расчете на поток	64
Используйте самые точные типы данных	64
Держите события узконаправленными	65
Пример: перегрузка определений событий	66
Минимизируйте размер событий	68
Привлекайте потенциальных потребителей к дизайну события	69
Не используйте события в качестве семафоров и сигналов	69
Резюме	70

Глава 4. Интеграция событийно-управляемых архитектур с существующими системами	71
Что такое освобождение данных?	72
Компромиссы для освобождения данных	73
Конвертация освобожденных данных в события	75
Шаблоны освобождения данных	75
Фреймворки освобождения данных	76
Освобождение данных по запросу	77
Массовая загрузка	77
Инкрементная загрузка временных меток	77
Загрузка с автоинкрементируемым ID	77
Выполнение пользовательских запросов	78
Инкрементное обновление	78
Выгоды от обновления по запросу	78
Недостатки обновления по запросу	79
Освобождение данных с помощью журналов захвата изменений в данных	80
Выгоды от использования журналов хранилища данных	81
Недостатки использования журналов базы данных	82
Освобождение данных с помощью исходящих таблиц	82
Некоторые мысли по поводу производительности	84
Изоляция внутренних моделей данных	84
Обеспечение совместимости схем	86
Выгоды от создания событий с помощью исходящих таблиц	88
Недостатки создания событий с помощью исходящих таблиц	89
Захват изменений в данных с помощью триггеров	89
Выгоды от использования триггеров	92
Недостатки использования триггеров	92
Внесение изменений определения данных в захватываемые наборы данных	93
Обработка произведенных постфактум изменений определения данных для шаблонов запросов и журналов CDC	93
Обработка изменений определения данных для шаблонов захвата таблиц изменений в данных	94
Запись данных о событиях в хранилища данных	94
Влияние на бизнес записи данных в приемники и получения их из источников	95
Резюме	97
Глава 5. Основы событийно-управляемой обработки	99
Топологии без поддержки состояния	100
Преобразования	100
Ветвление и слияние потоков	101
Переподразделение потоков событий	101
Пример переподразделения потока событий	102
Соподразделение потоков событий	103
Пример соподразделения потока событий	103
Назначение разделов экземпляру потребителя	104
Назначение разделов с помощью контроллера разделов	104
Назначение соподразделенных разделов	105
Стратегии назначения разделов	105
Назначение по круговой схеме	105

Статическое назначение	106
Индивидуально настраиваемое назначение	107
Восстановление после отказов экземпляра обработчика без поддержки состояния.....	107
Резюме	107
Глава 6. Детерминированная обработка потоков.....	108
Детерминизм событийно-управляемых рабочих процессов.....	109
Временные метки.....	109
Синхронизация распределенных временных меток.....	111
Обработка с помощью событий, помеченных временными метками	111
Пример: выбор порядка событий при обработке многочисленных разделов	112
Планирование событий и детерминированная обработка	112
Индивидуально настраиваемые планировщики событий.....	113
Обработка на основе времени события, времени обработки и времени получения.....	113
Извлечение временной метки потребителем	114
Вызовы к внешним системам в форме «запрос-ответ».....	114
Водяные знаки	114
Водяные знаки в параллельной обработке	115
Время потока.....	117
Время потока в параллельной обработке	118
Неупорядоченные и запоздалые события.....	119
Запоздалые события с водяными знаками и временем потока	120
Причины и последствия неупорядоченных событий	120
Загрузка из источников с неупорядоченными данными	120
Несколько производителей в несколько разделов	121
Чувствительные ко времени функции и управление окнами	122
Переворачивающиеся окна	122
Скользящие окна	123
Сеансовые окна	124
Обработка запоздалых событий	124
Повторная обработка или обработка в режиме, близком к реальному времени	126
Периодические сбои и запоздалые события	127
Проблемы с подключением производителя/брокера событий	127
Резюме и дополнительная литература	129
Глава 7. Потоковая передача с поддержкой состояния	130
Хранилища состояний и материализация состояний из потока событий	130
Запись состояния в поток событий журнала изменений.....	131
Материализация состояния во внутреннее хранилище состояний	132
Материализация глобального состояния.....	133
Преимущества использования внутреннего состояния	134
Требования к масштабируемости снимаются с разработчика	134
Высокопроизводительные возможности, основанные на использовании дисков	134
Гибкость использования подключенного через сеть внешнего диска	134
Недостатки использования внутреннего состояния	135
Ограничения при использовании локальных дисков	135
Потери при использовании дискового пространства	135

Масштабирование и восстановление внутреннего состояния	136
Использование «горячих реплик»	136
Восстановление и масштабирование из журналов изменений	138
Восстановление и масштабирование входных потоков событий.....	138
Материализация состояния во внешнее хранилище состояний	139
Преимущества внешнего состояния	140
Полная локализация данных	140
Технологии	140
Недостатки внешнего состояния.....	140
Управление многочисленными технологиями.....	140
Потеря производительности из-за сетевой задержки	141
Финансовые затраты на сервисы внешнего хранилища состояний	141
Полная локальность данных	141
Масштабирование и восстановление с помощью внешних хранилищ состояний.....	142
Использование исходных потоков	142
Использование журналов изменений.....	142
Использование моментальных снимков	143
Перестраивание хранилищ состояний или их миграция?	143
Перестраивание	143
Миграция.....	144
Транзакции и практически однократная обработка	145
Пример: сервис учета запасов	146
Практически однократная обработка с транзакциями клиент-бронкер	147
Практически однократная обработка без транзакции клиент-бронкер	148
Генерация дублирующихся событий	149
Идентификация дублирующихся событий.....	149
Защита от дубликатов.....	150
Поддержание согласованного состояния.....	151
Резюме	153
Глава 8. Построение рабочих процессов с помощью микросервисов	154
Шаблон «хореография»	155
Простой пример событийно-управляемой хореографии	156
Создание и изменение хореографического рабочего процесса.....	157
Мониторинг хореографического рабочего процесса	157
Шаблон «оркестровка»	158
Простой пример событийно-управляемой оркестровки	159
Простой пример оркестровки с прямым вызовом	160
Событийно-управляемая оркестровка в сравнении с оркестровкой с прямым вызовом	160
Создание и изменение рабочего процесса на основе оркестровки	162
Мониторинг процесса на основе оркестровки.....	162
Распределенные транзакции	162
Хореографические транзакции: шаблон «Сага».....	163
Пример хореографической транзакции	163
Транзакции с оркестровкой	165
Компенсационные процессы	167
Резюме	168

Глава 9. Микросервисы с использованием технологии «Функция как сервис».....	169
Проектирование функционально-ориентированных решений в качестве микросервисов	169
Обеспечивайте строгое членство в ограниченном контексте	169
Фиксируйте смещения только после завершения обработки.....	170
Когда функция завершила свою обработку.....	170
Когда функция только что запустилась	170
Меньше значит больше.....	171
Выбор провайдера FaaS	171
Построение микросервисов из функций.....	172
«Холодный старт» и «теплые старты»	173
Запуск функций с помощью триггеров	174
Запуск по новому событию: слушатель потока событий	174
Запуск по задержке группы потребителей.....	176
Запуск по расписанию.....	177
Запуск с использованием веб-перехватчиков	177
Запуск по событиям ресурсов	177
Решение бизнес-задач с помощью функций	178
Поддержание состояния.....	178
Функции, вызывающие другие функции.....	179
Шаблон событийно-управляемого обмена информацией	179
Шаблон прямого вызова	180
Хореография и асинхронный вызов функций.....	180
Оркестровка и синхронный вызов функций.....	182
Завершение работы функции и выключение	183
Тонкая настройка функций.....	184
Выделение достаточных ресурсов	184
Параметры пакетной обработки событий	184
Масштабирование решений FaaS.....	185
Резюме	186
Глава 10. Микросервисы на основе базового шаблона производителя и потребителя	187
Где ВРС работают хорошо?	187
Интеграция с существующими и унаследованными системами.....	188
Пример: шаблон «Коляска»	188
Бизнес-логика с поддержкой состояния без учета порядка событий	189
Пример: книгоиздание.....	189
Когда уровень данных выполняет значительную часть работы	190
Независимое масштабирование уровня обработки и данных	191
Пример: агрегирование данных о событиях для создания профилей взаимодействия с пользователями	191
Гибридные приложения ВРС с внешней потоковой обработкой.....	192
Пример: использование внешнего фреймворка обработки потоков для объединения потоков событий	192
Резюме	194

Глава 11. «Тяжеловесные» фреймворки для микросервисов.....	195
Краткая история «тяжеловесных» фреймворков	196
Внутренняя логика работы «тяжеловесных» фреймворков	197
Выгоды и ограничения.....	199
Варианты настройки кластера и режимы исполнения	200
Используйте сервис, размещенный на хосте провайдера.....	201
Постройте свой собственный полный кластер	201
Создайте кластер, интегрированный с CMS	201
Развертывание и запуск кластера с использованием CMS	201
Указание ресурсов для одного задания с использованием CMS	202
Режимы передачи приложений в кластер.....	203
Драйверный режим	203
Кластерный режим	203
Обработка состояния и использование контрольных точек	204
Масштабирование приложений и обработка разделов потока событий	205
Масштабирование приложения во время его работы	206
Масштабирование приложения путем его перезапуска.....	209
Автоматическое масштабирование приложений	209
Восстановление после отказов	210
Рекомендации по части мультитенантности	210
Языки и синтаксис	211
Выбор фреймворка	211
Пример: обработка щелчков и просмотров с помощью сеансовых окон	212
Резюме	215
Глава 12. «Легковесные» фреймворки для микросервисов.....	216
Выгоды и ограничения.....	216
«Легковесная» обработка.....	217
Обработка состояния и использование журналов изменений	218
Масштабирование приложений и восстановление после отказов	218
Перераспределение событий	219
Назначение состояния.....	219
Репликация состояния и «горячие реплики»	220
Выбор «легковесного» фреймворка.....	220
Apache Kafka Streams	221
Apache Samza: режим встраивания	221
Языки и синтаксис	221
Операция соединения «поток-таблица-таблица»: шаблон обогащения	222
Резюме	226
Глава 13. Интегрирование событийно-управляемых микросервисов с микросервисами типа «запрос-ответ».....	227
Обработка внешних событий	227
Автономно генерируемые события	228
Реактивно генерируемые события	228
Обработка автономно генерируемых аналитических событий	229
Интегрирование со сторонними API «запрос-ответ»	230
Обработка и обслуживание данных с поддержкой состояния	232
Обслуживание запросов реального времени с помощью внутренних хранилищ состояний	233

Обслуживание запросов реального времени с помощью внешних хранилищ состояний	236
Обслуживание запросов путем материализации событийно-управляемого микросервиса.....	236
Обслуживание запросов через отдельный микросервис	237
Обработка запросов внутри событийно-управляемого рабочего процесса	239
Обработка событий для пользовательских интерфейсов	240
Пример: рабочий процесс публикации газет (шаблон одобрения)	241
Разделение сервисов одобрения редактором и рекламодателем	245
Микрофронтенды в приложениях на основе запросов-ответов	247
Выгоды от микрофронтендов	248
Композиционные микросервисы	249
Простота привязки к бизнес-требованиям	249
Недостатки микрофронтендов.....	249
Потенциальное отсутствие единобразия элементов пользовательского интерфейса и стилизации	249
Различающаяся производительность микрофронтендов	250
Пример: приложение поиска источников впечатлений и отзывов о них	251
Резюме	254
Глава 14. Вспомогательные инструменты	255
Система закрепления микросервисов за командами	255
Создание и модификация потока событий.....	256
Разметка потока событий метаданными.....	256
Квоты	257
Реестр схем.....	258
Оповещение о создании и модификации схем.....	259
Управление смещением	259
Разрешения и списки контроля доступа для потоков событий	260
Управление состоянием и сброс приложения.....	261
Мониторинг запаздывания смещения потребителей	262
Оптимизированный процесс создания микросервисов.....	263
Элементы управления контейнерами	264
Создание кластеров и управление ими.....	264
Программная доводка брокеров событий	265
Программная доводка вычислительных ресурсов.....	265
Межкластерная репликация событийных данных	266
Программная доводка инструментария.....	266
Отслеживание зависимостей и визуализация топологии.....	266
Пример топологии.....	268
Резюме	271
Глава 15. Тестирование событийно-управляемых микросервисов.....	272
Общие принципы тестирования	272
Модульное тестирование функций топологии	273
Функции без поддержки состояния	273
Функции с поддержкой состояния.....	273
Тестирование топологии.....	274
Тестирование развития и совместимости схем	275

Интеграционное тестирование событийно-управляемых микросервисов	275
Локальное интеграционное тестирование	276
Создание временной среды внутри среды выполнения тестируемого кода	278
Создание временной среды, внешней по отношению к тестируемому коду	280
Интеграция размещенных на хосте сервисов с использованием возможностей имитации и симуляции	281
Интеграция удаленных сервисов, не имеющих локальных возможностей	282
Полное интеграционное тестирование	283
Программное создание временной среды интеграционного тестирования	283
Заполнение событиями с использованием реальных производственных данных.....	284
Заполнение событиями из подготовленного источника тестирования.....	285
Создание имитационных событий с использованием схем	285
Тестирование с использованием единой общей среды.....	286
Тестирование с использованием производственной среды.....	287
Выбор стратегии полного интеграционного тестирования	288
Резюме	289
Глава 16. Развёртывание событийно-управляемых микросервисов	291
Принципы развертывания микросервисов	291
Архитектурные компоненты развертывания микросервисов.....	292
Системы непрерывной интеграции, доставки и развертывания	292
Системы управления контейнерами и стандартное аппаратное обеспечение	294
Базовый шаблон полного развертывания	294
Шаблон непрерывного обновления	296
Шаблон разрушительных изменений схемы	297
Продолженная миграция через два потока событий.....	298
Синхронизированная миграция в новый поток событий.....	299
Шаблон «сине-зеленого» развертывания	300
Резюме	301
Глава 17. Заключение.....	302
Уровни обмена информацией.....	302
Предметные области бизнеса и ограниченные контексты	302
Совместные инструменты и инфраструктура	303
Схематизированные события	304
Освобождение данных и единственный источник истины	304
Микросервисы	305
Варианты реализации микросервисов	305
Тестирование	306
Развёртывание.....	307
Завершающие слова	308
Предметный указатель.....	309

Предисловие

Я написал эту книгу, потому что сам хотел бы иметь такую книгу, когда начинал свое путешествие в мир событийно-управляемых микросервисов. Она стала квинтэссенцией моего личного опыта, дискуссий с другими людьми и бесчисленных блогов, книг, постов, бесед, конференций и документов, связанных с той или иной частью предметной области событийно-управляемых микросервисов. Я обнаружил, что во многих работах, которые я читал, упоминались событийно-управляемые архитектуры либо только вскользь, либо с недостаточной глубиной. Некоторые из них охватывали только отдельные аспекты этих архитектур и, хотя и были полезными, но составляли лишь малую часть пазла. Другие работы показались мне упрощенными и пренебрежительными, поскольку утверждали, что событийно-управляемые системы на самом деле полезны лишь в том, чтобы одна система отправляла асинхронное сообщение непосредственно другой, подменяя синхронные системы «запросов-ответов». Как подробно рассказывается в этой книге, событийно-управляемые архитектуры — это гораздо большее, чем такое ограниченное представление.

Инструменты, которые мы используем, существенно влияют на то, какую пользу мы с их помощью для себя извлекаем. Архитектуры событийно-управляемых микросервисов становятся возможными благодаря целому ряду технологий, которые только недавно стали легкодоступными. В этой книге в основе архитектур и шаблонов дизайна лежат распределенные, отказоустойчивые, высокопроизводительные и высокоскоростные брокеры событий. Эти технологические решения основаны на конвергенции больших данных и необходимости обработки событий почти в реальном времени. Микросервисам способствует простота контейнеризации и получения вычислительных ресурсов, что обеспечивает простоту в размещении, масштабировании и управлении сотнями тысяч микросервисов.

Технологии, поддерживающие событийно-управляемые микросервисы, оказывают значительное влияние на то, что мы думаем о задачах и как их решаем, а также на то, как структурированы наши предприятия и организации. Событийно-управляемые микросервисы меняют характер работы предприятия, способы решения задач и взаимодействия коллективов, людей и бизнес-единиц. Эти инструменты дают нам действительно новый способ осуществлять виды деятельности, которые до недавнего времени были невозможны.

Условные обозначения, принятые в книге

В книге используются следующие типографские условные обозначения:

◆ *Курсив*

Им выделяются новые термины.

◆ **Полужирный шрифт**

Им выделяются интернет-адреса (URL) и адреса электронной почты.

◆ Шрифт Arial

Им выделяются пути к файлам, имена файлов и расширения файлов.

◆ Монодирический шрифт

Он используется для листингов программ, а также внутри абзацев для выделения упоминаемых там элементов программ — таких как переменные, инструкции языка и ключевые слова.

◆ Полужирный монодирический шрифт

Им выделяются команды либо другой текст, который должен быть напечатан непосредственно пользователем.

◆ Монодирический шрифт курсивом

Им выделяется текст, который должен быть заменен значениями пользователя либо значениями, определяемыми по контексту.



Такой значок обозначает подсказку или совет.



Такой значок обозначает общее примечание.



Такой значок обозначает предупреждение или предостережение.

На веб-странице этой книги по адресу <https://oreil.ly/building-event-driven-microservices> приведены описания ошибок, примеры и прочая дополнительная информация.

Благодарности

Я хотел бы выразить свое уважение и благодарность сотрудникам компании Confluent, которые, наряду с изобретением Apache Kafka, являются одними из первых, кто по-настоящему «понимает», когда речь идет о событийно-управляемых архитектурах. Мне посчастливилось получить от одного из них, Бена Стопфорда

(Ben Stopford), ведущего технолога офиса технического директора, обширные и ценные отзывы. Скотт Моррисон (Scott Morrison), технический директор РНЕМI Systems, также поделился со мной цennыми идеями, отзывами и рекомендациями. Я выражают свою благодарность и признательность Скотту и Бену за то, что они помогли сделать эту книгу тем, чем она является сегодня. Будучи главными ее рецензентами и техническими экспертами, они помогли мне усовершенствовать содержащиеся в ней идеи, улучшить качество контента, не дали мне разместить неверную информацию и помогли правильно рассказать о событийно-управляемых архитектурах.

Я также хотел бы выразить благодарность моим друзьям Джастину Токарчуку (Justin Tokarchuk), Гэри Грэму (Gary Graham) и Нику Грину (Nick Green), которые вычитали и отредактировали множество моих черновиков. Вместе со Скоттом и Беном они помогли мне выявить в моем повествовании наиболее существенные слабые места, предложили пути их улучшения, а также поделились своими мыслями и личным опытом в отношении излагаемого материала.

Я также благодарю сотрудников издательства O'Reilly за то, что они помогали мне самыми разнообразными способами. По ходу дела я работал со многими замечательными людьми, но, в частности, хотел бы поблагодарить моего редактора, Корбина Коллинза (Corbin Collins), за то, что он поддерживал меня в трудные времена и помогал мне удержаться на верном пути. Он стал отличным коллегой в этом начинании, и я ценю усилия, которые он приложил, чтобы меня поддержать.

Рэйчел Монаган (Rachel Monaghan), мой редактор, напомнила мне о школьных годах, когда мои сочинения возвращались ко мне расчерканными красным цветом. Я чрезвычайно благодарен ей за острый глаз и знание английского языка — она помогла сделать эту книгу намного проще для чтения и понимания. Спасибо, Рейчел.

Кристофер Фошер (Christopher Faucher) был со мной очень терпелив, предоставляя мне отличные подсказки и не моргнув глазом позволил мне в последний момент внести в книгу ряд важных нетривиальных изменений. Спасибо, Крис.

Майк Лукидес (Mike Loukides), вице-президент по контентной стратегии, был одним из моих первых контактов в O'Reilly. Когда я обратился к нему со своим престранным многословным предложением, он терпеливо со мной работал, чтобы конкретизировать его и превратить в основу для книги, которую вы сегодня читаете. Я благодарен ему за то, что он нашел время поработать со мной и в конечном счете помог мне продвинуться вперед в этой работе. Я изо всех сил старался прислушиваться к его предостережениям, чтобы не создать том, который соперничал бы по объему со словарем.

Хочу сказать моим маме и папе, что я благодарен им за то, что они научили меня по достоинству ценить письменную речь. Я благодарен вам за любовь и поддержку. Мой отец познакомил меня с Маршаллом Маклюэном (Marshall McLuhan), и хотя стоит признаться, что я не читал большинство из того, что он написал, я чрезвычайно признателен ему за его утверждение о том, что информационная среда влияет на сообщение. Это изменило мой взгляд на системные архитектуры и их оценку.

Наконец, спасибо всем остальным, кто так или иначе внес большой или малый вклад в поддержку меня и этой работы. И таких людей, которые внесли свой вклад каждый по-своему — в разговорах, блог-постах, презентациях, открытом исходном коде, курьезах, личном опыте, историях и импровизированных рассуждениях, — очень много. Спасибо вам всем и каждому.

Работа над этой книгой доставила мне и удовольствие, и разочарование. Нередко я проклинал себя за то, что ее начал, но, к счастью, еще больше я был рад, когда ее завершил. Надеюсь, что эта книга поможет вам, дорогой читатель, в какой-то мере прикоснуться к миру событийно-управляемых микросервисов и найти в нем место для себя.

Предисловие научного редактора к русскому изданию

Работа над современной книгой по информационным технологиям — большая ответственность для переводчика и редактора. Профессиональная литература в переводе доходит до русскоязычного читателя с опозданием минимум на год. Поэтому в нашем сообществе не всегда успевает сложиться русскоязычная терминология, отражающая соответствующие слова в английском первоисточнике. Отсюда — сложности в использовании тех или иных терминов в переводе книги.

При редактировании перевода этой книги я старался применять терминологию, которую ранее использовали в переводных книгах издательства O'Reilly по созданию микросервисов и платформы обработки потоковых данных Apache Kafka — например, «микросервис», «слабая связанность», «сильное зацепление», «производитель», «потребитель», «материализация». Если аналогов не было, обращался к терминологии в отрасли из собственного опыта работы — например, «оркестровка», «хореография», «шаблон», «предметно-ориентированное проектирование». Если на практике в большинстве случаев используется английский эквивалент, и он уже привычен и понятен читателю (например, «фреймворк»), я без раздумий использовал его. Если же таких аналогов в отрасли и в книгах нет (например, «grandfathered systems» — буквально «дедушкины системы»), то старался подбирать максимально точный аналог в русском языке.

Для сохранения точности перевода и отсылки читателя к англоязычным первоисточникам ко всем важным для понимания книги терминам в скобках приведен оригинальный английский вариант термина из первоисточника.

Верю, что изучение этой книги позволит сократить временной разрыв между получением столь важной для специалистов информации в английском и русском варианте.

Научный редактор
Тимур Напреев

Почему именно событийно-управляемые микросервисы?

The medium is the message¹. https://t.me/it_boooks

— Маршалл Маклюэн

Маклюэн утверждает, что на человечество влияет и вносит фундаментальные изменения в общество не содержимое средств массовой информации, а вовлеченность в их среду. Газеты, радио, телевидение, Интернет, мгновенные сообщения и социальные сети изменили человеческое взаимодействие и социальные структуры благодаря нашей коллективной в них вовлеченности.

То же самое относится и к архитектуре компьютерных систем. Достаточно взглянуть на историю компьютерных изобретений, чтобы увидеть, как обмен данными по сети, реляционные базы данных, разработки в области больших данных (*big data*) и облачные вычисления существенно изменили способы построения таких архитектур и выполнения работы с их помощью. Каждое из этих изобретений изменило не только то, как та или иная технология использовалась в различных программно-информационных проектах, но и то, как организации, коллективы и люди обменивались между собой информацией. От централизованных мейнфреймов до распределенных мобильных приложений — каждая новая среда принципиально изменила отношение людей к вычислительной технике.

Информационная среда асинхронно производимого и потребляемого события претерпела принципиальный сдвиг под воздействием современных технологий. Эти события теперь могут храниться бесконечно, в чрезвычайно больших масштабах и потребляться любым сервисом столько раз, сколько необходимо. Вычислительные ресурсы могут легко приобретаться и подключаться по требованию, что облегчает создание микросервисов и управление ими. Микросервисы могут хранить свои данные и управлять ими в соответствии со своими собственными требованиями, причем делать это в масштабах, которые ранее ограничивались пакетно-ориентированными решениями на основе больших данных. Эти усовершенствования скромной и простой событийно-управляемой среды имеют далеко идущие послед-

¹ Не утихают споры о том, что Маршалл Маклюэн в действительности хотел сказать своим знаменитым афоризмом: «The Medium is the Message». По мнению авторитетного российского маклюэноведа И. Б. Архангельской: «...Маклюэн, возможно, хотел сказать, что тип и форма медиа (generic form of media) важнее того значения (meaning) или содержания (content), которое оно передает, т. е. сама форма средства коммуникации меняет наше сознание». — Прим. ред.

ствия, которые не только изменяют архитектуру компьютеров, но и полностью меняют то, как коллективы, люди и организации создают системы и строят предпринятие.

Что такое событийно-управляемые микросервисы?

Микросервисы и архитектуры в стиле микросервисов существуют уже много лет в самых разных формах и под самыми разными названиями. Сервисно-ориентированные архитектуры (Service-Oriented Architectures, SOA) часто состоят из многочисленных сервисов, синхронно обменивающихся информацией непосредственно друг с другом. В архитектурах, основанных на передаче сообщений, события используются для асинхронного обмена данными между приложениями. Обмен информацией на основе событий, безусловно, не нов, но необходимость в обработке больших массивов данных в реальном времени является новинкой и требует изменения старых архитектурных стилей.

В современной архитектуре событийно-управляемых микросервисов системы обмениваются информацией, *отправляя* (issuing) и *потребляя* (consuming) события. Эти события не уничтожаются при потреблении, как в системах передачи сообщений, но вместо этого остаются легко доступными для других потребителей, которые могут их читать по мере необходимости. В этом состоит важное различие, поскольку оно позволяет задействовать мощные шаблоны, описанные в этой книге.

Сами сервисы малы и создаются специально, чтобы выполнять специфические функции бизнеса. В типичной ситуации под *малым размером сервиса* понимается то, что на его написание уходит не более двух недель. По другому определению сервис должен быть способен (концептуально) «укладываться в голове». Эти сервисы потребляют события из входных потоков событий, выполняют свою специфическую бизнес-логику и могут генерировать свои собственные исходящие события, предоставлять данные для доступа по схеме «запрос-ответ», обмениваться информацией со сторонним API или выполнять другие необходимые действия. Как будет подробно описано в этой книге, такие сервисы могут хранить или не хранить состояние, быть сложными или простыми, могут быть реализованы как автономные приложения или как функция с использованием технологии «Функция как сервис».

Эта комбинация потоков событий и микросервисов образует взаимосвязанный граф активности всей деловой организации. Традиционные компьютерные архитектуры, состоящие из приложений-монолитов и межмонолитного обмена информацией, имеют сходную графовую структуру. Оба этих графа показаны на рис. 1.1.

Выяснение того, как заставить эту графовую структуру работать эффективно, включает в себя рассмотрение двух основных компонентов: узлов и соединений. В этой главе мы рассмотрим каждый из них по очереди.

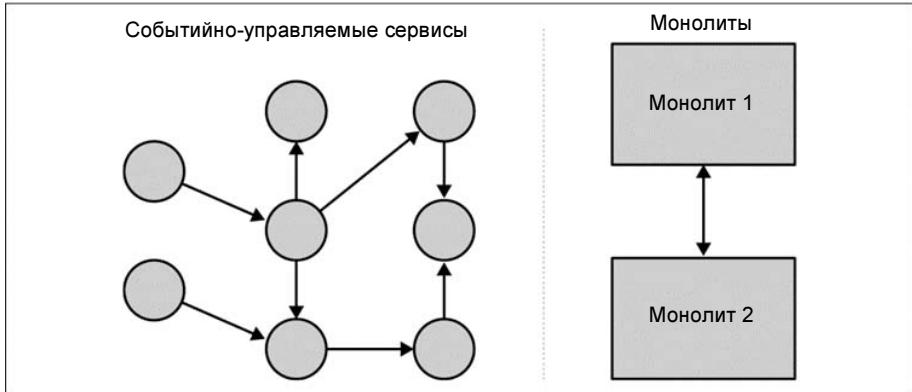


Рис. 1.1. Графовые структуры микросервисов (слева) и монолитов (справа)

Введение в предметно-ориентированное проектирование и ограниченные контексты

Дизайн на основе предметно-ориентированного проектирования, описанный Эриком Эвансом в его книге «Domain-driven design» (<https://oreil.ly/3fGwK>), вводит несколько концепций, необходимых для построения событийно-управляемых микросервисов. Обсуждая эту тему с учетом значительного количества готовых статей (<https://oreil.ly/zAXqd>), книг (<https://oreil.ly/XwjR3>) и лучших блог-постов месяца, я буду здесь предельно краток.

Итак, в основе предметно-ориентированного проектирования лежат следующие концепции:

◆ Домен.

Предметная область, в которой работает бизнес. Сюда входит все, с чем предприятие повседневно имеет дело, включая правила, процессы, идеи, специфичную для предприятия терминологию и все, что связано с его пространством решений, *независимо от того, описано ли это на предприятии или нет*. Предметная область существует независимо от существования предприятия.

◆ Поддомен.

Компонент предметной области. Каждый поддомен сфокусирован на конкретном подмножестве функций и, как правило, отражает некоторую организационную структуру предприятия (например, Склад, Продажи и Инжиниринг). Поддомен может рассматриваться как самостоятельный домен. Поддомены, как и домены, принадлежат пространству решений.

◆ Доменная (и поддоменная) модель.

Абстракция реальной предметной области, полезная для целей предприятия. Для генерирования модели используются наиболее важные для предприятия части и свойства предметной области. Главная модель предметной области предприятия

видна через продукты, которые предприятие предоставляет своим клиентам, через интерфейсы, с помощью которых клиенты взаимодействуют с продуктами, и через различные другие процессы и функции, с помощью которых предприятие достигает своих заявленных целей. Модели часто нуждаются в доработке по мере изменения домена и смены бизнес-приоритетов. Модель предметной области является частью пространства решений, поскольку это конструкция, которую предприятия используют для решения своих задач.

◆ *Ограниченный контекст.*

Логические границы, включая входы, выходы, события, требования, процессы и модели данных, относящиеся к поддомену. Хотя в идеале ограниченный контекст и поддомен будут полностью совпадать, унаследованные системы, технические взаимосвязи и интеграции со сторонними компонентами часто создают исключения. Ограничные контексты также являются свойством пространства решений и оказывают значительное влияние на то, как микросервисы взаимодействуют друг с другом.

Ограничным контекстам следует быть *сильно зацепленными* (*highly cohesive*). Внутренние операции контекста должны быть интенсивными и тесно взаимосвязанными, причем подавляющий объем обмена информацией происходит внутри него и не выходит за его границы. Сильное зацепление позволяет сократить объем проектирования и упростить реализацию.

Соединения между ограниченными контекстами должны быть *слабо связанными* (*loosely coupled*), поскольку изменения, вносимые в пределах одного ограниченного контекста, должны минимизировать или исключить их влияние на соседние контексты. Слабая связь может гарантировать, что изменение требований в одном контексте не потребует внесения зависимых изменений в соседние контексты.

Использование моделей предметной области и ограниченных контекстов

Каждая организация образует единый домен между собой и внешним миром. Каждый, кто работает в организации, предпринимает усилия для удовлетворения ее потребностей в ее предметной области.

Этот единый домен подразделяется на поддомены. Для технологически ориентированной компании — это могут быть, например, поддомены инженерного отдела, отдела продаж и отдела поддержки клиентов. Каждый поддомен имеет свои собственные требования и обязанности и сам может быть подразделен на еще более мелкие «подподдомены». Такой процесс разделения повторяется до тех пор, пока модели поддоменов не станут детализированными и действенными и пока команды разработчиков не смогут реализовать их в малые и независимые сервисы. Ограничные контексты устанавливаются вокруг этих поддоменов и формируют основу для создания микросервисов.

Привязка ограниченных контекстов к бизнес-требованиям

Бизнес-требования к продукту часто меняются в течение его срока службы — например, из-за организационных изменений или запросов на новые функции. Напротив, компании редко приходится менять базовую реализацию какого-либо продукта без сопутствующих изменений бизнес-требований. Вот почему ограниченные контексты должны строиться вокруг бизнес-требований, а не требований технологических.

Привязка ограниченных контекстов к бизнес-требованиям позволяет командам вносить изменения в реализации микросервисов с учетом слабой связанности и сильного зацепления². Она предоставляет команде автономию в дизайне и реализации решения для конкретных бизнес-потребностей, что значительно снижает зависимости между командами и позволяет каждой команде сосредоточиваться строго на своих собственных требованиях.

И наоборот, привязка микросервисов к техническим требованиям создает проблемы. Этот сценарий решений часто встречается в неправильно спроектированных синхронных микросервисах типа «точка-точка» и в традиционных вычислительных системах монолитного типа, где команды владеют определенными техническими слоями приложения. Главная трудность при технологической привязке заключается в том, что она распределяет ответственность за выполнение бизнес-функции между многочисленными ограниченными контекстами, в которых могут участвовать несколько команд с различающимися графиками и обязанностями. Поскольку ни одна команда не несет единоличной ответственности за внедрение решения, каждая служба становится связанный с другой через границы как команды, так и API, что затрудняет внесение изменений и делает их дорогостоящими. Безобидное на первый взгляд изменение, ошибка или отказалший сервис могут серьезно повлиять на возможности предприятия в части работы сервисов, и это касается всех сервисов, использующих техническую систему. Техническая привязка редко используется в архитектурах событийно-управляемых микросервисов (Event-Driven Microservice, EDM), и ее следует полностью избегать, когда это возможно. Устранение трансграничных зависимостей между технологиями и командами снизит чувствительность системы к изменениям.

На рис. 1.2 показаны оба сценария: единоличное владение (*слева*) и трансграничное владение (*справа*). При единоличном владении команда полностью организована вокруг двух независимых бизнес-требований (ограниченных контекстов) и сохраняет полный контроль над своим прикладным кодом и слоем базы данных. При трансграничном владении команды организованы в соответствии с техническими требованиями, где прикладной слой управляет отдельно от слоя данных. Это создает явные зависимости между командами, а также неявные зависимости между бизнес-требованиями.

² См., например, Ньюмен С. Создание микросервисов. СПб.: Питер, 2016. — Прим. перев.

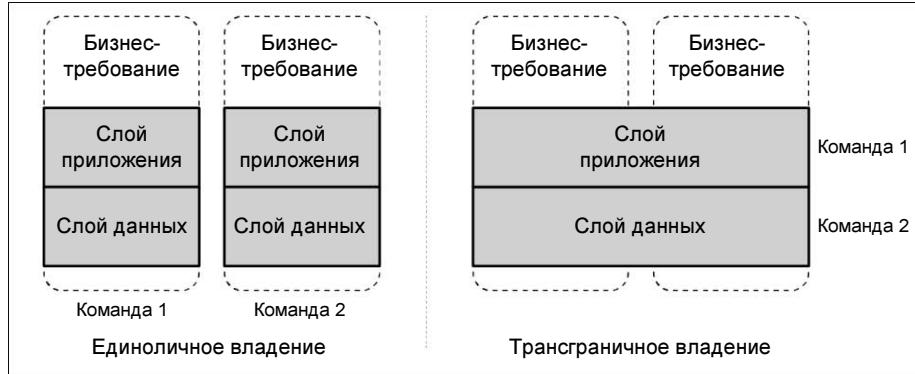


Рис. 1.2. Согласование бизнес-контекста и технологического контекста

Предпочтительным является моделирование архитектуры событийно-управляемых микросервисов с опорой на бизнес-требования, хотя такой подход и требует компромиссов. Код может реплицироваться несколько раз, и многие сервисы могут использовать одинаковые шаблоны доступа к данным. Разработчики продуктов могут попытаться уменьшить повторяемость, поделившись источниками данных с другими продуктами или установив границы. В этих случаях последующая тесная сцепка может быть гораздо более долгостоящей в долгосрочной перспективе, чем повторение логики и хранение дубликатов данных. Упомянутые компромиссы будут далее рассмотрены в этой книге подробнее.



Сохраняйте слабую связанность между ограниченными контекстами и сосредоточьтесь на минимизации межконтекстовых зависимостей. Это позволит при необходимости изменять реализации ограниченного контекста, не нарушая впоследствии многие (или любые) другие системы.

В дополнение к сказанному, от каждой команды может потребоваться опыт работы с полным стеком технологий, что может быть осложнено потребностью в специальных наборах навыков и разрешений на доступ. Организация должна ввести в действие наиболее распространенные требования таким образом, чтобы эти вертикальные группы могли поддерживать себя сами, в то время как более специализированные наборы навыков можно было бы предоставлять на межкомандной основе по мере необходимости. Эти передовые рекомендации подробнее рассматриваются в главе 14.

Структуры обмена информацией

Ради достижения своих целей команды, системы и люди в организации должны обмениваться между собой информацией. Этот обмен образует сложную топологию зависимостей, именуемую *структурой обмена информацией*, или *коммуникационной структурой*. Существуют три основные структуры обмена информацией, и каждая из них влияет на то, как работают предприятия.

Структуры обмена бизнес-информацией

Структура обмена бизнес-информацией (рис. 1.3) обусловливает взаимодействие между командами и подразделениями, при котором все они руководствуются возложенными на них требованиями и обязанностями. Например, отдел инженеринга разрабатывает программные продукты, отдел продаж продаёт продукты клиентам, а отдел поддержки клиентов обеспечивает, чтобы покупатели и клиенты были удовлетворены. Организация команд и постановка их целей — от основных бизнес-единиц до работы отдельного участника — подпадает под эту структуру. Бизнес-требования, их закрепление за командами и состав команд со временем меняются, что может сильно повлиять на взаимосвязь между структурой обмена бизнес-информацией и структурой обмена технической информацией о реализации решений.

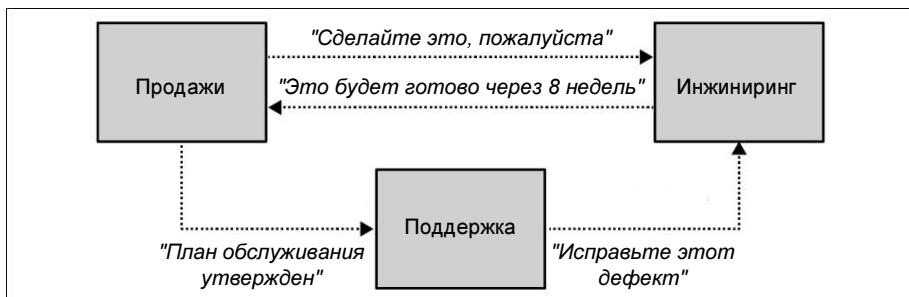


Рис. 1.3. Пример структуры обмена бизнес-информацией

Структуры обмена технической информацией

Структура обмена технической информацией (рис. 1.4) представляет собой данные и логику, относящиеся к модели поддомена и находящиеся под влиянием бизнеса организации. Она формализует бизнес-процессы, структуры данных и системный дизайн таким образом, чтобы бизнес-операции могли выполняться быстро и эффективно. Это приводит к компромиссу в гибкости для структуры обмена бизнес-информацией, поскольку переопределение бизнес-требований, которые должны быть удовлетворены реализацией, требует переписывания логики. Такие переделки чаще всего являются итеративными модификациями модели поддомена и ассоциированного с ней кода, которые с течением времени отражают эволюцию технической информации для выполнения новых бизнес-требований.

Квинтэссенцией структуры обмена информацией для инженеринга программно-информационного обеспечения является монолитное приложение базы данных. Бизнес-логика приложения взаимодействует внутри него посредством вызовов функций или набора состояний. Это монолитное приложение, в свою очередь, используется для удовлетворения бизнес-требований, обусловливаемых структурой обмена бизнес-информацией.

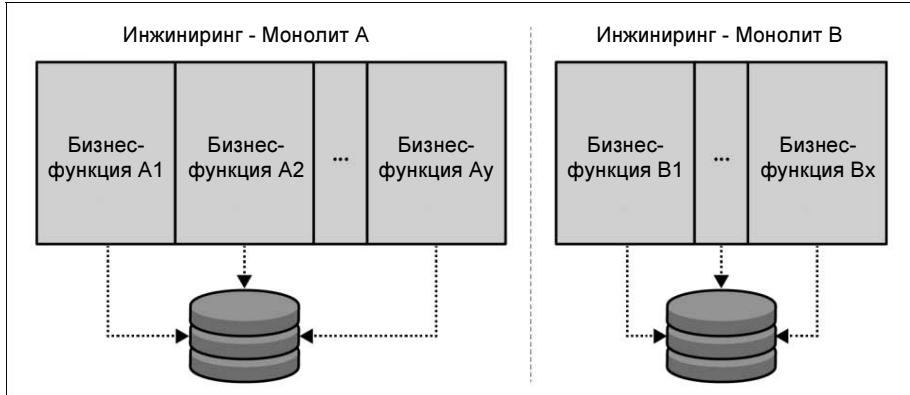


Рис. 1.4. Пример структуры обмена технической информацией

Структуры обмена данными

Структура обмена данными (рис. 1.5) представляет собой процесс, посредством которого данные передаются по всему предприятию и, в частности, между приложениями. Хотя структура обмена данными, включающая электронную почту, обмен мгновенными сообщениями и собрания, часто используется на предприятии для передачи изменений, она в значительной степени игнорировалась в программном обеспечении. Ее роль обычно от системы к системе не регламентировалась, а структура обмена информацией при этом часто выполняла двойную задачу, реализуя функции обмена данными в дополнение к своим собственным требованиям. Это создавало много проблем в ситуациях, когда компании с течением времени претерпевали рост и изменения, влияние которых оценивается в следующем разделе.

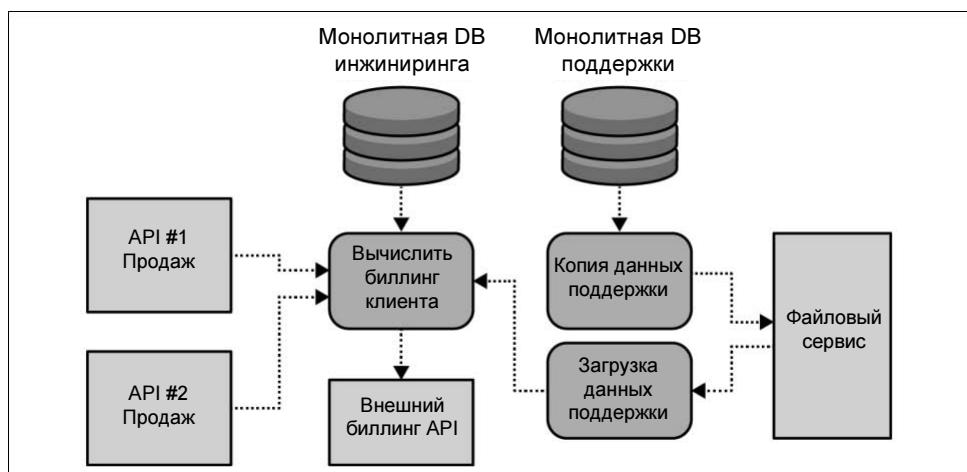


Рис. 1.5. Пример структуры нерегламентированного обмена данными

Закон Конвея и структуры обмена информацией

Организации, которые занимаются дизайном систем... ограничены порождать дизайны, которые являются копиями структур обмена информацией в этих организациях.

— Мелвин Конвей. *Как комитеты изобретают?* (Апрель 1968)

Эта цитата, известная как *закон Конвея*, подразумевает, что команда будет создавать продукты в соответствии со структурами обмена информацией в своей организации. Структуры обмена бизнес-информацией организуют людей в команды, и эти команды обычно разрабатывают продукты, которые ограничены рамками этих команд. Структуры обмена информацией обеспечивают доступ к моделям данных, относящимся к поддоменам для того или иного продукта, но также ограничивают доступ к другим продуктам из-за слабых возможностей по обмену данными.

Поскольку концепции предметной области охватывают весь бизнес, данные предметной области часто требуются другим ограниченным контекстам внутри организации. Коммуникационные структуры реализации, как правило, плохо справляются с предоставлением этого коммуникационного механизма, хотя они превосходно удовлетворяют потребности своего собственного ограниченного контекста. Они влияют на дизайн продуктов двояко. Во-первых, из-за неэффективности передачи необходимых данных предметной области в рамках организации они препятствуют созданию новых, логически отдельных продуктов. Во-вторых, они обеспечивают легкий доступ к существующим данным домена с риском постоянного расширения домена для соответствия новым бизнес-требованиям. Именно этот шаблон воплощен в монолитных конструкциях.

Структуры обмена данными играют ключевую роль в том, как организация проектирует и реализует продукты, но во многих организациях эта структура давно отсутствует. Как уже отмечалось, структуры обмена информацией часто играют эту роль в дополнение к своей собственной.

Некоторые организации пытаются смягчить невозможность доступа к данным предметной области из других реализаций, но их усилия в этом направлении имеют собственные недостатки. Например, зачастую используются совместные базы данных, хотя это и совершенно неверный подход, поскольку они не в состоянии масштабироваться в достаточной мере, чтобы удовлетворять всем требованиям к производительности. Базы данных также могут предоставлять реплики «только для чтения», однако даже это способно излишне раскрыть их внутренние модели данных. Пакетные процессы могут сбрасывать данные в файловое хранилище для чтения другими процессами, но такой подход может создавать проблемы с согласованностью данных и множественностью источников истинности. Наконец, все эти решения приводят к сильной сцепке между реализациями и еще больше укрепляют архитектуру в виде прямых связей «точка-точка».



Если вы обнаружите, что в вашей организации слишком трудно получить доступ к данным или что ваши продукты имеют ограниченную область применения, поскольку все данные локализованы в одной реализации, то вы, скорее всего, столкнетесь с последствиями плохой структуры обмена данными. Эта проблема будет усиливаться по мере роста организации, разработки новых продуктов и все большей потребности в доступе к часто используемым данным предметной области.

Структуры обмена информацией в традиционных вычислениях

Структуры обмена информацией в организации в значительной степени влияют на то, как в них создаются программы. Это верно и на уровне команды — структуры обмена информацией в команде влияют на решения, которые она принимает для реализации возложенных на нее конкретных бизнес-требований. Давайте посмотрим, как это работает на практике.

Рассмотрим следующий сценарий. У одной команды есть один сервис, поддерживаемый одним хранилищем данных. Он с радостью выполняет свои бизнес-функции, и в мире все хорошо. Но однажды руководитель команды приходит с новым бизнес-требованием. Оно связано с тем, что команда уже делает, и, пожалуй, может быть просто добавлено в их существующий сервис. Однако оно также в достаточной мере отличается от принятого контекста и может быть вынесено в свой собственный новый сервис.

Команда находится на перепутье: реализовывать им новые бизнес-требования в новый сервис или же просто добавить их в существующий? Давайте рассмотрим их варианты выбора подробнее.

Вариант 1: создать новый сервис

Бизнес-требования достаточно отличаются от принятых, и поэтому имеет смысл поместить их в новый сервис. Но как насчет данных? Новая бизнес-функция нуждается в некоторых старых данных, но эти данные в настоящее время заблокированы в существующем сервисе. Кроме того, у команды на самом деле нет процесса для запуска новых, полностью независимых сервисов. С другой стороны, команда становится больше, поскольку компания быстро растет. Если в будущем команду придется разделить, наличие модульных и независимых систем может значительно облегчить разделение обязанностей и прав владения.

С таким подходом связаны определенные риски. Команде нужно найти способ получить данные из исходного хранилища данных и скопировать их в новое хранилище данных. Этот способ должен обеспечить выполнение работы так, чтобы не выставлялась наружу внутренняя работа и чтобы изменения, которые он вносит в свои структуры данных, не повлияли на другие команды, которые копируют данные этой команды. Кроме того, копируемые данные всегда будут несколько устаревшими, поскольку команда может позволить себе копировать производственные данные в реальном времени только каждые 30 минут, чтобы не перегружать хранилище данных запросами. Это соединение необходимо будет контролировать и поддерживать, чтобы обеспечивать его правильную работу.

Существует также риск запуска и выполнения новых сервисов. Команде нужно будет управлять двумя хранилищами данных и двумя сервисами, а также создавать для них процессы журнализации, мониторинга, тестирования, развертывания и отката. Она также должна позаботиться о синхронизации любых изменений в структуре данных, чтобы не влиять на зависимую систему.

Вариант 2: добавить новое бизнес-требование в существующий сервис

Другой вариант — создать новые структуры данных и бизнес-логику в рамках существующего сервиса. Необходимые данные уже находятся в хранилище данных, а процессы журналирования, мониторинга, тестирования, развертывания и отката также определены и используются. Команда знакома с системой и может сразу приступить к реализации логики, а ее монолитные шаблоны поддерживают такой подход к дизайну сервисов.

С этим подходом также связаны риски, хотя их меньше. Границы внутри реализации могут размываться по мере внесения изменений, тем более что модули часто объединяются в одну кодовую базу. Слишком легко быстро добавлять функциональности, пересекая эти границы и образуя непосредственную сцепку через модуль. В быстром движении есть большое преимущество, но оно достигается за счет плотной сцепленности, пониженной связности и отсутствия модульности. Хотя команды могут от этого защититься, такая защита требует умелого планирования и строгого соблюдения границ, которые часто уходят на второй план перед лицом жестких графиков, неопытности и смены владельца сервиса.

Плюсы и минусы каждого варианта

Большинство команд выбрали бы второй вариант — добавление функциональности в существующую систему. В этом варианте выбора нет ничего плохого — монолитные архитектуры являются полезными и мощными структурами и могут обеспечивать исключительную ценность для предприятия. Первый вариант сталкивается лоб в лоб с двумя проблемами, связанными с традиционными вычислениями:

- ◆ доступ к данным другой системы трудно обеспечить надежно, в особенности в крупном масштабе и в реальном времени;
- ◆ создание и управление новыми сервисами сопряжено с существенными накладными расходами и рисками, в особенности если в организации нет установленного для этого способа.

Доступ к локальным данным всегда осуществляется проще, чем к данным в другом хранилище данных. Любые данные, инкапсулированные в хранилище данных другой команды, трудно получить, поскольку это требует пересечения границ обмена как бизнес-, так и технической информацией. По мере роста требований к данным, количеству подключений и производительности становится все труднее это поддерживать и масштабировать.

Хотя подход на основе копирования необходимых данных и вполне приемлем, он не является надежным. Эта модель поощряет много прямых связей «точка-точка», которые становится проблематично поддерживать по мере роста организации, изменения бизнес-единиц и владельцев, а также по мере устаревания продуктов и постепенного отказа от них. Это создает строгую техническую зависимость между структурами обмена информацией в обеих командах (команде, хранящей данные,

и команде, копирующей их), требуя, чтобы они работали синхронно всякий раз, когда происходит изменение данных. Особое внимание должно быть уделено тому, чтобы внутренняя реализованная модель данных не раскрывалась и не выставлялась чрезмерно наружу во избежание того, чтобы другие системы не завязывали на нее свою логику. Масштабируемость, производительность и доступность системы часто вызывает трудности для обеих систем, поскольку запрос на репликацию данных может создавать непосильную нагрузку на исходную систему. Отказ процессов синхронизации также может быть не замечен до тех пор, пока не возникнет аварийная ситуация. Такое своего рода «племенное знание»³ может привести к тому, что команда скопирует себе данные, полагая, что берет их из истинного первоисточника.

Скопированные данные всегда будут несколько устаревшими к моменту завершения запроса и передачи данных. Чем больше набор данных и чем сложнее его источник, тем больше вероятность того, что копия не будет синхронизирована с оригиналом. Это создает проблемы, когда системы исходят из того, что каждая из них обладает идеальными обновленными копиями, в особенности при обмене между собой информацией, касающейся этих данных. Например, из-за устаревания данных сервис отчетов может сообщать иные значения, чем сервис начисления платежей. Это может иметь серьезные последствия для качества сервисов, отчетности, аналитики и принятия финансовых решений.

Неспособность правильно распространять данные по всей компании не связана с фундаментальным недостатком концепции. Совсем наоборот — это происходит из-за слабой или несуществующей структуры обмена данными. В приведенном ранее сценарии структура обмена информацией в команде выполняет вторую функцию чрезвычайно ограниченной структуры обмена данными.



Один из догматов событийно-управляемых микросервисов состоит в том, что стержневые бизнес-данные должны находиться в легком доступе и использоваться любым сервисом, который в них нуждается. В рассмотренном сценарии структура нерегламентированного обмена данными замещается формализованной структурой обмена данными. Для гипотетической команды такая структура обмена данными могла бы устраниТЬ большинство трудностей, связанных с получением данных из других систем.

Командный сценарий: продолжение

Перенесемся на год вперед. Команда решила остановиться на варианте 2 и включить новые функциональности в имеющийся сервис. Сделать это было быстро и легко, и с тех пор они добавили в него ряд новых функциональностей. По мере роста команды росла и компания, и теперь пришло время реорганизовать эту команду в две более мелкие, более целенаправленные команды.

³ «Племенное знание» (tribal knowledge), или коллективная мудрость, — это информация или знание, которое известно внутри племени, но часто неизвестно за его пределами. Племя в этом смысле может быть группой или подгруппой людей, которые разделяют такое общее знание. — Прим. перев.

Теперь за каждой новой командой должны быть закреплены определенные бизнес-функции из предыдущего сервиса. Бизнес-требования у каждой команды четко разделены в зависимости от предметной области компании, которая требует наибольшего внимания. Однако разделить структуру обмена информацией оказывается непростым делом. Как и прежде, похоже, что для выполнения своих требований обе команды нуждаются в больших объемах одних и тех же данных. Возникают новые вопросы:

- ◆ какая команда должна владеть какими данными;
- ◆ где должны находиться данные;
- ◆ как быть с данными, в которых обе команды должны модифицировать значения?

Руководители команд решают, что, возможно, лучше будет просто поделиться сервисом и обе они смогут работать над разными частями данных. Это потребует гораздо большего обмена информацией между командами и синхронизации усилий, что может негативно сказаться на производительности. А как быть в будущем, если они снова удвоются в размере? Или если бизнес-требования изменятся настолько, что они больше не смогут выполнять все свои обязанности с той же структурой данных?

Конфликтующие обстоятельства

На исходную команду оказывают влияние два конфликтующих обстоятельства. Она вынуждена держать все свои данные локально в одном сервисе, чтобы ускорить и упростить добавление новой бизнес-функциональности за счет расширения структуры обмена. Впоследствии рост команды потребовал разделения структуры обмена бизнес-информацией — необходимость, возникающая вслед за перезакреплением бизнес-требований за новыми командами. Однако структура обмена информацией не может поддерживать изменения в их нынешнем виде и должна быть разбита на подходящие компоненты. Ни один из подходов не является масштабируемым, и оба указывают на необходимость выполнять работу по-разному. Все эти проблемы происходят из одной и той же корневой причины — слабых, плохо определенных средств обмена данными между сервисами.

Событийно-управляемые коммуникационные структуры

Событийно-управляемый подход предлагает альтернативу традиционному поведению структур обмена информацией и обмена данными. Событийно-управляемый обмен — это не прямая замена для обмена «запрос-ответ», а совершенно иной способ обмена информацией между сервисами. Обмен данными о событиях разделяет производство данных и владение ими от доступа к ним. Сервисы больше не сцеплены напрямую через API запросов-ответов, а вместо этого взаимодействуют через события, определенные в потоках событий (этот процесс подробнее рассматривается в главе 3). Обязанность производителей — создавать и передавать четко определенные данные в соответствующие потоки событий.

События являются основой обмена информацией

Все совместно используемые данные публикуются в наборе потоков событий, образуя непрерывный поток, подробно рассказывающий обо всем том, что произошло в организации. Он становится каналом, по которому системы обмениваются между собой. В качестве события может быть передано почти все что угодно — от простых уведомлений до сложных записей с поддержкой состояния. События являются данными — это не просто сигналы, указывающие на то, что где-то еще данные готовы, либо только средство прямой передачи данных от одной системы к другой. Скорее наоборот, они действуют и как хранилище данных, и как средство асинхронного обмена информацией между сервисами.

Потоки событий обеспечивают единственный источник истины

Каждое событие в потоке — это констатация факта, и вместе эти констатации образуют единый источник истины, являющийся основой обмена информацией для всех систем внутри организации. Структура обмена информацией хороша лишь настолько, насколько достоверна ее информация, поэтому крайне важно, чтобы организация приняла событийно-потоковый нарратив в качестве единственного источника истины. Если некоторые коллективы решат вместо этого поместить конфликтующие данные в другие места, то значимость функции потока событий как становового хребта обмена данными в организации значительно уменьшится.

Потребители занимаются своим моделированием и выполняют свои запросы

Событийно-управляемая структура обмена информацией отличается от чрезмерно расширенной структуры обмена информацией тем, что она не способна обеспечить какую-либо функциональность выполнения запросов или поиска данных. Вся бизнес-логика и логика приложения должны быть инкапсулированы внутри производителя и потребителя событий.

Требования по доступу к данным и моделированию полностью переложены на потребителя, причем каждый потребитель получает свою собственную копию событий из исходных потоков событий. Любая сложность запроса также смещается от структуры обмена информацией источника в сторону структуры потребителя. Потребитель остается полностью ответственным за любое смешивание данных из многочисленных потоков событий, специальной функциональности запросов или другой специфичной для предприятия логики. В противном случае как производители, так и потребители освобождаются от обязанности предоставлять механизмы осуществления запросов, механизмы передачи данных, API (интерфейсы прикладного программирования) и трансграничные сервисы для средств обмена данными. Теперь они ограничены только решением потребностей своего непосредственного ограниченного контекста.

Обмен данными улучшается по всей организации

Использование структуры обмена данными — это инверсия, когда все совместно используемые данные выставляются за пределы структуры обмена информацией. Не все данные должны использоваться совместно, и, следовательно, не все они должны публиковаться в наборе потоков событий. Однако любые данные, вызывающие интерес у любой другой команды или сервиса, должны публиковаться в общем наборе потоков событий таким образом, чтобы производство и владение данными были полностью разграничены друг от друга. Благодаря этому, обеспечивается структура формализованного обмена данными, которая долгое время отсутствовала в системных архитектурах и которая лучше соответствует ограниченно-контекстным принципам слабой связанности и сильной сцепленности.

Приложения теперь могут обращаться к данным, доступ к которым в противном случае было бы трудно получить посредством соединений «точка-точка». Новые сервисы могут просто получать любые необходимые данные из канонических потоков событий, создавать свои собственные модели и состояния, а также выполнять любые необходимые деловые функции, не зависящие от прямых соединений «точка-точка» или API с любым другим сервисом. Это открывает перед организацией возможность эффективнее использовать свои огромные объемы данных в любом продукте и даже уникально и мощно смешивать данные из многочисленных продуктов.

Доступные данные поддерживают изменения в обмене информацией

Потоки событий содержат ключевые события предметной области, которые занимают центральное положение в функционировании предприятия. Хотя команды могут реорганизоваться, а проекты — приходить и уходить, важные ключевые данные предметной области остаются легкодоступными для любого нового продукта, который этого требует, независимо от какой-либо конкретной структуры обмена информацией. Это дает предприятию беспрецедентную гибкость, поскольку доступ к ключевым событиям предметной области больше не зависит от какой-либо конкретной реализации.

Асинхронные событийно-управляемые микросервисы

Событийно-управляемые микросервисы позволяют выполнять трансформации и операции бизнес-логики, необходимые для удовлетворения требований ограниченного контекста. Таким приложениям поручено выполнять указанные требования и генерировать любые собственные необходимые события другим связанным потребителям. Вот несколько первичных выгод от использования событийно-управляемых микросервисов:

- ◆ *Гранулярность.*

Сервисы аккуратно укладываются в ограниченные контексты и могут быть легко переписаны при изменении деловых требований.

◆ **Масштабируемость.**

Отдельные сервисы могут масштабироваться вверх и вниз по мере необходимости.

◆ **Технологическая гибкость.**

Сервисы используют наиболее подходящие языки и технологии. Это также позволяет легко создавать прототипы с использованием передовых технологий.

◆ **Гибкость бизнес-требований.**

Владение гранулированными микросервисами легко реорганизуется. По сравнению с крупными сервисами существует меньше зависимостей между командами и организация может быстрее реагировать на изменения бизнес-требований, которые в противном случае были бы нарушены барьерами доступа к данным.

◆ **Слабая сцепленность.**

Событийно-управляемые микросервисы сцеплены с данными предметной области, а не с API конкретной реализации. Схемы данных могут использоваться для того, чтобы значительно улучшить управление изменениями данных, как это будет показано в главе 3.

◆ **Поддержка непрерывной поставки.**

Легко поставлять малый модульный микросервис и при необходимости откатывать ее назад.

◆ **Высокая тестопригодность.**

Микросервисы, как правило, имеют меньше зависимостей, чем крупные монолиты, что облегчает выполнение имитации необходимых тестовых конечных точек и обеспечивает надлежащее кодовое покрытие.

Пример команды, использующей событийно-управляемые микросервисы

Давайте вернемся к упомянутой ранее команде, но со структурой событийно-управляемого обмена данными.

В команде вводится новое бизнес-требование. Оно в какой-то степени связано с тем, что делают их нынешние продукты, но оно также достаточно отличается, чтобы его можно было бы направить в свой собственный сервис. Не нарушает ли его добавление в существующий сервис *принцип единственной ответственности* (*single responsibility*)⁴ и не расширяет ли оно в настоящее время определенный ограниченный контекст? Или же оно является простым расширением существующего сервиса, возможно, за счет добавления каких-либо новых родственных данных или функциональности?

⁴ Одна из формулировок указанного принципа заключается в том, что каждый объект должен иметь только одну причину для действия. — Прим. перев.

Предыдущие технические трудности — такие как выяснение того, откуда брать исходные данные и как загружать данные в приемник, обработка трудностей пакетной синхронизации и имплементация синхронных API, — теперь в значительной степени устранены. Команда может запустить новый микросервис и при необходимости извлекать необходимые данные из потоков событий, когда это ей понадобится. Вполне возможно, что команда будет подмешивать к ним общие данные, используемые в других своих сервисах, при условии, что эти данные используются исключительно для удовлетворения требований нового ограниченного контекста. Хранение и структура этих данных полностью зависят от команды, которая может выбирать, какие поля оставлять, а какие выбрасывать.

Бизнес-риски также снижаются, поскольку малые, более мелкие сервисы обеспечивают возможность их владения одной командой, позволяя командам масштабироваться и реорганизовываться по мере необходимости. Когда команда становится слишком большой, чтобы ею мог управлять сам владелец предприятия, она может подразделяться по мере необходимости и за ней может закрепляться владение микросервисами. Владение данными о событиях перемещается вместе с сервисом-производителем, а для уменьшения объема обмена информацией между командами, необходимого для выполнения будущей работы, могут приниматься соответствующие организационные решения.

Природа микросервисов не дает укорениться спагетти-коду и расширениям монолита при условии, что накладные расходы на создание новых сервисов и получение необходимых данных минимальны. Проблемы масштабирования теперь сосредоточены на индивидуальных сервисах обработки событий, и при этом эти сервисы могут масштабировать свой центральный процессор, память, диск и количество экземпляров по мере необходимости. Остальные требования к масштабированию перекладываются на структуру обмена данными, которая должна обеспечивать возможность обработки различных нагрузок сервисов, потребляющих из своих потоков события и производящих события в них.

Однако для того, чтобы все это сделать, команда должна обеспечить, чтобы данные действительно присутствовали в структуре обмена данными, и она должна иметь средства для легкого развертывания и управления парком микросервисов. Это требует общеорганизационного внедрения архитектуры событийно-управляемых микросервисов (Event-Driven Microservice, EDM).

Синхронные микросервисы

Микросервисы могут быть реализованы асинхронно с использованием событий (подход, который отстает в этой книге) или синхронно, что часто встречается в архитектурах с ориентацией на сервисы. Синхронные микросервисы, как правило, выполняются с использованием подхода «запрос-ответ», когда для выполнения бизнес-требований сервисы обмениваются данными непосредственно через API.

Недостатки синхронных микросервисов

Существует ряд проблем с синхронными микросервисами, которые затрудняют их использование в крупных масштабах. Это не означает, что компания не может добиться успеха, используя синхронные микросервисы, о чем свидетельствуют достижения таких компаний, как Netflix, Lyft, Uber и Facebook. Но многие компании также сколотили состояния, используя архаичные и ужасно запутанные монолиты из спагетти-кода, поэтому не следует путать конечный успех компаний с качеством ее опорной архитектуры. В ряде книг описываются способы реализации синхронных микросервисов, поэтому рекомендую прочитать их, чтобы лучше понять синхронные подходы⁵.

Более того, обратите внимание на то, что ни микросервисы «точка-точка» типа «запрос-ответ», ни асинхронные событийно-управляемые микросервисы не являются строго лучше одни других. И то и другое имеет свое место в организации, поскольку некоторые задачи гораздо лучше подходят для одной или другой из них. Однако мой собственный опыт, а также опыт многих моих знакомых и коллег показывает, что архитектура EDM обеспечивает беспрецедентную гибкость, отсутствующую в синхронных микросервисах «запрос-ответ». Возможно, вы придетете к такому же выводу, когда будете читать эту книгу. По меньшей мере вы поймете их сильные и слабые стороны.

Вот несколько самых больших недостатков синхронных микросервисов «запрос-ответ».

Связь «точка-точка»

Синхронные микросервисы опираются на другие сервисы, которые помогают им выполнять свои бизнес-задачи. Эти сервисы, в свою очередь, имеют свои собственные зависимые сервисы, которые имеют свои собственные зависимые сервисы, и т. д. Это может привести к чрезмерному разветвлению и трудностям в отслеживании того, какие сервисы отвечают за выполнение конкретных частей бизнес-логики. Количество соединений между сервисами может стать ошеломляюще большим, что еще сильнее укрепляет существующие структуры обмена информацией и затрудняет будущие изменения.

Зависимое масштабирование

Возможность масштабирования собственного сервиса зависит от способности всех зависимых сервисов также масштабироваться и напрямую связана со степенью разветвления обмена информацией. Узким местом при масштабировании могут стать технологии реализации. Это дополнительно осложняется сильно изменяющимися нагрузками и скачкообразными нагрузками из поступающих запросов, все из которых должны обрабатываться синхронно во всей архитектуре.

⁵ См., например, книгу «Создание микросервисов» Сэма Ньюмана (Building Microservices, Sam Newman, o'Reilly, 2015) и книгу «Микросервисы для предприятия» Касуна Индрасири и Прабата Сиривардена (Microservices for the Enterprise, Kasun Indrasiri, Prabath Siriwardena, Apress, 2018) (<https://oreil.ly/FPtLm>).

Обработка отказов сервисов

Если зависимый сервис неисправен, то необходимо принять решение о том, как обрабатывать исключение. Принимать решения о том, как справляться с перебоями, когда повторять попытку, когда выдавать отказ и как восстанавливать данные, чтобы обеспечить согласованность данных, становится все труднее, чем больше сервисов существует в экосистеме.

Управление версиями API и зависимостями

Часто требуется одновременное существование нескольких определений API и версий сервисов. Не всегда возможно или желательно заставлять клиентов переходить на самый последний API. Это может значительно усложнить оркестровку запросов на изменение API между многочисленными сервисами, в особенности если они сопровождаются изменениями в базовых структурах данных.

Доступ к данным с привязкой к реализации

Синхронные микросервисы имеют те же проблемы, что и традиционные сервисы, в том, что касается доступа к внешним данным. Несмотря на то что существуют стратегии дизайна сервисов, которые уменьшают необходимость доступа к внешним данным, микросервисам нередко все-таки требуется доступ к часто используемым данным из других сервисов. Это усложняет доступ к данным, и возможности масштабируемости снижаются до решений на основе обмена информацией.

Распределенные монолиты

Сервисы могут композиционно составляться таким образом, что они действуют как распределенный монолит, причем между ними совершаются много взаимопереплетенных вызовов. Такая ситуация часто возникает, когда команда разделяет монолит на компоненты и решает использовать синхронные вызовы «точка-точка», чтобы имитировать существующие границы внутри своего монолита. Сервисы «точка-точка» позволяют легко размывать границы между ограниченными контекстами, т. к. функциональные вызовы внешних систем могут вставляться построчно с существующим монолитным кодом.

Тестирование

Выполнять интеграционное тестирование сложно, т. к. каждый сервис требует полностью функционирующих зависимостей, которые, в свою очередь, требуют своих собственных. Установка заглушек может работать для модульных тестов, но редко оказывается достаточной для более обширных потребностей в тестировании.

Преимущества синхронных микросервисов

Синхронные микросервисы предоставляют ряд неоспоримых преимуществ. Некоторые шаблоны доступа к данным благоприятны для прямых цепочек «запрос-ответ» — таких как аутентификация пользователя и отчет об АВ-тесте. Интеграция

со сторонними решениями почти всегда использует синхронный механизм и, как правило, обеспечивает гибкий, не зависящий от языка механизм обмена по протоколу HTTP.

Операции трассировки между несколькими системами могут быть проще в синхронной среде, чем в асинхронной. Подробные журналы могут показывать, какие функции были вызваны в каких системах, что обеспечивает высокую отлаживаемость и видимость бизнес-операций.

Сервисы, размещающие на хосте веб- и мобильные приложения, в основном основаны на дизайне в стиле «запрос-ответ», независимо от их синхронной или асинхронной природы. Клиенты получают своевременный ответ, полностью соответствующий их потребностям.

Фактор наличия опыта у команды разработки также весьма важен, тем более что многие разработчики на современном рынке, как правило, гораздо опытнее в синхронном кодировании в стиле монолита. Это делает приобретение навыков для синхронных систем в целом проще, чем приобретение навыков для асинхронной событийно-управляемой разработки.



Архитектура компании редко, если вообще когда-либо, будет полностью основана на событийно-управляемых микросервисах. Гибридные архитектуры, безусловно, останутся нормой, когда синхронные и асинхронные решения развертываются бок о бок, как того требует пространство решений.

Резюме

Структуры обмена информацией определяют то, как программное обеспечение создается и управляет на протяжении всей жизни организации. Структуры обмена данными часто являются недостаточно развитыми и нерегламентированными, но введение прочного, простого в доступе набора событий предметной области, воплощенного в событийно-управляемых системах, делает возможными меньшие по размеру функциональные реализации сервисов.

Основы событийно-управляемых микросервисов

Событийно-управляемый микросервис — это малое приложение, построенное для исполнения конкретного ограниченного контекста. *Потребляющие* микросервисы потребляют и обрабатывают события из одного или нескольких входных потоков событий, в то время как *производящие* микросервисы производят события в потоки событий для использования другими сервисами. Обычно событийно-управляемый микросервис является потребителем одного набора входных потоков событий и производителем другого набора выходных потоков событий. Эти сервисы могут как поддерживать состояние (см. главу 5), так и не поддерживать его (см. главу 7), а также могут содержать синхронные API «запросов-ответов» (см. главу 13). Все эти сервисы имеют общую функциональность получения событий от брокера событий или их передачи брокеру событий. Обмен информацией между событийно-управляемыми микросервисами является полностью асинхронным.

Потоки событий обслуживаются *брокером событий*, который более подробно рассматривается в этой главе далее. Выполнение микросервисов в любом значимом масштабе часто требует использования конвейеров развертывания и систем управления контейнерами, также обсуждаемых в конце этой главы.

Построение топологий

Термин «топология» нередко встречается в обсуждениях событийно-управляемых микросервисов. Он часто используется для обозначения обрабатывающей логики отдельных микросервисов. Его также можно применить для обозначения графо-подобной связи между отдельными микросервисами, потоками событий и API «запросов-ответов». Давайте рассмотрим каждое определение по очереди.

Топология микросервисов

Топология микросервисов — это событийно-управляемая топология, внутренняя по отношению к одному микросервису. Она определяет управляемые данными операции, которые должны выполняться над входящими событиями, включая трансформацию, хранение и эмиссию.

На рис. 2.1 показана топология микросервиса, принимающего два входных потока событий.

Топология микросервиса принимает события из потока событий А и материализует их в хранилище данных. Операция материализации подробнее рассматривается

в этой главе далее. Тем временем поток событий В принимается, некоторые события отфильтровываются, трансформируются и затем объединяются с сохраненным состоянием. Результаты выводятся в новый поток событий. Прием событий, обработка и вывод их в выходной поток являются частями топологии микросервиса.

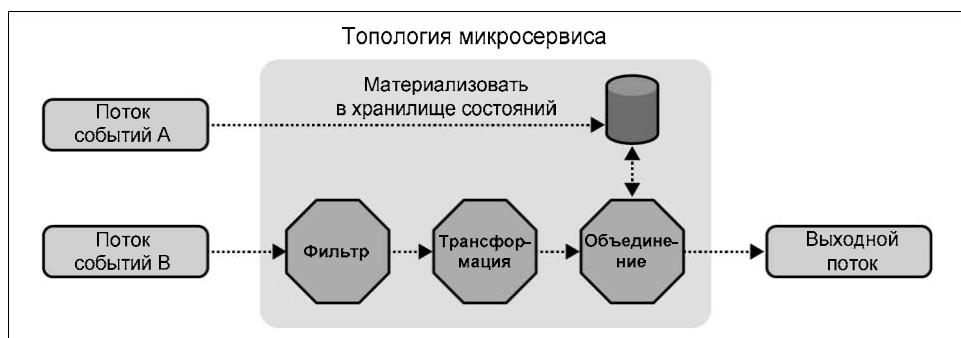


Рис. 2.1. Простая топология микросервиса

Бизнес-топология

Бизнес-топология — это набор микросервисов, потоков событий и API, выполняющих сложные бизнес-функции. Она является произвольной группой сервисов и может представлять сервисы, принадлежащие одной команде или отделу, а также сервисы, которые выполняют набор сложных бизнес-функций. Бизнес-топологию составляют структуры обмена бизнес-информацией, подробно описанные в главе 1. Микросервисы реализуют ограниченные контексты, а потоки событий обеспечивают механизм обмена данными для совместного использования трансграничных данных предметной области.



Топология микросервисов детализирует внутреннюю работу одного микросервиса. Бизнес-топология, напротив, детализирует связи между сервисами.

На рис. 2.2 показана бизнес-топология с тремя независимыми микросервисами и потоками событий. Обратите внимание, что бизнес-топология не детализирует внутреннюю работу микросервисов.

Микросервис 1 получает и обрабатывает данные из потока событий А и передает результаты в поток событий В. Микросервис 2 и микросервис 3 получают данные из потока событий В. Микросервис 2 действует строго как потребитель и предоставляет REST API, в котором данные могут быть доступны синхронно извне. Между тем микросервис 3 выполняет свои собственные функции в соответствии со своими требованиями по ограниченному контексту и выводит результатирующие данные в поток событий С. Новые микросервисы и потоки событий могут добавляться в бизнес-топологию по мере необходимости, объединяясь асинхронно через потоки событий.

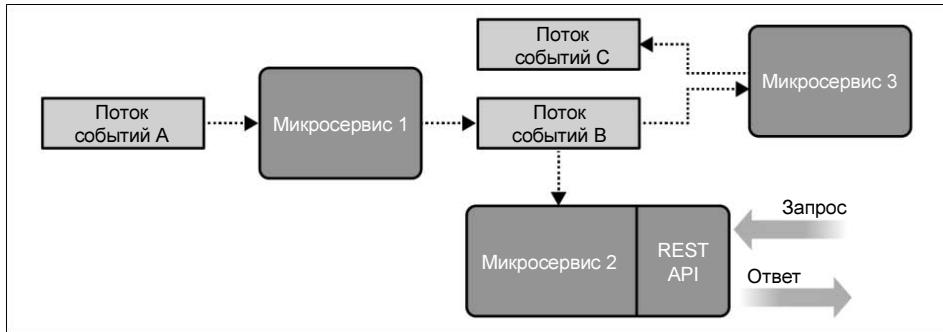


Рис. 2.2. Простая бизнес-топология

Содержимое события

Событием может быть *все*, что произошло в рамках структуры обмена деловой информацией. Получение счета, бронирование конференц-зала, заказ чашки кофе (да, вы можете подключать к потоку событий кофеварку), наем нового сотрудника и успешное выполнение произвольного кода — все это примеры событий, которые происходят на предприятии. Важно понимать, что события могут быть чем угодно, что для предприятия важно. Как только эти события начинают создаваться и передаваться, можно создавать событийно-управляемые системы, чтобы брать их под контроль и использовать по всей организации.

Событие — это запись того, что произошло, подобно тому, как журналы ошибок приложения и другой его информации регистрируют то, что происходит в приложении. Однако, в отличие от этих журналов, события также являются единственным источником истины, как показано в главе 1. Соответственно, они должны содержать всю информацию, необходимую для точного описания того, что произошло.

Структура события

События обычно представляются в формате ключ/значение. Значение хранит полную информацию о событии, в то время как ключ используется для целей идентификации, маршрутизации и агрегирования событий с *одним и тем же ключом*. Ключ не является обязательным полем для всех типов событий.

Ключ	Значение
Уникальный ID	Подробная информация, относящаяся к уникальному ID

Есть три главных типа событий, которые будут использоваться на протяжении всей этой книги и с которыми вы неизбежно столкнетесь в своих предметных областях.

Событие без ключа

События без ключа используются, чтобы описывать событие исключительно как констатацию факта. Примером может служить событие, указывающее на то, что клиент взаимодействовал с продуктом, — например, пользователь, открывавший сущность «Книга» на цифровой книжной платформе. Как следует из названия, в этом событии нет ключа.

Ключ	Значение
N/A	ISBN: 372719, метка времени: 1538913600

Сущностное событие

Сущностное событие является *уникальным объектом*, и доступ к его экземплярам осуществляется по уникальному ID экземпляра этого события. Сущностное событие описывает свойства и состояние сущности — чаще всего объекта в бизнес-контексте в текущий момент времени. В книжном издаельстве примером сущностного события может быть сущность «Книги» с ключом ISBN. Поле «Значение» содержит всю необходимую информацию, относящуюся к уникальному экземпляру сущности.

Ключ	Значение
ISBN: 372719	Автор: Адам Беллемар

Сущностные события особенно важны в событийно-управляемой архитектуре. Они обеспечивают непрерывную историю состояния сущности и могут использоваться для материализации состояния (рассматривается в следующем разделе). Для определения текущего состояния сущности необходимо только самое позднее сущностное событие.

Событие с ключом

Событие с ключом содержит ключ, но не представляет сущность. События с ключом обычно используются для разделения потока событий, чтобы обеспечить локальность данных в пределах одного раздела потока событий (подробнее об этом позже в этой главе). Примером может служить поток событий, выбираемый по ключу ISBN и указывающий на то, какой пользователь взаимодействовал с книгой.

Ключ	Значение
ISBN: 372719	UserId: A537FE
ISBN: 372719	UserId: BB0012

Обратите внимание, что события могут агрегироваться по ключу таким образом, что для каждого ISBN может быть составлен список пользователей, в результате чего будет получено одно сущностное событие, выбранное по ключу ISBN.

Материализация состояния из сущностных событий

Вы можете *материализовать* (materialize) таблицу с поддержкой состояний сущности, применяя сущностные события по порядку из их потока. Каждое сущностное событие *вставляется* (upserting) в таблицу ключ/значение таким образом, чтобы для каждого ключа в ней было представлено самое последнее событие чтения. И наоборот, вы можете конвертировать таблицу в поток сущностных событий, публикуя каждое обновление в потоке событий. Это называется *дуальностью табличного потока*, и она лежит в основе создания состояния в событийно-управляемом микросервисе. Это показано на рис. 2.3, где и AA, и CC имеют самые новые значения в своей материализованной таблице состояний.



Термин *upserting* (от update — обновить и insert — вставить), который можно представить как «обновление-вставка», подразумевает вставку в таблицу новой строки, если она там еще не существует, либо ее обновление, если существует.

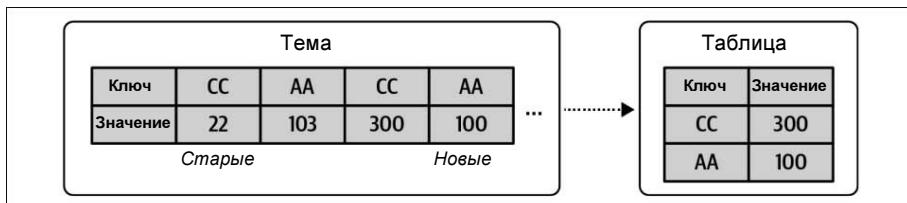


Рис. 2.3. Материализация потока событий в форме таблицы

Точно так же вы можете регистрировать в таблице все обновления и при этом создавать поток данных, представляющих состояние таблицы во временной динамике. В следующем примере (рис. 2.4) BB «обновляется-вставляется» (Upsert) дважды, в то время как DD — только один раз. Выходной поток здесь показывает три события обновления-вставки, представляющие эти операции.

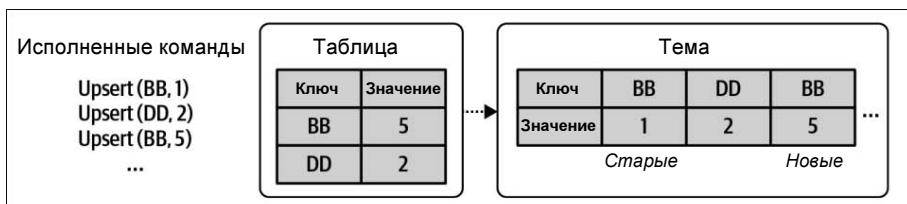


Рис. 2.4. Генерирование потока событий из изменений, примененных к таблице

Как можно видеть, таблица реляционной базы данных создается и заполняется с помощью серии команд вставки, обновления и удаления данных. Эти команды могут создаваться как события в неизменяемом (immutable) журнале выполненных операций (таком как локальный файл только для добавления записей — например, двоичный журнал в MySQL) либо как внешний поток событий. Воспроизведя все содержимое журнала, вы можете точно восстановить таблицу и все ее содержимое.



Эта двойственность табличного потока используется для обмена состояниями между событийно-управляемыми микросервисами. Любой потребляющий клиент может прочитать поток событий с ключами и материализовать его в своем локальном хранилище состояний. Этот простой, но мощный шаблон позволяет микросервисам делиться состоянием только через события, без какой-либо прямой сцепки между производящим и потребляющим сервисами.

Удаление события с ключом обрабатывается созданием *отметки об удалении* (tombstone). Отметка об удалении — это событие с ключом, значение которого равно `null`. Такое условное обозначение указывает потребителю на то, что событие с этим ключом должно быть удалено из материализованного хранилища данных, поскольку вышестоящий производитель объявил, что оно теперь удалено.

Неизменяемые журналы, в которые информация только добавляется, могут расти бесконечно, если их не уплотнять. *Уплотнение* (сжатие) выполняется брокером событий для уменьшения размера его внутренних журналов путем сохранения только самого последнего события для каждого конкретного ключа. Старые события того же ключа будут удалены, а оставшиеся события будут сжаты в новый и меньший набор файлов. Смещения потока событий поддерживаются таким образом, что для этого не требуется внесения каких бы то ни было изменений со стороны потребителей. На рис. 2.5 показано логическое уплотнение потока событий в брокере событий, включающее полное удаление записей, помеченных отметкой об удалении.



Рис. 2.5. После уплотнения для ключа сохраняется только самая последняя запись — все записи, помеченные отметкой на удаление, и их предшественники того же ключа удаляются

Уплотнение — за счет исключения истории событий — сокращает как использование диска, так и количество событий, которые должны быть обработаны для достижения текущего состояния, которое в противном случае предоставляется полным потоком событий.

Поддержание состояния для обработки бизнес-логики является чрезвычайно распространенным шаблоном в событийно-управляемой архитектуре. Почти наверняка вся ваша бизнес-модель не сможет вписаться в чисто потоковую предметную область — без поддержки состояния, поскольку прошлые решения будут влиять на решения, которые вы принимаете сегодня. Так, если вы, например, занимаетесь розничной торговлей, то вам нужно знать уровень запасов, чтобы выявлять моменты, когда настанет время размещать повторные заказы и избегать продаж клиентам товаров, которых у вас нет. Вы также захотите иметь возможность отслеживать

свою кредиторскую и дебиторскую задолженности. Возможно, вы захотите, чтобы информация о еженедельной рекламной акции рассыпалась всем клиентам, которые предоставили вам свои адреса электронной почты. Все эти системы требуют, чтобы у вас была возможность материализовывать потоки событий в представления текущего состояния.

Определения и схемы данных событий

Событийные данные служат средством долгосрочного и не зависящего от реализации хранения данных, а также механизмом обмена информацией между сервисами. Поэтому важно, чтобы и производители, и потребители событий имели общее понимание смысла данных. В идеале потребитель должен иметь возможность интерпретировать содержание и смысл события, не консультируясь с владельцем сервиса-производителя. Это требует общего языка для обмена информацией между производителями и потребителями и схоже с определением API запросов-ответов между синхронными сервисами.

Продукты, обеспечивающие схематизацию выборок, такие как Apache Avro (<https://avro.apache.org>) и Google Protobuf (<https://oreil.ly/zth68>), предоставляют две функциональности, которые в значительной степени задействуются в событийно-управляемых микросервисах. Во-первых, они обеспечивают фреймворк, в котором некоторые наборы изменений могут безопасно вноситься в схемы, не требуя от нижестоящих потребителей внесения изменений в код. Во-вторых, они также предоставляют средства для генерирования типизированных классов (где это применимо) для конвертирования данных в объекты на выбранном вами языке программирования. Такие возможности намного упрощают создание бизнес-логики при разработке микросервисов и делают это прозрачнее. В главе 3 эти темы рассматриваются подробнее.

Принцип единственного источника событий

Каждый поток событий генерируется одним и только одним микросервисом-производителем. Этот микросервис является владельцем каждого события, произведенного в соответствующем потоке. При этом любое событие знает об авторитетном источнике истины, что позволяет отслеживать происхождение данных во всей системе. Для установления владельца данных и обеспечения ограничений записи должны использоваться механизмы контроля доступа, описанные в главе 14.

Взаимодействие микросервисов с помощью брокера событий

В основе каждой платформы на основе событийно-управляемых микросервисов лежит *брокер событий*. Он представляет собой систему, которая получает события, сохраняет их в очереди или потоке событий и предоставляет их для использования другими процессами. События обычно публикуются в разных потоках на основе их

логического смысла, подобно тому, как база данных включает много таблиц, каждая из которых предназначена для хранения специфического типа данных.

Системы с использованием брокера событий, подходящие для крупных предприятий, как правило, следуют той же модели. Многочисленные распределенные брокеры событий работают вместе в кластере, обеспечивая платформу для производства и потребления потоков событий. Эта модель предоставляет функции, необходимые для реализации событийно-управляемой экосистемы в крупном масштабе:

◆ *Масштабируемость.*

В целях увеличения производительности, потребления и емкости хранилища данных в кластер могут быть добавлены дополнительные экземпляры брокера событий.

◆ *Долговечность.*

Событийные данные реплицируются между узлами. Это позволяет кластеру брокеров как сохранять, так и продолжать передавать данные, когда брокер отказывает.

◆ *Высокая доступность.*

Кластер узлов брокера событий позволяет клиентам подключаться к другим узлам в случае отказа брокера. Это позволяет клиентам поддерживать безотказную работу.

◆ *Высокая производительность.*

Многочисленные узлы с брокерами совместно несут нагрузку производства и потребления событий. В дополнение к этому каждый узел брокера должен быть высокоэффективным, чтобы иметь возможность обрабатывать сотни тысяч операций записи или чтения в секунду.

Хотя брокеры событий могут реализовывать хранение, репликацию и доступ к событийным данным разными способами, все они, как правило, обеспечивают своим клиентам одинаковые механизмы хранения и доступа.

Хранение и обработка событий

Далее приведены минимальные требования к обработке данных брокером:

◆ *Партиционирование.*

Потоки событий могут быть подразделены (партиционированы) на отдельные подпотоки, число которых может варьироваться в зависимости от потребностей производителя и потребителя. Такой механизм разделения позволяет многочисленным экземплярам потребителя обрабатывать каждый подпоток параллельно, что обеспечивает гораздо большую пропускную способность. Обратите внимание, что *очереди* не требуют разделения, хотя в любом случае их может быть полезно разбивать на подразделы для повышения производительности.

◆ *Строгая упорядоченность.*

Данные в подразделе потока событий строго упорядочены и подаются клиентам в том же порядке, в каком они были первоначально опубликованы.

◆ *Неизменяемость.*

Все событийные данные неизменны после публикации. Не существует механизма, который мог бы изменить событийные данные после их публикации. Вы можете изменить предыдущие данные, только опубликовав новое событие с обновленными данными.

◆ *Индексация.*

Событиям при записи в поток событий назначается индекс. Он используется потребителями для управления обработкой данных, поскольку они могут указывать, с какого смещения начинать чтение. Разница между текущим индексом потребителя и конечным его индексом представляет собой *запаздывание* потребителя. Эту метрику можно использовать для увеличения числа потребителей, когда она высока, и их уменьшения, когда она низка. Кроме того, она также может действовать для реализации логики «Функции как сервис».

◆ *Бесконечное хранение.*

Потоки событий должны быть способны хранить события в течение бесконечного периода времени. Это свойство является основополагающим для поддержания состояния в потоке событий.

◆ *Воспроизведимость.*

Потоки событий должны быть воспроизводимыми, чтобы любой потребитель мог читать любые необходимые ему данные. Благодаря этому обеспечивается база для единого источника истины и основа для обмена состояниями между микросервисами.

Дополнительные факторы, которые следует учитывать

Существует ряд дополнительных факторов, которые необходимо учитывать при выборе брокера событий:

◆ *поддерживающий инструментарий* — он необходим для эффективной разработки событийно-управляемых микросервисов. Многие из этих инструментов связаны с реализацией самого брокера событий. Некоторые из них включают в себя:

- просмотр событийных и схемных данных;
- квоты, контроль доступа и управление темами (топиками);
- показания мониторинга, пропускной способности и задержек.

Более подробная информация об инструментарии, который может вам понадобиться, приведена в главе 14;

◆ *сервисы провайдера* — размещенные у провайдера сервисы позволяют вам передать на аутсорсинг создание вашего брокера событий и управление им. При этом следует убедиться в следующем:

- существуют ли размещенные на хосте решения;
- приобретете ли вы размещеннное на хосте решение или разместите его внутри компании;

- обеспечивает ли провайдер мониторинг, масштабирование, аварийное восстановление, репликацию и многозональное развертывание;
 - связывает ли это вас с одним конкретным поставщиком сервисов;
 - существуют ли у провайдера профессиональные службы поддержки?
- ◆ *клиентские библиотеки и фреймворки обработки* — существует широкий выбор многочисленных реализаций брокера событий, каждая из которых имеет различные уровни поддержки клиентов. Важно, чтобы ваши часто используемые языки и инструменты хорошо работали с клиентскими библиотеками. Для этого следует убедиться в следующем:
- существуют ли клиентские библиотеки и фреймворки на требуемых языках;
 - сможете ли вы создавать библиотеки, если их не существует;
 - используете ли вы популярные фреймворки или же пытаетесь развернуть свои собственные?
- ◆ *наличие поддержки сообщества* — этот момент является чрезвычайно важным при выборе брокера событий. Хороший пример брокера событий с большой поддержкой сообщества представляет собой свободно доступный проект с открытым исходным кодом Apache Kafka (<https://kafka.apache.org>). Выбирая брокера, следует убедиться в следующем:
- есть ли у него поддержка онлайнового сообщества;
 - является ли эта технология зрелой и готовой к производству;
 - используется ли эта технология во многих организациях;
 - привлекательна ли эта технология для потенциальных сотрудников;
 - будут ли ваши сотрудники готовы строить свою работу на основе этой технологии?
- ◆ *долгосрочное и многоуровневое хранение* — в зависимости от размера потоков событий и продолжительности хранения бывает предпочтительнее хранить старые сегменты данных в более медленном, но более дешевом хранилище. Многоуровневое хранилище (tiered storage) обеспечивает несколько (обычно три: Tier 1, Tier 2 и Tier 3) уровня доступа к хранилищу, а выделенный диск, локальный для брокера событий или его узлов, которые обслуживают данные, обеспечивает самый высокий уровень производительности. Последующие уровни могут включать такие возможности, как выделенные крупномасштабные сервисы слоя хранения (например, Amazon S3, Google Cloud Storage и Azure Storage). При этом следует убедиться в следующем:
- поддерживается ли многоуровневое хранилище автоматически;
 - можно ли передавать данные на более низкие или более высокие уровни в зависимости от использования;
 - можно ли легко извлекать данные с любого уровня, на котором они хранятся?

Брокеры событий или брокеры сообщений?

Я обнаружил, что люди путаются в том, что представляет собой брокер сообщений и что представляет собой брокер событий. Брокеры событий могут использоваться вместо брокера сообщений, но брокер сообщений не может выполнять все функции брокера событий. Давайте сравним их подробнее.

Брокеры сообщений имеют долгую историю и использовались многими организациями в крупномасштабных архитектурах промежуточного программного обеспечения, ориентированных на сообщения. Брокеры сообщений позволяют системам обмениваться информацией по сети через очереди сообщений и модели публикации/подписки. Производители данных пишут сообщения в очередь, в то время как потребитель принимает эти сообщения и обрабатывает их соответствующим образом. Затем сообщения признаются потребленными и удаляются либо сразу, либо вскоре после этого.

Брокеры событий, в отличие от брокеров сообщений, предназначены для решения задач иного типа и предоставляют упорядоченный журнал фактов. Брокеры событий отвечают двум очень специфическим потребностям, которые не удовлетворяются брокерами сообщений. Прежде всего, брокер сообщений предоставляет только очереди сообщений, где чтение сообщения выполняется по очереди. Приложения, которые совместно потребляют из очереди, будут получать только подмножество записей. Это делает невозможным корректную передачу состояния через события, т. к. каждый потребитель не может получить полную копию всех событий. В отличие от брокера сообщений, брокер событий ведет единый реестр записей и управляет индивидуальным доступом через индексы, так что каждый независимый потребитель может получить доступ ко всем необходимым событиям. Кроме того, брокер сообщений удаляет события после подтверждения, в то время как брокер событий сохраняет их до тех пор, пока это необходимо организации. Удаление события после потребления не позволяет брокеру сообщений обеспечить бесконечно хранящийся, глобально доступный, воспроизводимый и единый источник истины для всех приложений.



Брокеры событий обеспечивают возможность ведения неизменяемого и только расширяемого журнала фактов, который сохраняет состояние упорядоченности событий. Потребитель может зайти и получить их из любой точки журнала в любое время. Этот шаблон необходим для событийно-управляемых микросервисов, но он недоступен брокерам сообщений.

Имейте в виду, что очереди, используемые в брокерах сообщений, по-прежнему играют определенную роль в событийно-управляемых микросервисах. Очереди предоставляют полезные шаблоны доступа, которые, однако, могут быть неудобны для реализации со строго подразделенными потоками событий. Шаблоны, вводимые системами с использованием брокера сообщений, безусловно, являются допустимыми шаблонами для архитектур EDM, но они недостаточны для полного объема обязанностей, требуемых такими архитектурами. Остальная часть книги посвящена именно использованию брокеров событий в архитектуре EDM, а не архитектуре брокера сообщений или дизайну приложений на их основе.

Потребление из неизменяемого журнала

Хотя это и не является окончательным стандартом, обычно имеющиеся брокеры событий используют *неизменяемый журнал*, в который записи только добавляются. События добавляются в конец журнала и получают идентификатор (ID) с автоматически инкрементируемым индексом. Потребители данных используют ссылку на индексный ID для доступа к данным. Затем события могут использоваться либо в форме *потока событий*, либо в форме *очереди* — в зависимости от бизнес-требований и имеющихся функций брокера событий.

Потребление в качестве потока событий

Каждый потребитель отвечает за обновление своих собственных указателей на ранее прочитанные индексы в потоке событий. Этот индекс, именуемый *смещением*, является мерой текущего события от начала потока событий. Смещения позволяют нескольким потребителям потреблять и отслеживать свое продвижение независимо друг от друга, как показано на рис. 2.6.

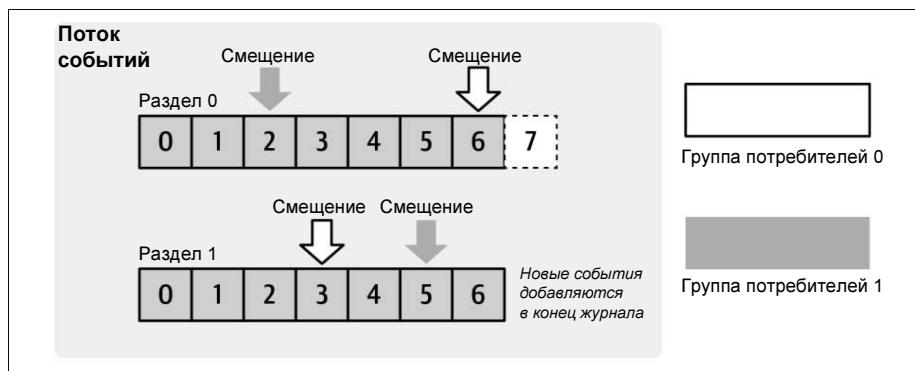


Рис. 2.6. Группы потребителей и их пораздельные смещения

Использованное здесь понятие *группа потребителей* позволяет рассматривать многочисленных потребителей как одну и ту же логическую сущность и может использоваться для горизонтального масштабирования потребления сообщений. Новый потребитель присоединяется к группе потребителей, вызывая перераспределение назначенных им разделов потока событий. Новый потребитель потребляет события только из назначенных ему разделов, точно так же, как старые экземпляры потребителей, ранее находившиеся в группе, продолжают потреблять только из оставшихся назначенных им разделов. Благодаря этому потребление событий может быть сбалансировано в одной и той же группе потребителей, при этом обеспечивая, чтобы все события для того или иного раздела потреблялись исключительно одним экземпляром потребителя. Число активных экземпляров потребителя в группе ограничено числом разделов в потоке событий.

Потребление в форме очереди

При потреблении на основе очередей каждое событие потребляется одним и только одним экземпляром микросервиса. После потребления это событие помечается брокером событий как «потребленное» и больше не предоставляется никакому другому потребителю. При потреблении в форме очереди количество разделов не имеет никакого значения, т. к. для потребления может быть использовано любое число экземпляров потребителя.



При обработке из очереди порядок событий не поддерживается. Параллельные потребители потребляют и обрабатывают события не по порядку, в то время как одиничный потребитель из-за невозможности обработать событие может вернуть его в очередь для обработки позже и перейти к следующему событию.

Очереди поддерживаются не всеми брокерами событий. Например, Apache Pulsar (<https://pulsar.apache.org>) в настоящее время поддерживает очереди, а Apache Kafka — нет. На рис. 2.7 показана реализация очереди с использованием подтверждения индивидуального смещения.

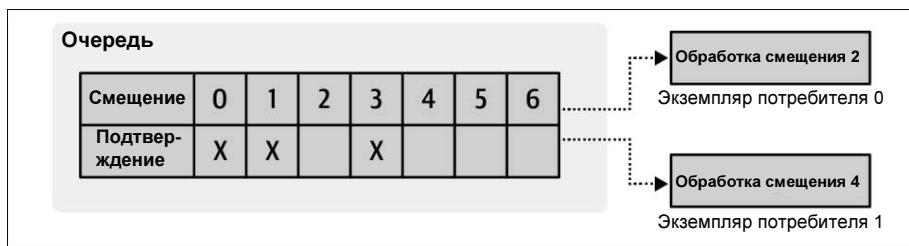


Рис. 2.7. Потребление из неизменяемого журнала в форме очереди

Обеспечение единственного источника истины

Постоянный и неизменяемый журнал обеспечивает механизм хранения единственного источника истины, при этом брокер событий становится единственным местом, в котором сервисы потребляют и производят данные. Благодаря этому каждый потребитель гарантированно получает идентичную копию данных.

Принятие брокера событий в качестве единственного источника истины требует изменения культуры в организации. Если раньше команда могла просто писать прямые SQL-запросы для доступа к данным в базе данных монолита, то теперь она должна также публиковать данные монолита у брокера событий. Разработчики, управляющие монолитом, должны обеспечивать, чтобы полученные данные были совершенно точными, потому что любая несогласованность между потоками событий и базой данных монолита будет считаться недоработкой команды разработчиков. Потребители данных больше не находятся в цепочке непосредственно на монолите, а вместо этого потребляют непосредственно из потоков событий.

Внедрение событийно-управляемых микросервисов позволяет создавать сервисы, которые используют для хранения данных и доступа к ним исключительно брокер

событий. И хотя локальные копии событий, безусловно, могут использоваться бизнес-логикой микросервисов, брокер событий остается единственным источником истины для всех данных.

Масштабное управление микросервисами

Управление микросервисами может все более усложняться по мере роста числа сервисов. Каждый микросервис требует конкретных вычислительных ресурсов, хранилищ данных, конфигураций, переменных среды и целого ряда других специфичных для микросервисов свойств. Каждый микросервис также должен быть пригодным для управления и развертывания командой, которой он принадлежит. Контейнеризация и виртуализация, а также ассоциированные с ними системы управления являются распространенными способами достижения этой цели. Оба варианта позволяют отдельным командам настраивать требования своих микросервисов посредством единого модуля развертывания.

Размещение микросервисов в контейнерах

Контейнеры, недавно ставшие популярными с появлением системы автоматизации развертывания с поддержкой контейнеризации Docker (<https://www.docker.com>), изолируют приложения друг от друга. Контейнеры используют существующую операционную систему сервера на основе модели совместного ядра. Благодаря этому обеспечивается базовое разделение между контейнерами, в то время как сам контейнер изолирует переменные среды, библиотеки и другие зависимости. Контейнеры обеспечивают большинство преимуществ виртуальной машины (рассмотренной далее) за небольшую стоимость, с быстрым временем запуска и низкими ресурсными накладными расходами.

Подход на основе совместно используемой контейнерами операционной системы влечет за собой некоторые сложности и опасности. Контейнеризированные приложения должны быть способны работать в виртуальной машине ОС сервера. Если для приложения требуется специализированная ОС, придется настроить независимый сервер. Одной из главных проблем при всем этом является безопасность, поскольку контейнеры получают общий доступ к ОС хост-машины. Уязвимость в ядре также может поставить под угрозу все контейнеры на этом хосте. Если на сервере используются контейнеры только с сервисами одного предприятия, это вряд ли создаст проблему, но современные модели совместной аренды в облачных вычислениях начинают требовать уделяния размещению контейнеров больше внимания.

Размещение микросервисов на виртуальных машинах

Виртуальные машины (ВМ) устраниют некоторые недостатки контейнеров, хотя их внедрение происходит медленнее. Традиционные виртуальные машины обеспечивают полную изоляцию с помощью автономной ОС и виртуализированного аппаратного обеспечения, специфичного для каждого экземпляра. Хотя эта альтернатива обеспечивает более высокую безопасность, чем контейнеры, она намного ее до-

роже. Каждая виртуальная машина имеет более высокие накладные расходы по сравнению с контейнерами, более медленное время запуска и большие ресурсные потребности системы.



В настоящее время предпринимаются усилия по удешевлению и повышению эффективности виртуальных машин. Говоря лишь о некоторых разработанных в этом направлении системах, можно упомянуть, например, Google gVisor (<https://oreil.ly/0GwA4>), Amazon Firecracker (<https://oreil.ly/tqVmO>) и Kata Containers (<https://katacontainers.io>). По мере совершенствования этих технологий виртуальные машины станут для ваших потребностей в микросервисах гораздо более конкурентоспособной альтернативой контейнерам. Если ваши потребности будут обусловлены в первую очередь требованиями безопасности, то стоит внимательно следить за этой областью.

Управление контейнерами и виртуальными машинами

Управление контейнерами и виртуальными машинами осуществляется с помощью различных специализированных программ, именуемых *системами управления контейнерами* (Container Management System, CMS). Они управляют развертыванием контейнеров, распределением ресурсов и интеграцией с базовыми вычислительными ресурсами. Популярные и часто используемые CMS включают Kubernetes (<https://oreil.ly/GHaef>), Docker Engine (<https://www.docker.com>), Mesos Marathon (<https://oreil.ly/a5y5V>), Amazon ECS (<https://aws.amazon.com/ecs>) и Nomad (<https://www.nomadproject.io>).

Микросервисы должны иметь возможность масштабирования вверх и вниз в зависимости от меняющихся рабочих нагрузок, *соглашений об уровне обслуживания* (Service Level Agreement, SLA) и требований к производительности. Необходимо поддерживать вертикальное масштабирование, при котором нагрузки на вычислительные ресурсы, такие как процессор, память и диск, увеличиваются или уменьшаются на каждом экземпляре микросервиса. Горизонтальное масштабирование также должно поддерживаться с добавлением или удалением новых экземпляров.

Каждый микросервис должен разворачиваться как единое целое. Для многих микросервисов один-единственный исполняемый компонент — это все, что необходимо для реализации бизнес-требований, и он может разворачиваться в одном контейнере. Другие микросервисы могут быть сложнее, с многочисленными контейнерами и внешними хранилищами данных, требующими координации. Именно здесь в игру вступает что-то вроде концепции модулей (pod) системы оркестровки контейнеров Kubernetes, позволяющей развертывать и возвращать несколько компонентов как единое действие. Kubernetes также позволяет производить однократные операции — например, миграция баз данных может осуществляться во время выполнения одного развертываемого объекта.

Управление виртуальными машинами также поддерживается рядом реализаций, но в настоящее время оно более ограничено, чем управление контейнерами. Системы Kubernetes и Docker Engine поддерживают контейнеры Google gVisor и Kata, а платформа Amazon поддерживает AWS Firecracker. Границы между контейнерами и виртуальными машинами продолжат размываться по мере развития технологий.

Убедитесь, что выбранная вами CMS будет обрабатывать необходимые вам контейнеры и виртуальные машины.



Существуют ресурсы, предоставляющие обширную и полезную информацию по Kubernetes, Docker, Mesos, Amazon ECS и Nomad. Эта информация выходит далеко за рамки того, что представлено в этой книге. Рекомендую вам изучить эти материалы для получения дополнительной информации.

Уплата налога на микросервисы

Налог на микросервисы — это сумма затрат, включая финансовые, трудовые ресурсы и возможности, связанные с внедрением инструментов и компонентов архитектуры микросервисов. Сюда входят затраты на управление, развертывание и эксплуатацию брокера событий, CMS, конвейеров развертывания, решений мониторинга и сервисов журналирования. Эти расходы неизбежны и оплачиваются либо централизованно организацией, либо независимо каждой командой, внедряющей микросервисы. Первый вариант приводит к масштабируемой, упрощенной и унифицированной структуре разработки микросервисов, в то время как второй — к чрезмерным накладным расходам, дублированию решений, фрагментированному инструментарию и неустойчивому росту.

Уплата налога на микросервисы — дело не тривиальное, и это одно из самых больших препятствий для начала работы с EDM. Малые организации, вероятно, скорее всего будут придерживаться архитектуры, которая лучше соответствует их бизнес-требованиям, — такой как модульная монолитная система. Более крупные организации должны учитывать общие затраты как на внедрение, так и на обслуживание платформы на основе микросервисов и определять, сможет ли долгосрочная дорожная карта их предприятия соответствовать прогнозируемым рабочим усилиям.

К счастью, в последние годы как открытые, так и размещенные у провайдера сервисы стали гораздо доступнее и проще в использовании. Налог на микросервисы неуклонно снижается благодаря новым интеграциям между CMS, брокерами событий и другими необходимыми инструментами. Убедитесь, что ваша организация готова выделить необходимые ресурсы для оплаты этих первоначальных расходов.

Резюме

В этой главе были рассмотрены базовые требования к событийно-управляемым микросервисам. Брокер событий — это главный механизм обмена данными, обеспечивающий поставку потоков событий в реальном времени в крупном масштабе для потребления другими сервисами. Системы контейнеризации и управления контейнерами позволяют осуществлять масштабное управление микросервисами. На конец, в этой главе также были рассмотрены важные принципы, лежащие в основе событий и событийно-управляемой логики, и дан первый взгляд на управление состоянием в распределенном событийно-управляемом мире.

Обмен информацией и контракты на передачу данных

Фундаментальная проблема обмена информацией состоит в том, чтобы в одной точке точно или приблизительно воспроизвести сообщение, выбранное в другой точке.

— Клод Шеннон

Родоначальник теории информации Клод Шеннон выявил самое большое препятствие для обмена информацией: обеспечить, чтобы потребитель сообщения смог точно воспроизвести сообщение производителя так, чтобы и содержание, и смысл сообщения были правильно переданы. Производитель и потребитель должны иметь общее понимание сообщения — в противном случае оно может быть неверно истолковано и обмен информацией окажется неполным. В событийно-управляемой экосистеме событие является сообщением и основной единицей обмена информацией. Событие должно как можно точнее описывать, *что* произошло и *почему*. Оно является констатацией факта и в сочетании со всеми другими событиями в системе дает полную историю того, что произошло.

Событийно-управляемые контракты на передачу данных

Формат передаваемых данных и логика, в соответствии с которой они создаются, формируют *контракт на передачу данных*. Такой контракт соблюдается как производителем, так и потребителем событийных данных. Это придает событию смысл и форму вне контекста, в котором оно производится, и расширяет удобство использования данных для потребительских приложений.

Существуют два компонента четко определенного контракта на передачу данных. Прежде всего это *определение данных* или то, что будет произведено (т. е. поля, типы и различные структуры данных). Второй компонент — это *триггерная логика* или причина его создания (т. е. конкретная бизнес-логика, которая вызвала создание события). По мере развития бизнес-требований можно вносить изменения как в определение данных, так и в триггерную логику.

При изменении определения данных необходимо соблюдать осторожность, чтобы не удалить и не изменить поля, используемые нижестоящими потребителями. Точно так же вы должны быть осторожными при изменении триггерной логики. Впрочем, изменения определения данных происходят гораздо чаще, чем изменения триггерного механизма, поскольку его изменение часто нарушает смысл исходного определения события.

Использование явных схем в качестве контрактов

Самый лучший способ обеспечить соблюдение контрактов на передачу данных и их согласованность — это определить схему для каждого события. Производитель определяет явную схему, детализирующую определение данных, и триггерную логику, причем все события одного типа придерживаются этого формата. При этом производитель обеспечивает механизм доведения своего формата события до всех потенциальных потребителей. Потребители, в свою очередь, могут уверенно строить свою бизнес-логику с использованием микросервисов на основе данных, описанных схемой.



Любая реализация обмена событиями между производителем и потребителем, в которой отсутствует явная предопределенная схема, неизбежно будет опираться на неявную схему. Неявные контракты на передачу данных хрупки и подвержены неконтролируемым изменениям, что может привести к большим неоправданным трудностям для потребителей.

Потребитель должен иметь возможность извлекать данные, необходимые для его бизнес-процессов, и он не может это сделать, не имея набора соглашений относительно того, какие данные должны быть доступны. Потребители часто должны полагаться на «племенные знания» и внутрикомандный обмен информацией для урегулирования вопросов с данными, а этот процесс не масштабируется по мере увеличения числа потоков событий и числа команд. Существует также существенный риск, когда требуется, чтобы каждый потребитель независимо интерпретировал данные, поскольку он может интерпретировать их иначе, чем его коллеги, что приводит к несогласованным взглядам на единый источник истины.



Может возникнуть соблазн построить общую библиотеку, интерпретирующую любое событие для всех сервисов, но это создает проблемы с многочисленными языковыми форматами, развитием событий и независимыми циклами выпуска. Дублирование усилий в разных сервисах для обеспечения единообразного представления неявно определенных данных имеет нетривиальный характер, и его лучше всего полностью избегать.

Производители также находятся в невыгодном положении с неявными схемами. Даже имея самые лучшие намерения, производитель может не заметить (или, возможно, его модульные тесты этого не показывают), что он изменил свое определение событийных данных. Без явной проверки формата событий их сервисов эта ситуация может остаться незамеченной до тех пор, пока не приведет к отказу нижестоящих потребителей. Так что именно явные схемы обеспечивают безопасность и стабильность как потребителям, так и производителям данных.

Комментарии к определению схемы

Поддержка интегрированных комментариев и произвольных метаданных в определении схемы необходима для передачи смысла события. Знания, связанные с производством и потреблением событий, должны быть максимально приближены к определению событий. Комментарии к схемам помогают устраниćть двусмысленность в отношении смысла данных и уменьшить вероятность их неправильного

толкования потребителями. Есть две главные области, где комментарии особенно ценные:

- ◆ конкретизация триггерной логики события. Обычно это делается в заголовке блока в верхней части определения схемы и должно четко указывать, почему событие было сгенерировано;
- ◆ предоставление контекста и ясности относительно конкретного поля в рамках структурированной схемы. Например, комментарии поля `datetime` могут указывать, какой формат времени используется: UTC, ISO или UNIX.

Полнофункциональное развитие схемы

Формат схемы должен поддерживать полный набор правил развития схемы. Развитие схемы позволяет производителям обновлять формат вывода своих сервисов, давая возможность потребителям продолжать непрерывное потребление событий. Изменения на предприятии могут потребовать добавления новых полей, привести к отказу от использования старых полей или расширению области действия поля. Правила развития схемы обеспечивают такой ход событий, при котором изменения могли бы происходить безопасно, а производители и потребители были способны обновляться независимо друг от друга.

Обновления сервисов становятся непомерно дорогими без поддержки развития схемы. Производители и потребители вынуждены тесно координировать свои действия, а старые, ранее совместимые данные, вполне могут больше не быть совместимыми с существующими системами. Неразумно ожидать, что потребители будут обновлять свои сервисы всякий раз, когда производитель изменяет схему данных. Фактически основная ценность микросервисов заключается в том, что они не должны зависеть от циклов выпуска других сервисов, за исключением исключительных случаев.

Явный набор правил развития схем позволяет как потребителям, так и производителям обновлять свои приложения в удобное для них время. Эти правила называются *типами совместимости*:

◆ Прямая совместимость.

Позволяет читать данные, произведенные с помощью более новой схемы, так же, как если бы они были произведены с помощью старой схемы. Это положение является особенно полезным в свете развития событийно-управляемой архитектуры, поскольку наиболее распространенный шаблон изменения системы начинается с того, что производитель обновляет свое определение данных и создает данные с более новой схемой. Потребителю останется только обновить свою копию схемы и кода, если ему потребуется доступ к новым полям.

◆ Обратная совместимость,

Позволяет читать данные, произведенные с помощью старой схемы, так же, как если бы они были произведены с помощью более новой схемы. Это дает потребителю данных возможность использовать новую схему для чтения старых данных. Вот несколько сценариев, где это особенно полезно:

- потребитель ожидает, что новая функциональность будет предоставлена вышестоящей командой. Если новая схема уже определена, то потребитель может выпустить свое собственное обновление до выпуска производителя;
- данные в кодировке схемы отправляются продуктом, развернутым на оборудовании клиента, — например приложением сотового телефона, которое сообщает о пользовательских метриках. Обновления могут быть внесены в формат схемы для новых выпусков производителей, сохраняя при этом совместимость с предыдущими выпусками;
- потребительскому приложению может потребоваться переработать данные в потоке событий, который был произведен с более старой версией схемы. Развитие схемы обеспечивает потребителю возможность перевести ее в знакомую версию. Если обратная совместимость не соблюдается, то потребитель сможет читать сообщения только в самом новом формате.

◆ *Полная совместимость.*

Сочетание прямой совместимости и обратной совместимости представляет собой полную гарантию, и вы должны использовать его всякий раз, когда это возможно. Вы всегда можете ослабить требования к совместимости на более позднем этапе, однако ужесточить их часто бывает гораздо сложнее.

Поддержка генератора кода

Генератор кода используется для преобразования схемы события в определение класса или эквивалентную структуру для используемого языка программирования. Это определение класса задействуется разработчиком для создания и заполнения новых событийных объектов. Компилятор или сериализатор (в зависимости от реализации) требует от производителя соблюдать типы данных и заполнять все не допускающие значений `null` поля, указанные в исходной схеме. Созданные производителем объекты затем конвертируются в их сериализованный формат и отправляются брокеру событий, как показано на рис. 3.1.



Рис. 3.1. Рабочий процесс производства событий производителя с использованием генератора кода

Потребитель событийных данных поддерживает свою собственную версию схемы, которая часто совпадает с версией производителя, но может быть более старой или более новой — в зависимости от развития схемы. Если соблюдается полная совместимость, то для генерирования определений схемы сервис может использовать любую ее версию. Пользователь читает событие и десериализует его, используя версию схемы, с помощью которой оно было закодировано. Формат события хранится либо вместе с сообщением, что может быть чрезмерно дорогостоящим при

крупном масштабе, либо в реестре схем и доступен по требованию (см. разд. «Регистр схем» главы 14). После десериализации в исходный формат событие может быть конвертировано в версию схемы, поддерживаемую потребителем. На этом этапе в игру вступают правила развития, причем к отсутствующим полям применяются значения, заданные по умолчанию, а неиспользуемые поля полностью удаляются. Наконец, данные конвертируются в объект на основе созданного схемой класса. В этот момент может начать свою работу бизнес-логика потребителя. Этот процесс показан на рис. 3.2.



Рис. 3.2. Процесс потребления и преобразования событий потребителем с использованием генератора кода (обратите внимание, что потребитель конвертирует события из схемы V2, созданной производителем, в формат схемы V1, используемый потребителем)

Самым большим преимуществом поддержки генератора кода является возможность написать приложение в соответствии с определением класса на языке по вашему выбору. Если вы используете компилируемый язык, генератор кода обеспечивает проверку компилятора, чтобы убедиться, что вы не ошиблись с типами событий или не пропустили заполнение любого заданного ненулевого поля данных. Ваш код не станет компилироваться, если он не соответствует схеме, и поэтому ваше приложение не будет отправлено без соблюдения определения схемных данных. Как компилируемые, так и некомпилируемые языки выигрывают от наличия реализации класса для кодирования. Современная IDE-среда оповестит вас, если вы попытаетесь передать неверные типы в конструктор или сеттер, тогда как вы не получите никакого оповещения, если вместо этого используете общий формат, такой как объект-словарь в формате ключ/значение. Снижение риска неправильного обращения с данными обеспечивает гораздо более стабильное качество данных во всей экосистеме.

Разрушительные изменения схемы

Бывают моменты, когда определение схемы должно измениться таким образом, что это приведет к разрушительному эволюционному изменению. Так может произойти по ряду причин, включающих развивающиеся бизнес-требования, которые меняют модель исходного домена, неправильное определение предметной области его действия и человеческую ошибку при определении схемы. И если производящий сервис может быть достаточно легко изменен в соответствии с новой схемой, последствия для нижестоящих потребителей могут оказаться фатальными и должны быть приняты во внимание.



Самое важное при работе с несовместимыми изменениями схемы — это ранний и четкий обмен информацией с потребителями. Убедитесь, что любые миграционные планы поняты и одобрены всеми участниками и что никто не будет застигнут врасплох.

Требование интенсивной координации между производителями и потребителями может показаться неуместным, однако пересмотр условий контракта на передачу данных и изменение модели предметной области требуют поддержки со стороны всех. Помимо пересмотра схемы вам необходимо предпринять некоторые дополнительные шаги, чтобы приспособить новую схему и новые потоки событий, которые создаются на ее основе. На события, срок действия которых истекает через заданный период времени, влияние разрушительных изменения схемы не столь критично, однако оно может быть весьма значительным для событий, которые существуют бесконечно.

Учет критических изменений схемы для событий

Критические разрушительные изменения в схеме сущностей встречаются весьма редко, поскольку это обстоятельство обычно требует переопределения исходной модели предметной области, так что текущую модель нельзя просто расширить. Новые сущности будут созданы по новой схеме, в то время как предыдущие сущности были созданы по старой. Это расхождение в определении данных оставляет вам два варианта:

- ◆ оставить в ходу как старую, так и новую схему;
- ◆ воссоздать все сущности в схеме нового формата (путем миграции или их воссоздания из исходного кода).

Первый вариант является наиболее простым для производителя, но он просто перекладывает разрешение различных определений сущностей на потребителя. Это противоречит цели *уменьшения* его потребности в индивидуальной интерпретации данных и увеличивает риск неправильной их интерпретации, несогласованной обработки между сервисами и значительно более высокой сложности технического сопровождения систем.



Реальность такова, что потребитель по части урегулирования расходящихся определений схем никогда не будет в лучшей ситуации, чем производитель. Перекладывание этой ответственности на потребителя — плохая практика.

Второй вариант сложнее для производителя, но обеспечивает согласованное переопределение бизнес-сущностей — как старых, так и новых. На практике производитель должен заново обработать исходные данные, которые привели к созданию старых сущностей, и применить новую бизнес-логику для воссоздания этих сущностей в новом формате. Такой подход вынуждает организацию решать, что означают эти сущности и как они должны пониматься и использоваться как производителем, так и потребителем.



Оставьте старые сущности по старой схеме в их исходном потоке событий, потому что они могут понадобиться вам для повторной обработки, проверки и судебных расследований. Создавайте новые и обновленные сущности, используя новую схему в новом потоке.

Когда вы вносите критические изменения, иметь дело с несуществующими событиями, как правило, проще. Самый простой вариант — создать новый поток собы-

тий и начать создавать новые события для этого потока. Потребители старого потока должны быть об этом уведомлены, чтобы они могли зарегистрироваться как потребители нового потока событий. Каждый сервис-потребитель также должен учитывать расхождение в бизнес-логике между двумя определениями событий.



Не смешивайте разные типы событий в потоке событий, в особенности типы событий, которые эволюционно несовместимы. Накладные расходы на поток событий дешевы, и логическое разделение событий важно для обеспечения того, чтобы потребители имели полную информацию и явные определения при работе с событиями, которые им необходимо обрабатывать.

Учитывая, что в старом потоке событий больше не создаются новые события, потребители каждого сервиса-потребителя в конечном итоге доберутся до последней записи. По прошествии времени период хранения потока приведет к его полной очистке, после чего все потребители смогут отменить регистрацию, а поток событий можно будет удалить.

Выбор формата события

Хотя существует много вариантов форматирования и сериализации событийных данных, контракты на передачу данных лучше всего выполнять в строго определенных форматах, таких как Avro, Thrift или Protobuf. Некоторые из наиболее популярных фреймворков брокеров событий поддерживают сериализацию и десериализацию событий, которые кодируются в этих форматах. Например, Apache Kafka (<https://oreil.ly/oVULy>) и Apache Pulsar (<https://oreil.ly/UjE7L>) поддерживают формат схем JSON, Protobuf и Avro. Механизмом поддержки этих технологий является реестр схем, более подробно описанный в разд. «Реестр схем» главы 14. Хотя детальное рассмотрение и сравнение указанных вариантов сериализации выходит за рамки этой книги, существует ряд онлайновых ресурсов, которые помогут вам выбрать один из предложенных конкретных вариантов.

У вас может возникнуть соблазн выбрать более гибкий вариант в форме текстовых событий с использованием простых пар ключ/значение, которые хотя и предлагают некоторую структуру, но не предоставляют явных схем или структур развития схемы. Однако будьте осторожны с этим подходом, поскольку он может поставить под угрозу способность микросервисов оставаться изолированными друг от друга из-за сильного контракта на передачу данных, требующего гораздо большего обмена информацией между командами.



Неструктурированные события в форме обычного текста обычно становятся временем как для производителя, так и для потребителя, в особенности по мере изменения вариантов использования и данных с течением времени. Как уже упоминалось, вместо этого рекомендуется выбрать строго определенный, явный схемный формат, поддерживающий управляемое развитие схемы — например Apache Avro или Protobuf. При этом я не рекомендую формат JSON, т. к. он не обеспечивает полностью совместимого развития схемы.

Дизайн событий

При создании определений событий следует придерживаться определенных рекомендаций, а также избегать ряда неверных подходов. Имейте в виду, что по мере увеличения числа архитектур событийно-управляемых микросервисов увеличивается и число определений событий. Хорошо продуманные события сведут к минимуму повторяющиеся болевые точки как для потребителей, так и для производителей. С учетом сказанного ни одно из приведенных далее правил не является безоговорочным требованием. Вы можете применять их все вместе или выборочно по своему усмотрению, хотя я рекомендую вам, прежде чем продолжить, очень тщательно проанализировать весь спектр возможных последствий и компромиссов для вашего пространства решений.

Говорите правду, всю правду и ничего, кроме правды

Хорошее определение события — это не просто сообщение, указывающее на то, что *что-то* произошло, а, скорее, полное описание *всего*, что произошло во время этого события. С точки зрения предприятия это результирующие данные, которые создаются при приеме входных данных и применении бизнес-логики. Такое выходное событие должно рассматриваться как единственный источник истины и должно быть представлено как непреложный факт для потребления нижестоящими потребителями. Это полный и абсолютный авторитет в отношении того, что фактически произошло, и потребителям нет нужды обращаться к какому-либо другому источнику данных, чтобы узнать, что такое событие имело место.

Используйте одно-единственное определение события в расчете на поток

Поток событий должен содержать события, представляющие одно-единственное логическое событие. Не рекомендуется смешивать разные типы событий в потоке событий, поскольку это может привести к путанице в определениях того, каким является событие и что представляет собой поток. Тогда будет трудно выполнить валидацию продуцируемых схем, т. к. в таком сценарии новые схемы могут добавляться динамически. Хотя существуют особые обстоятельства, при которых вы можете игнорировать этот принцип, подавляющее большинство потоков событий, производимых и потребляемых в рамках вашего архитектурного рабочего процесса, должны иметь строгое и единственное определение.

Используйте самые точные типы данных

Используйте самые точные типы для ваших событийных данных. Это даст вам возможность опираться на генераторы кода, проверку типов данных используемыми языками (если она поддерживается) и сериализационные модульные тесты для проверки границ ваших данных. Все это звучит просто, но существует много случаев, в которых — когда вы не используете правильные типы — может закрасться двусмысленность. Вот несколько легко предотвратимых реальных примеров:

◆ использование строкового типа `string` для хранения числового значения.

Это требует от потребителя разбора и конвертирования строки в числовое значение и часто приводит к получению чего-то вроде GPS-координат. Такой подход чреват ошибками и может привести к отказам, в особенности когда отправляется значение `null` или пустая строка.

◆ использование целочисленного типа `integer` в качестве булева значения.

В то время как 0 и 1 могут означать соответственно ложь и истину, что будет означать 2? А, например, -1?

◆ использование строкового типа `string` для перечисления.

Это создает для производителей проблемы, поскольку они должны гарантировать, что их опубликованные значения соответствуют принятому списку псевдо-перечислений. А опечатки и неверные значения неизбежно появятся. Потребителю, заинтересованному в этом поле, необходимо знать диапазон возможных значений, и для этого ему потребуется связаться с командой разработчиков, если такой диапазон не указан в комментариях к схеме. В любом случае здесь происходит неявное определение, поскольку производители не защищены от любых изменений диапазона значений в строке. Так что весь этот подход — просто плохая практика.

От перечислений часто отказываются, потому что производители боятся создавать новый токен перечисления, которого нет в схеме потребителя. Однако потребитель обязан рассмотреть токены перечисления, которые он не распознает, и определить, следует ли ему их обрабатывать с использованием значения, заданного по умолчанию, или просто выдавать фатальное исключение и останавливать обработку до тех пор, пока кто-то не сможет решить, что нужно сделать. И Protobuf, и Avro имеют элегантные способы обработки неизвестных токенов перечисления, и их следует использовать, если для вашего формата события выбран любой из них.

Держите события узконаправленными

Одним из распространенных неверных подходов является добавление поля `type` в определение события, где разные значения `type` указывают на специфические подсвойства события. Обычно это делается для данных, которые «похожи, но различаются», и часто является результатом того, что разработчик неправильно идентифицирует события как узконаправленные. Хотя это может показаться способом экономии времени или упрощением шаблона доступа к данным, перегрузка событий параметрами `type` редко бывает хорошей идеей.

С этим подходом связано несколько проблем. Каждое значение параметра `type` обычно имеет принципиально другое бизнес-значение, даже если его техническое представление почти такое же, как и у остальных. Кроме того, с течением времени эти значения могут меняться, и область, которую охватывает событие, может расширяться. Некоторые из этих типов могут потребовать добавления новых параметров для отслеживания специфичной для типа информации, в то время как другие типы требуют отдельных параметров. В конце концов вы можете получить ситуа-

цию, когда будете иметь несколько очень разных событий, но все они будут заполнять одну и ту же схему событий, что затруднит понимание того, что именно это событие в действительности представляет.

Эта сложность затрагивает не только разработчиков, которые должны поддерживать и заполнять события, но и потребителей данных, которые должны иметь согласованное представление о том, какие данные публикуются и почему. Если контракт на передачу данных изменится, то они рассчитывают на то, что смогут изолировать себя от этих изменений. Добавление избыточных типов полей требует от них фильтрации только тех данных, которые им небезразличны. Существует риск того, что потребитель не сможет полностью понять различные смыслы типов, что приведет к неправильному потреблению и логически неправильному коду. По каждому потребителю также необходимо выполнить дополнительную обработку, отбрасывая события, которые не имеют отношения к этому потребителю.



Очень важно отметить, что добавление полей `type` не уменьшает и не устраниет основную сложность, присущую создаваемым данным. На самом деле эта сложность просто смещается от многочисленных отдельных потоков событий с четко различимыми схемами в сторону объединения всех схем в один поток событий. Можно утверждать, что это только увеличивает сложность. Дальнейшее развитие схемы при этом становится сложнее, как и техническое сопровождение кода, который производит события.

Вспомните принципы определения контракта на передачу данных. События должны быть связаны с одним бизнес-действием, а не с общим событием, которое записывает крупные массивы данных. Если вам кажется, что вам нужно общее событие с различными параметрами `type`, то это обычно является признаком того, что ваше пространство решений и ограниченный контекст недостаточно четко определены.

Пример: перегрузка определений событий

Представьте себе простой веб-сайт, где пользователь может прочитать книгу или посмотреть фильм. Когда пользователь впервые подключается к такому веб-сайту, например, открывая книгу или запуская фильм, внутренний сервис публикует событие этого взаимодействия, именуемое `ProductEngagement`, в поток событий. Структура данных события этой поучительной истории может выглядеть примерно так:

```
TypeEnum: Book, Movie
```

```
ActionEnum: Click
```

```
ProductEngagement {
    productId: Long,
    productType: TypeEnum,
    actionPerformed: ActionEnum
}
```

Теперь представьте, что появилось новое бизнес-требование: вам нужно отслеживать тех, кто смотрел трейлер фильма, прежде чем смотреть сам фильм. Для книг нет предварительных просмотров, и хотя булево значение подходит для просмотра

фильмов, оно должно допускать значение `null`, чтобы разрешить взаимодействия с книгами:

```
ProductEngagement {
    productId: Long,
    productType: TypeEnum,
    actionType: ActionEnum,
    //Применимо только к type=Movie
    watchedPreview: {null, Boolean}
}
```

В настоящий момент параметр `watchedPreview` не имеет ничего общего с `Book`, но он все равно добавлен в определение события, поскольку мы именно таким образом фиксируем взаимодействие с продуктом. Если вы хотите помочь своим потребителям, то можете добавить в схему комментарий, сообщающий им, что это событие связано только с `type=Movie`.

Теперь появляется еще одно новое требование: вы должны отслеживать пользователей, которые помещают закладку в свою книгу, и регистрировать, на какой странице она находится. Опять же, поскольку существует только одна определенная структура событий для взаимодействия с продуктом, ваше решение ограничено добавлением новой сущности действия (`Bookmark`) и добавлением допускающего значение `null` поля `PageId`:

```
TypeEnum: Book, Movie
ActionEnum: Click, Bookmark
```

```
ProductEngagement {
    productId: Long,
    productType: TypeEnum,
    actionType: ActionEnum,
    //Применимо только к productType=Movie
    watchedPreview: {null, Boolean},
    //Применимо только к productType=Book, actionType=Bookmark
    pageId: {null, Int}
}
```

Как вы можете здесь видеть, всего несколько изменений в бизнес-требованиях могут значительно усложнить схему, которая пытается служить нескольким бизнес-целям. Это добавляет сложности и производителю, и потребителю данных, поскольку они оба должны проверять правильность логики данных. Собираемые и предоставляемые данные всегда будут сложными, но, следуя принципу единственной ответственности, вы можете разложить схему на что-то более управляемое. Давайте посмотрим, как будет выглядеть этот пример, если вы разделите каждую схему в соответствии с принципом единственной ответственности:

```
MovieClick {
    movieId: Long,
    watchedPreview: Boolean
}
```

```
BookClick {
    bookId: Long
}

BookBookmark {
    bookId: Long,
    pageId: Int
}
```

Перечисления `ProductType` и `ActionType` исчезли, и схемы соответственно откорректированы. Теперь вместо одного существуют три определения событий, и хотя количество определений схемы увеличилось, внутренняя сложность каждой схемы значительно уменьшилась. Следование рекомендации о единственном определении события для каждого потока ведет к созданию нового потока для каждого типа события. Определения событий не станут дрейфовать во времени, триггерная логика не изменится, и потребители могут быть уверены в стабильности определения узконаправленного одноцелевого события.

Вывод из этого примера состоит не в том, что первоначальный создатель определения события допустил ошибку. На самом деле его интересовали *любые* взаимодействия с продуктом, а не какие-либо *конкретные*, и поэтому первоначальное определение было вполне разумным. Но как только бизнес-требования изменились за счет включения в их состав отслеживания событий, связанных с фильмом, владелец сервиса должен был заново оценить, служит ли определение события по-прежнему одной цели или же оно теперь охватывает несколько целей. Из-за того, что бизнес потребовал подробных сведений о событии более низкого уровня, владелец сервиса должен был понять, что хотя одно событие может служить нескольким целям, его реализация быстро становится сложной и громоздкой.



Избегайте добавления полей `type` в события, которые перегружают смысл события. Это может вызвать значительные трудности в развитии и техническом сопровождении формата события.

Найдите время, чтобы подумать, как ваши схемы могут развиваться. Определите основную бизнес-цель производимых данных, объем, домен и то, создаете ли вы их как одноцелевые. Убедитесь, что схемы точно отражают бизнес-задачи, особенно для систем, которые охватывают широкий спектр ответственности бизнес-функций. Может случиться так, что размах бизнеса и его имеющаяся техническая реализация несовместимы. Наконец, развивающиеся бизнес-требования могут потребовать от вас пересмотреть определения событий и потенциально изменить их, выходя за рамки простых инкрементных определений одной схемы. Возможно — если произойдут существенные изменения в бизнесе — потребуется разделить события и полностью переопределить их.

Минимизируйте размер событий

События работают хорошо, когда они небольшие, четко определены и легко обрабатываются. Однако крупные события могут происходить и происходят. Как пра-

вило, эти более крупные события представляют собой много контекстной информации. Возможно, они содержат множество точек данных, связанных с указанным событием, и просто очень большие показатели того, что произошло.

Когда вы смотрите на дизайн, который производит очень крупное событие, следует учесть несколько соображений. Убедитесь, что данные непосредственно связаны с событием и имеют к нему отношение. Дополнительные данные могут быть добавлены в событие «на всякий случай», но при этом не приносить никакой реальной пользы нижестоящим потребителям. Если вы обнаружите, что все событийные данные действительно напрямую связаны, то сделайте шаг назад и посмотрите на свое пространство решений. Требуется ли вашему микросервису доступ к данным? Возможно, вы захотите оценить ограниченный контекст, чтобы увидеть, насколько разумный объем работы выполняет сервис. Может быть, имеет смысл уменьшить объем работы этого сервиса, выделив его излишнюю функциональность в отдельный сервис.

Однако сценария, связанного с крупными событиями, не всегда можно избежать — некоторые обработчики событий производят очень крупные выходные файлы (типа большого изображения), которые слишком велики, чтобы поместиться в одно сообщение потока событий. В подобных сценариях вы можете использовать указатель на фактические данные, но делайте это с осторожностью. Такой подход добавляет риск в виде множественных источников истинности и изменчивости полезной нагрузки, поскольку неизменяемый реестр не может гарантировать сохранение данных за пределами своей системы.

Привлекайте потенциальных потребителей к дизайну события

Во время работы над дизайном нового события важно привлечь к нему всех предполагаемых потребителей этих данных. Потребители лучше понимают свои собственные потребности и предполагаемые бизнес-функции, чем производители, и способны помочь в уточнении требований. Потребители также получат при этом более глубокое понимание поступающих к ним данных. Совместная встреча или обсуждение помогут утрясти любые вопросы вокруг контракта на передачу данных между двумя системами.

Не используйте события в качестве семафоров и сигналов

Избегайте использования событий в качестве семафоров или сигналов. Такие события просто указывают на то, что что-то произошло, но не являются единственным источником истины для результатов.

Рассмотрим очень простой пример, когда система выводит событие, указывающее на то, что работа над некоторым заданием завершена. Хотя само событие указывает на то, что работа выполнена, фактический результат работы не включается в событие. Это означает, что для правильного использования этого события вы должны

найти место, где на самом деле находится завершенная работа. Однако как только для части данных появляются два источника истины, возникают проблемы согласованности.

Резюме

Асинхронные событийно-управляемые архитектуры в значительной степени зависят от качества событий. Высококачественные события явно определяются с помощью развивающейся схемы, имеют четко определенную триггерную логику и включают полные определения схемы с комментариями и документацией. Неявные схемы, хотя их и легче реализовать и технически сопровождать производителю, перекладывают большую часть работы по их интерпретации на потребителя. Кроме того, они более подвержены неожиданным отказам из-за отсутствия событийных данных и непреднамеренных изменений. Явные схемы являются важным компонентом для широкого внедрения событийно-управляемой архитектуры, в особенности по мере роста организаций, когда распространение «племенных знаний» в масштабах всей организации становится невозможным.

Определения событий должны быть узкими и тесно ориентированными на предметную область события. Событие должно представлять собой конкретное бизнес-событие и содержать соответствующие поля данных для записи того, что произошло. Такие события формируют сведения о бизнес-операциях и могут использоваться другими микросервисами для собственных нужд.

Развитие схем является очень важным аспектом явных схем, поскольку оно позволяет управлять механизмом изменения событийной модели предметной области. Модель предметной области обычно развивается, в особенности по мере появления новых требований и расширения организации. Развитие схемы дает возможность производителям и потребителям изолировать себя от изменений, которые не являются существенными для их деятельности, в то же время позволяя тем сервисам, которые действительно заботятся об изменениях, обновлять себя соответствующим образом.

В некоторых случаях развитие схемы невозможно, и должно произойти разрушительное изменение. Заинтересованные стороны (производители и потребители) должны сообщать друг другу о причинах разрушительных изменений и собираться вместе, чтобы пересмотреть модель предметной области на будущее. Миграция старых событий в зависимости от ситуации может быть или не быть необходимой.

Интеграция событийно-управляемых архитектур с существующими системами

Переход организации на событийно-управляемую архитектуру требует интеграции существующих систем в единую экосистему. Ваша организация может иметь одно или несколько монолитных приложений на основе реляционных баз данных. Между различными их реализациями, скорее всего, будут существовать соединения «точка-точка». Возможно, у вас уже существуют событийно-подобные механизмы для передачи данных между системами — такие как регулярная синхронизация дампов баз данных через промежуточное хранилище файлов. В том случае, если вы создаете архитектуру событийно-управляемых микросервисов с нуля и не имеете унаследованных систем, отлично! Вы можете пропустить эту главу (хотя, тем не менее, вам следует учитывать, что архитектура EDM может не подходить для вашего нового проекта). Однако если у вас имеются действующие унаследованные системы, которые нуждаются в поддержке, то читайте дальше.

В любой предметной области предприятия существуют сущности и события, которые обычно требуются в многочисленных поддоменах. Например, розничный интернет-торговец должен предоставлять информацию о продукте, ценах, запасах и изображениях в различные ограниченные контексты. Возможно, платежи собираются одной системой, но должны подтверждаться в другой, а анализ шаблонов покупок выполняется в третьей. Предоставление этих данных в едином месте в качестве нового базового источника истины позволяет каждой системе потреблять их по мере их поступления. Миграция на событийно-управляемые микросервисы требует предоставления необходимых данных предметной области в брокере событий в виде потоков событий. Этот процесс именуется *освобождением данных* (data liberation) и предусматривает *получение* данных из существующих систем и хранилищ состояний, которые их содержат.

Данные, переданные в поток событий, могут быть доступны любой системе — событийно-управляемой или иной. В то время как событийно-управляемые приложения способны для чтения событий использовать потоковые фреймворки и обычных потребителей, унаследованные приложения могут оказаться не в состоянии получать к ним доступ так же легко из-за целого ряда факторов, таких как особенности технологии и ограничения в производительности. В этом случае вам может потребоваться *передать* события из потока событий в имеющееся хранилище состояний.

Существует целый ряд шаблонов и фреймворков для получения событийных данных из источника и их передачи получателям¹. Для каждого технического приема в этой главе будет рассмотрена причина, почему это необходимо, приемы, как это сделать, и компромиссы, связанные с различными подходами. Затем мы рассмотрим, как освобождение данных и передача данных получателю вписываются в организацию в целом, какое влияние они оказывают и как структурировать ваши усилия для достижения успеха.

Что такое освобождение данных?

Освобождение данных — это выявление и публикация трансграничных (cross-domain) наборов данных из разных предметных областей в соответствующие им потоки событий и является частью *стратегии миграции* для событийно-управляемых архитектур. Трансграничные наборы данных включают любые данные, хранящиеся в одном хранилище данных, которые требуются другим внешним системам. Зависимости «точка-точка» между существующими сервисами и хранилищами данных часто выявляют трансграничные данные, которые должны быть освобождены, как показано на рис. 4.1, где три зависимых сервиса напрямую делают запросы к унаследованной системе.

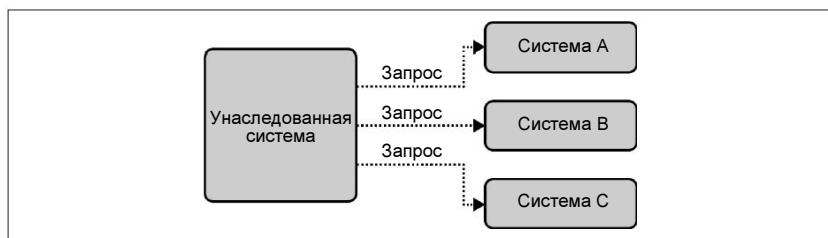


Рис. 4.1. Зависимости «точка-точка»: доступ к данным непосредственно из сервиса

Освобожденные потоки событий позволяют создавать новые управляемые событиями микросервисы в качестве потребителей, а существующие системы — своевременно переносить. Реактивные, управляемые событиями фреймворки и сервисы теперь могут использоваться для потребления и обработки данных, и последующим потребителям больше не требуется напрямую связываться с системой исходных данных.

Служа единственным источником истины, эти потоки также стандартизируют способ, которым системы во всей организации обращаются к данным. Системы больше не нуждаются в непосредственной связи с опорными хранилищами данных и приложениями, а вместо этого могут ориентироваться исключительно на контракты с потоками событий на передачу данных. Рабочий процесс после освобождения показан на рис. 4.2.

¹ Источники (source) и получатели (sink) данных — это инструменты, обеспечивающие направленность потоков. Источник данных генерирует поток трафика, а получатель данных завершает поток трафика. — Прим. перев.

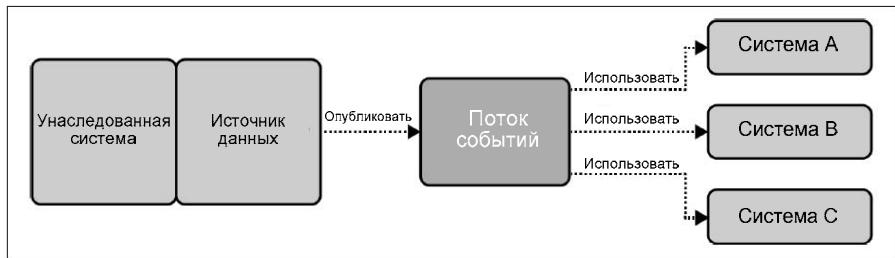


Рис. 4.2. Рабочий процесс после освобождения данных

Компромиссы для освобождения данных

Набор данных и его поток освобожденных событий должны быть полностью синхронизированы, хотя это требование ограничено возможностью согласованности (eventual consistency) из-за задержки распространения событий. Поток освобожденных событий должен материализоваться обратно в точную реплику исходной таблицы, и это свойство широко используется в событийно-управляемых микросервисах (как показано в главе 7). В отличие от этого, унаследованные системы не перестраивают свои наборы данных из каких-либо потоков событий, а вместо этого, как правило, имеют собственные механизмы резервного копирования и восстановления и не читают абсолютно ничего из потока освобожденных событий.

В идеальном мире все состояния будут создаваться, управляться, поддерживаться и восстанавливаться из единого источника истины потоков событий. Любое совместно используемое состояние должно быть *сначала опубликовано* в брокере событий и материализовано обратно в любые сервисы, которые должны материализовать это состояние, включая сервис, который исходно произвел данные (рис. 4.3).

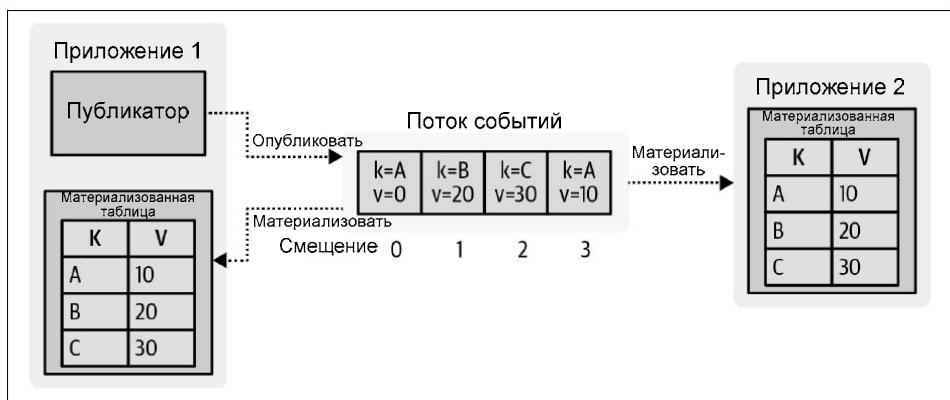


Рис. 4.3. Публикация в потоке перед материализацией

Хотя идеальная ситуация поддержания состояния в брокере событий возможна для новых микросервисов и переработанных унаследованных приложений, она *не обязательно* возможна или полезна для всех приложений. Это особенно верно для сер-

висов, которые вряд ли когда-либо будут переработаны или изменены после первоначальной интеграции с механизмами обработки изменений в данных. Унаследованные системы могут быть как чрезвычайно важными для организации, так и чрезвычайно трудными для модификации, причем самые сложные случаи таких систем называются «большим комком грязи» (*big ball of mud*)². Тем не менее, несмотря на сложность системы, их внутренние данные все равно должны быть доступны другим новым системам. И хотя модификация таких систем весьма желательна, существует ряд проблем, которые мешают этому произойти в реальности:

- ◆ *ограниченная поддержка разработчиков* — многие унаследованные системы имеют минимальную поддержку разработчиков и требуют ряда решений для создания освобожденных данных;
- ◆ *расходы на модификацию* — переработка ранее существовавших прикладных рабочих процессов в смесь асинхронной событийно-управляемой и синхронной логики веб-приложений на основе концепции MVC (Model-View-Controller) может оказаться непомерно дорогостоящей, в особенности для сложных унаследованных монолитов;
- ◆ *риск унаследованной поддержки* — изменения, внесенные в унаследованные системы, могут иметь непреднамеренные последствия, в особенности когда обязанности системы неясны из-за технических недоработок и неидентифицированных соединений «точка-точка» с другими системами.

Впрочем, и здесь есть возможность для компромисса. Вы можете использовать шаблоны освобождения данных для извлечения данных из хранилища данных и создания необходимых потоков событий в форме односторонней событийно-управляемой архитектуры, поскольку унаследованная система *не станет* читать данные из потока освобожденных событий (как это показано на рис. 4.3). Вместо этого главная цель состоит в том, чтобы поддерживать синхронизацию внутреннего набора данных с потоком внешних событий посредством строго контролируемой публикации событийных данных. И поток событий будет в конечном счете согласован с внутренним набором данных унаследованного приложения (рис. 4.4).

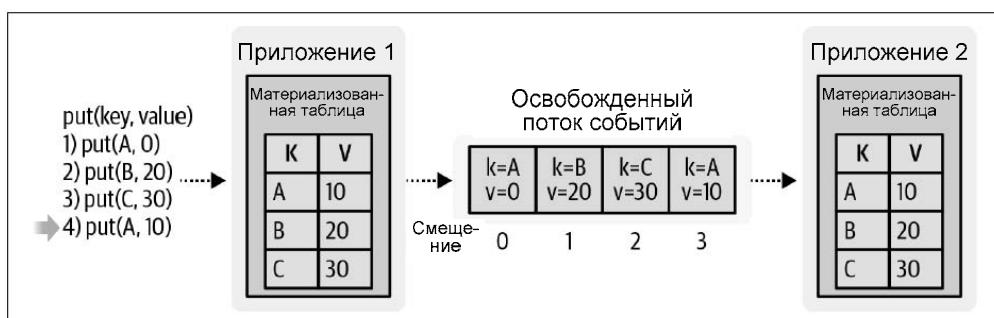


Рис. 4.4. Освобождение и материализация состояния между двумя сервисами

² См. <https://oreil.ly/8bJew>.

Конвертация освобожденных данных в события

Освобожденные данные, как и любое другое событие, подчиняются тем же правилам развития схем, которые были представлены в главе 3. Одним из свойств четко определенного потока событий является наличие явно определенной и версионно совместимой схемы для событий, которые он содержит. Вы должны убедиться, что потребители имеют базовые гарантии качества данных в рамках определенного схемой контракта на передачу данных. Изменения в схеме могут быть сделаны только в соответствии с правилами развития.



Используйте в своей организации один и тот же стандартный формат как для освобожденных событийных данных, так и для собственных событийных данных.

По определению данные, которые наиболее релевантны и используются в бизнесе, — это данные-кандидаты на освобождение. Изменения, внесенные в определения исходных данных, — такие как создание новых полей, изменение существующих или удаление каких-либо других, могут привести к динамическому изменению данных, передаваемых вниз по потоку потребителям. Отсутствие возможности использовать явно определенную схему для освобожденных данных вынудит нижестоящих потребителей принимать решения по устраниению любых несовместимостей. Это создает серьезные проблемы для обеспечения единого источника истины, поскольку нижестоящие потребители не должны пытаться анализировать или интерпретировать данные самостоятельно. Чрезвычайно важно обеспечить надежную и актуальную схему производимых данных и тщательно продумать развитие форматов данных во временной динамике.

Шаблоны освобождения данных

Существуют три главных шаблона освобождения данных, которые можно использовать для извлечения данных из базового хранилища данных. Поскольку освобожденные данные предназначены для формирования нового единого источника истины, они должны содержать весь набор данных из хранилища данных. Кроме того, эти данные должны постоянно обновляться с помощью новых вставок, обновлений и удалений.

◆ На основе запросов.

Вы извлекаете данные, выполняя запросы в базовое хранилище состояний. Это можно делать в любом хранилище данных.

◆ На основе журнала.

Вы извлекаете данные, отслеживая изменения в базовых структурах данных с помощью журнала, предназначенного только для добавления записей. Этот вариант доступен лишь для отдельных хранилищ данных, которые ведут журнал изменений, вносимых в данные.

◆ На основе таблицы.

В этом шаблоне вы сначала направляете данные в таблицу, используемую в качестве выходной очереди. Другой поток исполнения (или отдельный процесс) запрашивает эту таблицу и передает данные в соответствующий поток событий, а затем удаляет ассоциированные записи. Этот метод требует, чтобы хранилище данных поддерживало как транзакции, так и механизм выходной очереди — обычно это автономная таблица, настроенная для использования в качестве очереди.

Хотя каждый из этих трех шаблонов уникален, но между всеми ними есть одна общая черта. Каждый из них должен создавать свои события в отсортированном порядке временных меток, используя самое последнее время `updated_at` (обновленно в) исходной записи в заголовке выходного события. При этом будет генерироваться поток событий с меткой времени, соответствующей событию, а не времени, когда производитель опубликовал событие. Этот принцип особенно важен для освобождения данных, поскольку точно отражает время, когда события действительно произошли в бизнес-процессе. Чертежование событий на основе временных меток обсуждается далее в главе 6.

Фреймворки освобождения данных

Один из методов освобождения данных предусматривает использование выделенного централизованного фреймворка для извлечения данных в потоки событий. Примерами централизованных фреймворков для захвата потоков событий могут служить Kafka Connect (<https://oreil.ly/v0cpx>) — исключительно для платформы Kafka, Apache Gobblin (https://oreil.ly/pFRM_) и Apache NiFi (<https://oreil.ly/9sDb1>). Каждый фреймворк позволяет выполнять запрос к опорному набору данных, при этом его результаты передаются по конвейеру в выходные потоки событий. Каждый параметр также является масштабируемым, благодаря чему вы можете вносить в них дополнительные экземпляры, чтобы увеличивать емкость для исполнения заданий по захвату изменений в данных (Change-Data Capture, CDC). Они поддерживают различные уровни интеграции с реестром схем, предлагаемым библиотекой Confluent (Apache Kafka) (<https://oreil.ly/BkxyW>), но индивидуальная настройка, безусловно, может выполняться для поддержки и других реестров схем (см. разд. «Реестр схем» главы 14).

Не все процессы освобождения данных требуют выделенного фреймворка, и многие системы лучше подходят для того, чтобы брать на себя непосредственное владение собственным производством потока событийных данных. На самом деле выделенные фреймворки непреднамеренно поощряют неправильные шаблоны доступа к данным. Одним из наиболее распространенных таких шаблонов является доступность внутренних моделей данных внешним системам, что еще больше увеличивает сцепленность в противовес ее уменьшению, являющемуся одним из главных преимуществ событийно-управляемых архитектур. Это будет рассмотрено далее в оставшейся части главы.

Освобождение данных по запросу

Шаблон освобождения данных по запросу предусматривает запрос к хранилищу данных и передачу выбранных результатов в ассоциированный поток событий. Для запроса некоторого набора данных из хранилища данных используется клиент на основе соответствующего API, SQL или SQL-подобного языка. Набор данных — чтобы предоставлять всю историю событий — должен быть пригоден для массовой загрузки. Затем периодические обновления обеспечивают передачу изменений в выходной поток событий.

В этом шаблоне используется несколько типов запросов.

Массовая загрузка

Массовая загрузка запрашивает и загружает все данные из набора данных. Массовые загрузки организуются, когда вся таблица должна быть загружена в каждый интервал, а также перед текущими инкрементными обновлениями.

Массовая загрузка бывает дорогостоящей, т. к. она требует получения целиком всего набора данных из хранилища данных. Для малых наборов данных это, как правило, не является проблемой, но крупные наборы данных, в особенности с миллионами или миллиардами записей, могут оказаться трудными для загрузки. Для запросов и обработки очень крупных наборов данных я рекомендую вам изучить практические рекомендации для вашего конкретного хранилища данных, поскольку они могут значительно варьироваться в зависимости от их реализации.

Инкрементная загрузка временных меток

При инкрементной загрузке временных меток (`timestamp`) вы запрашиваете и загружаете все данные, начиная с самой последней временной метки результатов предыдущего запроса. Этот подход использует столбец или поле `updated-at` (обновлено в) в наборе данных, которые отслеживают время последнего изменения записи. Во время каждого инкрементного обновления запрашиваются только записи с временным метками `updated-at`, более поздними, чем время последней обработки.

Загрузка с автоинкрементируемым ID

Загрузка с автоматически инкрементируемым ID предусматривает запрос и загрузку всех данных, превышающих последнее значение ID. Для этого требуется строго упорядоченное автоматически инкрементируемое поле типа `Integer` или `Long`. Во время каждого инкрементного обновления запрашиваются только записи с ID, большим, чем последний обработанный ID. Этот подход часто используется для запросов к таблицам с неизменными записями — например, при использовании исходящих таблиц (см. разд. «*Освобождение данных с помощью журналов захвата изменений в данных*» далее в этой главе).

Выполнение пользовательских запросов

Пользовательский запрос ограничен только клиентским языком запросов. Этот подход часто используется, когда клиенту требуется лишь некоторое подмножество данных из более крупного набора данных или при соединении и денормализации данных из многочисленных таблиц во избежание чрезмерного раскрытия внутренней модели данных. Например, пользователь может фильтровать данные деловых партнеров в соответствии с некоторым полем, где данные каждого партнера отправляются в свой собственный поток событий.

Инкрементное обновление

На первом шаге любого инкрементного обновления следует обеспечить наличие необходимой временной метки или автоматически инкрементируемого поля ID в записях вашего набора данных. Должно существовать поле, которое запрос может использовать для фильтрации уже обработанных записей от тех, которые он еще не обработал. Наборы данных, в которых эти поля отсутствуют, должны быть оснащены этим полем, и хранилище данных должно быть сконфигурировано для заполнения необходимой временной метки `updated_at` или автоматически инкрементируемого поля ID. Если добавлять поля в набор данных невозможно, то и инкрементные обновления с помощью шаблона на основе запросов будут невозможны.

Второй шаг — определить частоту опроса и задержку обновлений. Более высокочастотные обновления обеспечивают меньшую задержку при обновлении данных в нижестоящем потоке, хотя это происходит за счет большей общей нагрузки на хранилище данных. Также важно учитывать и то, насколько является достаточным интервал между запросами для завершения загрузки всех данных. Запуск нового запроса в то время, пока старый еще загружается, может привести к конфликтной ситуации, когда старые данные перезаписывают новые данные в выходных потоках событий.

Когда поле инкрементного обновления выбрано и определена частота обновлений, на последнем шаге — перед тем, как задействовать инкрементные обновления, — выполняется однократная массовая загрузка. Эта массовая загрузка должна запрашивать и производить все существующие данные в наборе данных перед дальнейшими инкрементными обновлениями.

Выгоды от обновления по запросу

Обновление на основе запросов имеет ряд преимуществ, в том числе:

- ◆ *настраиваемость под индивидуальные нужды.*

Можно выполнять запросы к любому хранилищу данных, при этом доступен весь спектр клиентских возможностей для выполнения запросов.

- ◆ *Независимые периоды опрашивания.*

Конкретные запросы могут выполняться чаще, чтобы соответствовать более жестким соглашениям об уровне обслуживания (Service Level Agreements, SLA),

в то время как другие, более дорогие, запросы могут — ради экономии ресурсов — выполняться менее часто.

◆ *Изоляция внутренних моделей данных.*

Реляционные базы данных могут обеспечивать изоляцию от внутренней модели данных с помощью представлений (view) или материализованных представлений опорных данных. Этот метод можно использовать для скрытия информации о модели предметной области, которая не должна выставляться наружу, за пределы хранилища данных.



Помните, что освобожденные данные будут единственным источником истины. Подумайте о том, следует ли вместо этого освобождать какие-либо скрытые или пропущенные данные или же необходимо переработать исходную модель данных. Это часто происходит во время освобождения данных из унаследованных систем, где бизнес-данные и метаданные со временем переплетаются.

Недостатки обновления по запросу

У обновления на основе запросов есть и некоторые недостатки:

- ◆ *потребность во временной метке updated-at (обновлено в)* — опорная таблица или пространство имен опрашиваемых событий должны иметь столбец, содержащий их временну́ю метку updated-at. Это важно для отслеживания времени последнего обновления данных и для выполнения инкрементных обновлений;
- ◆ *неотслеживаемые жесткие удаления* — жесткие удаления (физические удаления записей из таблицы) не будут показаны в результатах запроса, поэтому отслеживание удалений ограничено мягкими удалениями на основе флагов, таких как столбец логического типа is_deleted (запись является удаленной);
- ◆ *хрупкая зависимость между схемой набора данных и схемой выходных событий* — могут произойти изменения схемы набора данных, несовместимые с правилами нижестоящей схемы событийного формата. Несоответствия становятся все более вероятными, если механизм освобождения отделен от кодовой базы приложения хранилища данных, что обычно имеет место в случае систем, основанных на запросах;
- ◆ *кратковременный захват* — данные синхронизируются лишь через интервалы опроса, и поэтому серия отдельных изменений в одной и той же записи может отображаться как одно событие;
- ◆ *потребление производственных ресурсов* — при выполнении запросов задействуются ресурсы базовой системы, что может привести к недопустимым задержкам в производственной среде. Эта проблема может быть устранена с помощью реплики данных «только для чтения», но при этом потребуются дополнительные финансовые затраты и увеличение сложности системы;
- ◆ *переменная производительность запросов из-за изменений в данных* — количество запрашиваемых и возвращаемых данных варьируется в зависимости от изменений, внесенных в базовые данные. В худшем случае каждый раз изменяется весь массив данных. Это может привести к конфликтным ситуациям, когда один запрос не будет завершен до начала следующего.

Освобождение данных с помощью журналов захвата изменений в данных

Еще один шаблон освобождения данных предусматривает использование базовых журналов регистрации *изменений в данных* в хранилищах данных (двоичные журналы в MySQL, журналы с упреждающей записью в PostgreSQL). Эти журналы представляют собой структуры регистрации только добавляемых данных, которые учитывают все, что произошло с отслеживаемыми наборами данных с течением времени. Регистрируемые изменения включают создание, удаление и обновление отдельных записей, а также создание, удаление и изменение отдельных наборов данных и их схем.

Технологические возможности для захвата изменений в данных более узки, чем для захвата данных на основе запросов. Не все хранилища данных реализуют неизменяемое журналирование событий, а из тех, которые это делают, не все они имеют готовые коннекторы, доступные для извлечения данных. Этот подход в основном применим для реляционных баз данных — таких как MySQL и PostgreSQL, хотя подходящим кандидатом может стать и любое хранилище данных, имеющее обстоятельные журналы. Многие другие современные хранилища данных предоставляют API-интерфейсы событий, которые действуют как прокси для физического журнала упреждающей записи. Например, MongoDB предоставляет интерфейс потоков изменений (<https://oreil.ly/v9gd0>), в то время как Couchbase обеспечивает доступ к репликации через свой внутренний репликационный протокол.

Журнал хранилища данных вряд ли будет включать все изменения с начала использования системы, поскольку это может быть огромный объем данных, и обычно его нет необходимости хранить. Поэтому вам до начала процесса захвата изменений в данных на основе журнала хранилища данных нужно будет сделать снимок существующих данных. Такой моментальный снимок обычно включает в себя большой, влияющий на производительность запрос к таблице и обычно называется *начальной загрузкой* (bootstrapping). Вы должны убедиться, что существует перекрытие между записями в результатах этого запроса и записями в журнале, чтобы случайно не пропустить ни одной записи.

При захвате событий из журналов изменений вы должны периодически размечать ход процесса с помощью контрольных точек, хотя в зависимости от используемого вами инструмента это может быть уже в него встроено. В случае отказа механизма захвата изменений в данных такая контрольная точка служит для восстановления последнего сохраненного индекса журнала изменений. Этот подход может обеспечить по меньшей мере хотя бы однократное производство записей, что, как правило, подходит для природы освобождения данных, основанной на сущностях. Создание дополнительных записей не имеет последствий, поскольку обновление сущностных данных является процессом идемпотентным³.

³ Идемпотентность — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом. — Прим. ред.

Разработано несколько вариантов извлечения исходных данных из журналов изменений. Одним из самых популярных вариантов для реляционных баз данных является технология Debezium (<https://debezium.io>), поскольку поддерживает наиболее распространенные из них (<https://oreil.ly/oFSax>). Debezium может производить записи как для Apache Kafka, так и для Apache Pulsar с его существующими реализациями. Безусловно возможна поддержка и дополнительных брокеров, хотя для этого может потребоваться некоторая собственная доработка. Технология Maxwell (<https://oreil.ly/Rr4Kp>) — это еще один пример варианта чтения двоичных журналов, хотя в настоящее время она ограничена поддержкой только баз данных MySQL и может производить данные лишь для Apache Kafka.

На рис. 4.5 показана база данных MySQL, эмитирующая свой двоичный журнал изменений. Сервис Kafka Connect, работающий с коннектором Debezium, потребляет «сырой» (raw) двоичный журнал. Debezium разбирает данные и конвертирует их в дискретные события. Затем маршрутизатор событий передает каждое событие в конкретный поток событий в Kafka — в зависимости от исходной таблицы этого события. Нижестоящие потребители теперь могут получать доступ к содержимому базы данных, потребляя соответствующие потоки событий из Kafka.

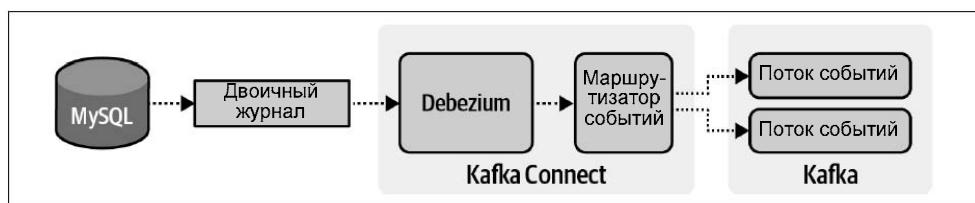


Рис. 4.5. Сквозной рабочий процесс Debezium, захватывающий данные из двоичного журнала базы данных MySQL и записывающий их в потоки событий в Kafka

Выгоды от использования журналов хранилища данных

Некоторые выгоды от использования журналов хранилища данных включают в себя следующие:

- ◆ *Отслеживание удалений.*

Двоичные журналы содержат данные о жестких удалениях записей. Они могут быть преобразованы в события удаления без необходимости мягкого удаления, как в обновлениях на основе запросов.

- ◆ *Минимальное влияние на производительность хранилища данных.*

Для хранилищ данных, использующих журнал с упреждающей записью и двоичные журналы, захват изменений в данных может выполняться без какого-либо влияния на производительность хранилища данных. Для тех, кто использует таблицы изменений, например в SQL Server, уровень такого влияния зависит от объема данных.

◆ *Обновления с низкой задержкой.*

Обновления могут распространяться сразу же после записи события в двоичный журнал и журнал с упреждающей записью. Это приводит к очень низкой задержке по сравнению с другими шаблонами освобождения данных.

Недостатки использования журналов базы данных

Далее приведены несколько недостатков использования журналов базы данных:

- ◆ *раскрытие внутренних моделей данных* — внутренняя модель данных в журналах изменений полностью выставляется наружу. Управление изоляцией базовой модели данных в такой ситуации должно быть тщательным и выборочным, в отличие от обновления на основе запросов, где для обеспечения изоляции могут использоваться представления (view);
- ◆ *денормализация⁴ вне хранилища данных* — журналы изменений содержат только событийные данные. Некоторые механизмы CDC могут извлекать данные из материализованных представлений, но для многих других денормализация должна происходить вне хранилища данных. Это может привести к созданию высоконормализованных потоков событий, требующих от нижестоящих микросервисов обработки соединений по внешнему ключу и денормализации;
- ◆ *хрупкая зависимость между схемой набора данных и схемой выходного события* — как и процесс освобождения данных на основе запросов, процесс на основе двоичного журнала существует вне приложения хранилища данных. Допустимые изменения хранилища данных — такие как изменение набора данных или переопределение типа поля, могут быть совершенно несовместимыми с конкретными правилами развития схемы событий.

Освобождение данных с помощью исходящих таблиц

Исходящая таблица (outbox) содержит заметные изменения, внесенные во внутренние данные хранилища, причем каждое значительное обновление в ней хранится в виде отдельной строки. Всякий раз, когда выполняется вставка, обновление или удаление в одной из таблиц хранилища данных, помеченных для захвата изменений в данных, соответствующая запись может быть опубликована в исходящей таблице. Каждая таблица в рамках захвата изменений в данных может иметь свою собственную исходящую таблицу или одна исходящая таблица может использоваться для всех изменений (подробнее об этом чуть позже).

Как внутренние обновления таблиц, так и обновления исходящих таблиц должны быть объединены в *единую транзакцию*, благодаря чему каждое из них происходит только в том случае, если вся транзакция завершается успешно. Неспособность

⁴ Денормализация — намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации, обычно проводимое с целью ускорения операций чтения из базы за счет добавления избыточных данных. — Прим. ред.

правильно провести этот процесс может в конечном счете привести к расхождению с потоком событий как единственным источником истины, которое бывает трудно обнаружить и исправить. Рассматриваемый в этом разделе шаблон обеспечивает более жесткий подход к захвату изменений в данных, поскольку требует модификации либо хранилища данных, либо слоя приложения, что, в свою очередь, требует участия разработчиков хранилища данных. Шаблон таблицы исходящих сообщений использует долговечность хранилища данных, обеспечивающую журналом упреждающей записи для событий, ожидающих публикации во внешних потоках событий.

Встроенные таблицы изменений в данных

Некоторые базы данных, такие как SQL Server, не используют журналы захвата изменений в данных, а вместо этого содержат таблицы изменений в данных. Эти таблицы часто используются для аудита операций базы данных и встроены в функциональность. Внешние сервисы, такие как ранее упомянутые Kafka Connect и Debezium, могут подключаться к базам данных, содержащим таблицу CDC вместо журнала CDC, и использовать шаблон на основе запросов для извлечения событий и продуцирования их в потоки событий.

Записи в исходящих таблицах должны иметь строгий идентификатор упорядочения, т. к. один и тот же первичный ключ может обновляться много раз в короткие сроки. Кроме того, вы можете перезаписывать предыдущее обновление для этого первичного ключа, хотя для этого сначала требуется найти предыдущую запись, и это вводит дополнительные накладные расходы на производительность. Это также означает, что перезаписанная запись не будет передаваться далее.

Автоматически инкрементируемый идентификатор, назначенный во время вставки, лучше всего использовать для определения порядка публикации событий. Также следует поддерживать столбец с временной меткой `created_at` (создано в), поскольку он отражает время события, когда запись была создана в хранилище данных, и может использоваться вместо физического времени при публикации в потоке событий. Это позволит планировщику точно чередовать события, как описано в главе 6.

На рис. 4.6 показан сквозной рабочий процесс для решения на основе исходящей таблицы CDC. Обновления внутренних таблиц, выполняемые клиентом хранилища

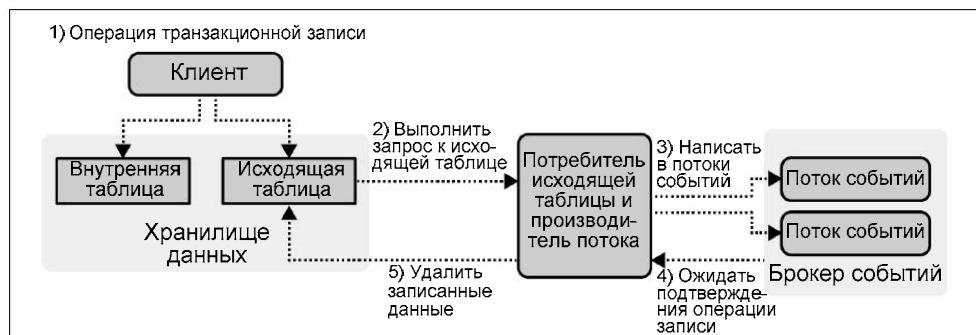


Рис. 4.6. Сквозной рабочий процесс для решения на основе исходящей таблицы CDC

данных, обертываются в транзакцию с обновлением исходящей таблицы, благодаря чему любые отказы не нарушают согласованность данных между ними. Между тем отдельный поток исполнения или процесс приложения используется для постоянного опроса исходящих таблиц и передачи данных в соответствующие потоки событий. После передачи данных соответствующие записи в исходящей таблице удаляются. В случае любого отказа, будь то в хранилище данных, у потребителя/производителя или в самом брокере событий, записи исходящей таблицы все равно будут сохранены без риска потери. Таким образом, этот шаблон обеспечивает гарантии по меньшей мере однократной доставки.

Некоторые мысли по поводу производительности

Включение исходящих таблиц создает дополнительную нагрузку на хранилище данных и его приложения по обработке запросов. Для малых хранилищ данных с минимальной нагрузкой накладные расходы могут остаться совершенно незамеченными. С другой стороны, в случае очень крупных хранилищ данных, в особенности тех, что имеют собственную значительную нагрузку и много таблиц, отслеживаемых захватом, эта дополнительная нагрузка может оказаться весьма существенной. Стоимость такого подхода должна оцениваться в каждом конкретном случае и быть сбалансирована с затратами на реактивную стратегию, подразумевающую в том числе анализ журналов захвата изменений в данных.

Изоляция внутренних моделей данных

Исходящую таблицу не нужно соотносить 1:1 с внутренней таблицей. Надо иметь в виду, что одна из главных выгод от исходящей таблицы состоит в том, что клиент хранилища данных может изолировать внутреннюю модель данных от нижестоящих потребителей. Внутренняя модель данных предметной области может использовать ряд высоконормализованных таблиц, оптимизированных для реляционных операций, но в значительной степени непригодных для использования потребителями. Даже простые предметные области могут содержать многочисленные таблицы, которые, если они будут выставлены наружу как независимые потоки, потребуют модификации для использования потребителями. Это быстро станет чрезвычайно дорогостоящим с точки зрения операционных накладных расходов, поскольку многочисленным командам, находящимся внизу потока, придется менять модель предметной области и заниматься обработкой реляционных данных в потоках событий.



Предоставление внутренней модели данных последующим потребителям — это самый неверный подход в полном смысле этого слова. Последующие потребители должны иметь доступ только к тем данным, которые отформатированы с помощью ориентированных вовне контрактов на передачу данных, как описано в главе 3.

Вместо этого клиент хранилища данных может денормализовать данные во время вставки таким образом, чтобы исходящая таблица отражала предполагаемый публичный контракт на передачу данных, хотя это и происходит за счет дополнительной производительности и пространства для хранения. Еще один вариант — под-

держивать соотношение 1:1 между изменениями и выходными потоками событий и денормализовывать потоки с помощью нижестоящего обработчика событий, предназначенного именно для этой задачи. Я называю это процесс *событизацией* (eventification), поскольку он конвертирует высоконормализованные реляционные данные в простые для потребления обновления отдельных событий. Событизация имитирует то, что *мог бы* сделать клиент хранилища данных, но делает это внешне по отношению к хранилищу данных, чтобы уменьшить нагрузку. Пример этого процесса показан на рис. 4.7, где пользователь User денормализуется на основе исходящих таблиц User (Пользователь), Location (Местоположение) и Employer (Работодатель).

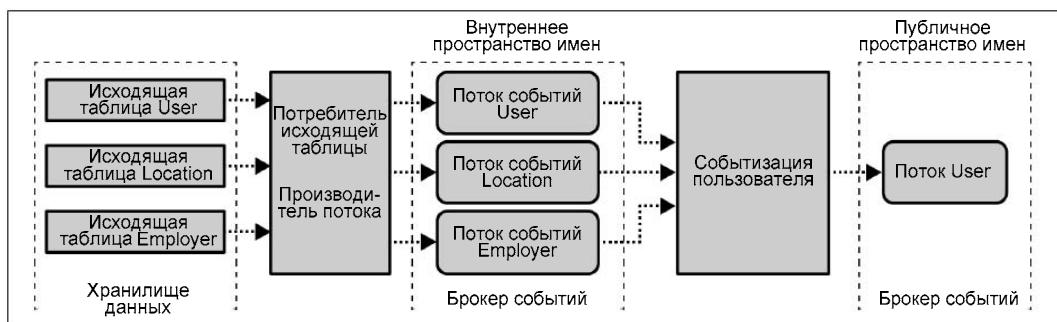


Рис. 4.7. Событизация публичных событий пользователя User с помощью приватных потоков событий User (Пользователь), Location (Местоположение) и Employer (Работодатель)

В этом примере пользователь User имеет ссылку в форме внешнего ключа на город, штат/провинцию и страну, в которой он живет, а также ссылку в форме внешнего ключа на своего текущего работодателя. Вполне разумно, что нижестоящий потребитель события User может просто захотеть иметь всю информацию о каждом пользователе в одном событии, вместо того чтобы вынужденно материализовывать каждый поток в хранилище состояний и использовать реляционные инструменты для его денормализации. Необработанные нормализованные события поступают из исходящих таблиц и направляются в их собственные потоки событий, но эти потоки — для защиты внутренней модели данных — хранятся во внутреннем пространстве имен, отделенном от остальной организации (см. разд. «Разметка потоков событий метаданными» главы 14).

Событизация пользователя осуществляется путем денормализации сущности User и отбрасывания любых структур внутренней модели данных. Этот процесс требует поддержания материализованных таблиц User, Location и Employer таким образом, чтобы любые обновления могли повторно использовать логику соединения и выдавать обновления для всех затронутых пользователей User. Конечное событие передается в публичное пространство имен организации для потребления любым потребителем.

Степень изоляции внутренних моделей данных от внешних потребителей, как правило, становится предметом разногласий в организациях, переходящих к событий-

но-управляемым микросервисам. Изоляция внутренней модели данных имеет важное значение для устранения сильного зацепления и обеспечения независимости сервисов, а также для гарантии того, что системы нужно будет изменять только в связи с новыми бизнес-требованиями, а не для изменения внутренней модели данных в восходящем направлении.

Обеспечение совместимости схем

Сериализация схемы (и, следовательно, валидация) также может быть встроена в рабочий процесс захвата. Это может выполняться как до, так и после записи события в исходящую таблицу. Успех означает, что событие может быть продолжено в рабочем процессе, в то время как отказ может потребовать ручного вмешательства, чтобы определить первопричину и избежать потери данных.

Сериализация перед фиксацией транзакции в исходящую таблицу обеспечивает самую сильную гарантию согласованности данных. Отказ сериализации приведет к отказу транзакции и откату всех изменений, внесенных во внутренние таблицы, обеспечивая, чтобы исходящая таблица и внутренние таблицы оставались синхронизированными. Этот процесс показан на рис. 4.8. После успешной валидации событие будет сериализовано и готово к публикации потока событий. Главное преимущество такого подхода заключается в том, что несогласованность данных между внутренним состоянием и выходным потоком событий значительно сокращается. Данные потока событий трактуются как приоритетные, и публикация правильных данных считается столь же важной, как и поддержание согласованного внутреннего состояния.

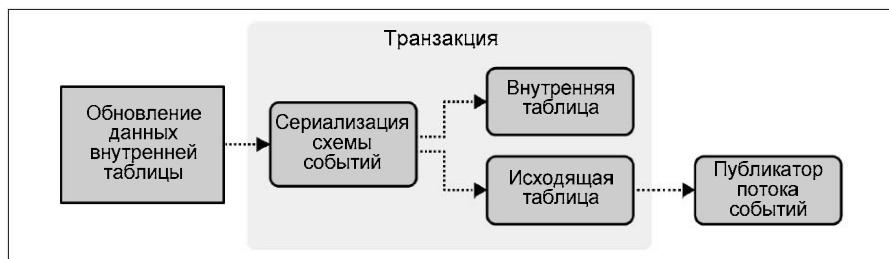


Рис. 4.8. Сериализация данных об изменениях перед записью в исходящую таблицу

Сериализация перед записью в исходящую таблицу также предоставляет вам возможность использовать одну исходящую таблицу для всех транзакций. Формат таблицы прост, т. к. ее содержимое представляет собой преимущественно сериализованные данные с соотнесением с целевым выходным потоком событий (рис. 4.9).

Одним из недостатков сериализации перед публикацией является то, что из-за накладных расходов на сериализацию может пострадать производительность. Это бывает не столь значимо для легких нагрузок, но при более тяжелых нагрузках может вызвать серьезные затруднения. Вам надо будет обеспечить соответствующую

Исходящая таблица				
id (автоинкремент)	created_at	serialized_key	serialized_value	output_stream
8273	2020-07-07T07:43:10	A0 FB 24	0112 C5 BB D4	Accounts
8274	2020-07-07T07:43:10	DE A8 EF	25 6B EA F9 76	Users

Рис. 4.9. Отдельная выходная таблица с уже валидированными и сериализованными событиями (обратите внимание на элемент `output_stream`, предназначенный для целей маршрутизации)

производительность системы, чтобы все эти процессы не оказывали на нее существенного влияния.

С другой стороны, сериализация может быть выполнена и после записи события в исходящую таблицу, как показано на рис. 4.10.

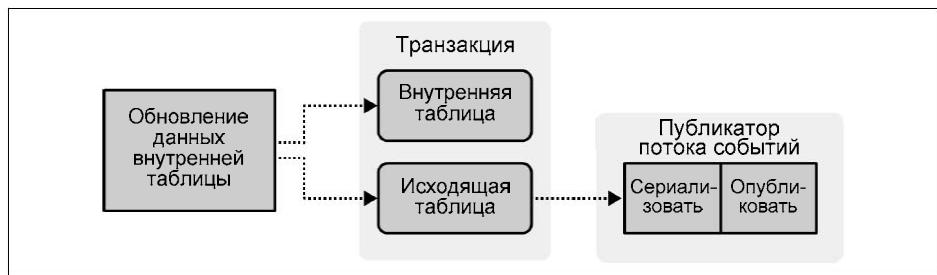


Рис. 4.10. Сериализация изменений в данных после записи в исходящую таблицу в рамках процесса публикации

При реализации этой стратегии у вас обычно имеются независимые исходящие таблицы, по одной для каждой модели предметной области, соотнесенные с публичной схемой соответствующего выходного потока событий. Процесс публикатора читает несериализованное событие из исходящей таблицы и пытается сериализовать его с ассоциированной схемой перед тем, как поместить его в выходной поток событий. На рис. 4.11 показан пример нескольких (двух) исходящих таблиц: одна — для сущности `User` и вторая — для сущности `Account`.

Исходящая таблица User				
id (автоинкремент)	created_at	name	address	country
23	2020-05-20T16:30:00	Justin T.	123 Road Lane	Canada
24	2020-05-20T16:30:01	Robert O.	60 3rd Street	USA

Исходящая таблица Account				
id (автоинкремент)	created_at	account_id	user_id	type
45	2020-06-02T09:10:10	623	2729	Cash
46	2020-06-03T11:53:01	638	4291	Credit

Рис. 4.11. Несколько исходящих таблиц (обратите внимание, что данные не сериализованы, а это означает, что они могут быть несовместимы со схемой выходного потока событий)

Отказ выполнить сериализацию указывает на то, что событийные данные не соответствуют определенной схеме и поэтому не могут быть опубликованы. Именно из-за этого становится сложнее поддерживать возможность сериализации после записи, поскольку уже завершенная транзакция будет помещать несовместимые данные в исходящую таблицу и нет никакой гарантии, что транзакция не будет отменена.

На самом деле вы, как правило, в итоге получаете в своей исходящей таблице значительное количество несериализуемых событий. Скорее всего, потребуется вмешательство человека, чтобы попытаться спасти некоторые данные, но решение этой проблемы будет трудоемким и трудным и может даже потребовать остановки процесса для предотвращения дополнительных проблем. Это усугубляется тем фактом, что некоторые события действительно могли оказаться совместимыми и уже были опубликованы, что может привести к неправильному порядку событий в выходных потоках.



Сериализация до публикации обеспечивает более сильную гарантию от несовместимых данных, чем после нее, и предотвращает распространение событий, нарушающих их контракт на передачу данных. Компромисс заключается в том, что этот вариант также не даст завершить бизнес-процесс в случае отказа сериализации, поскольку транзакция должна быть откачена назад.

Таким образом, валидация и сериализация перед записью обеспечивают, чтобы данные обрабатывались как приоритетные, и дают гарантию того, что события в выходном потоке событий будут в конечном счете согласованы с данными внутри исходного хранилища данных, а также будут поддерживать изоляцию внутренней модели данных источника. Это самая сильная гарантия, которую может предложить решение по захвату изменений в данных.

Выгоды от создания событий с помощью исходящих таблиц

Создание событий с помощью исходящих таблиц дает ряд существенных преимуществ:

- ◆ *Многоязычная поддержка.*

Этот подход поддерживается любым клиентом или фреймворком, предоставляющим транзакционные возможности.

- ◆ *Обеспечение валидации по схеме до публикации.*

Схемы могут валидироваться сериализацией перед вставкой в исходящую таблицу.

- ◆ *Изоляция внутренней модели данных.*

Разработчики приложений хранилища данных могут выбирать, какие поля записывать в исходящую таблицу, сохраняя внутренние поля изолированными.

- ◆ *Денормализация.*

Данные могут денормализовываться по мере необходимости перед записью в исходящую таблицу.

Недостатки создания событий с помощью исходящих таблиц

Создание событий посредством исходящих таблиц также имеет ряд недостатков:

- ◆ *необходимы изменения кода приложения* — чтобы реализовать этот шаблон, код приложения должен быть изменен, для чего необходимо привлечение ресурсов разработки и тестирования со стороны разработчиков приложения;
- ◆ *влияние на производительность бизнес-процессов* — такое влияние может быть весьма нетривиальным, в особенности при проверке схем с помощью сериализации. Отказавшие транзакции также могут препятствовать продолжению бизнес-операций;
- ◆ *влияние на производительность хранилища данных* — это влияние также может быть нетривиальным, в особенности когда значительное количество записей записывается, считывается и удаляется из исходящей таблицы.



Влияние на производительность должно быть сбалансировано другими затратами. Например, некоторые организации просто создают события, анализируя журналы захвата изменений в данных, и предоставляют их потребителям для дальнейшей очистки событий. Это влечет за собой их собственные затраты на обработку и стандартизацию событий, а также затраты человеческого труда на выявление несовместимых схем и учет эффектов сильной сцепки с внутренними моделями данных. Затраты, сэкономленные на стороне производителя, часто ничтожны по сравнению с затратами, понесенными на стороне потребителя для решения этих проблем.

Захват изменений в данных с помощью триггеров

Поддержка триггеров предшествует многим шаблонам аудита, двоичного журналирования и журналирования с упреждающей записью, рассмотренным в предыдущих разделах. Многие старые реляционные базы данных используют триггеры для создания таблиц аудита. Как следует из их названия⁵, триггеры настраиваются на автоматическое выполнение того или иного условия. Если он не срабатывает, то команда, которая вызвала триггер для исполнения, также не срабатывает, обеспечивая атомарность⁶ обновления.

Изменения можно захватывать на уровне строк в таблице аудита с помощью триггера AFTER. Например, после любой команды INSERT, UPDATE или DELETE этот триггер запишет соответствующую строку в таблицу изменений в данных. Благодаря этому обеспечивается, что изменения, внесенные в конкретную таблицу, будут отслеживаться соответствующим образом.

⁵ Триггер (англ. Trigger) в базах данных — это хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных. — Прим. ред.

⁶ Атомарность — это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). — Прим. ред.

Рассмотрим пример, показанный на рис. 4.12. Пользовательские данные обновляются-вставляются (upsert) в пользовательскую таблицу, а триггер захватывает события по мере их возникновения. Обратите внимание, что триггер также захватывает время, в которое произошла вставка, а также последовательно автоинкрементируемый ID для использования процессом публикатора событий.

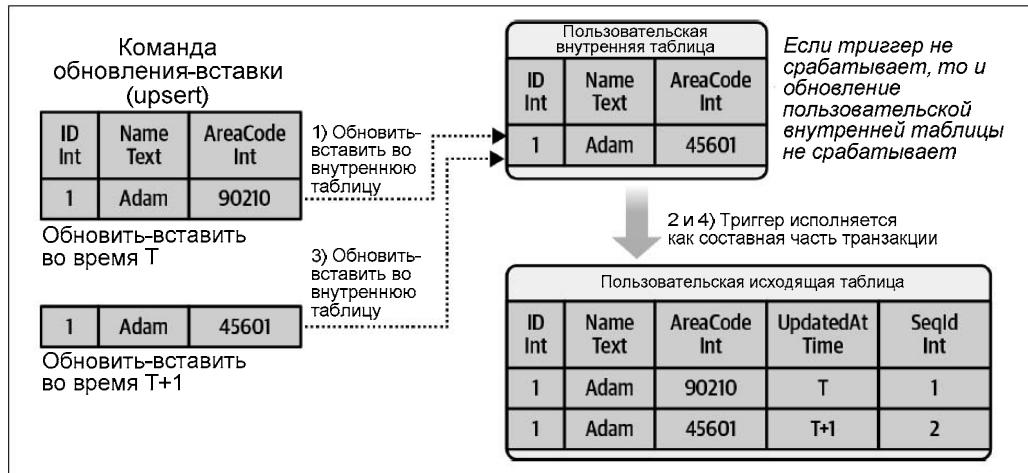


Рис. 4.12. Использование триггера для захвата изменений в пользовательской таблице

Как правило, вы не можете валидировать изменения в данных с помощью схемы событий во время исполнения триггера, хотя это не является невозможным. Одна из главных трудностей заключается в том, что это может просто не поддерживаться, поскольку триггеры исполняются в самой базе данных, и многие из них ограничены формами языка, которые они способны поддерживать. В то время как PostgreSQL поддерживает C, Python и Perl, на которых могут быть написаны пользовательские функции для выполнения валидации схемы, многие другие базы данных не обеспечивают многоязычную поддержку. Наконец, даже если триггер поддерживается, он может быть просто слишком дорогим. Каждый триггер срабатывает независимо и требует нетривиального объема накладных расходов для хранения необходимых данных, схем и валидационной логики, а для многих системных нагрузок эта стоимость слишком высока.

На рис. 4.13 показано продолжение предыдущего примера. Постфактумная валидация и сериализация выполняются для изменений в данных, при этом успешно валидированные данные передаются в выходной поток событий. Отказавшие данные должны быть обработаны на предмет ошибок в соответствии с бизнес-требованиями, но, скорее всего, потребуется вмешательство человека.

Схема таблицы захвата изменений в данных является мостом между схемой внутренней таблицы и схемой выходного потока событий. Совместимость между всеми тремя элементами имеет важное значение для обеспечения возможности передачи данных в выходной поток событий. Поскольку во время исполнения триггера вали-

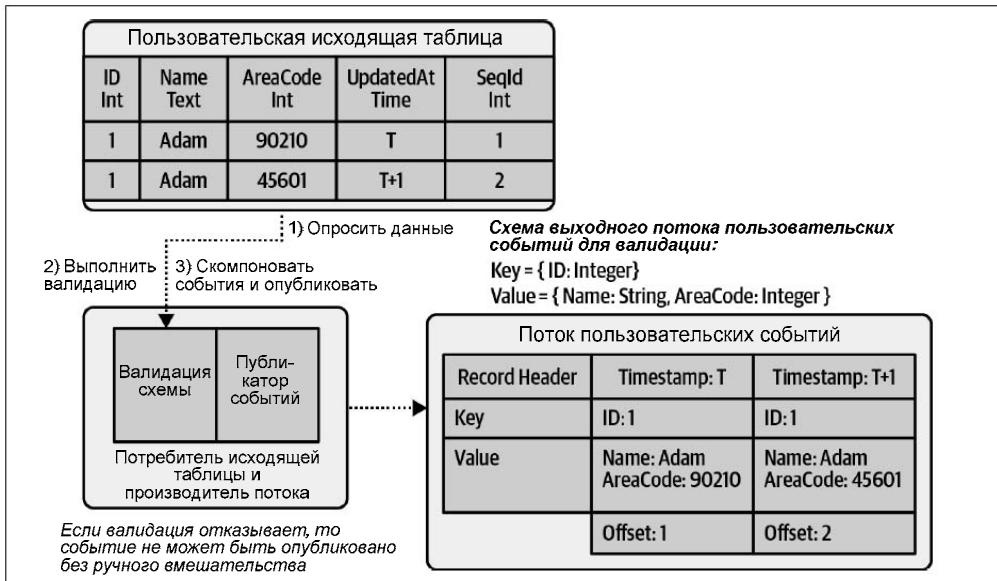


Рис. 4.13. Постфактумная валидация данных и их передача в выходной поток событий

дация выходной схемы обычно не выполняется, лучше всего синхронизировать таблицу изменений в данных с форматом схемы выходных событий.



Сравните формат схемы выходного события с таблицей изменений в данных во время тестирования. Перед производственным развертыванием могут проявиться несовместимости.

Тем не менее триггеры могут отлично работать во многих унаследованных системах. Унаследованные системы, как правило, по определению используют старые технологии, а триггеры существуют уже очень давно и вполне могут обеспечить необходимый механизм захвата изменений в данных. Шаблоны доступа и нагрузки, как правило, хорошо определены и стабильны, так что влияние добавления триггера может быть точно оценено. Наконец, хотя валидация схемы вряд ли произойдет во время самого процесса исполнения триггера, столь же маловероятно, что сами схемы изменятся просто из-за унаследованной природы системы. Постфактумная валидация является проблемой только в том случае, если ожидается, что схемы будут часто меняться.



Старайтесь избегать использования триггеров, если вместо этого вы можете применить более современную функциональность для генерирования изменений в данных или доступа к ним. Вы не должны недооценивать накладные расходы на производительность и управление, необходимые для решения на основе триггеров, в особенности когда задействовано много десятков или сотен таблиц и моделей данных.

Выгоды от использования триггеров

Использование триггеров влечет за собой следующие выгоды:

- ◆ *Поддерживается большинством СУБД.*
Триггеры разработаны для большинства реляционных баз данных.
- ◆ *Низкие накладные расходы для малых наборов данных.*
Техническое сопровождение и конфигурирование триггеров достаточно просты для малого объема данных.
- ◆ *Индивидуально настраиваемая логика.*

Код триггера можно индивидуально настроить так, чтобы он отображал только подмножество конкретных полей. За счет этого можно обеспечить некоторую изоляцию данных от нижестоящих потребителей.

Недостатки использования триггеров

Некоторые минусы использования триггеров заключаются в следующем:

- ◆ *накладные расходы на производительность* — триггеры исполняются параллельно с действиями над таблицами базы данных и могут потреблять ресурсы сервера. В зависимости от требований к производительности и соглашений об уровне обслуживания (Service Level Agreements, SLA) ваших сервисов этот подход может привести к неприемлемой нагрузке;
- ◆ *сложность управления изменениями* — изменения кода приложения и определений наборов данных могут потребовать соответствующих модификаций триггеров. При этом разработчиками системы могут быть упущены некоторые необходимые доработки базовых триггеров, что приведет к результатам освобождения данных, несовместимым с внутренними наборами данных. Для обеспечения того, чтобы рабочие процессы исполнения триггеров функционировали в соответствии с ожиданиями, следует выполнять всестороннее тестирование;
- ◆ *плохое масштабирование* — количество требуемых триггеров линейно масштабируется с количеством наборов данных, которые необходимо захватывать. Сюда не входят любые дополнительные триггеры, которые уже могут существовать в бизнес-логике, — например те, которые используются для обеспечения соблюдения зависимостей между таблицами;
- ◆ *валидация после публикации* — валидация схемы для выходного события происходит только после публикации записи в таблице исходящих сообщений, что может привести к появлению в таблице исходящих сообщений неопубликованных событий.



Некоторые базы данных допускают выполнение триггеров с помощью языков, которые могут проверять совместимость со схемами выходных событий во время исполнения триггера (например, Python для PostgreSQL). Это может увеличить сложность и затраты, но значительно снижает риск несовместимости нижестоящих схем.

Внесение изменений определения данных в захватываемые наборы данных

В рамках процесса освобождения данных интеграция изменений определения данных может быть затруднена. Миграция данных является обычной операцией для многих приложений реляционных баз данных и должна поддерживаться захватом. Изменения определения данных для реляционной базы данных могут включать добавление, удаление и переименование столбцов, изменение типа столбца, добавление или удаление значений, задаваемых по умолчанию. Хотя все эти операции являются допустимыми изменениями набора данных, они могут создавать проблемы для передачи данных в потоки освобожденных событий.



Определение данных — это формальное описание набора данных. Например, таблица в реляционной базе данных определяется с помощью языка **определения данных** (Data Definition Language, DDL). Результирующая таблица, столбцы, имена, типы и индексы являются частью определения данных.

К примеру, если требуется полная совместимость с версиями схемы, то вы не можете отбросить не допускающий значения `null` столбец без указания значения по умолчанию из захватываемого набора данных, т. к. пользователи, использующие ранее определенную схему, ожидают значения для этого поля. Потребители не смогут вернуться к какому-либо значению по умолчанию, поскольку ни одно из них не было указано во время исполнения контракта, поэтому они окажутся в двусмысленном состоянии. Если несовместимое изменение абсолютно необходимо и нарушение контракта на передачу данных неизбежно, то производитель и потребители данных должны договориться о новом контракте на передачу данных.



Валидные изменения в захватываемом наборе данных могут не быть валидными изменениями для схемы освобожденных событий. Эта несовместимость приведет к разрушительным изменениям схемы, которые повлияют на всех потребителей потока событий.

Захват изменений DDL зависит от шаблона интеграции, используемого для сбора измененных данных. Поскольку изменения DDL могут иметь значительное влияние на последующих потребителей данных, важно определить, обнаруживают ли ваши шаблоны захвата изменения в DDL до или после их совершения. Например, шаблон запроса и шаблон журнала CDC могут обнаруживать изменения DDL только постфактум, т. е. после того, как они уже были применены к набору данных. И наоборот, шаблон таблицы данных изменений интегрирован с циклом разработки исходной системы, так что изменения, внесенные в набор данных, требуют валидации с помощью таблицы данных изменений до выпуска продукта.

Обработка произведенных постфактум изменений определения данных для шаблонов запросов и журналов CDC

Для шаблона запроса схема может быть получена во время запроса, а схема события выведена логически. Новую схему событий можно сравнить со схемой выход-

ного потока событий, организованной на основе правил совместимости схемы, используемых для разрешения или запрещения публикации данных события. Этот механизм генерации схемы задействуется многочисленными коннекторами запросов — такими как те, которые предоставляются фреймворком Kafka Connect (<https://oreil.ly/9XRDrv>).

Для шаблона журнала CDC обновления определения данных обычно записываются в отдельную часть журнала CDC. Эти изменения необходимо извлечь из журналов и внести в схему, представляющую набор данных. Как только схема будет сгенерирована, она может быть валидирована на соответствие нижестоящей схеме событий. Однако поддержка этой функциональности ограничена. В настоящее время коннектор Debezium поддерживает только изменения определения данных MySQL.

Обработка изменений определения данных для шаблонов захвата таблиц изменений в данных

Таблица изменений в данных действует как мост между схемой выходного потока событий и схемой внутреннего состояния. Любая несовместимость кода проверки приложения или триггерной функции базы данных не позволит записать данные в таблицу измененных данных, а ошибка будет отправлена обратно в стек. Изменения, внесенные в таблицу отслеживания измененных данных, потребуют развития схемы, совместимой с выходным потоком событий, в соответствии с правилами совместимости схемы. Это двухэтапный процесс, который значительно снижает вероятность того, что непреднамеренные изменения попадут в производство.

Запись данных о событиях в хранилища данных

Запись данных из потоков событий состоит в потреблении событийных данных и вставке их в хранилище данных. Это реализуется либо централизованным фреймворком, либо автономным микросервисом. В хранилище данных может быть помещен любой тип событийных данных, будь то сущность, события с ключом или события без ключа.

Запись данных о событиях в хранилища данных особенно полезна для интеграции приложений, не управляемых событиями, с потоками событий. Процесс-приемник считывает потоки событий из брокера событий и вставляет данные в указанное хранилище данных. Он отслеживает свои собственные смещения потребления и записывает данные о событиях по мере их поступления на вход, действуя полностью независимо от приложения, не управляемого событиями.

Типичным применением записи событий является замена соединений типа «точка-точка» между унаследованными системами. Как только данные системы-источника высвобождаются в потоки событий, они могут быть записаны в принимающую систему с несколькими другими изменениями. Процесс записи работает как внешне, так и невидимо по отношению к принимающей системе.

Запись данных также часто используется командами, которым необходимо выполнять пакетный анализ больших данных (big data). Обычно они делают это, загружая

данные в распределенную файловую систему Hadoop (Hadoop Distributed File System), которая предоставляет инструменты анализа больших данных.

Использование общей платформы, такой как Kafka Connect, позволяет вам указывать приемники с простыми конфигурациями и выполнять их в совместной инфраструктуре. Альтернативное решение предоставляют автономные приемники с поддержкой микросервисов. Разработчики могут создавать и выполнять их на платформе микросервисов и управлять ими независимо.

Влияние на бизнес записи данных в приемники и получения их из источников

Централизованный фреймворк позволяет выполнять процессы с меньшими нагрузками для освобождения данных. Такой фреймворк может работать в масштабе одной команды, которая, в свою очередь, поддерживает потребности в освобождении данных других команд по всей организации. Стремящиеся к интеграции команды должны заботиться только о конфигурации и дизайне коннектора, а не о каких-либо оперативных обязанностях. Этот подход лучше всего работает в крупных организациях, где данные хранятся в многочисленных хранилищах в многочисленных командах, поскольку он позволяет быстро начинать освобождение данных без необходимости каждой команде создавать свое собственное решение.

Но при использовании централизованного фреймворка вы можете попасть в две ловушки. Во-первых, обязанности по получению данных из источников и их записи в приемники теперь распределены между командами. Команда, управляющая централизованным фреймворком, отвечает за стабильность, масштабирование и работоспособность как фреймворка, так и каждого экземпляра коннектора. Между тем команда, управляющая захватываемой системой, независима и может принимать решения, которые изменяют производительность и стабильность коннектора, — такие как добавление и удаление полей или изменение логики, влияющие на объем данных, передаваемых через коннектор. В результате между этими двумя командами образуется прямая зависимость. Производимые изменения могут нарушить работу коннекторов, но они могут быть обнаружены только командой, которая управляет коннекторами, что приводит к увеличению зависимости между командами. По мере роста числа изменений процесс становится трудноуправляемым.

Вторая проблема более распространена, особенно в организациях, где принципы, основанные на событиях, приняты лишь частично. Системы могут стать слишком зависимыми от фреймворков и коннекторов, чтобы выполнять за них работу, управляемую событиями. Как только данные будут освобождены из внутренних хранилищ состояний и опубликованы в потоках событий, организация может успокоиться и перейти к микросервисам. При этом команды могут чрезмерно полагаться на платформу коннекторов для получения и приема данных и отказаться от рефакторинга своих приложений в собственные приложения, управляемые событиями. В таком сценарии они вместо этого предпочтут просто запрашивать новые

источники и приемники по мере необходимости, оставляя все свое базовое приложение полностью игнорирующими события.



Инструменты CDC не являются главной целью при переходе к архитектуре, управляемой событиями, но вместо этого в первую очередь предназначены для помощи в начальной загрузке (bootstrap) процесса. Реальная ценность брокера событий как слоя передачи данных заключается в предоставлении надежного и правдивого источника данных о событиях, отделенного от слоев реализации, и брокер хороши лишь настолько, насколько высоки качество и надежность его данных.

Обе эти проблемы могут быть смягчены благодаря правильному пониманию роли фреймворка захвата изменений в данных (Change-Data Capture, CDC). Возможно, это противоречит здравому смыслу, но важно минимизировать использование фреймворка CDC и побудить команды реализовывать свой собственный захват изменений в данных (например, шаблон исходящих таблиц), несмотря на дополнительную предварительную работу, которая может потребоваться. При этом команды становятся единолично ответственными за публикацию и события своей системы, устранив зависимости между командами и хрупкие CDC на основе коннекторов. В результате работы, которую должна выполнять команда фреймворка CDC, сводится к минимуму, что позволяет ей сосредоточиться на поддержке продуктов, которые действительно в этом нуждаются.

Уменьшение зависимости от фреймворка CDC также продвигает мировоззрение «сначала событие». Вместо того чтобы думать о потоках событий как о способе перераспределения данных между монолитами, вы рассматриваете каждую систему как непосредственного публикатора и потребителя событий, присоединяясь к событийно-управляемой экосистеме. Став активным участником экосистемы EDM, вы начинаете думать о том, *когда и как* система должна производить события, о данных *там*, а не только о данных *тут*. Это важная часть культурного сдвига к успешному внедрению EDM.

Для продуктов с ограниченными ресурсами и тех, которые эксплуатируются только для технического сопровождения, централизованная система коннекторов источника с приемником может быть значительным благом. Для других продуктов, в особенности тех, которые являются более сложными, имеют значительные требования к потоку событий и находятся в стадии активной разработки, постоянное техническое сопровождение и поддержка коннекторов представляются неустойчивыми. В этих обстоятельствах лучше всего планировать время для выполнения переработки кодовой базы по мере необходимости, чтобы приложение стало действительно полностью событийно-управляемым.

Наконец, внимательно подумайте о достоинствах и недостатках каждой из стратегий CDC. Они часто становятся предметом дискуссий и споров внутри организации, поскольку команды пытаются выяснить свои новые обязанности и границы в отношении производства своих событий как единственного источника истины. Переход к событийно-управляемой архитектуре требует инвестиций в слой обмена данными, и полезность этого слоя будет столь же хорошей, что и качество данных

внутри него. Каждый сотрудник в организации должен изменить свое мышление, чтобы учесть влияние своих освобожденных данных на остальную часть организации и прийти к четким соглашениям об уровне обслуживания относительно схем, моделей данных, упорядоченности, задержки и правильности для событий, которые они представляют.

Резюме

Освобождение данных является важным шагом на пути к созданию зрелого и доступного слоя обмена данными. Унаследованные системы часто содержат основную часть данных стержневых бизнес-моделей предметной области, которая хранится в той или иной форме централизованной структуры обмена реализацией информацией. Такие данные должны быть высвобождены из этих унаследованных систем, чтобы позволить другим областям организации создавать новые, несцепленные продукты и услуги.

Для извлечения данных хранилищ и их трансформации существует ряд фреймворков, инструментов и стратегий. Все они имеют свои преимущества, недостатки и компромиссы. Ваши варианты их использования будут влиять на то, какие возможности вы выберете, либо вы обнаружите, что должны создать свои собственные механизмы и процессы.

Цель освобождения данных состоит в том, чтобы обеспечить чистый и согласованный единый источник истины для данных, важных для организации. Доступ к данным отделен от их производства и хранения, что устраняет необходимость в структуре обмена информацией, накладывающей дополнительные обязанности. Это простое действие сокращает границы доступа к важным данным предметной области из многочисленных реализаций унаследованных систем и непосредственно способствует разработке новых продуктов и услуг.

Существует полный спектр стратегий освобождения данных. Так, предусмотрен подход на основе тщательной интеграции с исходной системой, где события передаются брокеру событий по мере их записи в хранилище данных. Некоторые системы даже способны сначала производить поток событий, прежде чем потреблять его обратно для своих собственных нужд, тем самым обеспечивая работу потока событий как единственного источника истины. Производитель осознает свою роль первоисточника данных и устанавливает защиту для предотвращения непреднамеренных разрушительных изменений. Производители стремятся работать с потребителями, чтобы обеспечить качественный, четко определенный поток данных, минимизировать разрушительные изменения и обеспечить совместимость изменений в системе со схемами событий, которые они производят.

Существуют также высокореактивные стратегии. Владельцы исходных данных при этом практически не имеют представления о производстве данных в брокер событий. Они полностью полагаются на фреймворки, которые либо извлекают данные непосредственно из своих внутренних наборов данных, либо выполняют раз-

бор журналов захвата изменений в данных. Здесь часто встречаются несоответствующие схемы, нарушающие работу потребителей, так же как и раскрывающие наружу внутренние модели данных из исходной реализации. Эта модель в долгосрочной перспективе не является устойчивой, поскольку пренебрегает обязанностью владельца данных по обеспечению чистого и согласованного производства событий предметной области.

Культура организации обуславливает то, насколько успешными будут инициативы по освобождению данных при переходе к событийно-управляемой архитектуре. Владельцы данных должны серьезно относиться к необходимости создания чистых и надежных потоков событий и понимать, что наличия механизмов захвата данных недостаточно для организации перехода к архитектуре, управляемой событиями.

Основы событийно-управляемой обработки

Большинство событийно-управляемых микросервисов выполняют как минимум одни и те же три шага:

1. Потребить событие из входного потока событий.
2. Обработать это событие.
3. Произвести все необходимые выходные события.

Существуют также событийно-управляемые микросервисы, которые выводят свое входное событие из синхронного взаимодействия «запрос-ответ», подробно рассмотренного в главе 13. Эта же глава посвящена только микросервисам, источником событий которых являются потоки событий.

В событийно-управляемых микросервисах с потоками событий в качестве источника экземпляр микросервиса создает клиента-производителя и клиента-потребителя и регистрирует себя в любых необходимых группах потребителей, если это необходимо. Микросервис запускает цикл для опроса клиента-потребителя на наличие новых событий, обрабатывая их по мере поступления и создавая все необходимые выходные события. Этот рабочий процесс показан в следующем далее псевдокоде (ваша реализация, конечно, будет отличаться от него в зависимости от языка, фреймворка потоковой обработки, выбора брокера событий и других технических факторов):

```
Consumer consumerClient = new consumerClient(consumerGroupName, ...);
Producer producerClient = new producerClient(...);

while(true) {
    InputEvent event = consumerClient.pollOneEvent(inputStream);
    OutputEvent output = processEvent(event);
    producerClient.produceEventToStream(outputEventStream, output);

    //по меньшей мере однократная обработка.
    consumerClient.commitOffsets();
}
```

Особый интерес представляет функция `processEvent`. Именно здесь происходит реальная обработка событий — в первую очередь применение бизнес-логики и определение того, какие события, если таковые имеются, следует создавать. Этую обрабатывающую функцию лучше всего рассматривать как точку входа в обраба-

тывающую топологию микросервиса. Отсюда управляемые данными шаблоны трансформируют и обрабатывают данные в соответствии с бизнес-требованиями вашего ограниченного контекста.

Топологии без поддержки состояния

Построение топологии микросервисов требует событийного мышления, поскольку код выполняется в ответ на событие, поступающее на вход потребителя. Топология микросервисов — это, по сути, последовательность операций, выполняемых над событием. Она требует выбора необходимых фильтров, маршрутизаторов, трансформаций, материализаций, агрегаций и других функций, необходимых для выполнения необходимой бизнес-логики микросервиса. Те, кто знаком с функциональным программированием и фреймворками обработки big data в стиле MapReduce¹, могут чувствовать себя здесь как дома. Для других эта концепция может оказаться несколько в новинку.

Рассмотрим топологию, представленную на рис. 5.1. События потребляются по одному и обрабатываются на этапах 1 и 2.

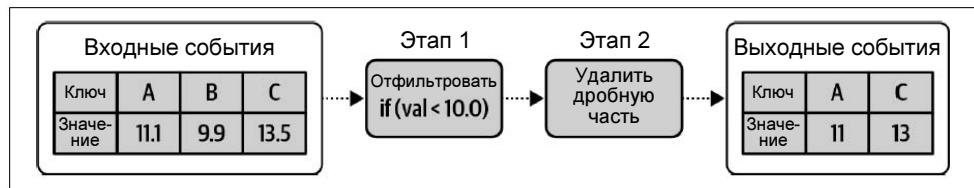


Рис. 5.1. Простая топология обработки событий

События с ключами А и С проходят через всю топологию. Они оба больше 10.0, что позволяет им пройти этап 1, а этап 2 просто отбрасывает дробную часть из значения события. Однако событие по ключу В отфильтровывается и отбрасывается, поскольку оно не соответствует критерию этапа 1.

Преобразования

Преобразование обрабатывает одно событие и генерирует ноль или более выходных событий. Преобразования, как вы могли догадаться, обеспечивают большую часть операций бизнес-логики, требующих преобразований. В зависимости от операций может возникнуть необходимость в *переподразделении* (repartitioning) событий (подробнее об этом рассказано далее). Часто встречающиеся преобразования включают в себя следующие, но не ограничиваются ими:

¹ MapReduce — модель распределенных вычислений от компании Google, используемая для параллельных вычислений над очень большими, вплоть до нескольких петабайт, наборами данных в компьютерных кластерах, а также соответствующий фреймворк для вычисления некоторых наборов распределенных задач с использованием большого количества компьютеров, образующих кластер. — Прим. ред.

◆ *Фильтр.*

Распространить событие, если оно соответствует необходимым критериям. Создает ноль или одно событие.

◆ *Операция Map.*

Изменить ключ и/или значение события, создавая ровно одно событие. Обратите внимание, что при изменении ключа может потребоваться переподразделение, чтобы обеспечить локальность данных в пределах раздела.

◆ *Операция mapValue.*

Изменить только значение события, но не ключ. Создает ровно одно событие. Переподразделение не требуется.

◆ *Индивидуально настраиваемое преобразование.*

Применить индивидуально настраиваемую логику, просмотреть состояние и даже синхронно обменяться информацией с другими системами.

Ветвление и слияние потоков

Приложению может потребоваться *ветвление* потоков событий, т. е. применение к событию логического оператора, а затем передача его в новый поток на основе результата. Одним из относительно распространенных сценариев является использование событийного «пожарного шланга» и принятие решения о том, куда его направлять, исходя из конкретных свойств (например, страны, часового пояса, происхождения, продукта или любого числа признаков). Второй распространенный сценарий заключается в передаче результатов в другие выходные потоки событий — например, вывод событий в поток недоставленных сообщений в случае ошибки обработки вместо того, чтобы полностью их отбрасывать.

Приложения также могут нуждаться в *слиянии* потоков, где события из многочисленных входных потоков потребляются, возможно, обрабатываются каким-то определенным образом, а затем выводятся в один выходной поток. Существует не так уж много сценариев, в которых важно объединить несколько потоков в один, поскольку микросервисы обычно объединяют столько входных потоков, сколько необходимо для реализации их бизнес-логики. Тема обработки потребляющих и обрабатывающих событий из многочисленных входных потоков в согласованном и воспроизводимом порядке обсуждается в главе 6.



Если вы все-таки в итоге объединяете потоки событий, то определите новую унифицированную схему, представляющую предметную область объединенного потока событий. Если эта область не имеет смысла, то, возможно, лучше оставить потоки не объединенными и пересмотреть дизайн вашей системы.

Переподразделение потоков событий

Потоки событий разделяются на разделы (partition) в соответствии с ключом события и логикой распределителя (event partitioner) событий. Для каждого события применяется алгоритм распределения событий и выбирается раздел, в который за-

писывается событие. *Переподразделение* (repartitioning) — это процесс производства нового потока событий с одним или несколькими из следующих свойств:

◆ *Другое количество разделов.*

Увеличивается количество разделов потока событий, чтобы увеличить нижестоящее распараллеливание или обеспечить соответствие числу разделов другого потока для совместного переподразделения (рассматривается далее в этой главе).

◆ *Другие ключи событий.*

Изменяется ключ события с целью обеспечения маршрутизации событий с одним и тем же ключом в один и тот же раздел.

◆ *Другой разделитель событий.*

Изменяется логика, используемая для выбора раздела, в который событие будет записано.

Редко бывает, чтобы обработчику, не поддерживающему сохранение состояния, потребовалось переподразделить поток событий, — за исключением случая увеличения количества разделов для обеспечения соответствия нижестоящему распараллеливанию. Тем не менее микросервис без поддержки состояния может использоваться для переподразделения событий, которые потребляются нижестоящим обработчиком с поддержкой состояния, что показано в следующем примере.



Алгоритм распределителя по заданному алгоритму соотносит ключ события с конкретным разделом, как правило, с помощью хеш-функции. Благодаря этому обеспечивается, что все события с одним и тем же ключом окажутся в одном и том же разделе.

Пример переподразделения потока событий

Предположим, существует поток пользовательских данных, который поступает из конечной точки, подключенной к Интернету. Действия пользователя преобразуются в события, причем данные этих событий содержат как ID пользователя, так и другие произвольные событийные данные, помеченные как `x`.

Потребители этого события заинтересованы в том, чтобы все данные, принадлежащие конкретному пользователю, содержались в *одном* разделе, независимо от того, как подразделен исходный поток событий. Для этого исходный поток можно переподразделить, как показано на рис. 5.2.

Перенос всех событий для каждого ключа в один раздел обеспечивает основу для *локализации данных*. Потребителю нужно только потреблять события из одного раздела, чтобы строить полную картину событий, относящихся к этому ключу. Такой подход позволяет потребительским микросервисам масштабироваться вверх до большого числа экземпляров, каждый из которых потребляет события из одного раздела, поддерживая одновременно полный учет состояния всех событий, относящихся к этому ключу. Переподразделение и локализация данных являются существенными частями выполнения их крупномасштабной обработки с поддержкой состояния.

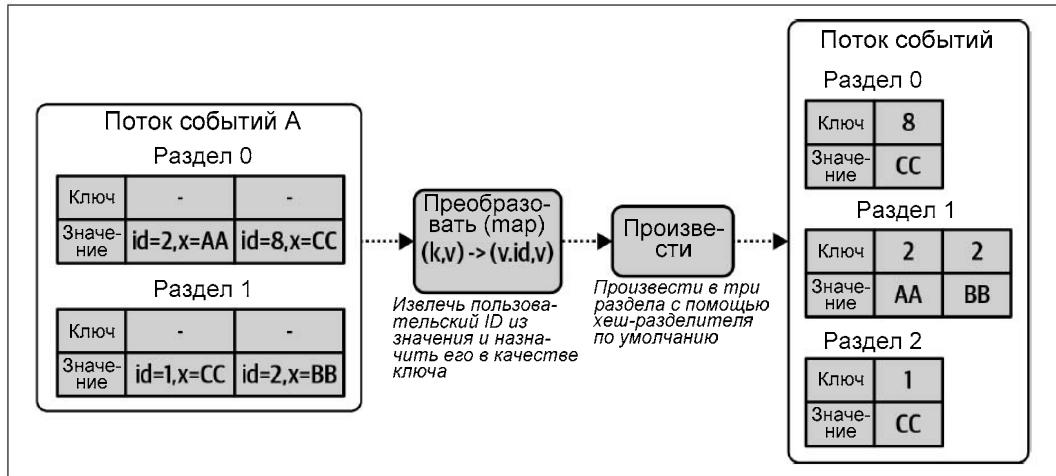


Рис. 5.2. Переподразделение потока событий

Соподразделение потоков событий

Соподразделение (copartitioning) — это переподразделение потока событий в новый поток с тем же количеством разделов и логикой закрепления разделов, что и другой поток. Оно требуется, когда события с ключами из одного потока событий должны быть расположены вместе (для обеспечения локализации данных) с событиями другого потока. Эта концепция важна для обработки потока с поддержкой состояния, поскольку многочисленные операции с поддержкой состояния (такие как потоковые соединения) требуют, чтобы все события для заданного ключа, независимо от того, из какого потока они происходят, обрабатывались через один и тот же узел. Подробнее об этом рассказано в главе 7.

Пример соподразделения потока событий

Рассмотрим еще раз пример переподразделения из рис. 5.2. Предположим, что теперь вам нужно соединить переподразделенный поток пользовательских событий с потоком пользовательских сущностей, доступ к которому осуществляется через тот же самый ID. Такие соединения этих потоков показаны на рис. 5.3.

Оба потока имеют одинаковое количество разделов, и оба были подразделены с использованием одного и того же алгоритма разделения. Обратите внимание, что распределение ключей каждого раздела совпадает с распределением ключей другого потока и что каждое соединение выполняется собственным экземпляром потребителя. В следующем разделе этой главы описываются способы, с помощью которых разделы потоков назначаются экземпляру микросервиса для использования соподразделенных потоков, как это было сделано в приведенном примере соединения.

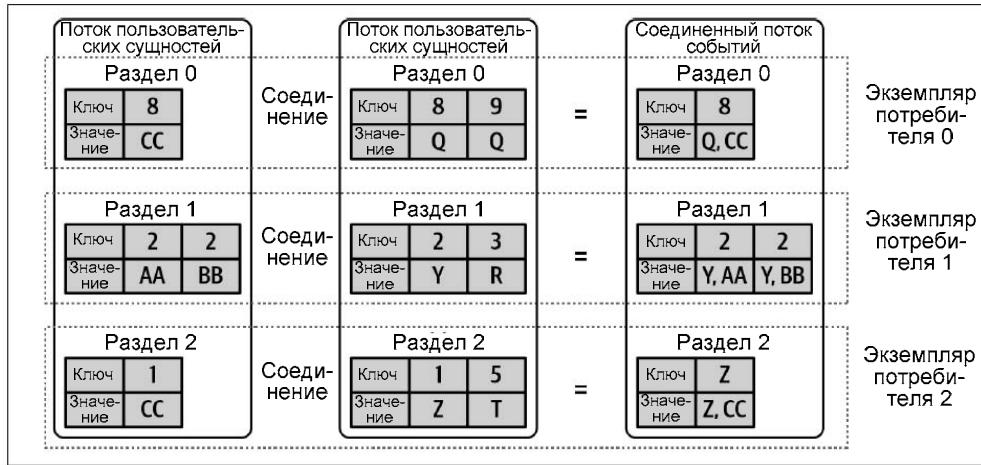


Рис. 5.3. Соподразделенные потоки пользовательских событий и пользовательских сущностей

Назначение разделов экземпляру потребителя

Каждый микросервис поддерживает свою собственную уникальную группу потребителей, представляющую совокупные смещения ее входных потоков событий. Первый экземпляр потребителя, который появится в сети, регистрируется в брокере событий, используя имя своей группы потребителей. После регистрации экземпляру потребителя необходимо будет назначить разделы.

Некоторые брокеры событий, такие как Apache Kafka, делегируют назначение раздела первому онлайн-клиенту для каждой группы потребителей. Как лидер группы потребителей этот экземпляр отвечает за выполнение обязанностей по назначению разделов, обеспечивая правильное назначение разделов входного потока событий всякий раз, когда новые экземпляры присоединяются к этой группе потребителей.

Другие брокеры событий, такие как Apache Pulsar, поддерживают централизованное владение назначением разделов внутри брокера. В этом случае назначение разделов и перебалансировка выполняются брокером, но механизм идентификации через группу потребителей остается прежним. Разделы назначаются, и работу можно начинать с последних известных смещений потребленных событий.

Работа на время переназначения разделов обычно на мгновение приостанавливается, чтобы избежать конфликтных ситуаций, связанных с переназначением. Это гарантирует, что любые отозванные разделы больше не будут обрабатываться другим экземпляром перед назначением новому экземпляру, что исключает любые возможности дублирования вывода.

Назначение разделов с помощью контроллера разделов

Для обработки больших объемов данных обычно требуется несколько экземпляров микросервиса-потребителя, будь то выделенный фреймворк потоковой обработки

или базовая реализация производителя/потребителя. Контроллер разделов гарантирует, что разделы распределяются между обрабатывающими экземплярами сбалансированным и справедливым образом.

Этот контроллер разделов также отвечает за переназначение разделов всякий раз, когда новые экземпляры потребителей добавляются или удаляются из группы потребителей. В зависимости от выбранного вами брокера событий этот компонент может быть встроен в клиент-потребитель или поддерживаться в брокере событий.

Назначение соподразделенных разделов

Контроллер разделов также несет ответственность за обеспечение выполнения любых требований к соподразделению. Все разделы, помеченные как соподразделяемые, должны быть назначены одному и тому же экземпляру потребителя. Это обеспечивает, что такому экземпляру микросервиса будет назначаться правильное подмножество событийных данных для выполнения его бизнес-логики. При этом рекомендуется проверять реализацию контроллера разделов, чтобы убедиться, что потоки событий имеют одинаковое количество разделов и генерируют исключение при неравенстве.

Стратегии назначения разделов

Цель алгоритма назначения разделов — обеспечить равномерное распределение разделов между экземплярами-потребителями при условии, что экземпляры-потребители равны по возможностям обработки. Алгоритм назначения разделов также может иметь второстепенные цели — такие как уменьшение количества разделов, переназначенных во время перебалансировки. Это особенно важно, когда вы имеете дело с материализованным состоянием, сегментированным по нескольким экземплярам хранилища данных, поскольку переназначение раздела может привести к тому, что будущие обновления попадут в неправильный сегмент. В главе 7 эта концепция более подробно рассматривается применительно к внутренним структурам хранилищ.

Существует ряд общих стратегий назначения разделов. Стратегия, используемая по умолчанию, может варьироваться в зависимости от вашего фреймворка или реализации, но следующие три, как правило, используются наиболее часто.

Назначение по круговой схеме

Все разделы сводятся в список и назначаются по круговой схеме каждому экземпляру потребителя. Отдельный список хранится для соподразделенных потоков, чтобы обеспечить их правильное назначение.

На рис. 5.4 показаны два экземпляра потребителя, каждый из которых имеет свой собственный набор назначенных разделов. У экземпляра C0 — два набора соподразделенных разделов по сравнению с одним у C1, т. к. назначение как началось, так и закончилось на C0.

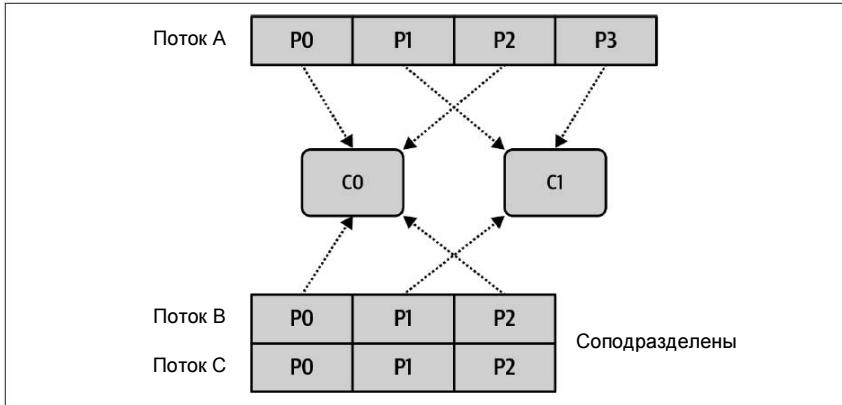


Рис. 5.4. Назначение разделов по круговой схеме для двух экземпляров потребителя

Когда число экземпляров потребителя для какой-либо группы потребителей увеличивается, назначения разделов должны быть перебалансированы, чтобы распределить нагрузку между вновь добавленными ресурсами. На рис. 5.5 показаны эффекты добавления еще двух экземпляров потребителя.

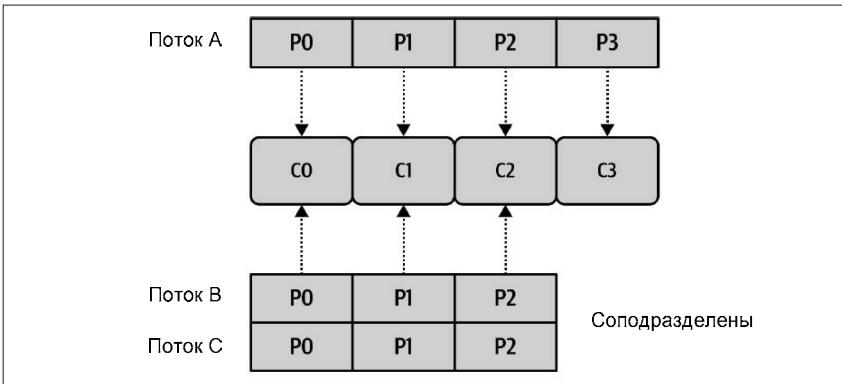


Рис. 5.5. Назначение разделов по круговой схеме для четырех экземпляров потребителя

Экземпляру C2 теперь назначаются соподразделенные разделы P2 потоков B и C, а также раздел P2 потока A. Экземпляр C3, с другой стороны, имеет только раздел P3 из потока A, потому что для назначения нет никаких дополнительных разделов. Добавление каких-либо дополнительных экземпляров не приведет к дополнительной параллелизации.

Статическое назначение

Протоколы статического назначения могут использоваться, когда определенные разделы должны быть назначены конкретным потребителям. Эта опция наиболее полезна, когда большие объемы данных с отслеживанием состояния материализуются в любом конкретном экземпляре, что обычно для внутренних хранилищ со-

стояний. Когда экземпляр потребителя покидает группу потребителей, статический контроллер не станет переназначать разделы, а вместо этого будет ждать, пока отсутствующий экземпляр потребителя не вернется в оперативный режим. В зависимости от реализации разделы могут быть динамически переназначены в любом случае, если исходный потребитель не сможет повторно присоединиться к группе потребителей в течение назначенного периода времени.

Индивидуально настраиваемое назначение

Используя внешние сигналы и инструменты, пользовательские задания можно адаптировать к потребностям клиента. Например, назначение может быть основано на текущей задержке во входных потоках событий, обеспечивая равное распределение работы по всем вашим экземплярам-потребителям.

Восстановление после отказов экземпляра обработчика без поддержки состояния

Восстановление после отказов без поддержки состояния фактически равносильно простому добавлению нового экземпляра в группу потребителей. Обработчики без поддержки состояния не требуют восстановления состояния, из чего следует, что они могут немедленно вернуться к обработке событий, как только им назначены разделы и установлено их потоковое время.

Резюме

Базовые событийно-управляемые микросервисы без поддержки состояния потребляют события, обрабатывают их и создают любые новые последующие события. Каждое событие обрабатывается независимо от других. Базовые преобразования позволяют изменять события в нужные форматы, которые затем можно переподразделять на новый поток событий с новым количеством разделов. Потоки событий с одним и тем же ключом, одним и тем же алгоритмом подразделения и одним и тем же количеством разделов называются *соподразделенными*, что гарантирует локализацию данных для заданного экземпляра потребителя. Контроллер разделов используется для обеспечения равномерного распределения разделов между экземплярами потребителя и правильного назначения соподразделенных потоков событий.

Соподразделение и назначение разделов являются важными понятиями для понимания обработки с поддержкой состояния, которая рассматривается в главе 7. Однако сначала вы должны подумать о том, как регулировать обработку многочисленных разделов из многочисленных потоков событий. Неупорядоченные (*out-of-order*) события, запоздалые (*late*) события и порядок, в котором события выбираются для обработки, оказывают значительное влияние на дизайн ваших сервисов. Рассмотрение этих вопросов и станет темой следующей главы.

Детерминированная обработка потоков

Событийно-управляемые микросервисы обычно имеют более сложные топологии, чем те, которые были представлены в предыдущей главе. События потребляются и обрабатываются из многочисленных потоков событий, в то время как для решения многих бизнес-задач необходима обработка с поддержкой состояния (рассмотренная в следующей главе). Микросервисы также подвержены тем же ошибкам и отказам, что и системы, не использующие микросервисы. И нет ничего необычного в том, что некоторые микросервисы обрабатывают события почти в реальном времени, в то время как другие, недавно запущенные микросервисы, их догоняют, обрабатывая отставшие данные.

Вот три главных вопроса, рассмотренных в этой главе:

- ◆ как микросервис выбирает порядок обработки событий при потреблении их из нескольких разделов;
- ◆ как микросервис обрабатывает неупорядоченные и запоздалые события;
- ◆ как нам обеспечить, чтобы наши микросервисы давали детерминированные¹ результаты при обработке потоков в почти реальном времени по сравнению с обработкой с самого начала потоков?

Мы можем ответить на эти вопросы, изучив временные² метки (timestamps), планирование событий, водяные знаки (watermarks) и время потока (stream times), а также их вклад в детерминированную обработку. Кроме того, отказы, ошибки и изменения в бизнес-логике приводят к необходимости повторной обработки, выдвигая на первый план важность получения детерминированных результатов. В этой главе также исследуется вопрос о том, как могут происходить неупорядоченные и опоздавшие события, рассматриваются стратегии их обработки и способы смягчения их влияния на наши рабочие процессы.



Эта глава весьма насыщена информацией, несмотря на все мои усилия найти простой и лаконичный способ объяснить ключевые понятия. В некоторых разделах я буду отсылать вас к сторонним ресурсам для самостоятельного изучения, поскольку рассматриваемые в них моменты часто выходят за рамки этой книги.

¹ Детерминизм — учение о взаимосвязи и взаимной определенности всех явлений и процессов, доктрина о всеобщей причинности. — Прим. ред.

² Далее в тексте — всюду «временные», если специально не указано иное. — Прим. ред.

Детерминизм событийно-управляемых рабочих процессов

Событийно-управляемый микросервис имеет два главных обрабатывающих состояния. Он может обрабатывать события в режиме, близком к реальному времени, что характерно для длительно работающих микросервисов. С другой стороны, он может обрабатывать события из прошлого в попытке догнать текущее время, что является общим для не полностью масштабированных и новых сервисов.

Если у группы потребителей переместить смещения входных потоков событий назад к стартовым событиям и снова запустили микросервис, будет ли он генерировать тот же результат, что и при первом запуске? Главная цель детерминированной обработки состоит в том, чтобы микросервис производил один и тот же результат, независимо от того, обрабатывает ли он данные в реальном времени или работает с отставшими данными.

Обратите внимание, что существуют явно недетерминированные рабочие процессы — основанные, например, на текущем физическом времени, и те, что запрашивают внешние сервисы. Внешние сервисы могут предоставлять разные результаты в зависимости от того, когда они запрашиваются, в особенности если их внутреннее состояние обновляется независимо от состояния сервисов, выдающих запрос. В этих случаях детерминированность не обещается, поэтому обязательно обратите внимание на любые неопределенные операции в вашем рабочем процессе.

Полностью детерминированная обработка является идеальным случаем, когда каждое событие приходит вовремя и нет задержек, отказов производителя или потребителя, а также проблем с разрывом сетевого соединения. Поскольку у нас нет иного выбора, кроме как иметь дело с подобными нарушениями идеальных сценариев, реальность такова, что наши сервисы могут лишь приблизиться к детерминированности в попытке ее достичь. Существует ряд компонентов и процессов, которые работают вместе, чтобы облегчить эту попытку, и в большинстве случаев так называемого детерминизма максимальных усилий, т. е. попытки достичь детерминизма, будет достаточно, чтобы удовлетворить ваши требования. Для достижения этой цели вам потребуется несколько компонентов: согласованные временные метки, хорошо подобранные ключи событий, назначение разделов, планирование событий и стратегии обработки опаздывающих событий.

Временные метки

События могут происходить в любом месте и в любое время и часто должны быть согласованы с событиями других производителей. Поэтому наличие синхронизированных и согласованных *временных меток* является жестким требованием для сравнения событий в распределенных системах.

Событие, хранящееся в потоке событий, имеет как смещение, так и временную метку. *Смещение* используется потребителем для определения того, какие события он уже прочитал, в то время как *временная метка*, указывающая на то, когда это

событие было создано, служит для определения момента, когда событие произошло относительно других событий, и для обеспечения того, чтобы события обрабатывались в правильном порядке.

Следующие далее понятия, связанные с временными метками, проиллюстрированы на рис. 6.1, где показано их временное положение в событийно-управляемом рабочем процесссе:

◆ *Время события.*

Локальная временная метка, назначенная событию производителем в момент его возникновения.

◆ *Время получения брокером (broker ingestion time).*

Метка времени, назначаемая событию брокером событий. Вы можете настроить ее и как время события, и как время получения, причем первое встречается гораздо чаще. В сценариях, где время события со стороны производителя ненадежно, время получения брокером может обеспечить достаточную замену.

◆ *Время получения потребителем.*

Время, когда событие считывается потребителем. Это может быть время события, указанное в брокерской регистрационной записи, или физическое время.

◆ *Время обработки.*

Физическое время, в которое событие было обработано потребителем.

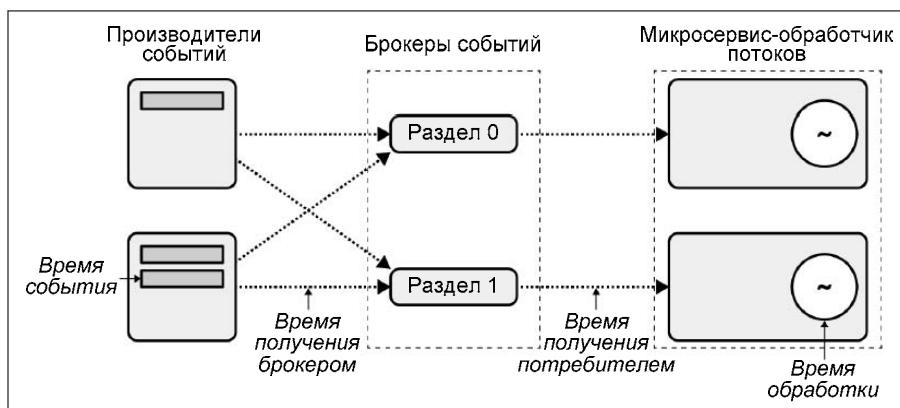


Рис. 6.1. Планировщик событий упорядочивает входные события по временной метке

Вы видите, что можно распространить время события через брокер событий на потребителя, позволяя логике потребителя принимать решения на основе того, когда событие произошло. Это поможет ответить на три вопроса, поставленные в начале главы. Теперь, когда мы наметили типы временных меток, давайте посмотрим, как они генерируются.

Синхронизация распределенных временных меток

Фундаментальное ограничение физики состоит в том, что нельзя гарантировать, что две независимые системы имеют строго одинаковое время системных часов. Различные физические свойства ограничивают то, насколько точными могут быть системные часы, — это и их предельные размеры в зависимости от материала в базовой тактовой схеме, изменения рабочей температуры чипа и несогласованные сетевые задержки связи во время синхронизации. Однако можно установить локальные системные часы, которые *почти* синхронизированы и в конечном счете достаточно хороши для большинства вычислительных целей.

Согласованные времена часов в основном достигаются за счет синхронизации с серверами *сетевого протокола времени* (Network Time Protocol, NTP). Поставщики облачных сервисов, такие как Amazon и Google, предлагают в своих различных регионах резервные спутниковые и атомные часы для мгновенной синхронизации.

Синхронизация с NTP-серверами в локальной сети может обеспечивать очень точные локальные системные часы с дрейфом всего в несколько миллисекунд по прошествии 15 минут (<https://oreil.ly/J6Zmo>). По словам Дэвида Миллса (David Mills), изобретателя NTP, в лучшем случае этот показатель можно уменьшить до 1 мс или меньше при более частых синхронизациях (<https://oreil.ly/oLEEs>), хотя кратковременные проблемы с сетью могут помешать достижению этой цели на практике. Синхронизация через открытый Интернет может привести к гораздо большим перекосам, когда точность снижается до диапазона ± 100 мс, и этот фактор следует учитывать, если вы пытаетесь синхронизировать события из разных областей земного шара.

NTP-синхронизация тоже подвержена отказам, поскольку перебои в работе сети, неправильное ее конфигурирование и периодически возникающие проблемы могут помешать синхронизации экземпляров. Да и сами NTP-серверы также могут стать ненадежными или не отвечать на запросы. На часы в экземпляре могут влиять проблемы с несколькими арендаторами, как в системах на основе виртуальных машин, совместно использующих базовое оборудование.

Однако в подавляющем большинстве случаев частая синхронизация с NTP-серверами может обеспечить достаточную согласованность времени системных событий. Улучшения в работе NTP-серверов и GPS-навигаторов (<https://oreil.ly/lBHD1>) начали уверенно сдвигать точность NTP-синхронизации в субмиллисекундный диапазон. Значения времени создания и времени получения, назначенные в качестве временных меток, могут быть высоко согласованными, хотя незначительные проблемы с неупорядоченностью все равно будут возникать. Обработка запоздалых событий рассматривается далее в этой главе.

Обработка с помощью событий, помеченных временными метками

Временные метки обеспечивают обработку событий, распределенных по многочисленным потокам событий и разделам в согласованном по времени порядке. Многие

варианты использования требуют, чтобы вы поддерживали порядок между событиями на основе времени и нуждались в согласованных, воспроизведимых результатах независимо от того, когда поток событий обрабатывается. Использование смещений в качестве средства сравнения работает только для событий в пределах одного раздела потока событий, в то время как события достаточно часто должны обрабатываться из нескольких разных потоков событий.

Пример: выбор порядка событий при обработке многочисленных разделов

Банку требуется, чтобы потоки событий размещения и снятия денежных средств обрабатывались в правильном временном порядке. Он ведет подсчет с поддержкой состояния размещения и снятия денежных средств с депозитов, назначая штраф за овердрафт, когда баланс счета клиента падает ниже нуля долларов. У банка в этом примере размещение денежных средств на депозитах находится в одном потоке событий, а снятие денежных средств — в другом (рис. 6.2).

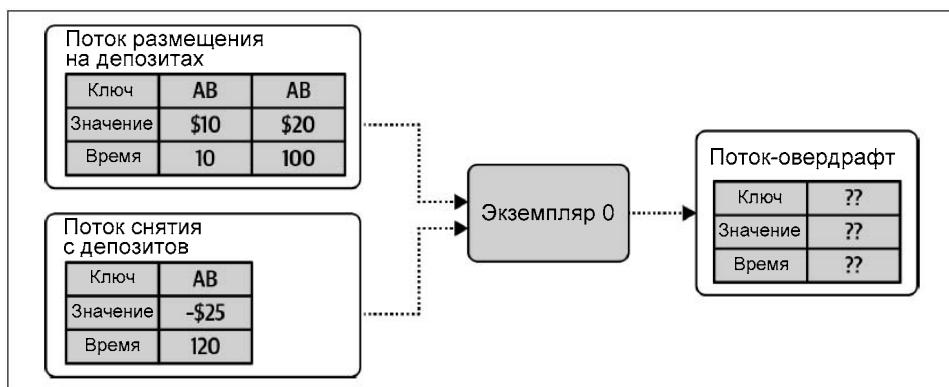


Рис. 6.2. В каком порядке следует обрабатывать события?

При лобовом подходе к потреблению и обработке записей обработчик, работающий по круговой схеме, мог бы в первую очередь обработать внесение на депозит в размере 10 долларов, затем провести вывод с депозита в размере 25 долларов (с отрицательным балансом и штрафами за овердрафт), после чего оформить внесение на депозит в размере 20 долларов. Однако это неверно и не отражает временной порядок, в котором события происходили. Приведенный пример ясно показывает, что при потреблении и обработке событий необходимо учитывать временную метку события. В следующем разделе это обсуждается подробнее.

Планирование событий и детерминированная обработка

Детерминированная обработка требует, чтобы события обрабатывались согласованно таким образом, чтобы результаты могли быть воспроизведены позже. *Планирование событий* — это процесс выбора следующих событий для обработки при

потреблении из многочисленных входных разделов. В случае неизменяемого потока событий на основе журнала записи потребляются в порядке смещения. Однако, как показано на рис. 6.2, чтобы обеспечить правильные результаты, порядок обработки событий должен формироваться на основе *времени события*, указанного в записи, независимо от того, из какого входного раздела оно поступает.



Наиболее распространенная реализация планирования событий выбирает и отправляет событие с самой старшей меткой времени из всех назначенных входных разделов в нижестоящую обрабатывающую топологию.

Планирование событий является особенностью многих фреймворков потоковой обработки, но обычно отсутствует в реализациях базовых потребителей. Вам нужно будет определиться, насколько это требуется для вашей реализации микросервиса.



Ваш микросервис будет нуждаться в планировании событий, если порядок, в котором события потребляются и обрабатываются, имеет значение для бизнес-логики.

Индивидуально настраиваемые планировщики событий

Некоторые потоковые фреймворки позволяют реализовать индивидуально настраиваемые планировщики событий. Например, Apache Samza позволяет использовать класс `MessageChooser`, где вы выбираете, какое событие обрабатывать, основываясь на ряде факторов — таких как приоритет некоторых потоков событий над другими, физическое время, время события, метаданные события и даже содержимое самого события. Однако при реализации собственного планировщика событий вам следует быть осторожными, т. к. многие индивидуально настраиваемые планировщики не детерминированы по своей природе и не смогут выдавать воспроизводимые результаты, если потребуется повторная обработка событий.

Обработка на основе времени события, времени обработки и времени получения

Основанный на времени порядок обработки событий требует, чтобы вы выбрали, какой момент времени будет использоваться в качестве временной метки события, как показано на рис. 6.1. Выбор делается между локально назначаемым временем события и временем получения брокером. В рабочем процессе «производство–потребление» обе временные метки встречаются только один раз, тогда как физическое время и время получения потребителем меняются в зависимости от того, когда приложение выполняется.

В большинстве сценариев, в особенности когда все потребители и все производители работают штатно и нет отставания событий для какой-либо группы потребителей, все четыре момента времени будут находиться в пределах нескольких секунд друг от друга. Напротив, в случае микросервисов, обрабатывающих отставшие

события, время события и время получения потребителем будут существенно различаться.

Для наиболее точного описания событий в реальном мире лучше всего использовать локально назначаемое время события *при условии, что вы можете положиться на его точность*. Если же временные метки производителя ненадежные (и вы не можете их исправить), то ваш следующий наилучший вариант выбора — устанавливать временные метки на основе того, когда события попадают в брокер событий. Только в редких случаях, когда брокер событий и производитель не могут обмениваться информацией, может возникнуть существенная задержка между истинным временем события и временем, назначенным брокером.

Извлечение временной метки потребителем

Потребитель должен знать временную метку записи, прежде чем он сможет решить, как заказать ее для обработки. Во время приема потребителем данных для извлечения метки времени из потребляемого события используется особое средство извлечения метки времени, называемое **экстрактором**. Этот экстрактор может извлекать информацию из любой части полезной нагрузки события, включая ключ, значение и метаданные.

Каждая потребляемая запись имеет назначенную временную метку события, установленную экстрактором. После того как такая временная метка установлена, она используется фреймворком потребителя на протяжении всей обработки.

Вызовы к внешним системам в форме «запрос-ответ»

Любые не событийно-управляемые запросы к внешним системам из управляемой событиями топологии могут привести к недетерминированным результатам. По определению внешние системы управляются извне по отношению к микросервису, что означает, что в любой момент их внутреннее состояние и их ответы на запрос микросервиса могут различаться. Насколько это важно, полностью зависит от бизнес-требований вашего микросервиса, и вам решать.

Водяные знаки

Водяные знаки (watermarks) используются для отслеживания времени события через топологию обработки и для объявления того, что все данные для указанного времени события (или более раннего) были обработаны. Этот технический прием обычно задействуется во многих ведущих фреймворках потоковой обработки, таких как Apache Spark, Apache Flink, Apache Samza и Apache Beam. Технический документ от Google (<https://oreil.ly/WO2OC>) описывает водяные знаки более подробно и содержит дополнительную информацию для тех, кто хотел бы узнать о них больше.

Водяной знак — это объявление нижестоящим узлам в пределах *одной и той же обрабатывающей топологии*, что все события времени t и предшествующие были

обработаны. Узел, получающий водяной знак, может затем обновить свое собственное внутреннее время события и распространить свой собственный водяной знак вниз по потоку к своим зависимым топологическим узлам. Этот процесс показан на рис. 6.3.

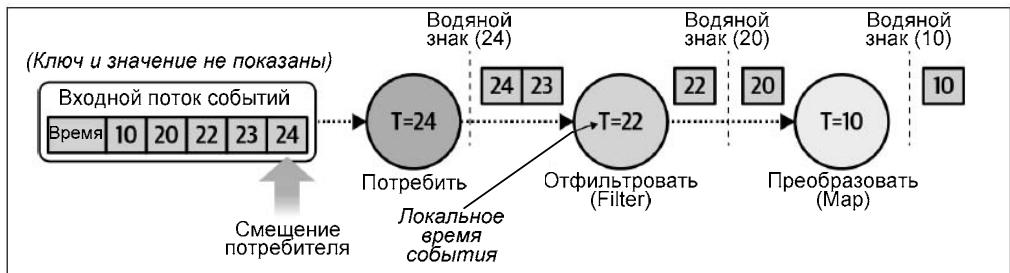


Рис. 6.3. Распространение водяных знаков между узлами в одной топологии

Здесь узел потребителя имеет наибольшее время водяного знака, поскольку он потребляет из входного потока событий. Новые водяные знаки генерируются периодически, например по истечении периода физического времени или времени события или после обработки некоторого минимального числа событий. Эти водяные знаки распространяются вниз по потоку к другим обрабатывающим узлам в топологии, которые соответственно обновляют свое собственное время события.



В этой главе мы лишь слегка касаемся темы водяных знаков, чтобы дать вам представление о том, как они используются для детерминированной обработки. Если вы хотите углубиться в тему водяных знаков, то обратитесь к главам 2 и 3 замечательной книги «Потоковые системы» Тайлера Акидау, Славы Черняка и Рувиша Лакса («Streaming Systems», Tyler Akidau, Slava Chernyak, Reuven Lax. O'Reilly, 2018).

Водяные знаки в параллельной обработке

Водяные знаки особенно полезны для координации времени событий между многочисленными независимыми экземплярами потребителей. На рис. 6.4 показана простая топология обработки двух экземпляров потребителя. Каждый экземпляр потребителя потребляет события из своего собственного назначенного ему раздела, применяя сначала функцию `groupByKey`, а затем агрегатную функцию `aggregate`. Это требует *перераспределения* (*shuffle*), когда все события с одним и тем же ключом отправляются в один нижестоящий агрегатный экземпляр. При этом события из экземпляра 0 и экземпляра 1 отправляются друг другу на основе ключа, чтобы гарантировать, что все события одного и того же ключа находятся в одном и том же разделе.

Чтобы пояснить, что демонстрирует приведенная на рис. 6.4 схема, давайте пройдемся по ней с самого начала.

Водяные знаки генерируются в исходной функции, где события потребляются из раздела потока событий. Водяные знаки определяют время события на этом узле

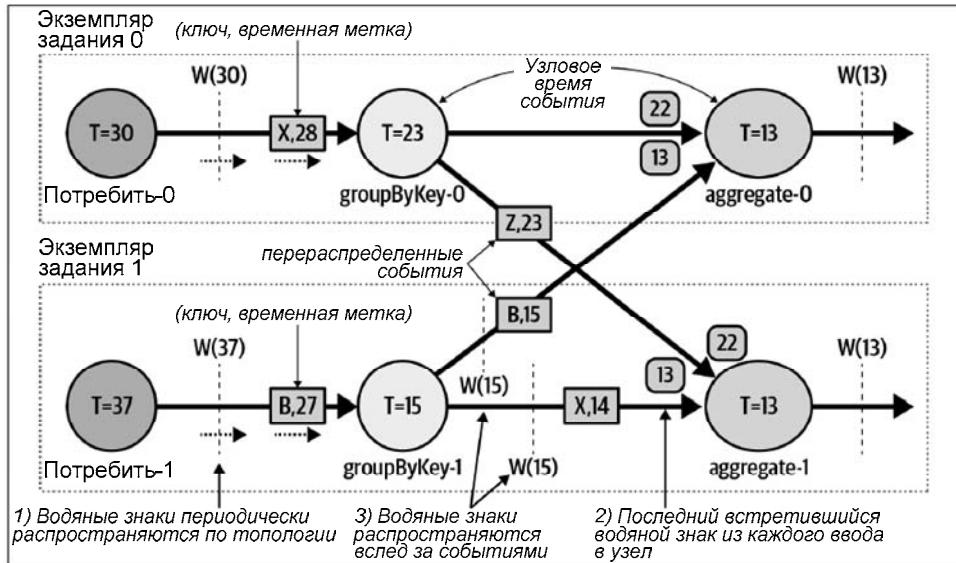


Рис. 6.4. Распространение водяных знаков между узлами в одной топологии с многочисленными обработчиками

и распространяются вниз по потоку по мере увеличения времени события в узле потребителя (см. запись 1 в нижней части схемы).

Нижестоящие узлы обновляют свое время события по мере поступления водяных знаков и, в свою очередь, генерируют свой собственный новый водяной знак для распространения вниз по потоку к его преемникам. Узлы с многочисленными входами, такие как `aggregate`, потребляют события и водяные знаки из многочисленных вышестоящих входов. Узловое время события — это *минимум* времени событий всех его входных источников, которое узел отслеживает внутри себя (см. запись 2).

В этом примере оба агрегатных узла `aggregate` обновят свое время события с 13 до 15, как только появится водяной знак из узла `groupByKey-1` (см. запись 3). Обратите внимание, что водяной знак не влияет на планирование событий в узле — он просто оповещает узел о том, что тот должен трактовать любые события с меткой времени раньше, чем водяной знак, как запоздалые. Обработка запоздалых событий рассматривается далее в этой главе.

Фреймворки Spark, Flink и Beam, наряду с другими тяжеловесными фреймворками обработки, требуют выделенного кластера обрабатывающих ресурсов для выполнения потоковой обработки в большом масштабе. Это особенно актуально, поскольку такой кластер также предоставляет средства для обмена информацией между задачами и централизованной координацией каждой задачи обработки. Для переподразделения событий, как, например, сделано в этом примере с помощью операции `groupByKey + aggregate`, вместо потоков событий в брокере событий используется внутрикластерный обмен информацией.

Время потока

Второй вариант поддержания времени в потоковом обработчике, именуемый просто *время потока*, — это подход, предлагаемый фреймворком Apache Kafka. Потребительское приложение, читающее данные из одного или нескольких потоков событий, поддерживает время потока для своей топологии — т. е. самую последнюю временную метку обработанных событий. Экземпляр потребителя собирает и буферизует события из каждого назначенного ему раздела потока событий, применяет алгоритм планирования событий для выбора следующего обрабатываемого события, а затем обновляет время потока, если оно больше, чем время предыдущего потока. Время потока никогда не уменьшается.

На рис. 6.5 показан пример времени потока. Узел потребителя поддерживает время одного потока на основе самого высокого значения времени события, которое он получил. Время потока в настоящее время установлено равным 20, т. к. это было время самого последнего обработанного события. Следующим обрабатываемым событием является наименьшее значение из двух входных буферов — в нашем случае это событие с временем события 30. Событие отправляется вниз к обрабатывающей топологии, и время потока будет обновлено до 30.

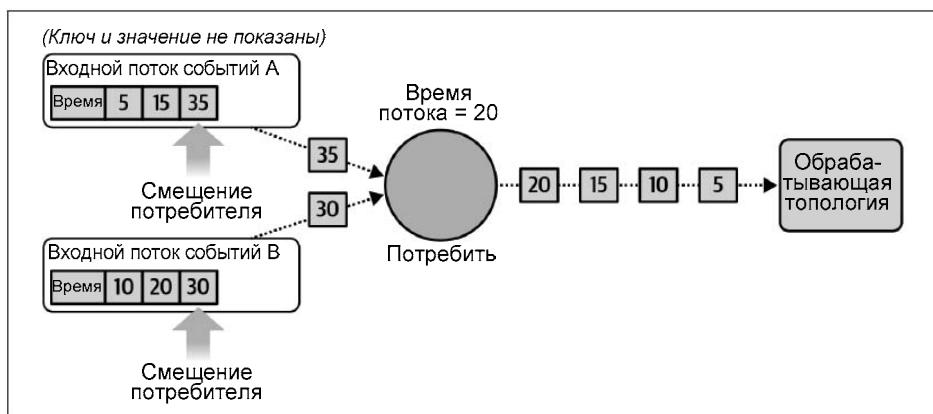


Рис. 6.5. Время потока при потреблении из многочисленных входных потоков

Время потока поддерживается за счет пропускания каждого события полностью через топологию перед обработкой следующего. В тех случаях, когда топология содержит переподразделяемый поток, каждая топология разделяется на две, и каждая субтопология сохраняет свое собственное время потока. События обрабатываются сначала в глубину, благодаря чему в любой момент времени в субтопологии обрабатывается только одно событие. Это отличается от подхода, основанного на водяных знаках, когда события могут буферизоваться на входах каждого обрабатывающего узла, причем время события в каждом узле обновляется независимо.

Время потока в параллельной обработке

Рассмотрим еще раз тот же пример потребителя с двумя экземплярами из рис. 6.4, но на этот раз с подходом на основе времени потока по технологии Kafka Streams (см. рис. 6.6). Заметным различием является то, что подход Kafka Streams отправляет репартицированные события *обратно* брокеру событий, используя то, что имеется *внутренним потоком событий*. Затем этот поток восстанавливается экземплярами, причем все репартицированные данные размещаются вместе по ключу внутри отдельных разделов. Функционально этот вариант похож на механизм перераспределения в тяжеловесном кластере, но не требует выделенного кластера (заметим для себя, что Kafka Streams очень удобен для микросервисов).

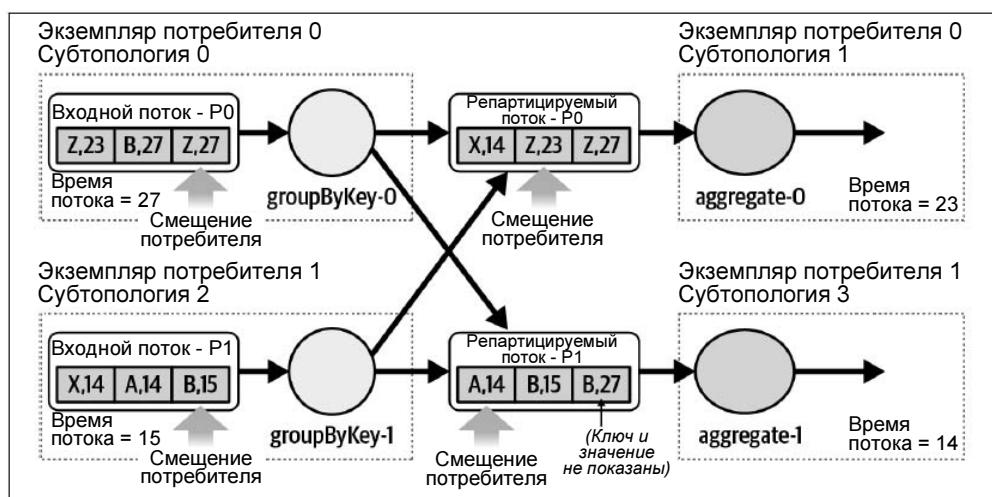


Рис. 6.6. Перераспределение событий посредством репартицируемого потока событий

В этом примере события из входного потока репартицируются в соответствии с их ключом и записываются в переподразделяемый поток событий. События, включенные в **A** и **B**, заканчиваются в потоке **P1**, тогда как события, включенные в **X** и **Z**, заканчиваются в потоке **P0**. Также обратите внимание, что время события поддерживается для каждого события и не перезаписывается текущим физическим временем. Напомним, что репартицирование должно трактоваться только как логическое перераспределение существующих событийных данных. Переписывание времени события полностью разрушило бы первоначальный временной порядок.

Обратите внимание на показанные здесь субтопологии. Из-за переподразделения потока событий обрабатывающая топология эффективно разрезается пополам, из чего следует, что работа над каждой субтопологией может выполняться параллельно. Субтопологии **1** и **3** потребляют из переподразделяемого потока и группируют события вместе, в то время как субтопологии **0** и **2** производят переподразделенные события. Каждая субтопология поддерживает свое собственное время потока, т. к. обе они состоят из независимых потоков событий.



Стратегии разметки водяными знаками также могут использовать переподразделяемые потоки событий. В Apache Samza предлагается автономный режим, похожий на тот, который используется в Kafka Streams, но там вместо времени потока используются водяные знаки.

Неупорядоченные и запоздалые события

В идеальном мире все события производятся без проблем и доступны потребителю с нулевой задержкой. К сожалению, для всех нас, живущих в реальном мире, это никогда не бывает так, поэтому мы должны планировать, чтобы приспособиться к неупорядоченным событиям. Событие считается *неупорядоченным* (out-of-order), если его временная метка не равна или больше событий, предшествующих ей в потоке событий. На рис. 6.7 событие **F** находится вне порядка, потому что его временная метка ниже, чем у **G**. Точно так же вне порядка находится событие **H**, поскольку его временная метка ниже, чем у **I**.



Рис. 6.7. Неупорядоченные события в разделе потока событий

Ограниченные наборы данных, такие как пакетно-обрабатываемые запоздалые данные, обычно весьма устойчивы к неупорядоченности данных. Весь пакет можно рассматривать как одно большое окно, и то, что событие прибывает вне порядка на много минут или даже часов, на самом деле не имеет особого значения, если обработка самого пакета еще не началась. Таким образом, ограниченный пакетно-обрабатываемый набор данных может давать результаты с высокой детерминированностью. Это обеспечивается за счет значительной задержки, в особенности для традиционных видовочных пакетных заданий обработки больших данных, где результаты доступны только после 24-часового периода плюс время пакетной обработки.

В случае неограниченных наборов данных, таких как постоянно обновляющиеся потоки событий, разработчик должен учитывать требования задержки и определенности в дизайне микросервиса. Это выходит за рамки технологических требований и переходит в бизнес-требования, поэтому любой разработчик событийно-управляемого микросервиса должен спросить: «обрабатывает ли мой микросервис неупорядоченные и запоздалые события в соответствии с бизнес-требованиями?» Неупорядоченные события требуют от предприятия принятия конкретных решений о том, как их обрабатывать, и определения того, являются ли задержки или детерминизм приоритетными.

Рассмотрим предыдущий пример банковского счета. Внесение на депозит, за которым следует немедленное снятие с депозита, должны — чтобы не был неправомерно применен штраф за овердрафт — обрабатываться в правильном порядке, независимо от порядка событий или того, насколько запоздалыми они могут быть. Чтобы смягчить это обстоятельство, логике приложения может потребоваться поддерживать состояние для обработки неупорядоченных данных в течение заданного предприятием периода времени — например, одн часового льготного окна.



События из одного раздела всегда должны обрабатываться в соответствии с порядком их смещения, независимо от их временной метки. Несоблюдение этого правила может привести к неупорядоченным событиям.

Событие можно считать *запоздалым* (*late*), только если оно рассматривается с точки зрения конкретного микросервиса. Один микросервис может считать любые неупорядоченные события запоздалыми, в то время как другой может быть достаточно «терпеливым», и пройдет много часов физического времени или времени события, прежде чем он сочтет событие запоздальным.

Запоздальные события с водяными знаками и временем потока

Рассмотрим два события — одно с временем t , другое со временем t' . Событие t' имеет более раннюю временную метку, чем событие t .

- ◆ *Водяные знаки.*

Событие t' считается запоздальным, когда оно приходит *после* водяного знака $W(t)$. То, как улаживать это событие, зависит от конкретного узла.

- ◆ *Время потока.*

Событие t' считается запоздальным, когда оно приходит *после* того, как время потока было увеличено за пределы t' . То, как улаживать это событие, зависит от каждого оператора в субтопологии.



Событие запаздывает только тогда, когда оно пропустило крайний для потребителя срок.

Причины и последствия неупорядоченных событий

Неупорядоченные события могут происходить по нескольким причинам.

Загрузка из источников с неупорядоченными данными

Наиболее очевидным, конечно же, является случай, когда события берутся из источника с неупорядоченными данными. Это может произойти, если данные потребляются из потока, который уже находится вне порядка, или если события

поступают из внешней системы с существующими неупорядоченными временными метками.

Несколько производителей в несколько разделов

Когда несколько производителей записывают данные в несколько выходных разделов, это также может создавать неупорядоченные события. Переподразделение существующего потока событий — это один из путей, который может к этому привести. На рис. 6.8 показано переподразделение двух разделов двумя экземплярами потребителя. В этом сценарии исходные события указывают на то, с каким продуктом пользователь взаимодействовал. Например, Гарри взаимодействовал с продуктами ID12 и ID77. Предположим, что аналитику данных необходимо переназначить эти события для ID пользователя, чтобы выполнить сеансовый анализ взаимодействия этого пользователя. Результатирующие выходные потоки могут в итоге оказаться с несколькими неупорядоченными событиями.

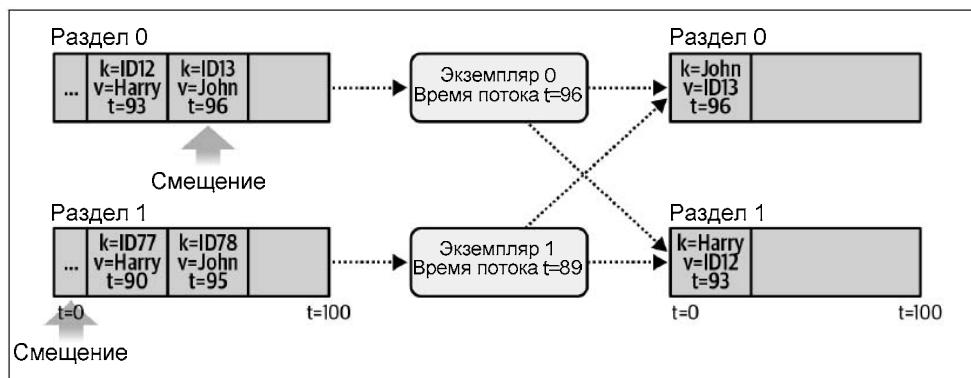


Рис. 6.8. Перемешивание событий через переподразделенный поток событий

Обратите внимание, что каждый экземпляр поддерживает свое собственное внутреннее время потока и что между этими двумя экземплярами нет синхронизации. Это может привести к перекосу во времени, который ведет к появлению неупорядоченных событий, как показано на рис. 6.9.

Экземпляр 0 лишь немногого опережал экземпляр 1 во времени потока, но из-за их независимого времени потока события времена $t = 90$ и $t = 95$ считаются неупорядоченными в переподразделенном потоке событий. Эта проблема усугубляется небалансированными размерами пакетов, неравными скоростями обработки и большим отставанием событий. Последствие здесь заключается в том, что ранее упорядоченные событийные данные теперь стали *неупорядоченными*, и, таким образом, вы, как потребитель, не можете рассчитывать на то, что время в каждом из ваших потоков событий будет постоянно увеличиваться.



Однопоточный производитель при нормальной работе не будет создавать неупорядоченные события, если только он не получает свои данные из неупорядоченно го источника.

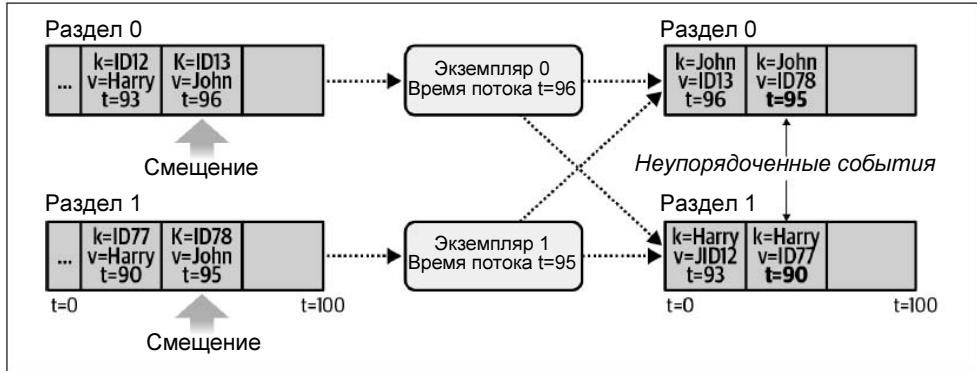


Рис. 6.9. Перемешивание событий через переподразделенный поток событий

Поскольку время потока увеличивается всякий раз, когда обнаруживается событие с более высокой меткой времени, можно попасть в сценарий, где большое количество событий считается запаздывающим из-за переупорядочения. Это может повлиять на обработку в зависимости от того, как потребители решают обрабатывать неупорядоченные события.

Чувствительные ко времени функции и управление окнами

Запоздалые события главным образом относятся к бизнес-логике, основанной на времени, — такой как агрегирование событий в определенный период времени или запуск события после определенного периода времени. *Запоздалое событие* — это событие, которое приходит после того, как бизнес-логика уже закончила обработку в течение определенного периода времени. Оконные функции являются отличным примером бизнес-логики, основанной на времени.

Управление окнами, называемое также *оконной обработкой* (windowing), означает группировку событий по времени. Это особенно полезно для событий с одним и тем же ключом, когда вы хотите увидеть, что произошло с событиями этого ключа за определенный период времени. Существует три основных типа окон событий, но, опять же, не забудьте проверить структуру потоковой обработки для получения дополнительной информации.



Оконная обработка может выполняться с использованием времени события или времени обработки, хотя оконная обработка на основе времени события обычно применяется чаще.

Переворачивающиеся окна

Переворачивающееся (tumbling) окно — это окно фиксированного размера. Предыдущие и последующие окна не перекрываются. На рис. 6.10 показаны три переворачивающихся окна, каждое из которых выровнено по t , $t + 1$ и т. д. Подобного

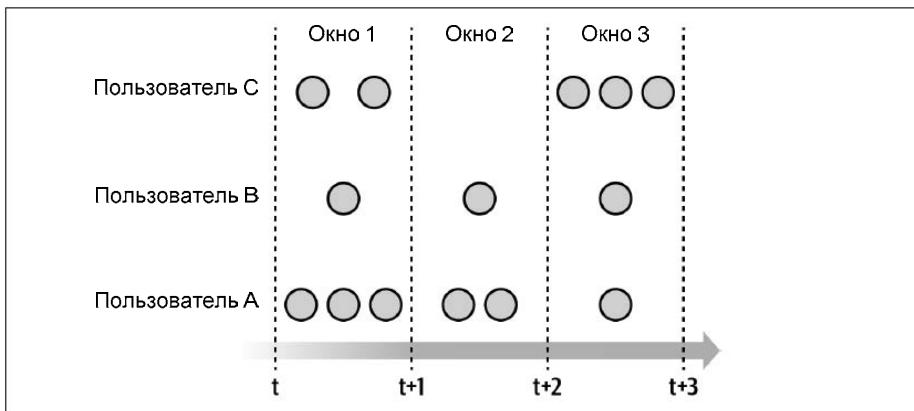


Рис. 6.10. Переворачивающиеся окна

рода окна помогают отвечать на вопросы типа: «Когда наступает пиковый час использования продукта?»

Скользящие окна

Скользящее (sliding) окно имеет фиксированный размер и инкрементный шаг, имеющийся в скольжении окна. Оно должно отражать только агрегацию событий, находящихся в окне в текущий момент. Скользящее окно помогает отвечать на вопросы типа: «Сколько пользователей кликнули на мой продукт за последний час?». На рис. 6.11 показан пример скользящего окна, включающий размер окна и величину его скольжения вперед.

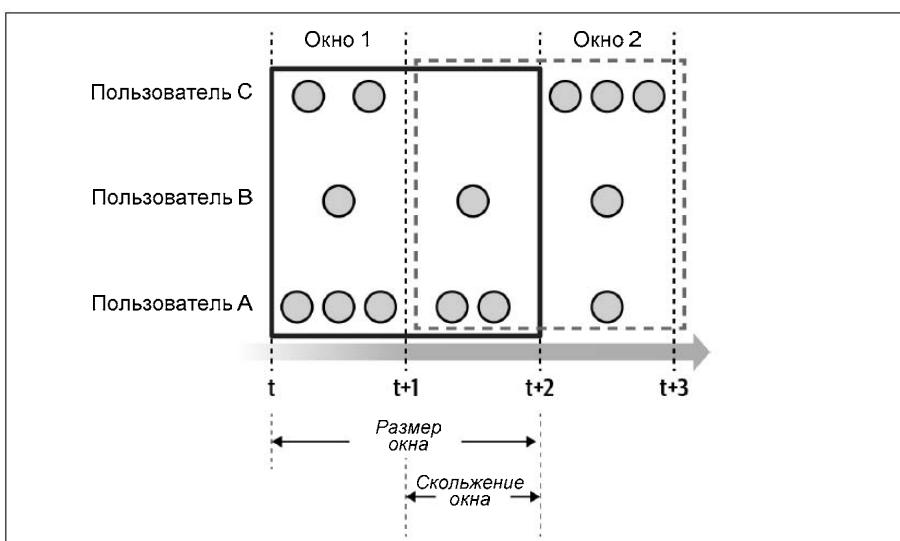


Рис. 6.11. Скользящие окна

Сеансовые окна

Сеансовое (session) окно — это окно с динамическим размером. Он завершается на основе тайм-аута из-за бездействия, и новый сеанс запускается для любой активности, происходящей после тайм-аута. На рис. 6.12 показан пример сеансовых окон с перерывом между сеансами из-за бездействия пользователя С. Такого рода окна помогают отвечать на вопросы типа: «На что смотрит пользователь в текущем сеансе просмотра?»

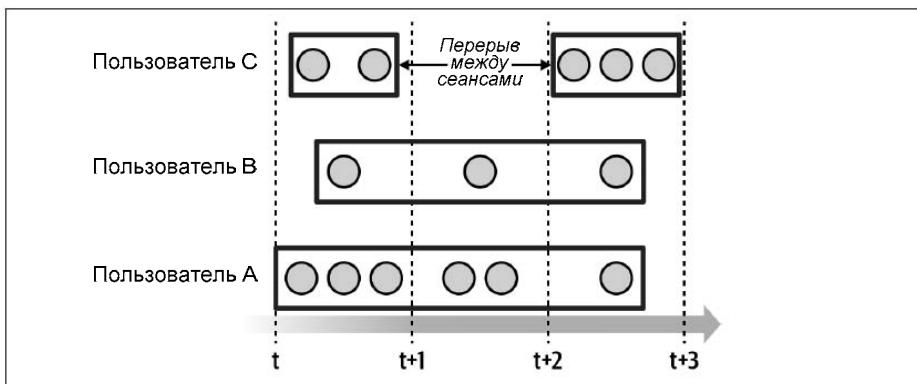


Рис. 6.12. Сеансовые окна

Каждая из этих оконных функций должна иметь дело с неупорядоченными событиями. Но вам придется решить, как долго ждать любых неупорядоченных событий, прежде чем считать их слишком запоздавшими для рассмотрения. Одна из фундаментальных проблем потоковой обработки заключается в том, что вы никогда не можете быть уверены, что получили все события. Ожидать прихода событий, связанных с нарушением порядка, конечно, сколько-то времени можно, но в конечном итоге вашему сервису придется отказаться от такого ожидания, поскольку ждать бесконечно тоже нельзя. Другие факторы, которые следует учитывать, включают объем хранимых данных о состоянии, вероятность запоздалых событий и влияние на предприятие отбрасывания запоздалых событий.

Обработка запоздалых событий

Стратегия обработки неупорядоченных и запоздалых событий должна быть определена на *бизнес-уровне* до разработки инженерного решения, поскольку стратегии будут варьироваться в зависимости от важности данных. Критически важные события, такие как финансовые транзакции и системные отказы, могут обрабатываться независимо от их положения в потоке. С другой стороны, события наподобие снятия показаний — такие как температурные или силовые измерения, могут быть просто отброшены как более не актуальные.

Бизнес-требования также определяют допустимую задержку, поскольку ожидание прибытия событий может повысить детерминизм, но за счет более высокого уровня

задержки. Это может негативно повлиять на характеристики производительности приложений, чувствительных ко времени, или приложений с жесткими соглашениями об уровне обслуживания. К счастью, микросервисы обеспечивают гибкость, необходимую для настройки характеристик детерминированности, задержки и обработки неупорядоченных событий для каждой услуги.

Существует несколько способов обработки запоздалого события, независимо от того, использует ли ваш фреймворк водяные знаки или время потока:

◆ *Событие отбрасывается.*

Событие просто отбрасывается. Окно закрыто, и любые агрегации на основе времени уже завершены.

◆ *Ожидание.*

Вывод окна откладывается до тех пор, пока не пройдет фиксированное время. Это влечет за собой более высокий детерминизм за счет увеличения задержки. Старые окна должны оставаться доступными для обновления до тех пор, пока не пройдет заданный промежуток времени.

◆ *Льготный период.*

Выполните оконный результат, как только окно будет сочтено завершенным. Затем оставьте окна доступными в течение заранее определенного льготного периода. Каждый раз, когда для этого окна приходит запоздалое событие, обновите агрегирование и выведите обновленную агрегацию. Это похоже на стратегию ожидания, за исключением того, что обновления генерируются по мере поступления запоздалых событий.

Независимо от того, как долго ожидает микросервис, в конечном итоге события просто будут слишком запоздальными, и их придется отбрасывать. Не существует четких технических правил того, как ваш микросервис должен обрабатывать запоздальные события, — просто убедитесь, что требования вашего бизнеса полностью удовлетворены. Если протокол обработки запоздальных событий не указан в бизнес-требованиях микросервиса, бизнес должен их соответствующим образом доработать.

Вот несколько вопросов, которые помогут вам определить рекомендации по обработке запоздальных событий:

- ◆ какова вероятность запоздальных событий;
- ◆ сколько времени требуется вашему сервису для защиты от запоздальных событий;
- ◆ каковы последствия отбрасывания запоздальных событий для предприятия;
- ◆ каковы бизнес-выгоды от длительного ожидания захвата запоздальных событий;
- ◆ сколько дискового пространства или памяти требуется для поддержания состояния;
- ◆ перевешивают ли затраты, понесенные в ожидании запоздальных событий, получаемые выгоды?

Повторная обработка или обработка в режиме, близком к реальному времени

Неизменяемые потоки событий обеспечивают возможность переносить смещения групп потребителей и воспроизводить обработку с произвольного момента времени. Эта функциональность называется *повторной обработкой* или *переработкой* (reprocessing), и ее необходимо учитывать при разработке каждого событийно-управляемого микросервиса. Повторная обработка обычно выполняется только на событийно-управляемых микросервисах, которые для обработки событий используют время событий, а не на тех, что опираются на агрегации и оконную обработку на основе реального времени.

Планирование событий — важная часть правильной повторной обработки поступающих из потоков событий запоздалых данных. Это гарантирует, что микросервисы будут обрабатывать события в том же порядке, что и в режиме, близком к реальному времени. Обработка неупорядоченных событий также является важной частью этого процесса, поскольку перераспределение потока событий через брокер событий (вместо использования тяжелых фреймворков типа Spark, Flink или Beam) может вызвать неупорядоченные события.

Вот шаги, которые необходимо предпринять, если вы хотите повторно обрабатывать (перерабатывать) свои потоки событий:

1. *Определить стартовую точку.* Рекомендуется, чтобы все потребители с поддержкой состояния повторно обрабатывали (перерабатывали) события с самого начала *каждого* потока событий, на который они подписаны. В частности, это относится к потокам событий сущностей, т. к. они содержат важные факты о рассматриваемых сущностях.
2. *Определить, какие потребительские смещения следуетбросить.* Любые потоки, содержащие события, которые используются в обработке с поддержкой состояния, должны быть сброшены в самое начало, т. к. трудно гарантировать, что вы получите правильное состояние, если начнете в неправильном месте (подумайте, что произойдет, если вы переработаете чей-то банковский баланс и случайно пропустите предыдущие выплаты).
3. *Подумать об объеме данных.* Некоторые микросервисы могут обрабатывать огромное количество событий. Уточните, сколько времени может потребоваться для повторной обработки событий и какие узкие места могут при этом выявиться. Чтобы гарантировать, что ваш микросервис не перегружает брокер событий вводом-выводом, вам может потребоваться установить квоты (см. разд. «*Квоты*» главы 14). В дополнение к этому вам, возможно, понадобится оповещать всех потребителей, если вы ожидаете, что придется генерировать большие объемы переработанных выходных данных. А им, в свою очередь, соответствующим образом масштабировать свои сервисы, если у них не задействовано их автоматическое масштабирование (см. разд. «*Автоматическое масштабирование приложений*» главы 11).
4. *Учитывайте время обработки.* Вполне возможно, что переработка может занимать много часов, поэтому стоит подсчитать, сколько времени простоя вам мо-

жет понадобиться. Убедитесь, что ваши последующие потребители также не возражают против возможного получения устаревших данных во время повторной обработки вашего сервиса. Увеличение числа экземпляров-потребителей до максимального параллелизма может значительно сократить время простоя и может быть уменьшено после завершения повторной обработки.

5. *Подумайте о последствиях.* Некоторые микросервисы выполняют действия, которые вы, возможно, не захотите выполнять при повторной обработке. Например, сервис, который посыпает пользователям электронные письма о том, что их заказы отправлены, не должен при повторной обработке событий посылать пользователям повторные электронные письма, т. к. это произвело бы на них ужасное впечатление и было бы совершенно бессмысленным мероприятием с точки зрения бизнеса. Внимательно подумайте о влиянии повторной обработки на бизнес-логику вашей системы, а также о потенциальных проблемах, которые могут возникнуть у нижестоящих потребителей.

Периодические сбои и запоздалые события

Событие может запаздывать при обработке почти в реальном времени (водяной знак или время потока увеличивается), но может быть доступно, как и ожидалось, в потоке событий во время его повторной обработки. Эту проблему бывает трудно обнаружить, но она действительно иллюстрирует связанную природу событийно-управляемых микросервисов и то, как вышестоящие проблемы могут повлиять на нижестоящих потребителей. Давайте разберемся, как это может произойти.

Проблемы с подключением производителя/брокера событий

При нормальной работе производители посыпают свои события по мере их возникновения, а потребители потребляют их почти в реальном времени (рис. 6.13). В сценарии с проблемами (рис. 6.14) записи *создаются* в порядке временных меток, но не могут быть *опубликованы* до наступления более позднего времени. При работе по такому сценарию сложно определить, когда это происходит, и он может остаться незамеченным даже в ретроспективе.

Как можно видеть на рис. 6.14, у производителя готовы к отправке несколько записей, но он не может подключиться к брокеру событий. Записи помечаются по *локальному* времени в момент, когда произошло событие. Производитель повторит попытку несколько раз и либо в конечном счете добьется успеха, либо прекратит попытки и откажется от них (в идеале выполнив отказ с предупреждением, чтобы неисправное соединение можно было идентифицировать и исправить). События из потока А по-прежнему потребляются, а водяной знак/время потока соответственно увеличивается. Однако при потреблении из потока В потребитель не получает никаких новых событий, поэтому он может просто предположить, что никаких новых данных нет.

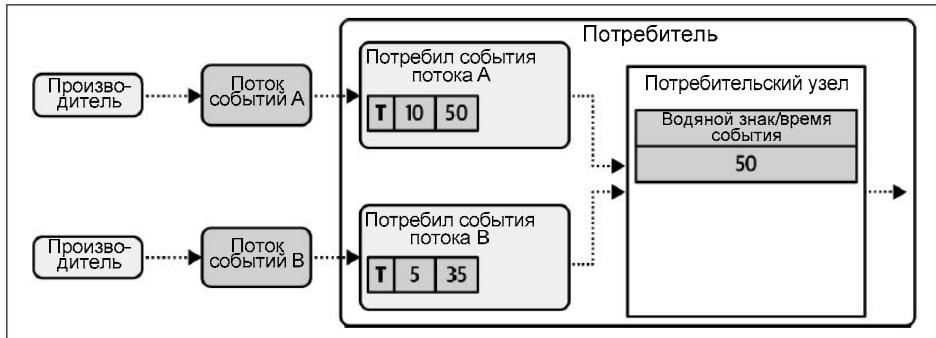


Рис. 6.13. Нормальная работа до момента отключения соединения производителя/брюкера

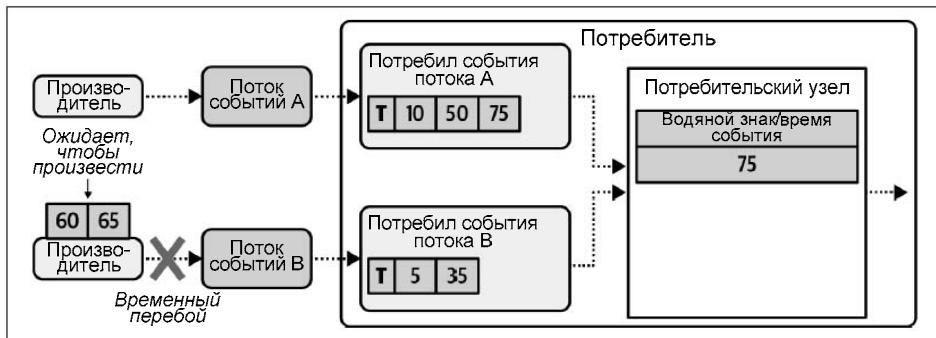


Рис. 6.14. Временный перебой в соединении производителя/брюкера

В конце концов производитель все же сможет передать записи в поток событий. Эти события публикуются в правильном порядке времени, в котором они действительно произошли, но из-за задержки физического времени потребители, работающие почти в реальном времени, отметят их как запоздалые и будут относиться к ним как к таковым. Это показано на рис. 6.15.

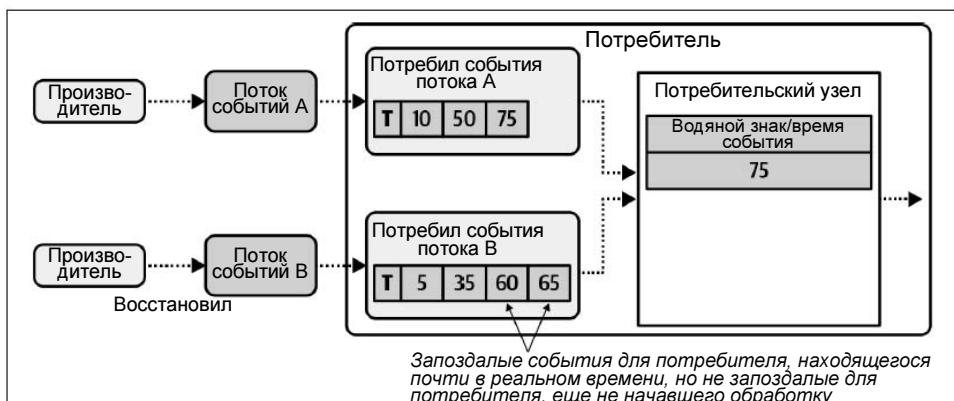


Рис. 6.15. Производитель может повторно подключиться и опубликовать свои временно отложенные события, когда потребитель уже увеличил время своего события

Один из способов смягчить эту проблему состоит в том, чтобы перед обработкой событий подождать заранее определенное количество времени, хотя этот подход все же требует затрат на задержку и будет полезен только тогда, когда производственные задержки короче времени ожидания. Еще один вариант — использовать в коде устойчивую логику обработки запаздывающих событий, обеспечивающую, чтобы этот сценарий не влиял на вашу бизнес-логику.

Резюме и дополнительная литература

В этой главе мы рассмотрели детерминизм и то, как лучше всего приблизиться к нему с неограниченными потоками. Мы также разобрались, как выбрать следующие события для обработки между несколькими разделами, чтобы обеспечить максимальную детерминированность при обработке как в режиме, близком к реальному времени, так и при повторной обработке. Сама природа неограниченного потока событий в сочетании с периодическими сбоями означает, что полный детерминизм не может быть достигнут никогда. Тем не менее разумные, оптимальные решения, работающие большую часть времени, способны обеспечить приемлемый компромисс между задержкой и правильностью.

Вы также должны учитывать в своем дизайне возможность появления неупорядоченных и запоздалых событий. В этой главе показано, как водяные знаки и время потока могут использоваться для идентификации и обработки таких событий. Если вы хотите узнать о водяных знаках больше, то ознакомьтесь с превосходными статьями Тайлера Акидау (Tyler Akida) «Мир за пределами пакетной потоковой передачи» 101 («The World Beyond Batch Streaming» 101, <https://oreil.ly/XoqNE>) и 102 (<https://oreil.ly/pkbAF>). Дополнительные соображения и идеи о распределенном системном времени можно найти в онлайновой книге Микито Такады (Mikito Takada) «Распределенные системы для удовольствия и прибыли» («Distributed Systems for Fun and Profit», <https://oreil.ly/IDT4D>).

Потоковая передача с поддержкой состояния

Потоковая передача с поддержкой состояния лежит в основе наиболее важных компонентов событийно-управляемых микросервисов, поскольку большинству приложений необходимо для своих целей поддерживать определенные состояния. В разд. «*Материализация состояния из сущностных событий*» главы 2 вкратце описываются принципы материализации потока событий в набор состояний. Здесь же мы подробнее рассмотрим вопрос создания, управления и использования состояний для событийно-управляемых микросервисов.

Хранилища состояний и материализация состояний из потока событий

Давайте начнем с нескольких определений:

◆ *Материализованное состояние.*

Проекция событий из исходного потока событий (неизменяемого).

◆ *Хранилище состояний.*

Место, где хранятся внутренние состояния вашего сервиса (изменяемое).

Как материализованные состояния, так и хранилища состояний требуются и широко используются в микросервисах с поддержкой состояния, но важно их различать. Материализованные состояния позволяют использовать общие бизнес-сущности в приложениях микросервисов, в то время как хранилища состояний дают возможность хранить внутренние состояния бизнес-логики и выполнять промежуточные вычисления.

При проектировании микросервисов нужно учитывать то, где сервис будет хранить свои данные. Существуют два основных варианта хранения состояния и доступа к нему:

- ◆ внутренний, когда данные хранятся в том же контейнере, что и обработчик, будучи размещенными в памяти или на диске;
- ◆ внешний, когда данные хранятся вне контейнера обработчика в разных формах внешних хранилищ. Доступ к ним часто выполняется через сеть.

На рис. 7.1 показаны примеры как внутреннего, так и внешнего хранилища состояний.

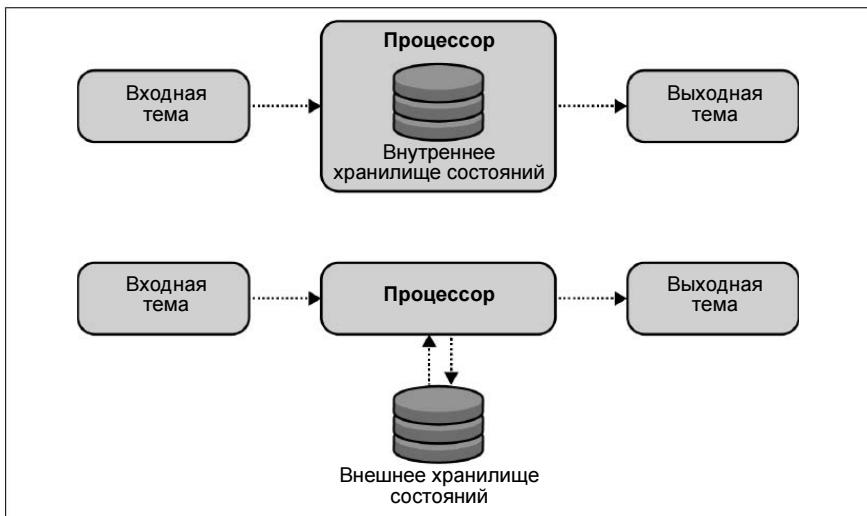


Рис. 7.1. Внутреннее (вверху) и внешнее (внизу) хранилища состояний

Выбор внутреннего или внешнего хранилища зависит в первую очередь от бизнес-функции микросервиса и технических требований. Но прежде чем оценивать эти два варианта подробнее, необходимо подумать о роли журнала изменений.

Запись состояния в поток событий журнала изменений

Журнал изменений — это запись всех изменений, внесенных в данные хранилища состояний. Он представляет собой поток, связанный с таблицей, где таблица состояний трансформирована в поток отдельных событий. Как точная копия состояния, поддерживаемого вне экземпляра микросервиса, журнал изменений может использоваться для пересчета состояния и служить способом отслеживания хода обработки событий в контрольных точках (рис. 7.2).



Журналы изменений оптимизируют задачу пересчета состояний отказавших сервисов, поскольку они хранят результаты предыдущей обработки, позволяя восстанавливющему обработчику избежать просмотра всех входных событий.

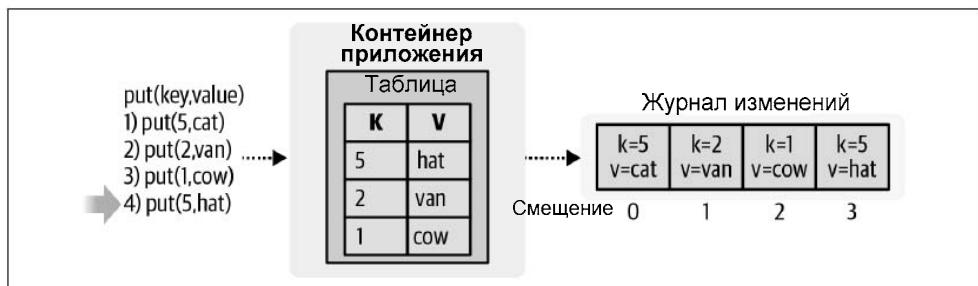


Рис. 7.2. Хранилище состояний с функцией пересчета состояний на основе журнала изменений

Потоки журнала изменений хранятся в брокере событий точно так же, как и любой другой поток, и, как уже отмечалось, обеспечивают средство перестройки хранилища состояний. Потоки журнала изменений должны уплотняться, т. к. для перестройки состояния им нужна только самая последняя пара ключ/значение.

Журналы изменений могут масштабироваться и восстанавливать состояние с высокой производительностью — в особенности для внутренних хранилищ состояний. В обоих случаях вновь созданный экземпляр приложения просто должен загрузить данные из соответствующих разделов журнала изменений, как показано на рис. 7.3.

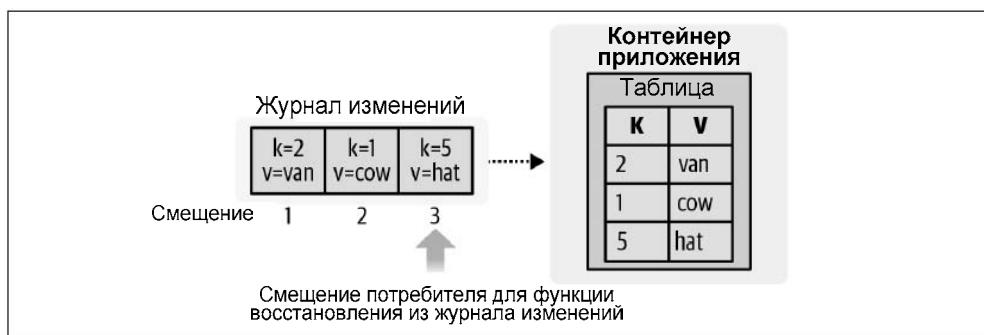


Рис. 7.3. Хранилище состояний восстанавливается из журнала изменений

Журналы изменений либо предоставляются в виде встроенной функции — например, в клиенте Kafka Streams (<https://oreil.ly/4hh3y>), либо реализуются разработчиком приложения. Базовые клиенты производителя/потребителя, как правило, не предоставляют никакой поддержки журналирования изменений или внутреннего состояния.

Материализация состояния во внутреннее хранилище состояний

Внутренние хранилища состояний существуют в том же контейнере или среде виртуальной машины, что и бизнес-логика микросервиса. В частности, существование внутреннего хранилища состояний связано с существованием экземпляра микросервиса, причем оба они работают на одном и том же базовом аппаратном обеспечении.

Каждый экземпляр микросервиса материализует события из назначенных ему разделов, поддерживая данные каждого раздела логически разделенными в хранилище. Такие отдельные материализованные разделы предоставляют экземпляру микросервиса возможность просто отбрасывать состояние для *отозванного раздела* (revoked partition) после перебалансировки группы потребителей. Это позволяет избежать утечек ресурсов и появления множественных источников истины путем организации существования материализованного состояния только на экземпляре, которому принадлежит раздел. Новые назначения разделов могут быть перестроен-

ны путем потребления входных событий из потока событий или из журнала изменений.

Для реализации внутренних хранилищ состояний обычно используются высокопроизводительные хранилища ключ/значения (такие как RocksDB), оптимизированные для эффективной работы с локальными твердотельными накопителями (SSD), что позволяет выполнять операции с наборами данных, превышающими объем доступной памяти. Впрочем, несмотря на то, что хранилища ключ/значения, как правило, являются наиболее распространенным решением для внутренних хранилищ состояний, вы можете задействовать любую форму хранилища данных. Использование реляционного или документного хранилища данных также возможно, однако оно тогда должно присутствовать в контейнере вместе с экземпляром микросервиса.

Материализация глобального состояния

Хранилище глобального состояния — это особая форма внутреннего хранилища состояний. Вместо того, чтобы материализовывать только назначенные ему разделы, хранилище глобального состояния материализует данные всех разделов для того или иного потока событий, предоставляя полную копию событийных данных каждому экземпляру микросервиса. Рис 7.4 иллюстрирует разницу между глобальным и неглобальным материализованными состояниями.

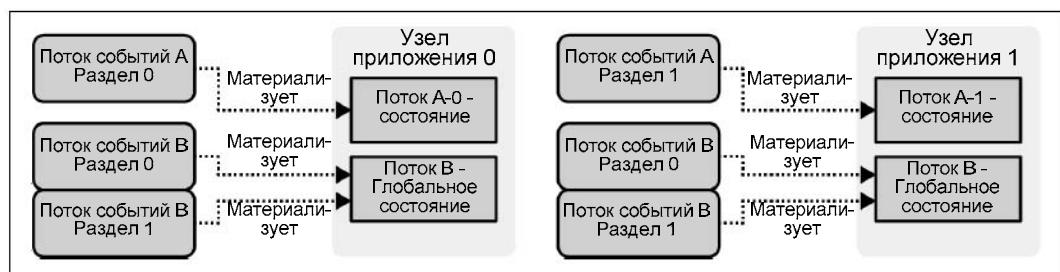


Рис. 7.4. Глобальное материализованное состояние и неглобальное материализованное состояние

Хранилища глобального состояния полезны тогда, когда для каждого экземпляра требуется полный набор данных, и, как правило, включают малые, часто используемые и редко изменяющиеся наборы данных. Глобальная материализация не может эффективно использоваться в качестве драйвера для событийно-управляемой логики, поскольку каждый экземпляр микросервиса обладает полной копией данных и, таким образом, будет производить дублированные выходные данные и неопределенные результаты. По этой причине лучше всего зарезервировать глобальную материализацию для просмотра общих наборов данных и таблиц измерений (dimension tables).

Преимущества использования внутреннего состояния

Требования к масштабируемости снимаются с разработчика

Главным преимуществом использования внутренних хранилищ состояний на локальном диске является то, что все требования к масштабируемости полностью переносятся на брокер событий и кластеры вычислительных ресурсов.

Это позволяет команде разработчиков приложений сосредоточиться строго на написании логики приложения, полагаясь при этом на то, что команды, обеспечивающие возможности микросервисов, будут поставлять механизмы масштабирования, общие для всех событийно-управляемых микросервисов. Такое решение обеспечивает единый подход к масштабируемости, где каждое приложение может быть масштабировано просто путем увеличения и уменьшения количества экземпляров.

Когда вы рассматриваете внутреннее хранилище состояний, важно понимать требования к производительности вашего приложения. С учетом возможностей современных облачных вычислений локальный диск не обязательно означает физически подключенный диск, поскольку накопитель, подключенный через сеть, может имитировать локальный диск и обеспечивать такую же логическую поддержку ваших приложений. Высокопроизводительный потоковый микросервис с поддержкой состояния может легко потреблять сотни тысяч событий в секунду. Вы должны тщательно оценить требуемые приложением характеристики производительности, чтобы убедиться, что они могут быть достигнуты с учетом возможных задержек.

Высокопроизводительные возможности, основанные на использовании дисков

В событийно-управляемом микросервисе — в особенности в недорогих решениях — обработка всех возможных состояний в памяти не всегда возможна. Для большинства случаев использования микросервисов достаточную производительность может обеспечить физически подключенный локальный диск. Варианты реализации с локальным диском, как правило, отдают предпочтение решениям с высокой скоростью доступа (random-access read), обычно представленным твердотельными накопителями (SSD). Например, задержка для операции чтения из произвольного места с твердотельного накопителя с использованием RocksDB составляет примерно 65 микросекунд (<https://oreil.ly/60t6J>), т. е. один поток будет иметь потолок последовательного доступа примерно в 15,4 тыс. запросов в секунду. При этом производительность памяти значительно выше — в нормальных условиях она обслуживает миллионы запросов произвольного доступа в секунду (<https://oreil.ly/QEbW3>). Подход на основе локального диска и локальной памяти обеспечивает чрезвычайно высокую пропускную способность и значительно сокращает ограничения доступа к данным.

Гибкость использования подключенного через сеть внешнего диска

Микросервисы также могут вместо локального диска использовать подключенный через сеть диск, что значительно увеличивает задержку чтения/записи. Поскольку

для поддержания порядка по времени и смещению события обычно должны обрабатываться по одному, один обрабатывающий поток исполнения будет тратить много времени на ожидание ответов на чтение/запись, что приведет к значительно более низкой пропускной способности системы. Это, как правило, хорошо для любого сервиса с поддержкой состояния, который не нуждается в высокой производительности, но может стать проблемой, если количество событий значительно возрастет.

Доступ к «локальным» данным, хранящимся на диске, подключенном через сеть, имеет гораздо более высокую задержку, чем доступ к физически локальным данным, хранящимся в памяти системы или на подключенном локально диске. В то время как RocksDB в паре с локальным SSD имеет расчетную пропускную способность 15,4 тыс. запросов в секунду, введение в идентичный шаблон доступа сетевой задержки всего в 1 мс туда и обратно (<https://oreil.ly/rsl6a>) снижает предел пропускной способности до всего 939 запросов в секунду. Хотя вы можете кое-что сделать по части обеспечения параллельного доступа и сокращения этого разрыва, помните, что события должны обрабатываться в той последовательности смещения, в которой они потребляются, и что распараллеливание во многих случаях невозможно.

Одной из выгод от подключенного через сеть диска является то, что состояние тома можно поддерживать и при необходимости переносить на новое обрабатывающее аппаратное обеспечение. Когда узел обработки возвращается в рабочее состояние, сетевой диск можно подключить повторно, и обработка может возобновиться с того места, где она была остановлена, вместо того, чтобы восстанавливать ее из потока журнала изменений. Это значительно сокращает время простоя, поскольку состояние больше не является полностью зависимым, как в случае с локальным диском, а также увеличивает гибкость микросервисов для миграции между вычислительными ресурсами — например, когда вы используете недорогие узлы по запросу (on-demand nodes).

Недостатки использования внутреннего состояния

Ограничения при использовании локальных дисков

Внутренние хранилища состояний ограничены использованием только того диска, который определен и присоединен к узлу во время выполнения сервисов. Изменение размера или количества присоединенных томов обычно требует остановки сервиса, настройки томов и перезапуска сервиса. Кроме того, многие решения по управлению вычислительными ресурсами допускают только увеличение размера тома, поскольку уменьшение размера тома означает необходимость удаления данных.

Потери при использовании дискового пространства

Циклические по своей природе шаблоны данных — такие как трафик, генерируемый на торговый сайт в три часа дня, по сравнению с тремя часами ночи, могут по-

требовать циклического изменения объема хранения — максимального объема диска для пикового трафика и очень малого объема в обычных случаях. Резервирование полного объема диска на все время — по сравнению с использованием внешних сервисов, которые взимают плату только за объемы хранящихся данных, — приводит к излишним денежным затратам и нерациональному использованию дискового пространства.

Масштабирование и восстановление внутреннего состояния

Масштабирование обработки до нескольких экземпляров и восстановление откавшего экземпляра — это идентичные процессы с точки зрения восстановления состояния. Новый или восстановленный экземпляр должен материализовать любое состояние, определенное его топологией, прежде чем он сможет начать обработку новых событий. Самый быстрый способ сделать это состоит в перезагрузке темы журнала изменений для каждого хранилища состояний, которое материализовано в приложении.

Использование «горячих» реплик

Хотя чаще всего для каждого раздела используется только одна реплика материализованного состояния, дополнительные реплики могут быть созданы с помощью управления состоянием или напрямую задействованы клиентским фреймворком. В Apache Kafka эта функциональность обеспечивается встроенным фреймворком Streams (<https://oreil.ly/VGbuo>) за счет простой настройки его конфигурации. Такая настройка обеспечивает высокую доступность хранилищ состояний и позволяет микросервису устойчиво реагировать на отказ экземпляра с нулевым временем простоя.

На рис. 7.5 показано развертывание из трех экземпляров с коэффициентом репликации внутреннего хранилища состояний, равным 2. Каждый раздел с поддержкой состояния материализуется дважды: один раз в качестве ведущего и один раз в качестве реплики. Каждая реплика должна управлять своими собственными смещениями с целью исключения отставания от смещения ведущей реплики. Экземпляр 0 и экземпляр 1 обрабатывают события потока B и соединяют их в соподразделенном материализованном состоянии. Экземпляр 1 и экземпляр 2 также поддерживают «горячие реплики» (hot replicas) соответственно потоков A-P0 и A-P1, при этом экземпляр 2 в противном случае не обрабатывает никаких других событий.

Когда лидирующий экземпляр останавливается, группа потребителей должна перебалансировать назначение разделов. Контроллер разделов (partition assignor) определяет местоположение «горячей реплики» (он предварительно назначил все разделы и знает все соотнесения раздел-экземпляр) и соответствующим образом переназначает разделы. На рис. 7.6 экземпляр 1 остановлен, и оставшиеся экземпляры микросервисов вынуждены перебалансировать свои назначения разделов. Экземплярам с «горячими репликами» предоставляется приоритет для утверждения права

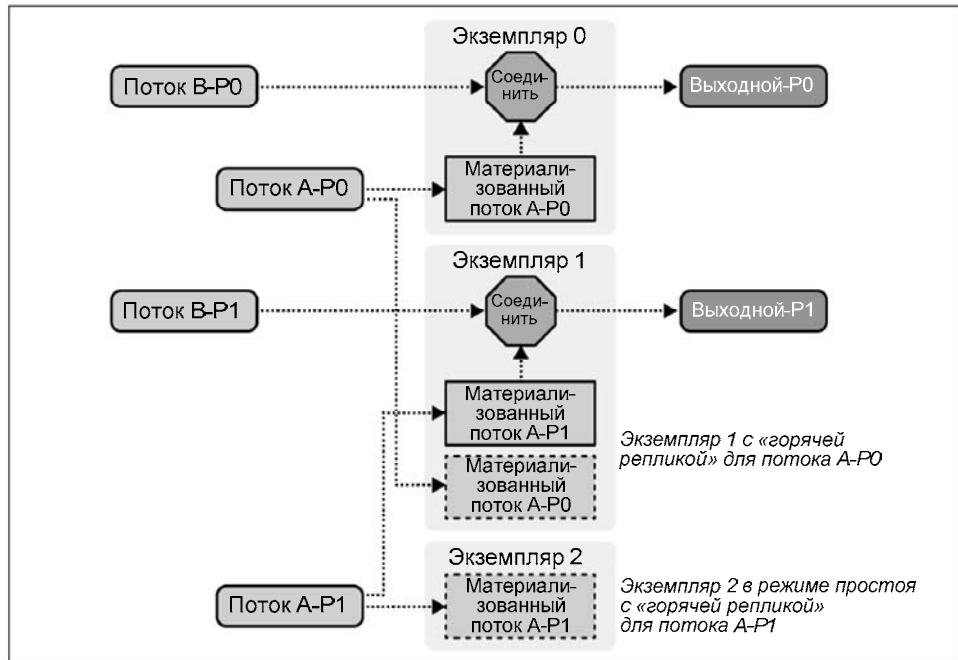


Рис. 7.5. Соединение поток-таблица с тремя экземплярами и двумя «горячими» репликами» в расчете на материализованный входной раздел

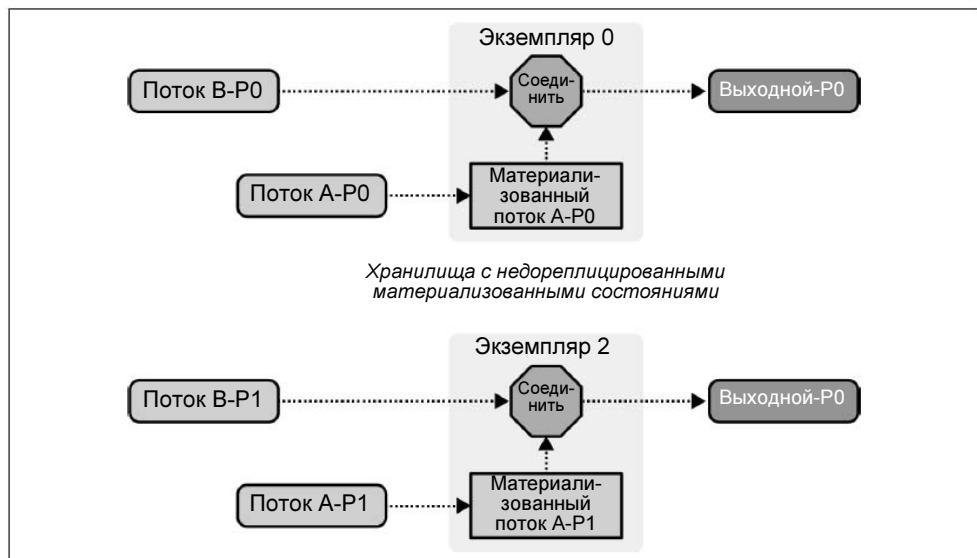


Рис. 7.6. Перебалансировка из-за остановки экземпляра 1

владения разделом и немедленного возобновления обработки. Как можно видеть, контроллер раздела выбрал экземпляр 2 для возобновления обработки потока B-P1.

После возобновления обработки новые «горячие реплики» должны быть построены из журнала изменений, чтобы поддерживать минимальное количество реплик. Новые «горячие реплики» создаются и добавляются к оставшимся экземплярам, как показано на рис. 7.7.

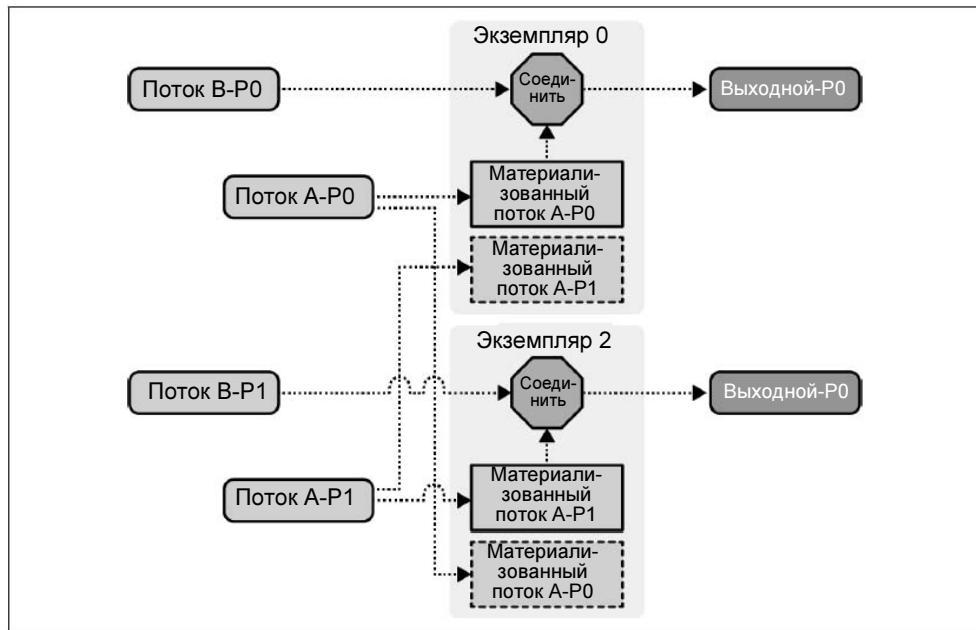


Рис. 7.7. Нормальная обработка с двумя экземплярами и двумя «горячими репликами» в расчете на материализованный входной раздел



Одним из основных компромиссов при использовании подхода на основе «горячих реплик» является использование дополнительного диска для обслуживания реплик в обмен на сокращение времени простоя из-за отказа экземпляра.

Восстановление и масштабирование из журналов изменений

Когда вновь созданный экземпляр микросервиса присоединяется к группе потребителей, любые разделы с поддержкой состояния, которые ему назначены, могут быть перезагружены просто путем потребления из его журнала изменений. В течение этого времени экземпляр не должен обрабатывать новые события, т. к. это может привести к неопределенным и ошибочным результатам.

Восстановление и масштабирование входных потоков событий

Если журнал изменений не поддерживается, то экземпляр микросервиса может перестроить свои хранилища состояний из входных потоков. Он должен повторно

прочитать и обработать все свои входные события с самого начала назначенных ему разделов потока событий. Каждое событие должно быть прочитано и обработано в строгом порядке, его состояние обновлено, и все последующие выходные события созданы.



Возьмем, к примеру, влияние событий, происходящих во время полной переработки. Потребителям может потребоваться обработать их идемпотентно или устраниить их как дубликаты.

Восстановление состояния с использованием такого процесса может занять гораздо больше времени, чем восстановление из журнала изменений. Так что этот подход лучше всего применять только для простых топологий, где дублирование выходных данных не является проблемой, удержание входного потока событий короткое, а потоки сущностных событий — редкие.

Материализация состояния во внешнее хранилище состояний

Внешние хранилища состояний существуют вне контейнера микросервиса или виртуальной машины, но обычно расположены в одной локальной сети. Вы можете реализовать внешнее хранилище данных, используя предпочтительную вами технологию, но выбрать ее вы должны, исходя из потребностей пространства решений микросервиса. Некоторые распространенные примеры сервисов внешнего хранения включают реляционные базы данных, документные базы данных, геопространственную поисковую систему на основе Lucene и распределенные высокодоступные хранилища ключ/значения.

Имейте в виду, что хотя внешнее хранилище состояний конкретного микросервиса может действовать общепринятую платформу хранения данных, сам набор данных должен оставаться логически изолированным от всех других экземпляров микросервиса. Совместное использование материализованного состояния между микросервисами — это вполне себе неверный подход, распространенный среди разработчиков внешних хранилищ данных, которые стремятся задействовать общий материализованный набор данных для удовлетворения многочисленных бизнес-требований. Следование ему может привести к тесной сцепке между другими совершенно не связанными продуктами или функциональностями, и этого желательно избегать.

Не делитесь прямым доступом к состоянию с другими микросервисами. Вместо этого все микросервисы должны материализовывать свою собственную копию состояния. Это исключает прямую сцепленность и изолирует микросервисы от непредвиденных изменений, но за счет использования дополнительных ресурсов обработки и хранения данных.

Преимущества внешнего состояния

Полная локализация данных

В отличие от внутренних хранилищ состояний, внешние хранилища состояний могут предоставлять доступ ко всем материализованным данным для каждого экземпляра микросервиса, хотя каждый экземпляр по-прежнему отвечает за материализацию собственных назначенных ему разделов. Единый материализованный набор данных устраняет необходимость в локальности разделов при выполнении поиска, реляционных запросов по внешним ключам и геопространственных запросов между большим числом элементов.



Используйте хранилища состояний с надежными гарантиями чтения после записи, чтобы устраниТЬ несогласованные результаты при использовании многочисленных экземпляров.

Технологии

Внешние хранилища данных могут использовать технологии, с которыми команда уже знакома, сокращая время и усилия, необходимые для внедрения микросервисов в производство. Особенно хорошими кандидатами для использования внешних хранилищ данных представляются шаблоны базового потребителя/производителя (см. главу 10). Решения на основе технологии «Функция как сервис», описанные в главе 9, также являются превосходными кандидатами на внешнее хранилище данных.

Недостатки внешнего состояния

Управление многочисленными технологиями

Внешние хранилища состояний управляются и масштабируются независимо от решения выбранного для бизнес-логики микросервиса. Один из рисков внешнего хранилища данных заключается в том, что владелец микросервиса теперь «находится на крючке», занимаясь реализацией его надлежащего технического сопровождения и масштабирования. Каждая команда должна распределять ресурсы, определять политику масштабирования и системный мониторинг, чтобы обеспечить сервису обработки данных отказоустойчивость и возможность работы с необходимой нагрузкой. Наличие управляемых сервисов обработки данных, предоставляемых собственными командами организации или сторонним поставщиком облачных платформ, помогает распределять часть этой ответственности.



Каждая команда должна полностью управлять внешними хранилищами состояний для своих микросервисов. Не делегируйте ответственность за управление внешним хранилищем состояний его команде, т. к. это создает техническую зависимость. Составьте список приемлемых внешних сервисов обработки данных с инструкциями по их правильному управлению и масштабированию. Это избавит команды от необходимости самостоятельно находить собственные управленические решения.

Потеря производительности из-за сетевой задержки

Доступ к данным, хранящимся во внешнем хранилище состояний, имеет гораздо более высокую задержку, чем доступ к данным, хранящимся локально в памяти или на диске. В разд. «Преимущества использования внутреннего состояния» этой главы было показано, что использование подключенного к сети диска добавляет небольшую сетевую задержку и может значительно снизить пропускную способность и производительность.

В то время как кэширование и параллелизация могут уменьшить влияние сетевой задержки, основанный на них компромисс часто добавляет сложности и увеличивает стоимость дополнительной памяти и процессора. Не все шаблоны микросервисов также поддерживают кэширование и параллелизацию, и многие из них требуют, чтобы обрабатывающий поток исполнения просто блокировался и ждал ответа из внешнего хранилища данных.

Финансовые затраты на сервисы внешнего хранилища состояний

Финансовые затраты при использовании внешних хранилищ данных, как правило, выше, чем при работе с аналогичными по размерам внутренними хранилищами данных. Провайдеры, у которых размещены решения для внешнего хранения состояний, часто взимают плату за число транзакций, размер полезной нагрузки данных и срок их хранения. Они также могут потребовать избыточного резервирования для обработки пиковых и несогласованных нагрузок. Модели ценообразования с гибкими характеристиками производительности помогают снизить затраты, но вы должны быть уверены, что они по-прежнему отвечают вашим потребностям.

Полная локальность данных

Хотя полная локальность данных также указывается в качестве преимущества, она может вызывать и некоторые проблемы. Данные, доступные во внешнем хранилище состояний, поступают из нескольких обработчиков и нескольких разделов, каждый из которых обрабатывает данные со своей скоростью. При этом становится трудно получать представление о вкладе любого конкретного экземпляра обработки в коллективное общее состояние (и его отлаживать).

Необходимо также тщательно избегать конфликтных ситуаций и неопределенного поведения, поскольку каждый экземпляр микросервиса работает в своем собственном независимом времени потока. Гарантии времени потока одного экземпляра микросервиса не распространяются на всех остальных.

Например, один экземпляр может попытаться присоединиться к событию по внешнему ключу, который еще не был заполнен отдельным экземпляром. Повторная обработка тех же данных в более позднее время может привести к появлению дубликата. Поскольку потоковая обработка каждого экземпляра полностью отделена от других, любые результаты, полученные с помощью этого подхода, скорее всего, будут неопределенными и невоспроизводимыми.

Масштабирование и восстановление с помощью внешних хранилищ состояний

Масштабирование и восстановление микросервисов с использованием внешнего хранилища состояний просто требует добавления нового экземпляра с необходимыми учетными данными для доступа к хранилищу состояний. Напротив, масштабирование и восстановление опорного хранилища состояний полностью зависит от выбранной технологии и является гораздо более сложным.

Повторю ранее сказанное: наличие списка приемлемых внешних сервисов обработки данных с руководствами по правильному управлению, масштабированию, резервному копированию и восстановлению данных имеет важное значение для обеспечения разработчикам возможности устойчиво следовать по выбранному ими пути. К сожалению, число технологий хранилищ состояний непомерно велико, и практически невозможно обсудить их все в этой книге. Вместо этого я просто обобщу стратегии восстановления состояний в три главных технических приема: восстановление из исходных потоков, использование журналов изменений и создание моментальных снимков.

Использование исходных потоков

Потребление событий с начала времени из исходных потоков создает свежую копию хранилища состояний. Входные смещения группы потребителей устанавливаются для всех входных потоков на начало времени. Этот метод имеет самое длительное время простоя из всех вариантов, но легко воспроизведим и полагается только на систему обеспечения постоянства данных брокера событий для поддержания исходных данных. Имейте в виду, что этот вариант в действительности представляет собой полный сброс приложения и также приводит к воспроизведению любых выходных событий в соответствии с бизнес-логикой микросервиса.

Использование журналов изменений

Внешние хранилища состояний обычно не опираются на использование хранимых брокером журналов изменений для регистрации и восстановления состояния, хотя нет никаких правил, этому препятствующих. И так же, как и при работе с внутренними хранилищами состояний, внешние хранилища состояний тоже могут быть повторно заполнены из журнала изменений. И точно так же, как и при перестроении из исходных потоков, вам надо будет создать новую копию хранилища состояний. При перестроении из журналов изменений потребительские экземпляры микросервисов должны обеспечивать перестраивание всего состояния в том виде, в котором оно сохранено в журнале изменений, прежде чем возобновлять обработку.



Перестраивание внешних хранилищ состояний из исходных потоков событий или журналов изменений может занять непомерно много времени из-за накладных расходов на задержку сети. Убедитесь, что в таком сценарии вы по-прежнему можете удовлетворить SLA (соглашение об уровне предоставления услуги) микросервисов.

Использование моментальных снимков

Гораздо более типично для внешних хранилищ состояний обеспечивать собственный процесс резервного копирования и восстановления, и многие службы размещения хранилищ состояний предоставляют для этого простые решения «одним щелчком мыши». В зависимости от конкретной реализации хранилища состояний следует придерживаться лучших практических приемов захвата и восстановления состояния.

Если сохраненное состояние является идемпотентным, то нет необходимости обеспечивать точное соответствие смещений материализованному состоянию. В этом случае установка смещений потребителей на значения за несколько минут до создания моментального снимка гарантирует, что никакие данные не будут пропущены. Это также обеспечивает условие, что события обрабатываются с гарантией «хотя бы один раз».

Если сохраненное состояние не является идемпотентным и любые дублирующиеся события неприемлемы, то вы должны хранить смещения разделов вашего потребителя вместе с данными в хранилище данных. Благодаря этому обеспечивается согласованность смещений потребителя и ассоциированного с ним состояния. Когда состояние восстанавливается из моментального снимка, потребитель может установить смещения своей группы потребителей на те, которые были найдены в моментальном снимке, начиная с точного времени создания моментального снимка (более подробно об этом рассказано далее в разд. «Поддержание согласованного состояния»).

Перестраивание хранилищ состояний или их миграция?

Новые бизнес-требования часто сопровождаются изменениями в существующих структурах данных хранилища состояний. Так, микросервису может потребоваться добавить новую информацию в существующие события, выполнить некоторые дополнительные шаги соединения с другой материализованной таблицей или иным образом сохранить вновь полученные бизнес-данные. В таком случае существующее хранилище состояний необходимо будет обновить, чтобы отразить данные — путем перестройки либо путем миграции.

Перестройка

Перестройка хранилищ состояний микросервиса обычно является наиболее распространенным методом обновления внутреннего состояния приложения. Микросервис сначала останавливается, и смещения входного потока потребителя сбрасываются в начало. Любое промежуточное состояние — например, сохраненное в журнале изменений или находящееся во внешнем хранилище состояний, должно быть удалено. В завершение запускается новая версия микросервиса, и состояние перестраивается по мере того, как события читаются заново из входных потоков

событий. Этот подход обеспечивает, чтобы состояние строилось точно так, как указано в новой бизнес-логике. Все новые выходные события также создаются и распространяются к подписавшимся потребителям. Они не рассматриваются как дублирующиеся события, поскольку бизнес-логика и формат вывода могут изменяться, и эти изменения должны быть распространены вниз по потоку.

Перестраивание состояния требует, чтобы все необходимые события входного потока событий по-прежнему существовали, — в частности те, что нужны для материализации состояния и агрегаций. Если ваше приложение критически зависит от набора входных данных, то вы должны обеспечить, чтобы такие исходные данные были легкодоступными за пределами хранилища данных вашей реализации микросервиса.



Перестраивание требует времени, и это важно учитывать в SLA микросервиса. Одной из главных выгод от практики перестройки является то, что она помогает вам проверить готовность к аварийному восстановлению, выполнив процесс восстановления, необходимый при отказе микросервисов и потере всего состояния.

Наконец, некоторые бизнес-требования однозначно требуют переработки данных с самого начала времени — например, тех, которые извлекают поля, присутствующие только во входных событиях. Вы не можете получить эти данные никаким другим способом, кроме как воспроизведя входные события, и в этот момент перестраивание хранилища состояний является единственным жизнеспособным вариантом.

Миграция

Перестраивание крупных хранилищ состояний может занять много времени или привести к непомерно высоким затратам на передачу данных по сравнению с последствиями изменений. Например, рассмотрим такое изменение бизнес-требований, когда дополнительное, но необязательное поле должно быть добавлено в выходной поток событий микросервиса. Это изменение может потребовать добавления еще одного столбца или поля в хранилище состояний микросервиса. Однако может оказаться, что предприятие не нуждается в переработке старых данных и в будущем хочет применять логику только к новым входным событиям. В случае хранилища состояний, поддерживаемого реляционной базой данных, вам придется просто обновить бизнес-логику вместе с ассоциированным определением таблицы. Вы можете выполнить простую вставку нового столбца с допускающим значение `null` типом и после быстрой серии тестов повторно развернуть приложение.

Тем не менее *мигрирование* оказывается гораздо более рискованным процессом, когда бизнес-требование и изменяемые данные становятся сложнее. Сложные миграции подвержены ошибкам и могут приводить к ошибочным результатам по сравнению с результатами полного перестройки хранилища данных. Логика миграции базы данных не является частью бизнес-логики, поэтому вы можете получить несогласованность данных, которая в противном случае не возникла бы при полном перестройке приложения. Такие типы ошибок миграции бывает трудно

обнаружить, если их не перехватить во время тестирования, и они могут вызывать несогласованность данных. При использовании подхода, основанного на миграции, обязательно выполняйте строгое тестирование и применяйте репрезентативные наборы тестовых данных для сравнения этого подхода с подходом, основанным на перестраивании.

Транзакции и практически однократная обработка

Практически однократная обработка (*effectively once processing*) обеспечивает, чтобы любые обновления единственного источника истины применялись согласованно, независимо от любого отказа для производителя, потребителя или брокера событий. Практически однократную обработку также иногда называют *строго одноразовой обработкой* (*exactly once processing*), хотя это не совсем точно. Микросервис может обрабатывать одни и те же данные несколько раз, скажем, из-за отказа потребителя и последующего восстановления, но не фиксировать свои смещения и не увеличивать время потока. Логика обработки будет выполняться каждый раз при обработке события, *включая любые побочные эффекты, которые может создать код*, — например, публикацию данных на внешние конечные точки или обмен данными со сторонней службой. Так что для большинства брокеров событий и большинства случаев использования термины «строго одноразовая» и «практически однократная» считаются вполне взаимозаменяемыми.

Идемпотентная запись — это одна из наиболее часто поддерживаемых функциональностей среди реализаций брокеров событий, таких как Apache Kafka и Apache Pulsar. Они позволяют записывать событие в поток событий один и только один раз. В случае отказа производителя или брокера событий во время записи события функциональность идемпотентной записи обеспечивает, чтобы при повторной попытке дубликат этого события не создавался.

Вашим брокером событий также могут поддерживаться и *транзакции*. В настоящее время полную транзакционную поддержку предлагает только Apache Kafka, хотя ожидается, что Apache Pulsar также обретет свою собственную реализацию. Подобно тому, как реляционная база данных может поддерживать множественные обновления в одной транзакции, реализация брокера событий также может поддерживать атомарную запись многочисленных событий в многочисленные отдельные потоки событий. Это позволяет производителю публиковать свои события в многочисленных потоках событий в одной атомарной транзакции. Конкурирующие реализации брокера событий, которым не хватает транзакционной поддержки, требуют, чтобы клиент обеспечивал свою собственную практическую однократную обработку. Далее мы рассмотрим оба этих варианта и посмотрим, как вы сможете использовать их для своих собственных микросервисов.



Транзакции чрезвычайно мощны и дают Apache Kafka значительное преимущество перед конкурентами. В частности, они могут приспосабливаться к новым бизнес-требованиям, которые в противном случае потребовали бы сложной переработки кода с целью обеспечения атомарности.

Пример: сервис учета запасов

Сервис учета запасов отвечает за выдачу оповещения о том, что запасы какого-либо товара находятся на низком уровне. Микросервис должен собрать воедино текущий запас по каждому продукту на основе цепочки поступлений и выбытий, произошедших с течением времени. Продажа товаров клиентам, потеря товаров из-за повреждения и потеря товаров из-за кражи — все это события, которые уменьшают запасы, в то время как получение поставок и прием возвратов от клиентов их увеличивают. Для простоты в нашем примере эти события показаны в одном и том же потоке событий (рис. 7.8).

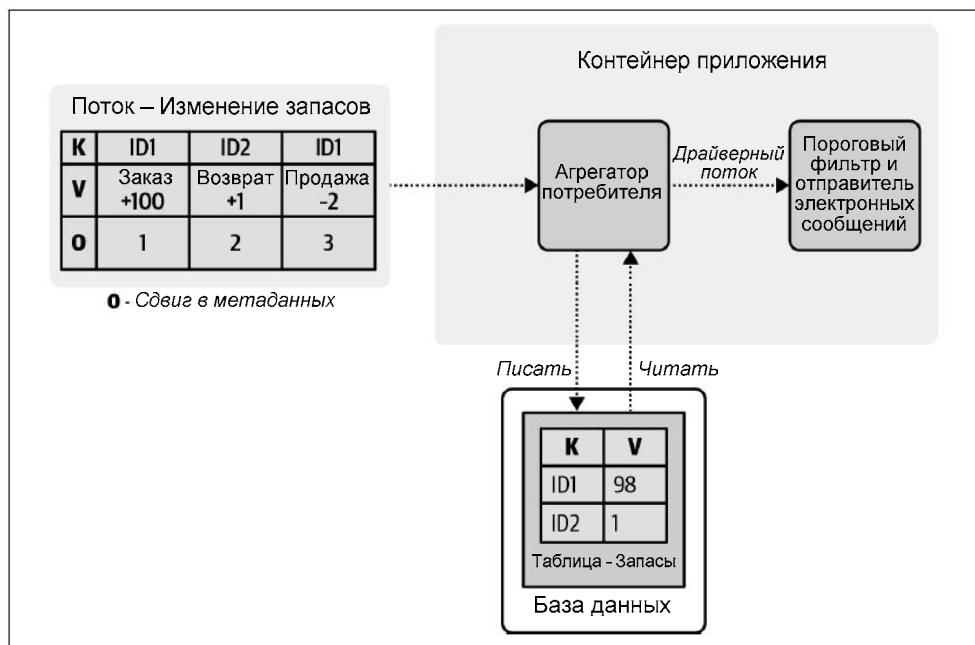


Рис. 7.8. Простой сервис учета запасов

Рассматриваемый здесь сервис учета запасов весьма прост. Он вычисляет текущую скользящую сумму запасов на основе изменений потока событий и сохраняет ее в своем хранилище данных. Бизнес-логика выполняет фильтрацию по пороговому значению и принимает решение о том, следует или нет выдавать оповещение службе управления запасами о низком или перепроданном запасе. При этом необходимо обеспечить, чтобы каждое входное событие практически применялось к агрегированному состоянию только один раз, т. к. применять его более одного раза неправильно, как и не применять его вообще. Именно здесь в игру вступает практически однократная обработка.

Практически однократная обработка с транзакциями клиент-брокер

Практически однократную обработку может обеспечить любой брокер событий, поддерживающий транзакции. При таком подходе любые выходные события и обновления *внутреннего состояния*, поддерживаемые журналом изменений, а также увеличение смещения потребителя обертываются вместе в одну атомарную транзакцию. Это возможно только в том случае, когда все три эти обновления хранятся в своем собственном потоке событий в брокере: обновление смещения, обновление журнала изменений и выходное событие фиксируются атомарно в рамках одной транзакции, как показано на рис. 7.9.

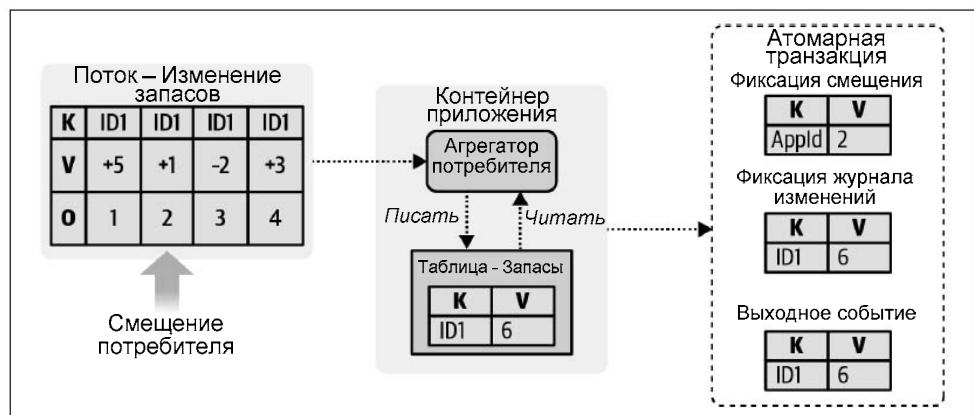


Рис. 7.9. Транзакции клиент-брокер — фиксация смещений и журналов изменений

Атомарная транзакция между клиентом-производителем и брокером событий публикует все события в соответствующие им потоки событий. В случае долговременного отказа со стороны производителя брокер будет следить за тем, чтобы ни одно из событий в транзакции не было зафиксировано (рис. 7.10). Потребители потока событий обычно воздерживаются от обработки событий, находящихся в незафиксированных транзакциях. Потребитель должен соблюдать порядок смещения, и поэтому он будет блокировать транзакцию, ждать ее завершения и лишь затем приступать к обработке события. В случае кратковременных ошибок производитель может просто повторить попытку совершения своей транзакции, поскольку эта операция является идемпотентной.

В случае, когда производитель сталкивается во время транзакции с фатальной исключительной ситуацией, его заменяющий экземпляр может быть просто перестроен путем восстановления из журналов изменений, как показано на рис. 7.11. Смещения группы потребителей входных потоков событий также сбрасываются в соответствии с последней известной не ошибочной позицией, сохраненной в потоке событий смещения.

Новые транзакции могут начаться после восстановления производителя, а все предыдущие незавершенные транзакции будут завершены отказом и очищены брокером.

ром событий. Механизмы транзакций могут в некоторой степени различаться в зависимости от реализации брокера, поэтому обязательно ознакомьтесь с правилами работы того, который вы используете.

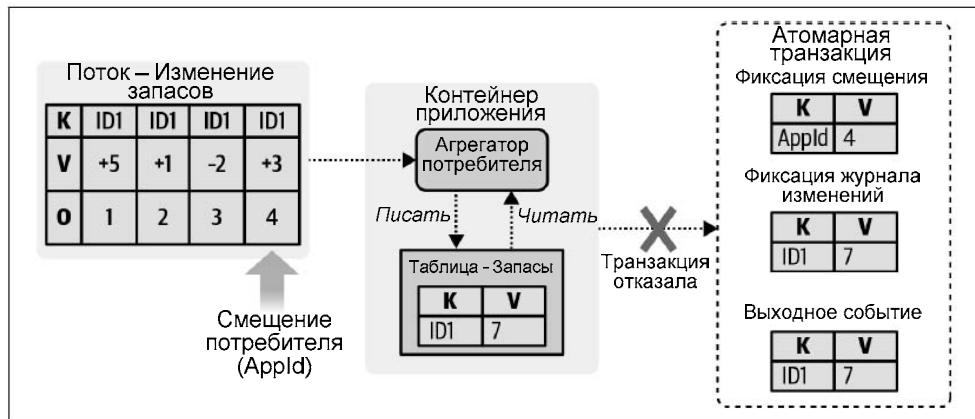


Рис. 7.10. Отказавшая фиксация транзакции клиент-брокер

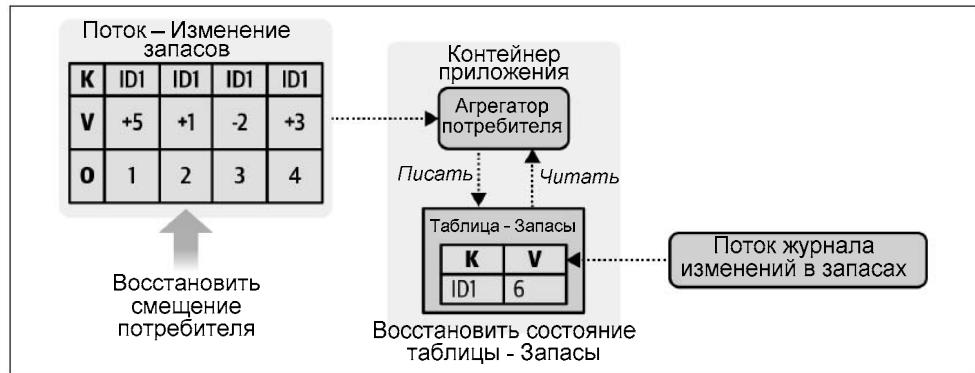


Рис. 7.11. Восстановление состояния из брокера с помощью журналов изменений и предыдущих смещений

Практически однократная обработка без транзакции клиент-брокер

Практически однократная обработка событий также возможна и для реализаций, которые не поддерживают транзакции клиент-брокер, хотя это требует дополнительной работы и внимательного рассмотрения событий на предмет появления их дубликатов. Во-первых, если вышеуказанные сервисы не могут предоставлять гарантии практической однократной обработки событий, то вполне возможно, что они способны создавать их дубликаты. При этом любые дублирующиеся события, созданные вышеуказанными процессами, должны быть выявлены и отфильтрованы. Во-

вторых, управление состоянием и смещением должно обновляться в локальной транзакции с целью обеспечения того, чтобы обработка событий применялась к состоянию системы только один раз. Следуя этой стратегии, клиенты могут быть уверены в том, что внутреннее состояние, генерируемое их обработчиком, согласуется с логикой входных потоков событий. Давайте рассмотрим эти шаги подробнее.



Лучше использовать брокер событий и клиент, которые поддерживают идемпотентную запись, чем пытаться решить проблему дубликатов постфактум. Первый метод хорошо масштабируется для всех потребительских приложений, в то время как второй является дорогостоящим и трудным для масштабирования.

Генерация дублирующихся событий

Дублирующиеся события генерируются, когда производитель успешно записывает события в поток событий, но либо не получает подтверждения записи и повторяет попытку, либо дает сбой перед обновлением собственных потребительских смещений. Эти сценарии немного различаются:

- ◆ *Производитель не получает подтверждения от брокера и повторяет попытку.*
В этом сценарии у производителя по-прежнему есть копии событий, которые он должен произвести в своей памяти. Эти события, если они будут опубликованы снова, могут иметь те же временные метки (если в них используется время создания событий) и те же событийные данные, но им будут назначены новые смещения.
- ◆ *Производитель аварийно дает сбой сразу после записи, прежде чем обновить свои собственные потребительские смещения.*

В этом случае производитель успешно записывает свои события, но не успевает обновить свои потребительские смещения. Это означает, что когда производитель восстановится, он повторит работу, которую делал ранее, создавая логически идентичные копии событий, но с новыми временными метками. Если обработка является детерминированной, то события будут иметь те же самые данные. Кроме того, будут назначены новые смещения.



Идемпотентное производство поддерживается многочисленными брокерами событий и может помочь при отказах из-за аварийных выходов из строя и повторных попыток записи, как в двух рассмотренных здесь сценариях. Однако оно не способно решить проблемы дублирования, возникающие из-за неправильной бизнес-логики.

Идентификация дублирующихся событий

Если идемпотентное производство событий недоступно и в потоке событий есть дубликаты (с уникальными смещениями и уникальными метками времени), то вы должны сами смягчить их влияние. Во-первых, определите, действительно ли дубликаты вызывают какие-либо проблемы. Во многих случаях дубликаты имеют незначительный, если не минимальный, эффект и могут быть просто проигнорированы. Для тех сценариев, где дублирующиеся события действительно вызывают про-

блемы, вам нужно будет выяснить, как их идентифицировать. Один из способов сделать это — побудить производителя присваивать каждому событию уникальный идентификатор (ID), благодаря чему любые дубликаты будут генерировать один и тот же уникальный хеш.

Такая хеш-функция часто основывается на свойствах внутренних данных события, включая ключ, значения и время создания события. Этот подход, как правило, хорошо работает для событий, имеющих значительную область данных, но плохо для событий, которые логически эквивалентны друг другу. Вот несколько сценариев, в которых вы можете генерировать уникальный ID:

- ◆ денежный перевод с банковского счета с указанием источника, места назначения, суммы, даты и времени;
- ◆ заказ в электронном магазине с подробным описанием каждого продукта, покупателя, даты, времени, итоговой суммы и поставщика платежей;
- ◆ запасы, дебитованные для целей отгрузки, где с каждым событием ассоциирован ID заказа (используется существующий уникальный ID данных).

Одним из факторов, который связывает все эти примеры, является то, что каждый ID состоит из элементов с очень высокой *мощностью* (т. е. уникальностью). Это значительно уменьшает шансы появления дубликатов этих идентификаторов. ID дедупликации (*dedupe ID*) может быть либо сгенерирован вместе с событием, либо присвоен потребителем при его получении, причем первый вариант предпочтительнее для распространения среди всех потребителей.



Защита от дублирующихся событий, произведенных без ключа, чрезвычайно сложна, поскольку нет никакой гарантии локальности раздела. Создавайте события с ключом, соблюдайте локальность раздела и используйте идемпотентные записи всякий раз, когда это возможно.

Защита от дубликатов

Любой потребитель с практически однократной обработкой должен либо идентифицировать и отбрасывать дубликаты, выполняя идемпотентные операции, либо потреблять из потоков события, имеющие идемпотентных производителей. Идемпотентные операции возможны не для всех бизнес-случаев, так что вы должны найти способ защитить свою бизнес-логику от дублирующихся событий и без идемпотентного производства. Такое мероприятие может быть дорогостоящим, поскольку оно требует, чтобы каждый потребитель поддерживал хранилище состояний ранее обработанных идентификаторов дедупликации. Хранилище может стать очень большим в зависимости от объема событий и смещения или временного диапазона, от которых приложение должно защищаться.

Идеальная дедупликация требует, чтобы каждый потребитель в течение неопределенного времени вел поиск каждого уже обработанного идентификатора дедупликации, но если будет предпринята попытка защиты от слишком большого диапазона событий, требования по времени и пространству для этого могут стать чрезмерно дорогими. На практике дедупликация обычно выполняется только для определенного скользящего временного окна или окна смещения.



Поддерживайте приемлемый размер хранилища дедупликации с помощью времени жизни (time-to-live, TTL), максимального размера кэша и периодических удалений. Необходимые конкретные настройки будут варьироваться в зависимости от чувствительности вашего приложения к дубликатам и влияния возникающих дубликатов.

Дедупликация должна предприниматься только в пределах одного раздела потока событий, т. к. дедупликация между разделами будет непомерно дорогой. События с ключами имеют дополнительное преимущество перед событиями без ключей, поскольку они будут согласованно распределяться по одному и тому же разделу.

На рис. 7.12 показано хранилище дедупликации в действии. Здесь вы видите рабочий процесс, через который событие проходит, прежде чем перейти к фактической бизнес-логике. В этом примере время жизни (TTL) произвольно устанавливается в 8000 секунд, но на практике его необходимо назначать на основе бизнес-требований.

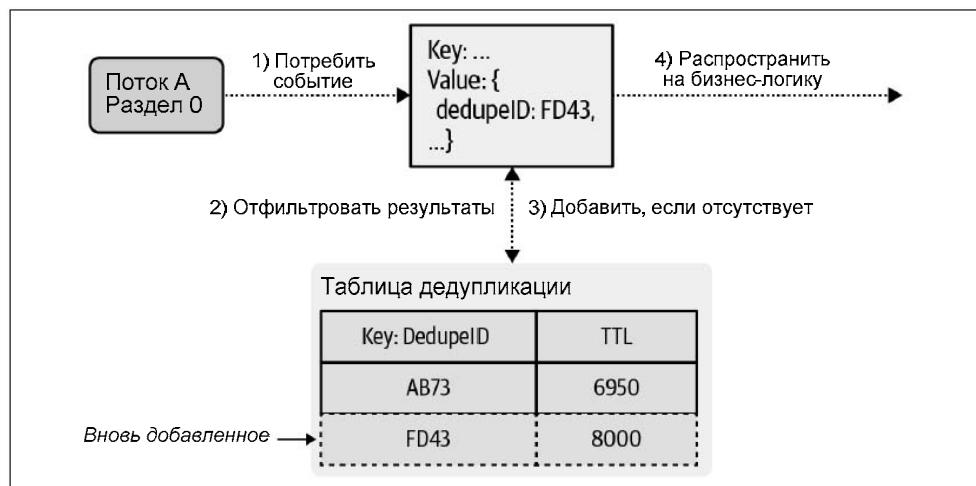


Рис. 7.12. Дедупликация с использованием сохраняемого состояния



Максимальный размер кэша используется в хранилище дедупликации для ограничения числа поддерживаемых событий — в особенности во время обработки.

Обратите внимание, что вы несете ответственность за поддержание надежных резервных копий таблицы дедупликации, как и за любую другую материализованную таблицу. В случае отказа таблица должна быть перестроена до того, как возобновит обработку новых событий.

Поддержание согласованного состояния

Микросервисы могут задействовать для практически однократной обработки транзакционные возможности своего хранилища состояний вместо брокера событий.

Это требует перемещения управления смещением группы потребителей из брокера событий в сервис данных, позволяя одной транзакции с поддержкой состояния атомарно обновлять как смещения состояния, так и смещения входных данных. Любые изменения, вносимые в состояние, полностью совпадают с изменениями, вносимыми в потребительские смещения, что обеспечивает согласованность внутри сервиса.

В случае отказа сервиса, такого как тайм-аут при фиксации в сервисе обработки данных, микросервис может просто отказаться от транзакции и вернуться к последнему известному безошибочному состоянию. Все потребление прекращается до тех пор, пока сервис обработки данных не начнет реагировать, и в этот момент потребление восстанавливается с последнего известного безошибочного смещения. Поддерживая официальную запись смещений синхронизированной с данными в сервисе обработки данных, вы получаете постоянное представление о состоянии, из которого сервис может восстанавливаться. Этот процесс проиллюстрирован на рис. 7.13–7.15.

Обратите внимание, что этот подход дает вашему обработчику практически однократную обработку, но не практически однократное производство событий. Любые события, производимые этим сервисом, подпадают под ограничения по меньшей мере однократного производства, т. к. нетранзакционное производство событий клиент-брюкер подвержено созданию дубликатов.

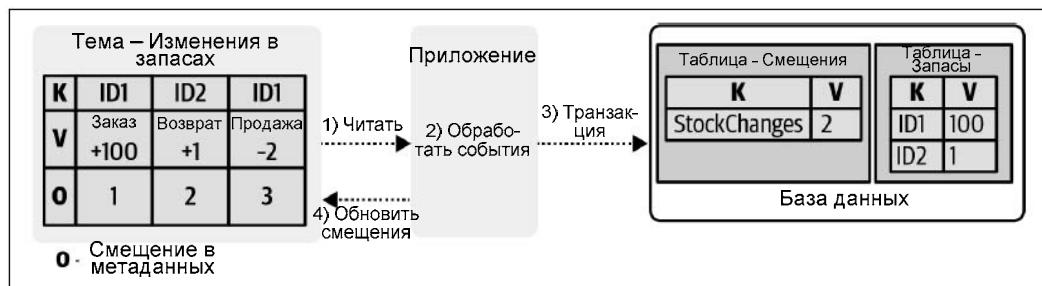


Рис. 7.13. Нормальная транзакционная обработка событий

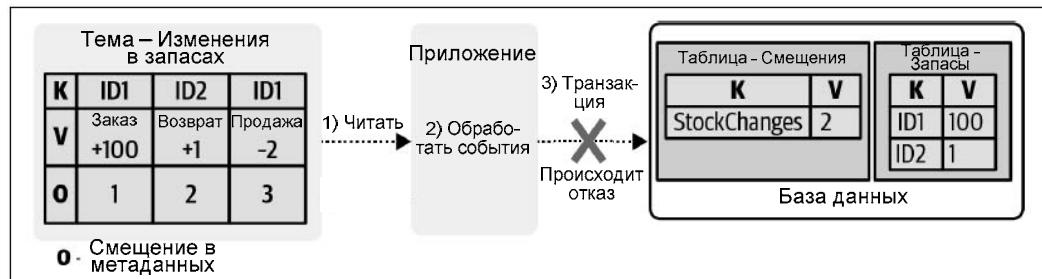


Рис. 7.14. В транзакционной обработке происходит отказ

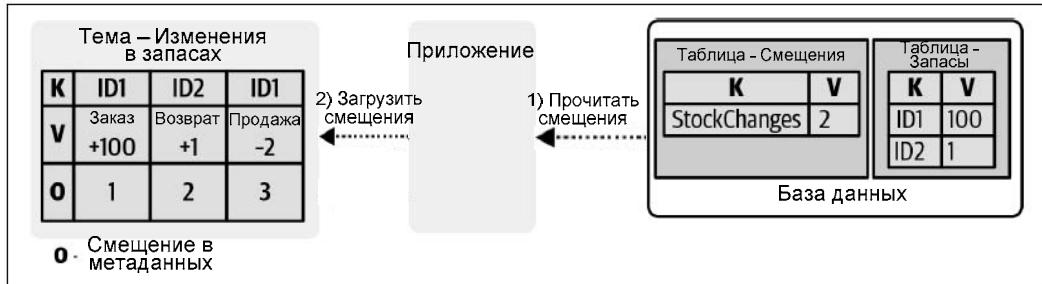


Рис. 7.15. Восстановление смещений в процессе восстановления состояния

Если вы должны иметь события, которые транзакционно создаются вместе с обновлениями в хранилище состояний, то обратитесь к рекомендациям, приведенным в главе 4. Использование таблицы изменений в данных может обеспечить в конечном счете согласованное, практически однократное обновление хранилища состояний с по меньшей мере однократным производством в выходной поток событий.

Резюме

В этой главе рассматривались внутренние и внешние хранилища состояний, характер их работы, их преимущества, недостатки и случаи, когда их следует использовать. Локальность данных играет большую роль в задержке и пропускной способности систем, позволяя им масштабироваться во время большой нагрузки. Внутренние хранилища состояний способны поддерживать высокопроизводительную обработку, в то время как внешние хранилища состояний могут предоставлять широкий спектр гибких вариантов для поддержания бизнес-требований ваших микросервисов.

Журналы изменений играют важную роль в резервном копировании и восстановлении хранилищ состояний микросервисов, хотя эта роль может также выполняться поддерживающими транзакции базами данных и регулярно планируемыми моментальными снимками. Брокеры событий, поддерживающие транзакции, могут обеспечивать чрезвычайно мощную практическую однократную обработку, снимая ответственность за предотвращение дублирования с потребителя, в то время как в системах без такой поддержки обеспечивать практическую однократную обработку могут специально предпринимаемые усилия по дедупликации.

Построение рабочих процессов с помощью микросервисов

Микросервисы по самому своему определению работают лишь в небольшой части общего бизнес-процесса организации. Так что *рабочий процесс* (workflow) — это определенный набор действий, составляющих бизнес-процесс, включающий любые логические ветвления и компенсирующие действия. Рабочие процессы обычно требуют многочисленных микросервисов, каждый из которых имеет свой собственный ограниченный контекст, выполняющий свою работу и создающий новые события для потребителей. Большая часть того, с чем мы знакомились до сих пор, касалась работы отдельных микросервисов «под капотом» общих решений. Теперь мы собираемся взглянуть на то, как несколько микросервисов могут работать вместе для выполнения более крупных бизнес-процессов, а также на некоторые «подводные камни» и проблемы, возникающие из подхода на основе событийно-управляемых микросервисов.

Вот несколько главных соображений, которые помогут нам разобраться с реализацией рабочих процессов *управления данными предприятия* (Enterprise Data Management, EDM):

◆ *Создание и модификация процессов:*

- как связаны сервисы в рамках процесса;
- как мы можем изменить существующие процессы без:
 - нарушения работы, которая уже идет;
 - необходимости изменений в многочисленных микросервисах;
 - нарушения мониторинга и видимости?

◆ *Мониторинг процессов:*

- как определить, когда процесс для события завершен;
- как определить успешность/неуспешность обработки события или что оно застряло где-то в процессе;
- как мониторить совокупное состояние процесса?

◆ *Реализация распределенных транзакций:*

- многие процессы требуют, чтобы ряд действий выполнялся одновременно или не выполнялся вообще. Как реализовать распределенные транзакции;
- как откатывать распределенные транзакции назад?

В этой главе рассматриваются два главных шаблона рабочих процессов: хореография и оркестровка — и дается их оценка с учетом приведенных соображений.

Шаблон «хореография»

Термин «хореографическая архитектура» (также известный как «реактивная архитектура») обычно относится к сильно изолированным (т. е. слабо связанным) микросервисным архитектурам. В рамках шаблона хореографии микросервисы реагируют на свои входные события по мере их поступления, без какой-либо блокировки или ожидания, полностью независимо от любых вышестоящих производителей или последующих потребителей. Это очень похоже на танцевальное представление, где каждый танцор должен знать свою собственную роль и выполнять ее самостоятельно, без какого-либо управления или указаний, что ему делать во время танца.

Хореография широко распространена в архитектуре событийно-управляемых микросервисов. Событийно-управляемые архитектуры сосредоточены исключительно на том, чтобы предоставлять *многократно используемые событийные потоки* революционной бизнес-информации, где потребители могут появляться и исчезать без каких-либо нарушений высокого рабочего процесса. Весь обмен информацией осуществляется строго через входные и выходные потоки событий. Производитель в системе с хореографической архитектурой не знает, ни кто является потребителем его данных, ни какую бизнес-логику или операции он намерен выполнять. Вышестоящая бизнес-логика полностью изолирована от нижестоящих потребителей.

Хореография желательна в большинстве случаев обмена информацией между командами, поскольку она позволяет создавать слабо связанные системы и снижает требования к координации между ними. Поскольку многие бизнес-процессы независимы друг от друга и не требуют строгой координации, это делает подход на основе шаблона хореографии идеальным для обмена информацией. Новые микросервисы можно легко добавлять в архитектуру на основе шаблона хореографии, в то время как существующие могут так же легко удаляться.

Рабочий процесс хореографической архитектуры определяют связи между микросервисами. Так, серия работающих вместе микросервисов может отвечать за обеспечение бизнес-функциональности процесса. Такой хореографический рабочий процесс является формой *эмерджентного поведения*¹, где не только отдельные микросервисы управляют рабочим процессом, но и связи между ними.



Архитектуры микросервисов прямого вызова сосредоточены на предоставлении повторно используемых сервисов, которые будут задействованы в качестве строительных блоков для бизнес-процессов. В то же время архитектуры событийно-управляемых микросервисов сосредоточены на предоставлении повторно используемых сервисов.

¹ Эмерджентность (от англ. emergent — возникающий, неожиданно появляющийся) — появление у системы свойств, не присущих ее элементам в отдельности; несводимость свойств системы к сумме свойств ее компонентов. — Прим. ред.

заемых событий, не зная заранее о нижестоящем потреблении. Это свойство таких архитектур позволяет использовать в высшей степени отцепленные, хореографические архитектуры.

Важно отметить, что хореография относится к сфере событийно-управляемых архитектур, поскольку отцепление сервисов производителя от сервисов потребителя позволяет им выполнять свои функции независимо. Сравните это с архитектурой микросервисов прямого вызова, где в фокусе внимания находится предоставление повторно используемых сервисов для создания более обширной бизнес-функциональности и процессов и где один микросервис напрямую вызывает API другого. По определению это означает, что вызывающий микросервис должен знать две вещи:

- ◆ *какой* сервис должен быть вызван;
- ◆ *почему* сервис должен быть вызван (*ожидаемая бизнес-ценность*).

Как видите, микросервис прямого вызова тесно сцеплен и полностью зависит от ограниченных контекстов существующего микросервиса.

Простой пример событийно-управляемой хореографии

На рис. 8.1 показан выход из процесса на основе хореографического шаблона, в котором сервис А посыпает данные непосредственно в сервис В, который, в свою очередь, посыпает данные в сервис С. В этом конкретном случае вы можете сделать логический вывод о том, что сервисы связаны зависимым процессом: A → B → C. Выход сервиса С указывает на результат процесса в целом.

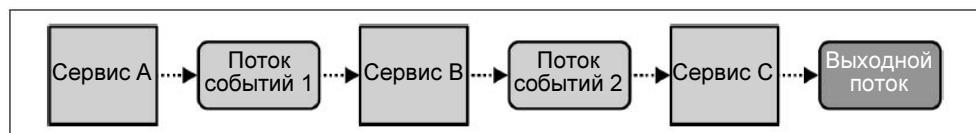


Рис. 8.1. Простой событийно-управляемый рабочий процесс на основе шаблона хореографии

Теперь предположим, что рабочий процесс должен быть перекомпонован таким образом, чтобы бизнес-действия в сервисе С выполнялись раньше, чем в сервисе В (рис. 8.2).

Оба сервиса: С и В — для чтения из потоков 1 и 2 должны быть изменены. Формат данных в потоках может больше не соответствовать потребностям нового процесса, требуя критических изменений схемы, которые могут существенно повлиять на других потребителей потока событий 2 (здесь не показаны). Для потока событий 2 может потребоваться создать совершенно новую схему событий, при этом старые данные в потоке событий будут перенесены в новый формат, удалены или оставлены на месте. Наконец, необходимо обеспечить, чтобы обработка была завершена для всех выходных событий сервисов А, В и С, прежде чем менять местами топологию, во избежание неполной обработки некоторых событий и их оставления в незавершенном состоянии.

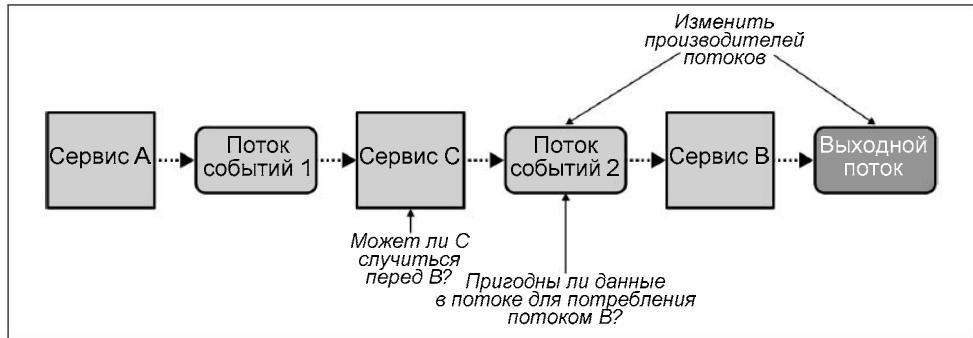


Рис. 8.2. Бизнес-изменения, необходимые для простого событийно-управляемого рабочего процесса на основе шаблона хореографии

Создание и изменение хореографического рабочего процесса

В то время как хореография позволяет просто добавлять новые шаги в конец процесса, вставлять шаги в его середину или же изменять порядок процесса иногда достаточно проблематично. Связи между сервисами также бывает трудно понять вне контекста процесса, и эта проблема усугубляется по мере увеличения числа сервисов в процессе. Хореографические рабочие процессы могут быть нестабильными, особенно когда бизнес-функции пересекаются с несколькими экземплярами микросервисов. Такие ситуации можно смягчить за счет тщательного соблюдения ограниченных контекстов сервисов и обеспечения того, чтобы полная бизнес-функциональность оставалась локальной для одного сервиса. Однако даже при правильной реализации изменений бизнес-логики может потребоваться доработать многочисленные сервисы, в особенности те, которые изменяют порядок самого процесса.

Мониторинг хореографического рабочего процесса

При мониторинге хореографического рабочего процесса необходимо учитывать его масштаб и объем. В изоляции распределенные процессы на основе хореографии могут затруднить процесс обработки конкретного события. В событийно-управляемых системах мониторинг критически важных бизнес-процессов может потребовать прослушивания каждого выходного потока событий и материализации его в хранилище состояний, чтобы вести учет того, где событие не удалось обработать или где оно застряло в обработке.

Например, изменение порядка процесса, показанного на рис. 8.2, требует также изменения системы видимости процесса. Это предполагает, что наблюдателей интересует *каждый* поток событий в этом рабочем процессе и они хотят знать о каждом его событии все. Но как быть с видимостью в процессах, где вас, возможно, *не интересует* каждый поток событий в процессе? Как насчет процессов, распределенных по всей организации?

Возьмем теперь гораздо более масштабный пример — такой как процесс выполнения заказа в крупном международном интернет-магазине. Клиент просматривает товары, выбирает некоторые из них для покупки, производит оплату и ожидает оповещения о доставке. Для поддержки этого процесса работы с клиентами может быть задействовано много десятков или даже сотен сервисов. Видимость такого процесса будет варьироваться в зависимости от потребностей наблюдателя.

Клиента, возможно, в действительности интересует только то, где находится заказ в процессе от оплаты до его исполнения и оповещения о доставке. Вы можете мониторить это, создавая события из трех отдельных потоков событий. Такой подход был бы достаточно устойчивым к изменениям как из-за его «публичной» природы, так и из-за малого числа потоков событий, из которых он потребляет. Между тем представление полного сквозного процесса может потребовать использования десятков потоков событий. Выполнить это будет сложнее как из-за объема событий, так и из-за независимости потоков событий, в особенности если процесс подвержен регулярным изменениям.



Убедитесь, что вы знаете, что именно пытаетесь сделать видимым в хореографическом рабочем процессе. У разных наблюдателей разные требования, и не все этапы рабочего процесса могут требовать явного воздействия.

Шаблон «оркестровка»

В шаблоне оркестровки (orchestration pattern) центральный микросервис-оркестратор выдает команды подчиненным микросервисам и ожидает от них ответов. Оркестровку можно представить себе как симфонический оркестр, где дирижер руководит оркестрантами во время исполнения ими музыкального произведения. Оркестратор содержит всю логику рабочего процесса для того или иного бизнес-процесса и отправляет конкретные события обрабатывающим микросервисам, чтобы сообщить им, что делать.



Оркестратор ожидает ответов от обрабатывающих микросервисов и обрабатывает результаты в соответствии с логикой рабочего процесса. В этом заключается отличие процесса на основе шаблона оркестровки от процесса на основе шаблона хореографии, в котором отсутствует централизованная координация.

Шаблон оркестровки позволяет гибко определять рабочий процесс в рамках отдельного микросервиса. Оркестратор отслеживает то, какие части процесса были завершены, какие находятся в процессе, а какие еще не запущены. Он также выдает командные события подчиненным микросервисам, которые выполняют требуемую задачу и предоставляют результаты оркестратору, как правило, путем выдачи запросов и ответов через поток событий.

Если платежный микросервис пытается исполнить платеж три раза, прежде чем отказать, то он должен сделать эти три попытки *внутренними* для платежного микросервиса. Он не делает *одну* попытку, чтобы затем оповестить оркестратора о том, что произошел отказ, и не ждет, что ему скажут повторять попытку или нет. Орке-

стратор не должен иметь права голоса в решении, как обрабатываются платежи, и в том числе сколько попыток делать, поскольку это является частью ограниченного контекста платежного микросервиса. Единственное, что должен знать оркестратор, так это то, что оплата полностью прошла или не прошла. Исходя из этого, он может действовать соответствующим образом, основываясь на логике процесса.



Убедитесь, что ограниченный контекст оркестратора строго лимитирован логикой рабочего процесса и содержит минимальную бизнес-логику выполнения. Сам оркестратор включает только логику рабочего процесса, в то время как сервисы под управлением оркестратора содержат основную часть бизнес-логики.

Обратите внимание, что бизнес-обязанности микросервиса, не связанного с оркестровкой, в оркестрованном шаблоне идентичны обязанностям того же микросервиса в хореографическом шаблоне. Оркестратор отвечает только за оркестровку и логику рабочего процесса, а вовсе не за выполнение бизнес-логики самих микросервисов. Давайте рассмотрим простой пример, иллюстрирующий эти ограничения.

Простой пример событийно-управляемой оркестровки

На рис. 8.3 приведена оркестрованная версия архитектуры, показанной на рис. 8.1.

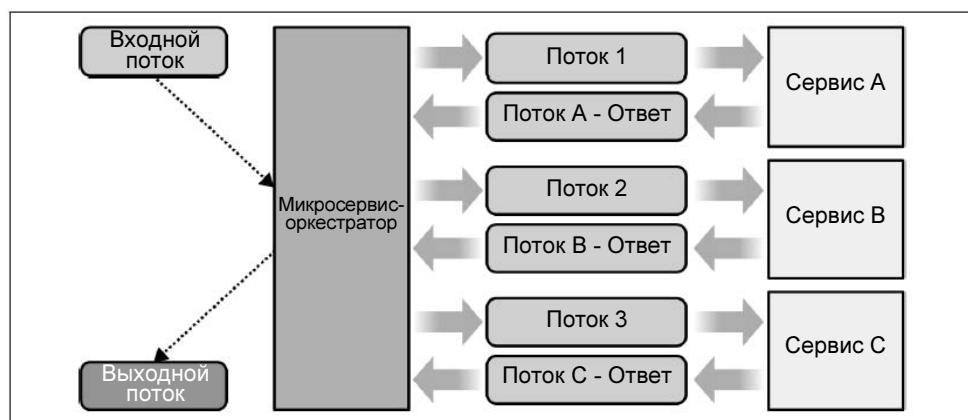


Рис. 8.3. Простой оркестрованный событийно-управляемый рабочий процесс

Оркестратор сохраняет материализацию событий, созданных сервисами А, В и С, и обновляет свое внутреннее состояние на основе результатов, возвращаемых рабочим микросервисом (табл. 8.1).

Таблица 8.1. Материализация событий, выдаваемых оркестратором

ID входного события	Сервис А	Сервис А	Сервис А	Статус
100	<результаты>	<результаты>	<результаты>	Сделано
101	<результаты>	<результаты>	Отправлено	Обработка
102	Отправлено	null	null	Обработка

ID события 100 был успешно обработан, в то время как ID событий 101 и 102 находится на разных стадиях процесса. Оркестратор может принимать решения на основе этих результатов и выбирать следующий шаг в соответствии с логикой процесса. Как только события обработаны, оркестратор может также брать необходимые данные из результатов сервисов A, B и C и формировать окончательный вывод. Исходя из того, что операции в сервисах A, B и C независимы друг от друга, вы можете внести изменения в процесс, просто изменив порядок отправки событий. В следующем далее фрагменте оркестровочного кода события просто потребляются из каждого входного потока и обрабатываются в соответствии с бизнес-логикой процесса:

```

while (true) {
    Event[] events = consumer.consume(streams)

    for (Event event : events) {
        if (event.source == "Входной поток") {
            //обработать событие + обновить материализованное состояние
            producer.send("Поток 1", ...) // Отправить данные в поток 1
        } else if (event.source == "Поток 1-Ответ") {
            //обработать событие + обновить материализованное состояние
            producer.send("Stream 2", ...) // Отправить данные в поток 2
        } else if (event.source == "Поток 2-Ответ") {
            //обработать событие + обновить материализованное состояние
            producer.send("Stream 3", ...) // Отправить данные в поток 3
        } else if (event.source == "Поток 3-Ответ") {
            // Обработать событие, обновить материализованное состояние
            // и построить выходные данные
            producer.send("Выход", ...) // Отправить результаты в выход
        }
    }
    consumer.commitOffsets()
}

```

Простой пример оркестровки с прямым вызовом

Оркестровка также может использовать шаблон «запрос-ответ», в котором оркестратор синхронно вызывает API микросервисов и ожидает ответа для получения результатов. Топология, показанная на рис. 8.4, почти идентична топологии, показанной на рис. 8.3, если не считать подстановки прямых вызовов.

Здесь также действуют обычные преимущества и ограничения сервисов с прямым вызовом. Тем не менее этот шаблон особенно полезен для реализации рабочих процессов с решениями на основе технологии «Функция как сервис» (см. главу 9).

Событийно-управляемая оркестровка в сравнении с оркестровкой с прямым вызовом

Оркестровка событийно-управляемых процессов (event-driven workflows) и оркестровка процессов с прямым вызовом (direct-call workflows) весьма схожи, если рас-

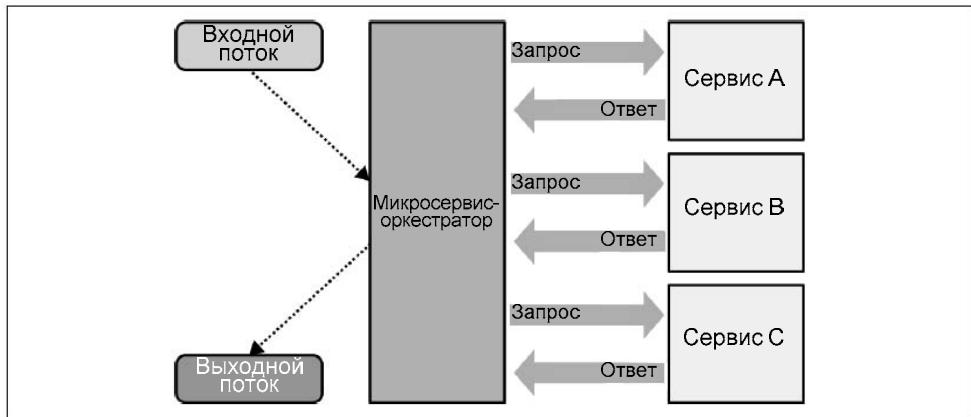


Рис. 8.4. Простой оркестрованный рабочий процесс прямого вызова

сматривать их в общем. Например, может показаться, что событийно-управляемая система на самом деле является просто системой «запрос-ответ», и в простых примерах это действительно так. Но когда вы глубже вникаете в ситуацию, появляется целый ряд факторов, которые следует учитывать при выборе того, какой вариант использовать:

◆ события-управляемые процессы:

- могут использовать тот же инструментарий мониторинга ввода/вывода и функциональность масштабирования, что и другие событийно-управляемые микросервисы;
- разрешают потокам событий по-прежнему потребляться другими сервисами, включая те, которые находятся вне оркестровки;
- как правило, более надежны, поскольку и оркестратор, и зависимые сервисы изолированы от периодических сбоев друг у друга через брокер событий;
- имеют встроенный механизм повторных попыток обработки при отказах, поскольку события могут оставаться в потоке событий для выполнения повторных попыток обработки.

◆ процессы прямого вызова:

- как правило, более быстрые, т. к. у них нет никаких накладных расходов на производство и потребление потоков событий;
- имеют периодические проблемы с подключением, которые, возможно, придется решать с помощью оркестратора.

Подводя итог, можно сказать, что процессы прямого вызова, также именуемые *синхронными потоками «запрос-ответ»*, как правило, действуют быстрее, чем событийно-управляемые, при условии, что все зависимые сервисы функционируют в рамках своих SLA. Они, как правило, хорошо работают, когда требуется очень быстрая обработка данных — например, в реальном времени. Между тем событийно-управляемые процессы имеют более устойчивые потоки ввода/вывода, выпол-

няя более медленные, но более надежные операции, и особенно хорошо справляются с кратковременными отказами.

Имейте в виду, что существует весьма много возможностей для смешивания и сочетания этих двух вариантов. Например, процесс на основе оркестровки может быть преимущественно событийно-управляемым, но требует, чтобы вызов процедуры «запрос-ответ» выполнялся непосредственно к внешнему API или уже существующему сервису. При соединении этих двух вариантов обеспечьте, чтобы отказы каждого сервиса обрабатывались должным образом.

Создание и изменение рабочего процесса на основе оркестровки

Оркестратор может отслеживать события в рабочем процессе, материализуя каждое из входящих и исходящих потоков событий и результатов вызова процедуры «запрос-ответ». Сам рабочий поток определяется исключительно в рамках оркестровочного сервиса, что позволяет использовать единую точку изменения для изменения рабочего процесса. Во многих случаях можно вносить изменения в рабочий процесс, не нарушая частично обработанные события.

Оркестровка приводит к тесной сцепленности между сервисами. При этом связи между оркестратором и зависимыми бизнес-сервисами должны быть явно определены.

Важно убедиться, что оркестратор отвечает только за оркестровку бизнес-процесса. Часто встречается неверный подход, заключающийся в создании единого сервиса типа «Бог» («God» service), который выдает отдельные команды многим подчиненным микросервисам. Этот подход распределяет бизнес-логику рабочего процесса между оркестратором и сервисами, что приводит к плохой инкапсуляции, нечетким ограниченным контекстам и трудностям масштабирования микросервисов бизнес-процесса за пределами команды разработчиков. Оркестратор должен делегировать полную ответственность подчиненным сервисам, чтобы минимизировать объем выполняемой им бизнес-логики.

Мониторинг процесса на основе оркестровки

Вы получаете видимость рабочего процесса оркестровки, запрашивая материализованное состояние, поэтому легко можете увидеть ход процесса любого конкретного события и любые проблемы, которые могли возникнуть в рабочем процессе. Вы можете реализовать мониторинг и ведение журнала на уровне оркестратора для обнаружения любых событий, которые приводят к сбоям рабочего процесса.

Распределенные транзакции

Распределенная транзакция (distributed transaction) — это транзакция, охватывающая два или более микросервиса. Каждый микросервис отвечает за обработку своей части транзакции, а также за отмену этой обработки в случае отмены транзакции

и ее отката. И логика исполнения, и логика отмены должны находиться в одном и том же микросервисе, как в целях технического сопровождения, так и для обеспечения того, чтобы новые транзакции не могли запускаться, если их также нельзя будет откатить назад.



Лучше всего избегать реализации распределенных транзакций, когда это возможно, поскольку они могут добавлять в процесс значительный риск и сложность. Вы должны учитывать целый ряд трудностей — таких как синхронизация работы между системами, облегчение отката, управление временными отказами экземпляров сервисов и сетевого подключения, и это лишь некоторые из имеющихся.

Тем не менее, несмотря на рекомендацию избегать, когда это возможно, реализации распределенных транзакций, они все же применяются и могут потребоваться при некоторых обстоятельствах, в особенности когда отказ от их применения привел бы к еще большему риску и сложности.

Распределенные транзакции в событийно-управляемом мире часто называются *сагами* и могут быть реализованы с помощью либо хореографического шаблона, либо шаблона оркестровки.

Шаблон «Сага» требует, чтобы участвующие микросервисы могли обрабатывать действия по откату, отменяющие свою часть транзакции. Как обычные действия обработки, так и действия по откату должны быть идемпотентными, чтобы любые периодические сбои участвующих микросервисов не оставляли систему в несогласованном состоянии.

Хореографические транзакции: шаблон «Сага»

Распределенные транзакции с хореографическими рабочими процессами могут быть сложными, поскольку каждый сервис должен иметь возможность откатывать изменения в случае сбоя. Это создает сильную связь между слабо связанными сервисами и может привести к тому, что некоторые не связанные службы приобретут сильную зависимость друг от друга.

Хореографический шаблон саги подходит лишь для простых распределенных транзакций — в особенности имеющих строгие требования к упорядоченности процесса, которые вряд ли изменятся с течением времени. Мониторинг исполнения транзакции с хореографией может быть затруднен, поскольку требует полной материализации каждого участвующего потока событий, как и в подходе с оркестровкой.

Пример хореографической транзакции

Продолжая предыдущий пример хореографического рабочего процесса, рассмотрим серию микросервисов A, B и C. Входной поток событий для обслуживания сервиса A запускает транзакцию, при этом работа сервисов A, B и C либо полностью завершается, либо отменяется и откатывается назад. Отказ на любом этапе цепочки отменяет транзакцию и начинает ее откат. Результирующий процесс показан на рис. 8.5.

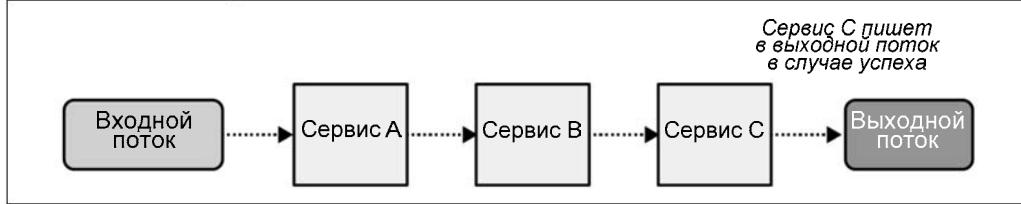


Рис. 8.5. Успешная хореографическая транзакция

Предположим теперь, что сервис С не может завершить свою часть транзакции. Тогда он должен полностью перевернуть рабочий процесс, либо отправив события, либо ответив на запрос предыдущего сервиса. Сервисы В и А также должны вернуть свои части транзакции, как показано на рис. 8.6.

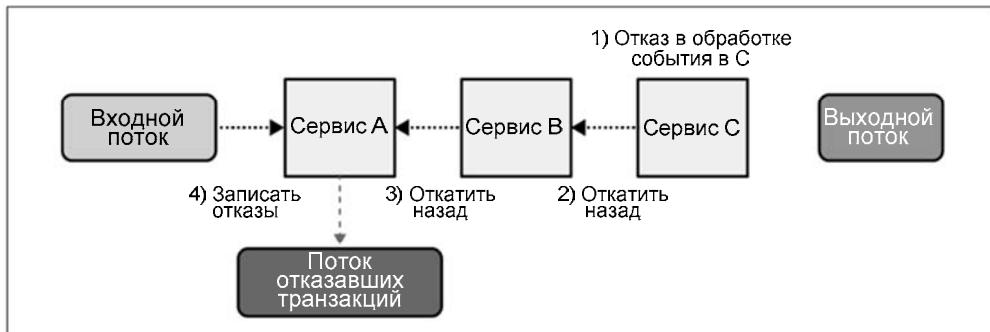


Рис. 8.6. Отказ хореографической транзакции с ее откатом

Сервис А, исходный потребитель входного события, теперь должен решить, что делать с результатами отказавшей транзакции. Эту любопытную ситуацию как раз и иллюстрируют два предыдущих рисунка: статус успешной транзакции исходит из вывода сервиса С (см. рис. 8.5), однако статус прерванной транзакции исходит из сервиса А (см. рис. 8.6), поэтому потребителю необходимо прослушивать как вывод сервиса С, так и поток неудачной транзакции из сервиса А, чтобы получить полную картину завершенных транзакций.



Вспомните принцип единственного автора. В поток событий может публиковать не более чем один сервис.

Даже потребляя сообщения как из выходных потоков, так и из потоков отказавших транзакций, потребитель все равно не сможет получить статус текущих транзакций или транзакций, застрявших в обработке. Это потребовало бы материализации каждого потока событий или того, чтобы внутреннее состояние каждого микросервиса было бы представлено через API, как обсуждалось ранее в этой главе. Внесение изменений в рабочий процесс хореографической транзакции требует решения тех же проблем, что и для нетранзакционного рабочего процесса, но с дополнительны-

ми накладными расходами на откат изменений, внесенных в рабочий процесс предыдущим микросервисом.

Хореографические транзакции могут быть в некотором смысле неустойчивыми, обычно требуют строгого упорядочивания и могут быть проблематичными для отслеживания. Они лучше всего работают в сервисах с очень небольшим количеством микросервисов — обычно с двумя или тремя — с очень строгим порядком и низкой вероятностью необходимости изменения рабочего процесса.

Транзакции с оркестровкой

Оркестрованные транзакции основаны на модели оркестратора с добавлением логики для отмены транзакции из любой точки рабочего процесса. Вы можете откликнуть эти транзакции назад, изменив логику рабочего процесса и обеспечив, чтобы каждый микросервис был способен предпринять дополнительное действие отмены транзакции.

Транзакции с оркестровкой также могут поддерживать различные сигналы — такие как тайм-ауты и ввод данных человеком. Вы можете использовать тайм-ауты для периодической проверки локального материализованного состояния, чтобы узнать, как долго транзакция обрабатывалась. Вводимые человеком через REST API данные (см. главу 13) могут обрабатываться наряду с другими событиями с выполнением отменяющих инструкций по мере необходимости. Централизованная природа оркестратора позволяет внимательно мониторить выполнение и состояние любой конкретной транзакции.

Транзакция может быть отменена в любой момент рабочего процесса из-за возвращаемого значения в одном из рабочих микросервисов, тайм-аута или прерывания, отправленного человеком-оператором.

Простая двухэтапная топология оркестрованных транзакций показана на рис. 8.7.

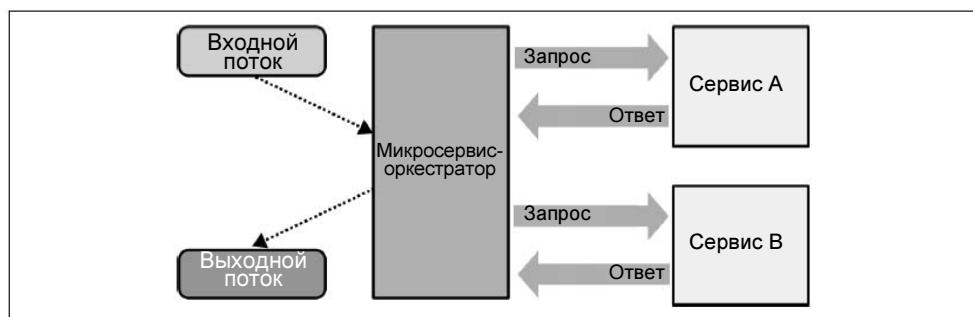


Рис. 8.7. Простая топология оркестрованных транзакций

Как можно видеть, здесь события потребляются из входного потока и обрабатываются оркестратором. Оркестратор использует прямые вызовы микросервисов рабочего процесса в форме «запрос-ответ»: делается запрос к сервису А и оркестратор

блокируется в ожидании ответа. Получив ответ от сервиса А, оркестратор обновляет свое внутреннее состояние и вызывает сервис В, как показано на рис. 8.8.

Здесь ситуация иная — сервис В не может выполнить необходимую операцию и, исчерпав свои собственные повторные попытки и логику обработки ошибок, в конечном счете возвращает в оркестратор ответ с отказом. Теперь уже оркестратор должен задействовать свою логику отката транзакции, основанную на текущем состоянии события, обеспечивая выдачу команд отката всем необходимым микросервисам.

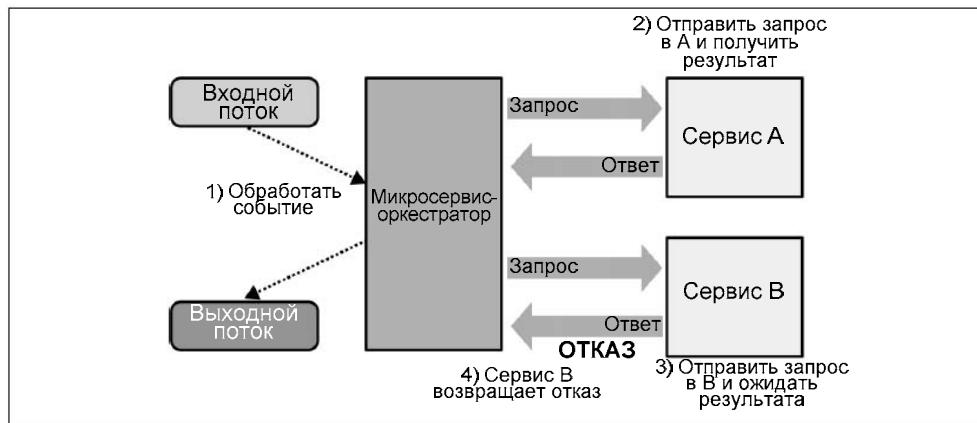


Рис. 8.8. Простая оркестрованная транзакция с отказом в процессе транзакции



Каждый микросервис несет полную ответственность за обеспечение собственной политики повторных попыток, обработку ошибок и управление кратковременными отказами. Оркестратор же не управляет ни одной из упомянутых операций.

На рис. 8.9 показано, как оркестратор выдает команду отката сервису А (ответ сервиса В с отказом, показанный на рис. 8.8, говорит о том, что он ничего не записал

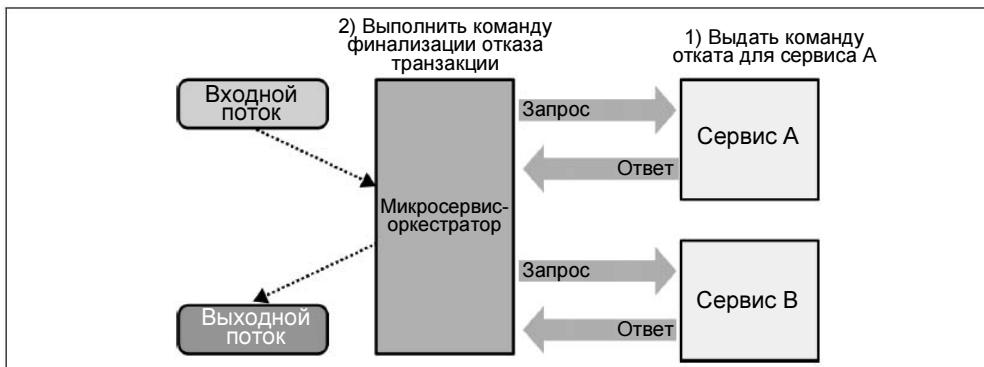


Рис. 8.9. Выдача команд отката в оркестрованной транзакции

во внутреннее хранилище данных). В этом примере сервис А успешно выполняет откат, но если бы он отказал во время отката, то оркестратор должен был бы определить, что делать дальше. Оркестратор может повторно ввести команду несколько раз, выдать предупреждение с помощью систем мониторинга или завершить работу приложения, чтобы предотвратить дальнейшие проблемы.

Итак, сразу после отката транзакции оркестратор должен решить, что делать дальше, чтобы завершить обработку этого события. Он может повторить событие несколько раз, отбросить его, завершить работу приложения или вывести событие отказа. Оркестратор, будучи единственным производителем, публикует откат транзакции в выходном потоке и позволяет нижестоящему потребителю его обработать. Этот подход отличается от хореографического, где нет единого потока, из которого можно потреблять все выходные данные, не отбросив принцип единственного автора.



Помимо того, что каждый микросервис полностью отвечает за свои собственные изменения состояния, он также отвечает и за то, чтобы его состояние было после отката согласованным. Ответственность оркестратора в этом сценарии ограничивается выдачей команд отката и ожиданием подтверждений со стороны зависимых микросервисов.

Оркестратор также может отображать состояние выполняемых транзакций в том же выходном потоке, обновляя субъект транзакции по мере того, как рабочие сервисы возвращают результаты. За счет этого может обеспечиваться высокая видимость состояния опорных транзакций и осуществление потокового мониторинга.

Оркестрованные транзакции обеспечивают более четкую видимость зависимостей рабочих процессов, более высокую гибкость для изменений и более четкие параметры мониторинга, чем транзакции на основе хореографии. Экземпляр оркестратора добавляет накладные расходы в рабочий процесс и требует управления, но может обеспечить сложным рабочим процессам ясность и структуру, которые не способны обеспечить транзакции с хореографией.

Компенсационные процессы

Не все процессы должны быть полностью отменяемыми и ограниченными транзакциями. В том или ином рабочем процессе может возникнуть много непредвиденных проблем, и во многих случаях вам, возможно, просто придется сделать все возможное, чтобы его завершить. И на случай отказа также есть действия, которые вы можете предпринять постфактум, чтобы исправить ситуацию.

В системах продажи билетов и инвентаризации этот подход используется часто. Например, веб-сайт, который продает физические продукты, может не иметь их достаточного запаса на момент продажи для обработки серии одновременных транзакций. После приема платежей и оценки имеющихся запасов розничный торговец может обнаружить, что для выполнения принятых заказов имеющихся запасов недостаточно. На этот случай у него есть несколько вариантов.

Строгий подход, основанный на транзакциях, потребовал бы отката самых последних транзакций — при этом деньги клиенту были бы возвращены и он был бы проинформирован о том, что товар отсутствует на складе и его заказ отменен. Хотя технически это правильно, такой подход может привести к оттоку клиентов и потере доверия у клиента к розничному продавцу. *Компенсационный рабочий процесс* может исправить ситуацию на основе политики, предусматривающей удовлетворение запросов клиентов.

Применяя компенсационный подход, предприятие может заказать новый запас товара, оповестить клиента о задержке, извиниться и предложить ему скидку на следующую покупку. Клиенту может быть также предоставлена возможность отменить заказ или дождаться пополнения запаса товара. Музыкальные, спортивные и различного рода концертные площадки часто используют такой подход в случае перепроданности билетов, это также делают авиакомпании и туристические агентства. Компенсационные процессы не всегда возможны, но они часто полезны для обработки распределенных рабочих процессов с продуктами, ориентированными на клиента.

Резюме

Хореографический шаблон допускает слабую связность между бизнес-единицами и независимыми процессами. Он подходит для простых распределенных транзакций и простых нетранзакционных рабочих процессов, где количество микросервисов невелико и порядок выполнения операций вряд ли когда-либо изменится.

Оркестрованные транзакции и процессы обеспечивают более четкую видимость и мониторинг рабочих процессов, чем хореография. Они способны обрабатывать более сложные распределенные транзакции и часто могут модифицироваться только в одном месте. Рабочие процессы, которые подвержены изменениям и содержат много независимых микросервисов, хорошо подходят для оркестровочного шаблона.

Наконец, не все системы требуют для успешной работы распределенных транзакций. Некоторые процессы могут обеспечить компенсационные действия в случае отказа, опираясь на организационные мероприятия в решении проблем, связанных с обслуживанием клиентов.

Микросервисы с использованием технологии «Функция как сервис»

Технология «Функция как сервис» (Functions-as-a-Service, FaaS) — это «бессерверное» решение, которое в последние годы становится все популярнее. Решения FaaS позволяют отдельным лицам создавать, управлять, развертывать и масштабировать функциональность приложений без необходимости нести накладные расходы на инфраструктуру. Они могут принести значительную пользу системам, управляемым событиями, выступая как средство реализации простых и умеренно сложных решений.

Функция — это фрагмент кода, который выполняется при возникновении определенного триггерного условия. Функция запускается, работает до завершения, а затем прекращает работу. Решения FaaS могут легко масштабировать число вызовов функций вверх и вниз в зависимости от нагрузки, обеспечивая тщательное отслеживание сильно изменяющихся нагрузок.

Полезно думать о решении FaaS как о базовой реализации взаимосвязи потребителя и производителя, которая регулярно дает сбой. Функция всегда завершается через заданный промежуток времени, и все связанные с ней соединения и состояния исчезают. Помните об этом при разработке своих функций.

Проектирование функционально-ориентированных решений в качестве микросервисов

Решения FaaS могут включать в себя множество различных функций, при этом сумма их операций составляет решение для ограниченного бизнес-контекста. Существует много способов создания функционально-ориентированных решений — гораздо больше, чем может охватить эта глава, но есть несколько общих принципов проектирования, которые помогут вам в этом процессе.

Обеспечивайте строгое членство в ограниченном контексте

Составляющие решение функции и внутренние потоки событий должны строго принадлежать ограниченному контексту, чтобы владелец функции и данных был четко идентифицирован. При реализации решений с участием микросервисов в больших количествах у организаций часто возникают вопросы о владельце функции, сервиса и потоков событий. В то время как многие решения на основе микро-

сервисов соотносятся 1:1 с ограниченным контекстом, не является редкостью и соотношение n:1, поскольку для одного ограниченного контекста могут использоваться многочисленные функции. Важно выявить, какая функция какому ограниченному контексту принадлежит, потому что высокая степень гранулярности функций может осложнить понимание.

Вот несколько практических способов поддержки ограниченных контекстов с функциями:

- ◆ обеспечивать, чтобы хранилища данных оставались закрытыми от внешних контекстов;
- ◆ использовать стандартные интерфейсы «запрос-ответ» или событийно-управляемые интерфейсы при соединении с другими контекстами;
- ◆ поддерживать строгие метаданные о том, какая функция какому контексту принадлежит (соотнесение функции с продуктом 1:1);
- ◆ поддерживать код функции внутри репозитория, соответствующего ограниченному контексту.

Фиксируйте смещения только после завершения обработки

Смещения фиксируются (commit) в один из двух моментов: при запуске функции или после того, как она завершила свою обработку. Фиксация смещений только после завершения обработки того или иного события или пакета событий является наилучшим приемом FaaS. Вам важно понимать, как происходит смещение в конкретных функционально-ориентированных решениях, поэтому давайте рассмотрим последствия каждого подхода.

Когда функция завершила свою обработку

Подход с фиксацией смещений после завершения обработки согласуется с тем, как другие реализации микросервисов фиксируют смещения, независимо от того, основаны ли они на базовом производителе/потребителе или фреймворке потоковой обработки. Эта стратегия обеспечивает самую сильную гарантию того, что события будут обработаны хотя бы один раз, и эквивалентна стратегиям управления смещением, используемым с решениями, не относящимися к FaaS.

Когда функция только что запустилась

Смещения могут фиксироваться и после того, как пакет обрабатываемых событий был передан функции. Этот простой подход используется во многих фреймворках FaaS, которые опираются на специфические для фреймворка механизмы повторных попыток и оповещения с целью смягчения повторяющихся отказов обработки событий. Функции, вызывающие другие функции в хореографическом шаблоне, часто опираются на эту стратегию, поскольку она значительно упрощает отслеживание обработки событий.

Однако фиксация смещений до завершения обработки может принести проблемы. Если функция не в силах успешно обработать событие и несколько попыток завершаются неудачно, вполне вероятна потеря данных. Такое событие обычно помещается в очередь недоставленных сообщений или просто отбрасывается. И хотя многие микросервисы, основанные на функциях, не чувствительны к потере данных, чувствительные к потере данных микросервисы не должны использовать эту стратегию.

Меньше значит больше

Часто упоминаемая особенность фреймворков FaaS заключается в том, что они позволяют легко написать одну функцию и повторно использовать ее в нескольких сервисах. Однако следование этому подходу может привести к сильно фрагментированному решению, которое затрудняет точное распознавание того, что происходит в ограниченном контексте. Кроме того, владение такой функцией становится неоднозначным, и может быть неясно, могут ли изменения функции отрицательно повлиять на другие сервисы. Хотя управление версиями функций может помочь в решении этой проблемы, оно также способно привести к конфликтам, когда несколько продуктов должны поддерживать и улучшать разные версии функции.

Решения FaaS могут включать в себя несколько функций для решения бизнес-требований ограниченного контекста, однако хорошим эмпирическим правилом для решений по технологии FaaS является использование как можно меньшего количества функций. Тестировать и отлаживать только одну функцию, а также управлять ею намного проще, чем делать то же самое для нескольких функций.

Выбор провайдера FaaS

Так же как и брокеры событий и системы управления контейнерами (Container Management Systems, CMS), фреймворки FaaS доступны и в качестве бесплатных решений с открытым исходным кодом, и в качестве платных продуктов облачных провайдеров. Организация, которая действует собственную внутреннюю CMS для своих операций на основе микросервисов, может также получить выгоду от работы с собственным решением FaaS. Ряд бесплатных вариантов с открытым исходным кодом, такие как OpenWhisk, OpenFaaS и Kubeless, способны использовать существующие сервисы управления контейнерами. А вот технология Apache Pulsar предлагает свое собственное встроенное решение FaaS, которое работает вместе со своим брокером событий. Используя общий фреймворк предоставления ресурсов, ваше решение FaaS может стать в ряд с вашим решением, работающим на микросервисах.

Провайдеры сервисов, такие как Amazon Web Services (AWS), Google Cloud Platform (GCP) и Microsoft Azure, также имеют свои собственные проприетарные фреймворки FaaS, каждый из которых предлагает привлекательные свойства и функциональность, но остается тесно интегрированным с проприетарным брокером событий своего провайдера. Это существенная проблема, потому что в настоящее

время все три провайдера ограничивают хранение данных в своих брокерах событий семью днями. Интеграция между облачными провайдерами и брокерами событий с открытым исходным кодом действительно существует (например, Kafka Connect (<https://oreil.ly/74sum>)), но может потребовать дополнительных усилий по ее настройке и управлению. Тем не менее если ваша организация уже является субподрядчиком сервисов AWS, GCP или Azure, то накладные расходы на запуск системы FaaS будут невелики.

Построение микросервисов из функций

При работе с функционально-ориентированными решениями вы должны учитывать четыре главных компонента, независимо от вашего фреймворка FaaS или брокера событий:

- ◆ функцию;
- ◆ входной поток событий;
- ◆ триггерную логику;
- ◆ политики ошибок и масштабирования с учетом метаданных.

Первым компонентом реализации FaaS является сама функция. Она может быть создана на любом языке, поддерживаемом фреймворком FaaS:

```
public int myfunction(Event[] events, Context context) {
    println ("здравствуй, мир!");
    return 0;
}
```

Параметр `events` содержит массив отдельных событий, подлежащих обработке, причем каждое событие включает `key`, `value`, `timestamp`, `offset` и `partition_id`. Параметр `Context` содержит информацию о функции и ее контексте — такую как ее имя, ID потока событий и оставшееся время жизни (`lifespan`) функции.

Затем для созданной функции нужно подключить триггерную логику. Это будет рассмотрено подробнее в следующем разделе, но пока предположим, что функция запускается всякий раз, когда новое событие поступает в один из ее потоков событий, на которые она подписана.

Логика запуска часто связана с функцией через карту запуска функции «функция-триггер», которая обычно скрыта за кулисами вашего фреймворка FaaS. Вот пример:

Функция	Поток(и) событий	Триггер	Политики и метаданные
myFunction	myInputStream	onNewEvent	< ... >

Как можно видеть, функция `myFunction` будет запущена, когда сработает триггер, — т. е. когда новое событие `onNewEvent` поступит в поток `myInputStream`. Здесь также присутствует столбец **Политики и метаданные**, содержащий следующие связанные параметры:

- ◆ группа потребителей;
- ◆ свойства потребителей — такие как размер пакета и окно пакета;
- ◆ политики повторных попыток и обработки ошибок;
- ◆ политика масштабирования.

Если триггеры, метаданные и политики установлены, функция готова к обработке входящих событий. Когда новое событие поступает в ее входной поток, функция запускается фреймворком FaaS, получает пакет событий и начинает обработку. По завершении функция останавливает работу и будет ждать появления новых событий. Это является типичной реализацией шаблона *слушателя потока событий* (*listener pattern*), который подробнее обсуждается в следующем разделе.



Каждый функционально-ориентированный микросервис должен иметь свою собственную независимую группу потребителей, как и в случае других микросервисов, не относящихся к FaaS.

Имейте в виду, что это всего лишь логическое представление компонентов, необходимых для успешного запуска и работы функции. Требования к созданию функций, управлению функциями и механизмам запуска со стороны фреймворка FaaS варьируются в зависимости от провайдера и реализации, поэтому обязательно обратитесь к документации для вашего фреймворка FaaS.

Существует также совокупность взаимодействий между механизмами запуска, потреблением событий, смещениями потребителей,ложенными функциями, отказами и по меньшей мере однократной (*at-least-once*) обработкой событий. Ее рассмотрение является предметом остальной части этой главы.

«Холодный старт» и «теплые старты»

«Холодный старт» (*cold start*) — это состояние функции по умолчанию при первом запуске или после достаточного периода бездействия. Контейнер должен быть запущен, и код загружен. Возможно, должны быть также созданы соединения с брокером событий и установлены любые другие клиентские соединения с внешними ресурсами. Как только все будет стабилизировано и готово к работе, функция приходит в «теплое» состояние и готова обрабатывать события. «Теплая» функция начинает обработку событий и по истечении срока или по завершении обработки приостанавливается и переводится в спящий режим.

Большинство фреймворков FaaS пытаются вторично использовать остановленные функции, когда это возможно. Во многих сценариях функция, обрабатывающая стабильный поток событий, достигнет истечения заданного тайм-аута и будет остановлена, чтобы через мгновение вернуться обратно с помощью триггерного механизма. Приостановленный экземпляр просто используется вторично, и при условии, что соединения с брокером событий и любыми хранилищами состояний не были разорваны в течение промежуточного периода, обработка может быть немедленно возобновлена.

Запуск функций с помощью триггеров

Триггеры используются, чтобы сообщить функции о том, что нужно запуститься и начать обработку. Поддерживаемые триггеры варьируются в зависимости от вашего фреймворка FaaS, но, как правило, все они попадают в одни и те же общие категории, как показано далее. А пока давайте посмотрим, какие сигналы можно задействовать для запуска функций, чтобы дать вам представление о том, как это все работает.

Запуск по новому событию: слушатель потока событий

Функции могут запускаться, когда событие создается в потоке событий. *Триггер слушателя потока событий* (event-stream listener trigger) скрывает потребление событий за существующей реализацией потребителя событий, уменьшая объем служебного кода, который должен писать разработчик. События поступают непосредственно в функцию в виде массива событий, в последовательном порядке из потока событий или в качестве кластера неупорядоченных событий, если они потребляются из очереди. Вы можете создать многочисленные соотнесения потоков событий с функциями, благодаря чему функция сможет потреблять события из многих разных потоков.

Решения FaaS от Google, Microsoft и Amazon предоставляют такой триггер для использования с их собственными брокерами событий, но в настоящее время не поддерживают запуск непосредственно от брокеров с открытым исходным кодом. Обобщенная структура этого подхода показана на рис. 9.1.

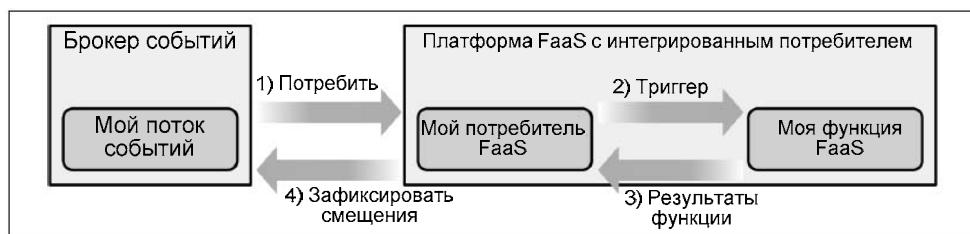


Рис. 9.1. Интегрированный слушатель потоков событий, поставляемый с фреймворком FaaS

И наоборот, решения с открытым исходным кодом — такие как OpenFaaS, Kubeless, Nuclio и другие, предоставляют разнообразные триггерные плагины с различными брокерами событий — такими как Kafka, Pulsar и NATS, и это только некоторые из них. Например, Apache Kafka Connect позволяет запускать функции сторонних фреймворков FaaS (<https://oreil.ly/y6zYz>). Поскольку Kafka Connect работает вне фреймворка FaaS, вы получите слушатель потока событий, как показано на рис. 9.2.

Хотя это и не показано в предыдущих примерах, результаты функций могут быть выведены в их собственные потоки событий не только для целей вывода данных, но и для отслеживания успеха функции.



Рис. 9.2. Внешнее приложение слушателя потоков событий, предоставляемое фреймворком Kafka Connect

Синхронные триггеры требуют, чтобы функция завершилась до того, как они выдаут следующие события. Это особенно важно для поддержания порядка обработки и ограничено параллельностью обрабатываемого потока событий. И наоборот, асинхронный запуск может выдавать многочисленные события многочисленным функциям, каждая из которых отчитывается по мере завершения. Однако такой режим *не будет* поддерживать порядок обработки и должен использоваться только тогда, когда порядок обработки не важен для бизнес-логики.

Размер пакета и *окно пакета* — это два важных свойства, которые следует учитывать в триггерах слушателя потока. Размер пакета определяет максимальное число событий для отправки на обработку, в то время как окно пакета указывает максимальное время ожидания дополнительных событий вместо немедленного запуска функции. Оба этих параметра используются для обеспечения того, чтобы накладные расходы на запуск функции распределялись между пакетами записей для снижения затрат.

Функция, исполняемая триggerом слушателя потока, выглядит примерно следующим образом:

```

public int myEventfunction(Event[] events, Context context) {
    for(Event event: events)
        try {
            println (event.key + ", " + event.value);
        } catch (Exception e) {
            println ("ошибка печати " + event.toString());
        }
    // Указывает фреймворку FaaS, что пакетная обработка была завершена.
    context.success();
    return 0;
}

```



Подобно контейнеризированному микросервису триггеры для шаблона слушателя потока событий могут быть сконфигурированы на запуск обработки событий из последних смещений потока, самых ранних смещений или где-либо между ними.

Запуск по задержке группы потребителей

Метрика *задержки группы потребителей* (consumer group's lag metric) — это еще один способ запуска функций. Вы можете обнаружить задержку, периодически опрашивая смещения отдельных групп потребителей приложения и вычисляя дельту между текущим смещением потребителя и начальным смещением потока (подробнее о мониторинге задержек см. в разд. «Мониторинг задержек смещения потребителей» главы 14). Подобно триггеру слушателя потока, мониторинг задержки также может использоваться для масштабирования микросервисов, не относящихся к FaaS.

Мониторинг задержки обычно предусматривает вычисление и представление метрик задержки в выбранный вами фреймворк мониторинга. Затем фреймворк мониторинга может вызвать фреймворк FaaS, чтобы сообщить ему о запуске функций, зарегистрированных в потоке событий. Высокое значение задержки может указывать на то, что для более быстрой обработки нагрузки могут быть запущены несколько экземпляров функции, в то время как низкое значение задержки может потребовать только одного экземпляра функции для обработки отставания. Вы можете регулировать соотношение между задержками и запуском функции в зависимости от того или иного микросервиса, обеспечивая соблюдение SLA.

Одно из важных различий между ранее упомянутым триггером слушателя потока событий и рассматриваемым здесь заключается в том, что при запуске по задержке функция не потребляет события до тех пор, пока не будет запущена. Функции, запускаемые триггером по задержке, имеют более широкую область ответственности, включая установление клиентского соединения с брокером событий, потребление событий и фиксацию любых обновлений смещения. Это делает запускаемые задержкой функции гораздо более похожими на базовые клиенты производителя/потребителя, хотя и с ограниченным сроком жизни. Следующий далее пример функции иллюстрирует этот рабочий процесс:

```
public int myLagConsumerfunction(Context context) {
    String consumerGroup = context.consumerGroup;
    String streamName = context.streamName;

    EventBrokerClient client = new EventBrokerClient(consumerGroup, ...);

    Event[] events = client.consumeBatch(streamName, ...);

    for(Event event: events) {
        // Выполнить работу по обработке события
        doWork(event);
    }
    // Зафиксировать смещения в брокере событий
    client.commitOffsets();

    // Указывает фреймворку FaaS, что функция выполнена успешно.
    context.success();
}
```

```
// Вернуться, сообщив системе запуска по задержке об успехе
return 0;
}
```

Группа потребителей и имя потока передаются в качестве параметров в контексте. Клиент создается, события потребляются и обрабатываются, а смещения фиксируются у брокера событий. Функция передает успешный результат обратно в фреймворк FaaS, а затем возвращается к обработке.

Если функция часто запускается монитором задержек, то существует шанс, что она все еще остается «теплой» с последней итерацией, и накладные расходы на подключение к клиенту брокера событий могут быть незначительными. Это, конечно, зависит от установленных тайм-аутов, используемых клиентом и конфигурациями брокера событий. При более длительных периодах бездействия перебалансирование группы потребителей и «холодный» запуск клиента несколько уменьшат объем работы, которую экземпляр функции может выполнять.

Запуск по расписанию

Функции также могут запускаться по расписанию в определенные дни и время. Запланированные расписанием функции запускаются с заданным интервалом, опрашивают исходные потоки событий на предмет новых событий и обрабатывают их или завершают работу по мере необходимости. Период опроса должен быть небольшим, чтобы поддерживать SLA, но слишком частый опрос может привести к чрезмерной нагрузке как на фреймворк FaaS, так и на брокер событий.

Клиентский код для триггера на основе расписания выглядит идентично тому, который был приведен в примере триггера по задержке группы потребителей.

Запуск с использованием веб-перехватчиков

Функции также могут запускаться прямым вызовом с использованием механизма получения уведомлений о событии, на которое подписано клиентское приложение (webhooks), что позволяет настраивать их интеграцию с фреймворками мониторинга, планировщиками и другими сторонними приложениями.

Запуск по событиям ресурсов

Изменения, вносимые в ресурсы, также могут быть источником триггеров. Например, создание, обновление или удаление файла в файловой системе может запускать функции, так же как и модификация строки данных в хранилище. Поскольку большинство событий в событийно-управляемых микросервисах генерируются посредством потоков событий, этот конкретный ресурсный триггер используется в бизнес-процессах не часто. Он, однако, весьма полезен, когда вы интегрируетесь с внешними источниками данных, которым требуется FTP или другой файловый сервис для передачи своих файлов.

Решение бизнес-задач с помощью функций

Решения, основанные на технологии FaaS, особенно хорошо подходят для ситуаций, в которых может быть задействован гибкий механизм выделения ресурсов обработки по требованию. Простые топологии являются при этом отличными кандидатами для использования, как и те технологии, которые не поддерживают состояние, не требуют детерминированной обработки нескольких потоков событий и масштабируются очень широко, — например, технологии обработки на основе очередей. Решения FaaS могут быть полезны для всего, что имеет сильно изменяемый объем, поскольку их возможности горизонтального масштабирования и характеристики по требованию позволяют быстро выделять и освобождать вычислительные ресурсы.

Решения FaaS могут работать очень хорошо, когда для вас не важны параллелизм и детерминизм. Однако как только детерминизм вступает в игру, вы должны очень внимательно следить за тем, чтобы обеспечить правильность и согласованность обработки потока событий. Подобно основным потребительским решениям из следующей главы, решения FaaS требуют наличия планировщика событий для обеспечения согласованных результатов обработки. Соподразделенные данные могут быть успешно и согласованно обработаны только одной функцией за раз, подобно тому, как полнофункциональные легкие и тяжелые фреймворки должны использовать только один поток.

Поддержание состояния

Учитывая короткий срок службы функций, большинство решений на основе FaaS с поддержкой состояния требуют использования внешней службы с поддержкой состояния. Частично причина заключается в том, что многие поставщики услуг FaaS стремятся предоставить быстрые и хорошо масштабируемые параметры вычислительной мощности независимо от местоположения данных. Наличие функций, которым требуется локальное состояние из предыдущих запусков, ограничивает текущее выполнение узлами, которые имеют это состояние в том же разделе. Такая зависимость значительно снижает гибкость поставщиков FaaS, поэтому они часто применяют политику «без локального состояния» (*«no local state» policy*) и требуют, чтобы все, имеющее состояние, хранилось вовне по отношению к обработчикам.

И хотя предыдущее локальное состояние может быть доступно, если функция запускается в «теплом» состоянии, нет никакой гарантии, что так будет всегда. Функции подключаются к внешним хранилищам состояний точно так же, как и любой другой клиент, — путем создания соединения с хранилищем состояний и использования соответствующего API. Любое состояние должно при этом сохраняться и извлекаться функцией явным образом.



Обязательно используйте строгие права доступа к состоянию вашей функции, чтобы ничего за пределами ее ограниченного контекста не было разрешено.

Некоторые фреймворки FaaS добавили сопровождение долговременных функций с поддержкой состояния — таких как *долговечные функции* Microsoft Azure's Durable Functions (<https://oreil.ly/ShsMI>), которые абстрагируют явное управление состоянием и дают возможность использовать локальную память, автоматически сохраняемую во внешнем состоянии. Это позволяет разработчикам приостанавливать функции и возвращать их к работе без необходимости писать код для явного хранения и извлечения состояния. Такой подход значительно упрощает бизнес-процессы с поддержкой состояния и предоставляет возможность автономного управления состоянием во всех реализациях функций.

Фреймворки FaaS будут продолжать развиваться и включать в себя новые функциональности. Простое управление состоянием является часто встречающейся потребностью в функционально-ориентированных решениях, поэтому следите за улучшениями обработки состояний в применяемых вами фреймворках FaaS.

Функции, вызывающие другие функции

Функции часто используются для исполнения других функций, а также могут за-действоваться в бизнес-процессах с применением шаблонов оркестровки или хореографии. Связь между функциями может обеспечиваться асинхронно через события, через вызовы «запрос-ответ» или же с помощью комбинации того и другого. Этот выбор в значительной степени зависит от фреймворка FaaS и пространства ограниченного контекста. При реализации многофункциональных решений обычно используются шаблоны архитектурного дизайна на основе оркестровки или хореографии.



Чтобы избежать проблем с обработкой событий не по очереди, прежде чем обрабатывать следующее событие, убедитесь, что вся обработка для предыдущего события завершена.

Шаблон событийно-управляемого обмена информацией

Выход одной функции может быть передан в поток событий для другой функции. Ограниченный контекст может состоять из многих функций и многих внутренних потоков событий с различной триггерной логикой и логикой масштабирования для каждой функции. Каждая функция обрабатывает входящие события со своей скоростью, при этом соответственным образом потребляются события, выполняется работа и производятся выходные данные. Пример такого подхода показан на рис. 9.3.

Функция А запускается здесь независимо от триггеров для функций В и С. Потоки событий 2 и 3 рассматриваются как внутренние потоки событий, причем доступ к их содержимому полностью запрещен любыми функциями вне ограниченного контекста. Каждая функция потребляет события из своего исходного потока, используя одну и ту же группу потребителей, поскольку все функции расположены в одном ограниченном контексте. Благодаря этому обеспечивается столь же эф-

фективная работа функции, что и микросервисов, не основанных на технологии FaaS.

Использование шаблона событийно-управляемого обмена информацией имеет ряд преимуществ. Каждая функция в топологии может управлять своими собственными смещениями потребителей, фиксируя каждое смещение после завершения своей работы. Вне обработки потока событий между функциями не должно быть никакой координации. На рис. 9.3 показан шаблон архитектурного дизайна, основанный на хореографии, хотя вы также можете использовать и шаблон оркестровки. В такой архитектуре любые отказы в обработке событий не приведут к потере данных, т. к. события надежно хранятся в брокере событий и просто будут обработаны следующим экземпляром функции.

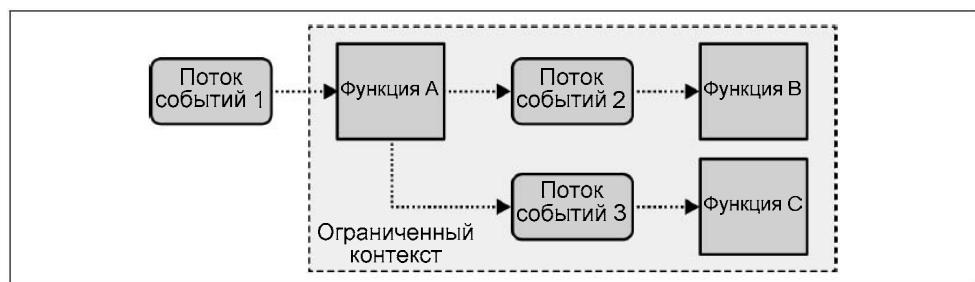


Рис. 9.3. Многофункциональная событийно-управляемая топология FaaS, представляющая собой единый микросервис

Шаблон прямого вызова

В шаблоне прямого вызова (direct-call pattern) функция может вызывать другие функции непосредственно из своего собственного кода. Прямой вызов других функций может выполняться асинхронно, что, по сути, является подходом «выстрелить и забыть», или синхронно, когдазывающая функция ожидает возвращаемого значения.

Хореография и асинхронный вызов функций

Асинхронные прямые вызовы реализуются в качестве решений FaaS на основе шаблона хореографии. Одна функция согласно своей бизнес-логике просто вызывает следующую, предоставляя ей и фреймворку FaaS возможность обрабатывать дальнейшие шаги, включая любые отказы или ошибки. Топология асинхронных функций с прямым вызовом — это простой способ соединения вызовов функций в цепочку (рис. 9.4).

Здесь функция А, обработав свой пакет событий, вызывает функцию В, после чего функция А может просто обновить свои потребительские смещения и завершить работу. Тем временем функция В продолжает свою обработку и выдает любые выходные данные в выходной поток событий.

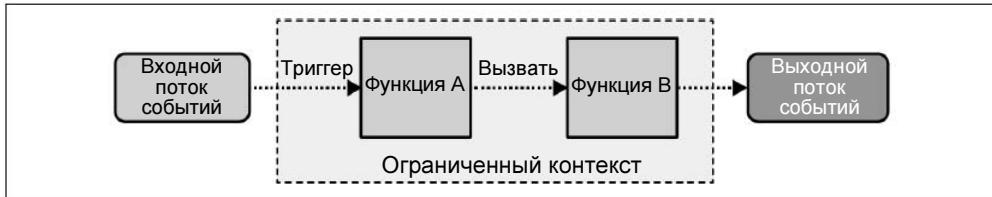


Рис. 9.4. Асинхронные вызовы функций на основе шаблона хореографии в ограниченном контексте

Одним из недостатков асинхронных прямых вызовов является то, что смещения потребителей обновляются только в случае успешной обработки. В приведенном на рис. 9.3 примере функция В не имеет обратной связи с функцией А, и поэтому только ошибки в функции А предотвратят неверную фиксацию смещений групп потребителей в рабочем процессе. Однако потеря событий может не иметь значения для некоторых бизнес-процессов, и в таких случаях эту проблему можно просто отклонить.

Еще одна потенциально серьезная проблема заключается в том, что события из-за многократных вызовов функции В могут обрабатываться не по порядку. Рассмотрим код функции А:

```

public int functionA(Event[] events, Context context) {
    for(Event event: events) {
        // Выполнить работу функции А
        // Вызывать функцию В асинхронно для каждого события
        // Не ждать возвращаемого значения
        asyncfunctionB(event);
    }

    context.success();
    return 0;
}

```

Функция В вызывается здесь командой из функции А. В зависимости от настройки вашего фреймворка FaaS это может привести к созданию многочисленных экземпляров функции В, каждый из которых работает независимо от других. Это создаст конфликтную ситуацию, при которой некоторые экземпляры функции В завершатся раньше других, что потенциально приведет к обработке событий не по порядку.

Точно так же код, приведенный далее, тоже не решит проблему упорядочения. Обработка все равно станет происходить не по порядку, т. к. функция А будет выполняться для всех событий в пакете перед выполнением функции В:

```

public int functionA(Event[] events, Context context) {
    for(Event event: events) {
        // Выполнить функцию А
    }

    // Вызвать функцию В асинхронно со всем пакетом событий целиком
    asyncFunctionB(events);
}

```

```

context.success()
return 0;
}

```

Обработка по порядку требует строгого выполнения функции А перед функцией В для каждого события и перед обработкой следующего события. Событие должно быть полностью обработано, прежде чем может быть начато следующее, — в противном случае, скорее всего, возникнет неопределенное поведение. Это особенно важно, если функции А и В действуют в одном и том же внешнем хранилище данных, поскольку функция В может опираться на данные, записанные функцией А.

Во многих случаях для нужд ограниченного контекста асинхронных вызовов недостаточно. В этих случаях подумайте о том, не является ли более подходящим вариантом использование синхронных вызовов на основе шаблона оркестровки.

Оркестровка и синхронный вызов функций

Синхронные вызовы функций позволяют вызывать другие функции и ожидать результатов, прежде чем продолжать выполнение оставшейся бизнес-логики. Реализовать такую возможность предоставляет нам шаблон оркестровки, рассмотренный в главе 8.

Обработка, запускаемая потоком событий

На рис. 9.5 показан пример функционально-ориентированной оркестровки внутри одного ограниченного контекста. Здесь одна функция-оркестратор запускается при поступлении новых событий в подразделенный поток событий и начинает обрабатывать входной пакет событий, последовательно отправляя событие для каждой функции.

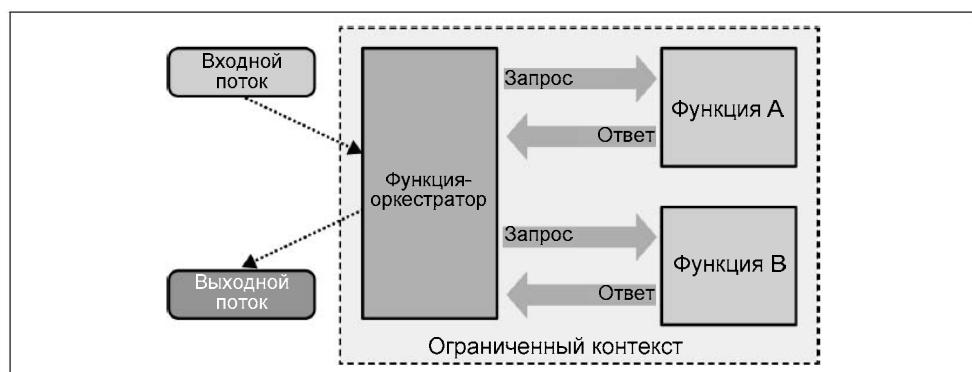


Рис. 9.5. Оркестрованные синхронные вызовы функций в ограниченном контексте

А вот пример соответствующего оркестровочного кода:

```

public int orchestrationFunction(Event[] events, Context context) {
    for(Event event: events) {

```

```
// Синхронные вызовы функций
Result resultFromA = invokeFunctionA(event);
Result resultFromB = invokeFunctionB(event, resultFromA);
Output output = composeOutputEvent(resultFromA, resultFromB);
// Записать в выходной поток
producer.produce("Output Stream", output);
}

// Это сообщит фреймворку FaaS, что нужно обновить потребительские смещения
context.success();
return 0;
}
```

Функция-оркестратор вызывает функции А и В по порядку и ожидает результатов от каждой функции. Выходы из функции А могут быть отправлены в функцию В, если это необходимо. Каждое событие полностью обрабатывается в бизнес-процессе перед запуском следующего, обеспечивая соблюдение порядка смещений. Как только функция завершит обработку каждого события в пакете, она может выдать сообщение об успехе и соответствующим образом обновить смещения.

Обработка событий, запускаемая очередью

Если вы используете очередь с возможностью индивидуальной фиксации, триггерный механизм может просто запускать индивидуальную функцию оркестровки для каждого события. Оркестратору необходимо будет отправить подтверждение обработки обратно в очередь после того, как он завершит свою работу. В случае, если оркестратор не сможет выполнить обработку, она будет просто перехвачена следующим созданным экземпляром оркестратора.

Завершение работы функции и выключение

Выполнение функции завершается, как только она заканчивает свою работу или достигает конца выделенного ей времени жизни (lifespan), обычно не превышающего 5–10 минут. Экземпляр функции приостанавливается и переходит в состояние гибернации, откуда он может быть немедленно восстановлен. Приостановленная функция также может быть в конечном счете удалена из кэша гибернации из-за ограничений ресурсов или времени.

Вам нужно будет решить, как обрабатывать любые открытые соединения и назначения ресурсов, сделанные в отношении функции до ее завершения. Так, в случае клиента-потребителя экземпляру функции могут быть назначены конкретные разделы потока событий. Тогда завершение экземпляра функции без отмены этих назначений может привести к задержкам обработки, т. к. владение ими в группе потребителей не будет переназначено до тех пор, пока не будет достигнут тайм-аут. События из этих разделов не станут обрабатываться до тех пор, пока не будет выполнена перебалансировка группы потребителей или же завершенный экземпляр функции не вернется в онлайн и не возобновит обработку.

Если ваша функция почти всегда находится онлайн и обрабатывает события, то закрытие соединений и перебалансирование группы потребителей могут оказаться

ненужными. Скорее всего, функция будет временно приостановлена в конце ее жизненного срока, ненадолго перейдет в режим гибернации и немедленно восстановится во время выполнения. И наоборот, для функции, которая работает только периодически, лучше всего закрыть все соединения и отказаться от назначения ей разделов потока событий. Следующий экземпляр функции должен будет заново создать соединение, независимо от того, является ли это «теплым запуском» или «холодным». Когда возникают сомнения, как правило, лучше всего очистить соединения — это облегчает нагрузку на внешние хранилища данных и на брокер событий и снижает вероятность того, что приостановленные функции будут претендовать на владение разделом.

Тонкая настройка функций

Каждая функция имеет конкретные потребности, основанные на ее рабочей нагрузке. Оптимизирование ресурсов, используемых функцией во время ее исполнения, может обеспечить высокую производительность при низких затратах. При формировании ресурсов и тонкой настройке параметров ваших функций следует учитывать несколько следующих моментов.

Выделение достаточных ресурсов

Каждой функции должен быть выделен *конкретный объем процессорного ресурса и памяти*. Важно настроить эти параметры в соответствии с потребностями вашей функции — избыточное выделение может оказаться дорогостоящим, в то время как недостаточное выделение может привести к отказу или слишком длительному выполнению ваших функций.

Максимальное время исполнения — вот еще один важный фактор, поскольку он ограничивает время выполнения функции. Этот параметр тесно связан с размером пакета, т. к. время, необходимое функции для обработки событий, очень часто в среднем линейно связано с числом обрабатываемых событий. Установите максимальное время исполнения выше максимального ожидаемого времени обработки конкретного размера пакета событий, чтобы избежать ненужных ошибок тайм-аута функции.

Наконец, вы должны рассмотреть *любой внешний ввод/вывод в хранилища состояний*, принадлежащие к ограниченному контексту функционально-ориентированного решения. Рабочая нагрузка функции изменяется в зависимости от потока входных событий, причем некоторые рабочие нагрузки требуют последовательного ввода/вывода во внешнее состояние, а другие — только нерегулярного ввода/вывода. Неспособность обеспечить достаточные ресурсы ввода/вывода может привести к снижению пропускной способности и производительности системы.

Параметры пакетной обработки событий

Если функция не способна обрабатывать назначенный ей пакет событий в течение ее времени жизни, то исполнение функции считается неуспешным и пакет должен

быть обработан снова. Однако в случае, если не будут произведены любые изменения выделенного времени жизни функции или размера пакета входных событий, она, скорее всего, просто снова откажет. Следовательно, должно быть выполнено хотя бы одно мероприятие их двух следующих:

- ◆ увеличение максимального времени жизни функции;
- ◆ уменьшение максимального размера пакета обрабатываемых функцией событий.



Функции, устанавливающие свои собственные соединения с брокером событий и управляющие потреблением событий, также могут периодически фиксировать смещения во время исполнения, обеспечивая частичное завершение пакета. Но это не работает, когда функция передает пакет событий, т. к. у нее нет возможности обновлять смещения потребителя во время обработки.

Кроме того, некоторые триггерные системы со слушателем событий, такие как предлагаемые Amazon и Microsoft, дают вам возможность автоматически уменьшать размер пакета в два раза при отказе и повторно выполнять отказавшую функцию. Если снова последует отказ, входной пакет снова сократится вдвое, т. е. функция повторно будет выполняться до тех пор, пока не достигнет точки, где она окажется способна завершить свою обработку вовремя.

Масштабирование решений FaaS

Решения FaaS предоставляют исключительные возможности для параллелизации работы, в особенности для очередей и потоков событий, где порядок обработки данных не имеет какого-либо значения. В случае подразделенных потоков событий, если порядок событий действительно важен, то максимальный уровень параллелизации ограничен числом разделов в потоках событий, как и для всех реализаций микросервисов.

Политики масштабирования, как правило, являются предметной областью фреймворка FaaS, поэтому обратитесь к документации своего фреймворка, чтобы узнать, какие варианты он предлагает. Типичные варианты предусматривают масштабирование, основанное на задержке входных данных потребителя, времени суток, пропускной способности обработки и характеристиках производительности.

В случае функций, которые устанавливают свои собственные соединения с брокером событий и управляют ими, остерегайтесь последствий перебалансирования назначения разделов, когда потребитель входит в группу потребителей либо ее покидает. Группа потребителей может оказаться в состоянии почти постоянного перебалансирования, если потребители часто присоединяются к группе потребителей и покидают ее, что препятствует нормальному ходу процесса. В экстремальных обстоятельствах можно застрять в *виртуальном тупике* (deadlock) перебалансирования, где функции проводят свое время жизни, неоднократно назначая и удаляя разделы. Эта проблема может возникнуть при использовании многих недолговечных функций с малыми размерами пакетов потребителей и может быть усугублена масштабированием политик, которые слишком чувствительны к задержке. Внедрение пошаговой политики масштабирования или использование петли гистерезиса

может обеспечить достаточную оперативность масштабирования, не приводя группу потребителей в состояние чрезмерного перебалансирования.

Статическое назначение разделов устраниет накладные расходы на перебалансирование динамически назначенных групп потребителей, а также может использоватьсь для соподразделения потоков событий. Функции будут запускаться, имея предварительное понимание о том, из каких разделов они станут потреблять данные, не будет никакого перебалансирования, и события могут просто потребляться всякий раз, когда функция запускается. Этот подход требует анализа работы, которую выполняет ваша функция, поскольку вам нужно убедиться, что используется каждый раздел.



Будьте осторожны с частым использованием триггеров и политики масштабирования. Частое перебалансирование назначений разделов может стать для брокеров событий весьма дорогостоящим явлением. Пытайтесь масштабировать свое количество функций вверх или вниз не чаще одного раза в несколько минут.

Резюме

«Функция как сервис» — это область облачных вычислений, которая быстро растет. Многие фреймворки FaaS предлагают различные инструменты разработки, управления, развертывания, запуска, тестирования и масштабирования функций, позволяющие создавать микросервисы с их использованием. Функции могут вызываться новыми событиями в потоке событий, задержками группы потребителей, физическим временем или индивидуально настроенной логикой.

Функционально-ориентированные решения особенно полезны в случаях работы с бизнес-задачами без поддержки состояния и простыми задачами с поддержкой состояния, которые не требуют планирования событий. Шаблон оркестратора позволяет вызывать многочисленные функции в строгом последовательном порядке, а также соблюдать порядок событий из потока событий. Поскольку пространство фреймворков FaaS быстро растет и эволюционирует, важно идти в ногу с новейшими функциональными возможностями этих платформ, представляющих интерес для вас и вашей организации.

Микросервисы на основе базового шаблона производителя и потребителя

Микросервисы, использующие *базовый шаблон производителя и потребителя* (Basic Producer and Consumer, ВРС), принимают события из одного или нескольких потоков событий, выполняют любые необходимые трансформации или бизнес-логику и выдают результирующие события в выходные потоки событий. Синхронный ввод/вывод в формате «запрос-ответ» также может быть частью такого бизнес-процесса, но он подробнее рассматривается в главе 13. Эта же глава посвящена исключительно событийно-управляемым компонентам.

Микросервисы ВРС характеризуются использованием базовых клиентов потребителя и производителя. Базовые клиенты потребителя не включают в себя планирование событий, водяные знаки, механизмы материализации, журналы изменений или горизонтальное масштабирование экземпляров обработки с локальными хранилищами состояний. Такие возможности, как правило, реализованы только в более полнофункциональных фреймворках, которые в главах 11 и 12 будут рассмотрены подробнее. Вы, можете разработать свои собственные библиотеки для обеспечения указанной функциональности, но вопрос такой разработки выходит за рамки этой главы. Так что вы должны решить, будет ли шаблон ВРС работать для ваших бизнес-требований или нет.

Клиент-производитель и клиент-потребитель доступны на наиболее часто используемых языках программирования, что снижает издержки на обучение персонала при начале работы с событийно-управляемыми микросервисами. Весь бизнес-процесс ограниченного контекста содержится в коде одного приложения микросервиса, оставляя обязанности микросервиса локализованными и простыми для понимания. Бизнес-процесс также может быть легко обернут в один или несколько контейнеров (в зависимости от сложности реализации), которые затем могут быть развернуты с помощью решения по управлению контейнерами микросервисов.

Где ВРС работают хорошо?

Микросервисы с использованием ВРС могут выполнять широкий спектр бизнес-требований, несмотря на отсутствие у них большинства полнофункциональных компонентов фреймворка. Простые шаблоны, такие как трансформации без поддержки состояния, легко реализуются, как и шаблоны с поддержкой состояния, где точное планирование событий не требуется.

В реализациях ВРС внешние хранилища состояний используются чаще, чем внутренние, поскольку масштабирование локального состояния между многочислен-

ными экземплярами и восстановление после отказов экземпляра затруднены без полнофункционального потокового фреймворка. Внешние хранилища состояний могут предоставлять многочисленным экземплярам микросервисов единый доступ, а также механизмы резервного копирования и восстановления данных.

Давайте рассмотрим несколько примеров, в которых реализация базовых ВРС работает особенно хорошо.

Интеграция с существующими и унаследованными системами

Унаследованные системы могут участвовать в событийно-управляемых архитектурах, интегрируя базовых клиентов производителя/потребителя в свою кодовую базу. Эта интеграция часто начинается на ранних этапах внедрения событийно-управляемых микросервисов и даже может быть частью вашей стратегии встраивания унаследованных систем в событийно-управляемую экосистему (см. главу 4). Унаследованные системы передают свои собственные данные в единственный источник истины — брокер событий — по мере необходимости и потребляют любые события, которые им нужны из других потоков событий.

В некоторых сценариях невозможно безопасно модифицировать унаследованную кодовую базу для получения и использования данных из потоков событий. К таким сценариям особенно применим шаблон «Коляска»¹ (sidecar), поскольку он обеспечивает некоторую событийно-управляемую функциональность, не затрагивая исходную кодовую базу.

Пример: шаблон «Коляска»

Магазин электронной коммерции имеет фронтенд, который показывает все данные о запасах и продуктах, которые он содержит. Ранее фронтенду монолит получал все свои данные путем синхронизации с подчиненным хранилищем данных «только для чтения» с помощью планируемого пакетного задания, как показано на рис. 10.1.

Сегодня же существуют два потока событий: один — с информацией о продукте, а другой — с уровнем запасов продукта. Вы можете использовать реализацию

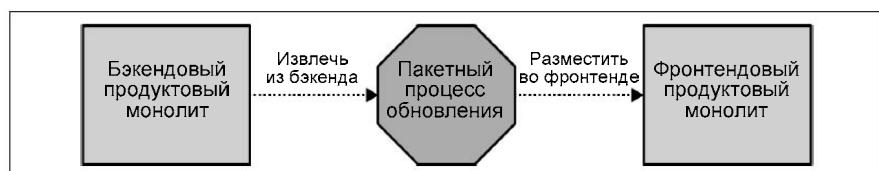


Рис. 10.1. Передача пакета между монолитами

¹ Шаблон называется коляской, потому что напоминает коляску, присоединенную к мотоциклу, — он «сбоку» присоединяется к родительскому приложению и предоставляет для него вспомогательные функции. — Прим. ред.

«коляски», чтобы загрузить эти данные в хранилище данных, где ВРС потребляет события и вставляет их в ассоциированные наборы данных. Фронтенд получает доступ к потоку данных обновлений продукта почти в реальном времени без необходимости изменять какой-либо системный код (рис. 10.2).

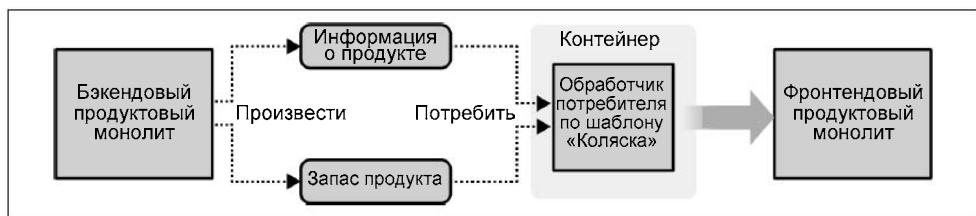


Рис. 10.2. Использование шаблона «Коляска» для вставки данных во фронтендовое хранилище данных

«Коляска» находится внутри своего собственного контейнера, но должна быть частью единого развертываемого фронтендового сервиса. Необходимо провести дополнительные испытания, чтобы убедиться, что интегрированная «коляска» работает должным образом. Здесь важно отметить, что шаблон «Коляска» позволяет добавлять новую функциональность в систему, не требуя значительных изменений в унаследованной кодовой базе.

Бизнес-логика с поддержкой состояния без учета порядка событий

Многие бизнес-процессы не имеют никаких ограничений по части порядка поступления событий, но требуют, чтобы *все* необходимые события в конечном счете поступили. Этот подход обеспечивается *шаблоном фильтрации* (gating pattern), и в нем хорошо реализуется ВРС.

Пример: книгоиздание

Допустим, вы работаете в издательстве, и прежде чем книга будет готова к отправке в типографию, необходимо осуществить три мероприятия. Не важно, в каком порядке происходят эти события, но важно, чтобы каждое из них происходило до выпуска книги в печать:

- ◆ *Содержимое.*

Должно быть написано содержимое книги.

- ◆ *Обложка.*

Должна быть создана обложка для книги.

- ◆ *Ценообразование.*

Должны быть установлены цены в соответствии с регионами и форматами.

Каждый из этих потоков событий действует как драйвер логики. Когда в любом из этих потоков появляется новое событие, оно сначала материализуется в соответст-

вующей таблице, а затем используется для просмотра всех остальных таблиц, чтобы увидеть, присутствуют или нет другие события (рис. 10.3).

В этом примере книга, имеющая ISBN 0010, будет опубликована в потоке событий выхода книги. Между тем книга, имеющая ISBN 0011, в настоящее время ждет появления обложки и в выходном потоке пока еще опубликована не будет.

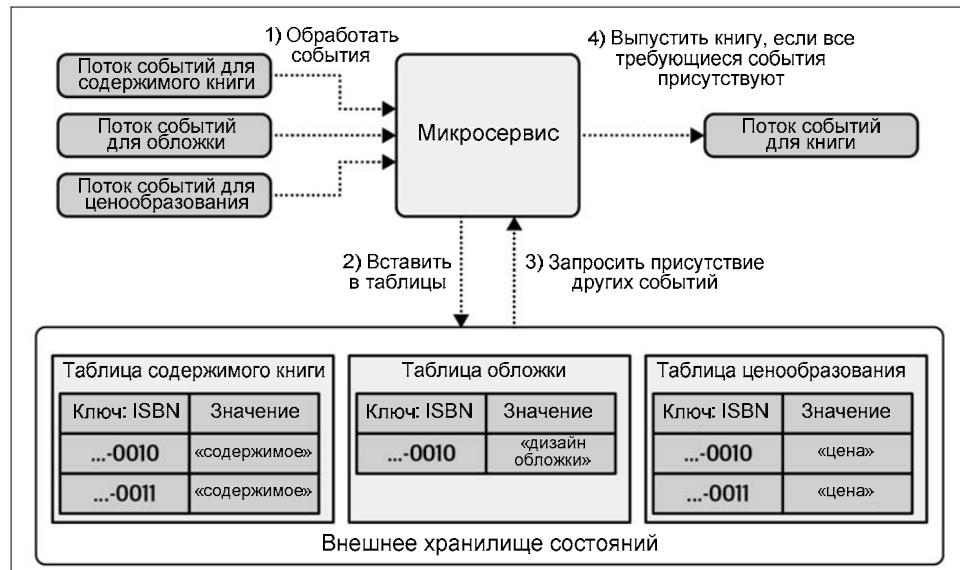


Рис. 10.3. Шаблон фильтрации проверяет готовность книги



В шаблоне фильтрации также может потребоваться явное одобрение со стороны человека. Подробнее об этом рассказано в разд. «Пример: рабочий процесс публикации газет (шаблон одобрения)» главы 13.

Когда уровень данных выполняет значительную часть работы

Использование ВРС также вполне приемлемо, когда базовый уровень данных выполняет большую часть бизнес-логики — такой как хранение геопространственных данных, свободный текстовый поиск, машинное обучение, искусственный интеллект и нейронные сети. Так, компания электронной коммерции может получать данные о новых продуктах, автоматически собирая их с веб-сайтов, и выполнять их классификацию с помощью микросервиса ВРС, при этом уровень данных серверной части является классификатором, обучающимся на пакетах данных. В качестве альтернативы события поведения пользователя, такие как открытие приложения, могут быть коррелированы с хранилищем геопространственных данных для определения ближайших розничных продавцов, рекламу которых следует показывать. В этих сценариях сложность обработки события полностью переносится на ниже-

лежащий уровень данных, а компоненты производителя и потребителя действуют как простые механизмы интеграции.

Независимое масштабирование уровня обработки и данных

Потребности в обработке и хранении данных микросервиса не всегда связаны линейно. Например, объем событий, которые должен обрабатывать микросервис, может меняться со временем. Так, часто встречающийся шаблон нагрузки (load pattern), на котором построен следующий далее пример, отражает цикл сна/бодрствования местного населения с интенсивной активностью в течение дня и очень низкой активностью в ночное время.

Пример: агрегирование данных о событиях для создания профилей взаимодействия с пользователями

Рассмотрим сценарий, в котором события поведения пользователя агрегируются в 24-часовые сеансы. Данные этих сеансов используются для определения продуктов, которые являются самыми популярными в последнее время, и для организации их рекламы. Как только 24-часовой сеанс агрегации завершен, он сбрасывается из хранилища данных и передается в выходной поток событий, освобождая пространство для хранения данных. Каждый пользователь имеет агрегацию, которая поддерживается во внешнем хранилище данных. Она выглядит примерно так:

Ключ	Значение
userId, timestamp	List (productId)

Уровень обработки нуждается в сервисе в соответствии с циклами сна/бодрствования людей, использующих продукт. В ночное время, когда большинство пользователей спят, для выполнения агрегаций требуется очень мало вычислительной мощности по сравнению с тем, что расходуется днем. Поэтому в ночное время сервис масштабируется вниз в целях экономии денег на вычислительную мощность.

Контроллер разделов может переназначать разделы входного потока событий одному экземпляру обработчика, поскольку он может обрабатывать потребление и обработку всех пользовательских событий. Обратите внимание, что, несмотря на малый объем событий, сфера потенциальных пользователей остается постоянной, и поэтому сервис требует полного доступа ко всем агрегациям пользователей. Масштабирование обработки не влияет на размер состояния, который сервис должен поддерживать.

В течение дня к сети могут быть подключены дополнительные обрабатывающие экземпляры для обработки увеличенной событийной нагрузки. В этом конкретном сценарии скорость запросов к хранилищу данных также увеличится, но кэширование, подразделение и пакетирование помогают держать нагрузку легче, чем линейное увеличение обрабатывающих мощностей.



Поставщики сервисов, такие как Google, Amazon и Microsoft, предлагают высокомасштабируемые хранилища данных с оплатой за чтение/запись, которые очень хорошо подходят для этого шаблона.

Гибридные приложения ВРС с внешней потоковой обработкой

Микросервисы с использованием ВРС также могут задействовать внешние системы потоковой обработки для выполнения работы, которую в противном случае было бы слишком сложно выполнять локально. Речь идет о *шаблоне гибридного приложения* (hybrid application pattern), в котором бизнес-логика распределена между ВРС и внешним фреймворком потоковой обработки. «Тяжеловесные» фреймворки, описание которых приводится в главе 11, являются отличными кандидатами на этот шаблон, поскольку могут обеспечивать крупномасштабную потоковую обработку с помощью простых интеграций.

Рассматриваемая здесь реализация ВРС может выполнять операции, которые в ином случае были бы для этого приложения недоступны, но при этом иметь доступ к любым необходимым языковым функциональностям и библиотекам. Например, можно использовать внешний фреймворк потоковой обработки для выполнения сложных агрегаций по нескольким потокам событий и задействовать микросервисы ВРС для заполнения локального хранилища данных результатами и обслуживания заданий «запрос-ответ».

Пример: использование внешнего фреймворка обработки потоков для объединения потоков событий

Допустим, ваш сервис ВРС должен использовать возможности объединения данных фреймворка потоковой обработки, который особенно хорошо подходит для объединения крупных наборов материализованных потоков событий. Внешний потоковый обработчик просто материализует потоки событий в таблицы и объединяет те строки, которые имеют один и тот же ключ. Эта простая операция объединения показана на рис. 10.4.

Гибридная реализация ВРС должна использовать совместимого клиента для запуска работы на внешнем фреймворке потоковой обработки. Такой клиент трансформирует код в инструкции для фреймворка, который сам будет обрабатывать потребление, объединение и выдачу событий в объединенный выходной поток событий. Эта конструкция передает работу на аутсорсинг внешнему сервису обработки, который возвращает результаты в форме потока событий. Рабочий процесс (workflow) такой структуры будет выглядеть так, как показано на рис. 10.5.



ВРС запускает клиента для выполнения внешней работы по потоковой обработке. Когда ВРС завершается, внешний экземпляр потоковой обработки также должен быть завершен, чтобы гарантировать, что никакие побочные процессы не останутся запущенными.

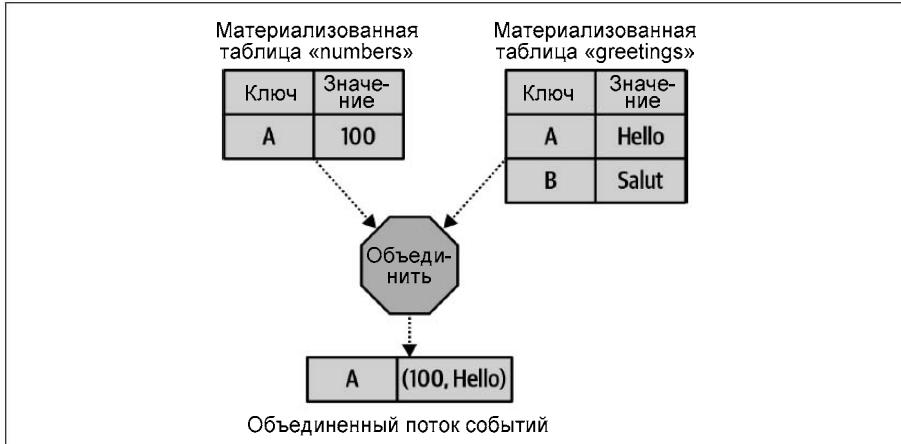


Рис. 10.4. Типичная аутсорсинговая операция обработки, выполняемая в крупном масштабе фреймворком потоковой обработки

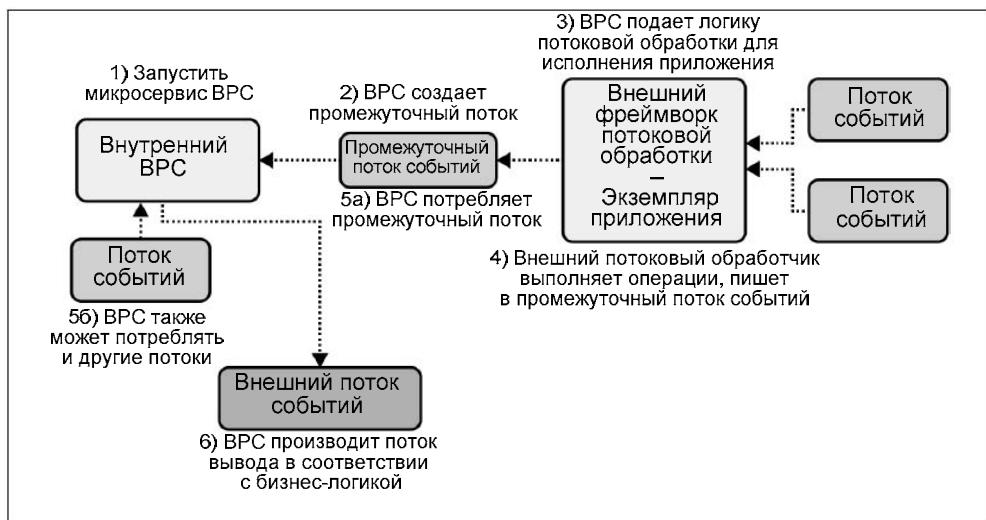


Рис. 10.5. Гибридный рабочий процесс, показывающий внешнее приложение обработки потока, отправляющее результаты обратно в ВРС через промежуточный поток событий

Преимущество шаблона гибридного приложения заключается в том, что он открывает функциональности потоковой обработки, которые в ином случае могут быть недоступны для вашего микросервиса. Доступность ограничена языками с соответствующими клиентами потоковой обработки, и не все функциональности могут поддерживаться для всех языков. Этот шаблон часто применяется в тандеме как с «легковесными», так и с «тяжеловесными» фреймворками, — например, он является одним из основных вариантов использования потоковых операций на основе SQL, таких как те, что предоставляются KSQL Confluent (<https://oreil.ly/iSLYt>). Подобные технологические варианты дают возможность расширять предложение

BPC, предоставляя ему доступ к мощным возможностям потоковой обработки, которых у него не было бы в противном случае.

Недостатки шаблона гибридного приложения связаны с увеличением сложности системы. Тестирувать приложения становится намного труднее, т. к. вы также должны найти способ интеграции внешнего фреймворка потоковой обработки в вашу среду тестирования (см. главу 15). Сложность отладки и разработки также возрастает, поскольку внедрение фреймворка потоковой обработки увеличивает число взаимодействующих сервисов и вероятность появления ошибок. Наконец, работа с ограниченным контекстом микросервиса может также усложниться, поскольку вам необходимо обеспечивать легкое управление развертыванием, откатом и операциями с развертываниями гибридных приложений.

Резюме

Шаблон BPC — простой и мощный. Он формирует основу многих шаблонов событийно-управляемых микросервисов без поддержки состояния и с поддержкой состояния. Используя шаблон BPC, можно легко реализовать потоковую передачу без поддержки состояния и простые приложения с поддержкой состояния.

Шаблон BPC также весьма гибок. Он хорошо сочетается с реализациями, где уровень хранения данных выполняет большую часть бизнес-логики. Вы можете использовать его в качестве интерфейсного слоя между потоками событий и унаследованными системами, а также задействовать внешние системы потоковой обработки для расширения его возможностей.

Однако из-за своей базовой природы он требует инвестиций в библиотеки для доступа к таким механизмам, как простая материализация состояний, планирование событий и принятие решений на основе временных меток. Эти компоненты пересекаются с предложениями, представленными в главах 11 и 12, и поэтому вам нужно будет принять решение о том, какой объем собственных доработок вы хотели бы выполнить, чтобы не обращаться к более специализированным решениям.

«Тяжеловесные» фреймворки для микросервисов

В этой и следующей главах рассматриваются полнофункциональные фреймворки, которым в событийно-управляемой обработке отводится особое место. Именуемые также *потоковыми фреймворками*, они предоставляют механизмы и API для обработки потоков данных и часто используются для потребления и создания событий в брокере событий. Эти фреймворки можно условно разделить на «тяжеловесные» фреймворки, которые рассматриваются в этой главе, и «легковесные» фреймворки, рассматриваемые в следующей. Обе эти главы приводятся здесь не для сравнения технологий, а, скорее, для того, чтобы дать общий обзор специфики их работы. Однако в некоторых разделах исследуются специфичные для фреймворков функциональности — в особенности в той части, что касается реализации приложений в стиле микросервисов. Для оценки «тяжеловесных» фреймворков в этой главе рассматриваются возможности Apache Spark (<https://spark.apache.org>), Apache Flink (<https://flink.apache.org>), Apache Storm (<https://storm.apache.org>), Apache Heron (<https://heron.apache.org>) и модели Apache Beam (<https://beam.apache.org>) — на примерах тех видов технологий и операций, которые обычно предоставляются фреймворками.

Одной из определяющих характеристик «тяжеловесного» потокового фреймворка является то, что он требует независимого кластера обрабатывающих ресурсов для своей работы. Этот кластер обычно состоит из нескольких совместно используемых рабочих узлов, а также нескольких ведущих узлов, которые планируют и координируют их работу. В дополнение к этому лидирующие решения Apache традиционно опираются на Apache Zookeeper — еще одну кластерную службу — для обеспечения поддержки высокой доступности и координации отбора лидеров кластеров. Хотя Zookeeper не является абсолютно необходимым для создания «тяжеловесного» кластера, вы должны тщательно оценить, насколько он вам нужен, если вы создаете свой собственный кластер.

Другая определяющая характеристика заключается в том, что «тяжеловесный» фреймворк использует свои собственные внутренние механизмы для обработки отказов, восстановления, размещения ресурсов, распределения задач, хранения данных, обмена информацией и координации между экземплярами обработки и задачами. Это контрастирует с реализациями таких «легковесных» фреймворков, как FaaS и BPC, которые для выполнения таких функций в значительной степени опираются на систему управления контейнерами (CMS) и брокеры событий.

Указанные характеристики являются главными причинами, по которым рассматриваемые в этой главе фреймворки называются «тяжеловесными». Необходимость

управлять и поддерживать дополнительные кластерные фреймворки независимо от брокера событий и CMS — задача не из легких.



Некоторые «тяжеловесные» фреймворки движутся в сторону облегченных режимов исполнения. Эти облегченные режимы хорошо интегрируются с CMS, используемой для работы с другими реализациями, работающими на основе микросервисов.

Возможно, вы заметили, что «тяжеловесный» фреймворк обрабатывает функции, которые обрабатываются CMS и брокером событий. Так, CMS может управлять распределением ресурсов, отказами, восстановлением и масштабированием систем, в то время как брокер событий способен обеспечивать событийно-управляемый обмен информацией между экземплярами одного микросервиса. Так что «тяжеловесный» фреймворк — это единое решение, которое объединяет отмеченные возможности CMS и брокера событий. Мы рассмотрим эту тему немного подробнее в следующей главе, посвященной «легковесным» фреймворкам.

Краткая история «тяжеловесных» фреймворков

«Тяжеловесные» фреймворки потоковой обработки происходят от своих предшественников — «тяжеловесных» фреймворков пакетной обработки. Один из самых известных таких фреймворков, Apache Hadoop, был выпущен в 2006 году, предоставляя любому пользователю технологии с открытым исходным кодом по обработке чрезвычайно больших пакетов данных (также известных как *big data*). Hadoop объединил в себе ряд технологий, чтобы предложить массовую параллельную обработку, восстановление после отказов, долговечность данных и межузловую связь, позволяя пользователям дешево и легко получать доступ к вычислительным мощностям для решения задач, требующих многих тысяч (или более) узлов.

Одним из первых широко доступных средств обработки пакетов данных *big data* стал уже упоминавшийся в главе 5 фреймворк MapReduce, однако, несмотря на свою мощь, он работает медленно по сравнению со многими современными решениями. Размер пакетов *big data* с течением времени неуклонно увеличивался — хотя в начале пути рабочие нагрузки в сотни (или тысячи) гигабайт были обычным явлением, сегодня они масштабируются до размеров в терабайтном и петабайтном диапазоне. По мере роста этих наборов данных растет и спрос на более быструю обработку, более мощные возможности, более простые варианты исполнения и решения, которые могут обеспечивать возможности потоковой обработки почти в режиме реального времени.

И именно здесь появляются Spark, Flink, Storm, Heron и Beam — решения, предназначенные для обработки потоков данных и обеспечения значимых результатов гораздо быстрее, чем те, которые обеспечивались пакетной обработкой MapReduce. Некоторые из них (такие как Storm и Heron) являются чисто потоковыми технологиями и в настоящее время не обеспечивают пакетной обработки. Другие же — такие как Spark и Flink, объединяют пакетную и потоковую обработку в единое ре-

шение. Эти технологии, несомненно, знакомы большинству специалистов, работающих с big data, и, вероятно, уже в той или иной степени используются в научных отраслях и для анализа данных во многих организациях. На самом деле именно так команды многих организаций начинают увлекаться обработкой, управляемой событиями, поскольку получают возможность преобразовывать свои существующие пакетные задания в конвейеры на основе потоковой передачи.

Внутренняя логика работы «тяжеловесных» фреймворков

Все упомянутые ранее «тяжеловесные» фреймворки Apache с открытым исходным кодом работают примерно одинаково. Проприетарные решения, такие как Dataflow от Google, которое выполняет приложения, написанные с использованием API Apache Beam, вероятно, действуют аналогичным образом, но это только предположение, учитывая, что источник закрыт, а бэкенд не описан подробно. Одна из трудностей подробного описания «тяжеловесных» фреймворков заключается в том, что каждый из них имеет свои собственные эксплуатационные и дизайнерские нюансы, и полный охват каждого такого фреймворка выходит далеко за рамки этой главы.



Убедитесь, что вы внимательно прочитали и поняли документы, подробно описывающие характер работы вашего конкретного «тяжеловесного» фреймворка.

«Тяжеловесный» кластер потоковой обработки — это группа выделенных ресурсов обработки и хранения, разбитая на две основные роли. Первая роль — это *ведущий узел* (master node), который определяет приоритеты, назначает исполнителей и задачи, выполняемые ими, и управляет ими. Вторая роль — *исполнитель* (executor), который выполняет эти задачи, используя вычислительную мощность, память, локальный и удаленный диск, доступные его *рабочему узлу* (worker node). В событийно-управляемой обработке эти задачи будут подключаться к брокеру событий и потреблять события из потоков событий. На рис. 11.1 приведена схема того, как это работает.

Здесь также показан Apache Zookeeper, который играет поддерживающую роль для всего этого потокового кластера. Zookeeper обеспечивает высоконадежную распределенную координацию и используется для определения того, какой узел является ведущим, — поскольку узлы нередко выходят из строя, будь то рабочие, ведущие или узлы Zookeeper. Так, при отказе ведущего узла Zookeeper помогает решить, какой из оставшихся резервных ведущих узлов станет новым лидером, чтобы обеспечить непрерывность операций.



Zookeeper исторически был одним из основных компонентов в обеспечении координации распределенных «тяжеловесных» фреймворков. Более новые фреймворки могут использовать или не использовать Zookeeper. В любом случае для надежного выполнения распределенных рабочих нагрузок распределенная координация необходима.

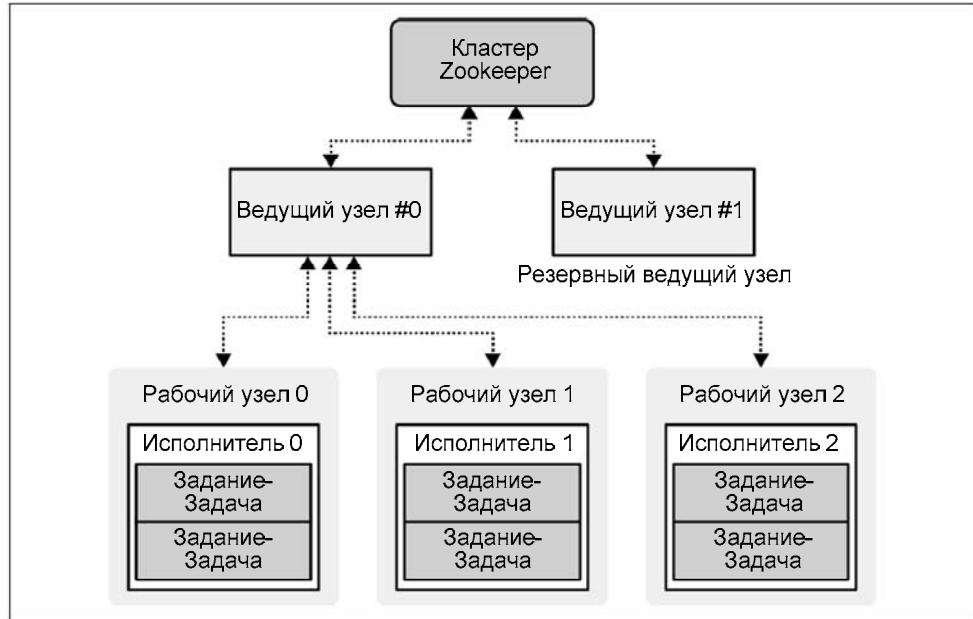


Рис. 11.1. Общий вид «тяжеловесного» фреймворка потоковой обработки

Задание (job) — это топология потоковой обработки, построенная с использованием пакета разработки программного обеспечения (SDK) фреймворка и предназначенная для решения задач конкретного ограниченного контекста. Она работает на кластере беспрерывно, обрабатывая события по мере их поступления, как и любой другой микросервис, описанный в этой книге.

После принятия кластером определенная топология потоковой обработки (т. е. задание) разбивается на *задачи* (task) и назначается рабочим узлам. Диспетчер задач (task manager) следит за прохождением задач и обеспечивает их выполнение. При возникновении отказа диспетчер задач перезапускает работу в одном из доступных исполнителей. Диспетчеры задач обычно настраиваются с высокой доступностью, благодаря чему, если узел, на котором работает диспетчер задач, выходит из строя, резервная копия может взять на себя управление, предотвращая отказ всех запущенных заданий.

На рис. 11.2 показано задание, отправляемое в кластер через ведущий узел 1, которое, в свою очередь, транслируется в задачи для обработки исполнителями. Эти длительные задачи устанавливают соединения с брокером событий и начинают потреблять события из потока событий.

Хотя в этом примере показано соотношение 1:1 между задачами и разделами потока, вы можете настроить степень параллелизма, которую вы хотели бы иметь в своем приложении. Одна задача может потреблять из всех разделов, или многие задачи могут потреблять из одного и того же раздела, скажем, в случае очереди.

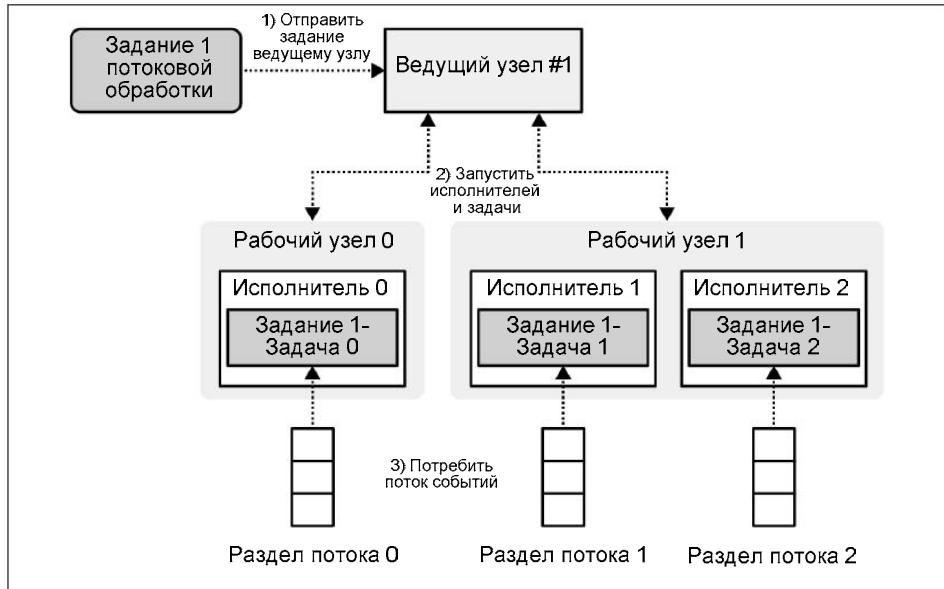


Рис. 11.2. Отправка задания потоковой обработки для чтения из потока событий

Выгоды и ограничения

Рассматриваемые в этой главе «тяжеловесные» фреймворки представляют собой преимущественно аналитические технологии. Они обеспечивают существенные достижения при анализе больших объемов событий в режиме, близком к реальному времени, что позволяет быстрее принимать решения. Некоторые весьма распространенные шаблоны их использования включают в себя следующее:

- ◆ извлечение данных, трансформацию их и загрузку в новое хранилище данных (Extract, Transform and load, ETL);
- ◆ выполнение сеансового и оконного анализа;
- ◆ выявление аномальных шаблонов поведения;
- ◆ агрегирование потоков и поддержку состояния;
- ◆ выполнение любых потоковых операций без поддержки состояния.

Все эти фреймворки мощные и достаточно зрелые, и многие организации их используют, внося свой вклад в совершенствование их исходного кода. В этом плане вам доступны многочисленные книги и посты в блогах, отличная документация и много примеров приложений.

Однако у них имеется несколько весьма существенных недостатков, которые ограничивают, хотя и не полностью исключают, создание приложений с использованием микросервисов, основанных на этих фреймворках.

Во-первых, эти «тяжеловесные» фреймворки изначально не были разработаны с намерением выполнять развертывания в стиле микросервисов. Для развертывания

этих приложений требуется выделенный кластер ресурсов, выходящий за рамки брокера событий и CMS, что усложняет управление большим числом приложений. Впрочем, существуют способы уменьшить эту сложность за счет применения новых технологических разработок для развертывания — некоторые из них подробно рассматриваются далее в этой главе.

Во-вторых, большинство этих фреймворков основаны на JVM (Java Virtual Machine), что ограничивает языки реализации, на которых вы можете создавать приложения, опирающиеся на микросервисы. Распространенным обходным путем здесь является использование для выполнения трансформаций «тяжеловесного» фреймворка, выступающего в роли собственного автономного приложения, в то время как другое автономное приложение, написанное на доступном вам языке, обслуживает бизнес-функциональность из трансформированного хранилища состояний.

В-третьих, материализация потока сущностей в до бесконечности хранящую данные таблицу поддерживается не всеми фреймворками. Это может препятствовать созданию соединений таблица-таблица и поток-таблица и реализации ряда шаблонов — например, таких как шаблон фильтрации (*gating pattern*), показанный на рис. 10.3.

Даже когда «тяжеловесные» фреймворки поддерживают потоковую материализацию и соединения, это часто не сразу видно в документации на них. Некоторые из этих фреймворков в значительной степени фокусируются на временных агрегациях с примерами, постами в блогах и рекламными объявлениями с упором на анализ временных рядов и агрегирование на основе ограниченных размеров окна. Небольшое, но тщательное исследование показывает, что ведущие фреймворки обеспечивают *глобальное окно*, которое позволяет материализовывать потоки событий. Исходя из этого, вы можете реализовывать свои собственные индивидуально настроенные функции соединения, хотя я нахожу, что они все еще гораздо хуже документированы, чем следовало бы, учитывая их важность для обработки потоков событий в масштабах организации.

Опять же, эти недостатки указывают на типы аналитических рабочих нагрузок, которые были предусмотрены для этих фреймворков при их разработке и внедрении. Технологические усовершенствования отдельных их реализаций и инвестиции в общие API-интерфейсы, которые не зависят от реализации (например, Apache Beam (<https://beam.apache.org>)), приводят к постоянным изменениям в сфере «тяжеловесных» фреймворков, и нужно следить за лидерами отрасли, чтобы увидеть результаты их нововведений.

Варианты настройки кластера и режимы исполнения

Существует несколько вариантов построения вашего «тяжеловесного» кластера обработки потоков и управления им, каждый из которых имеет свои преимущества и недостатки.

Используйте сервис, размещенный на хосте провайдера

Первый и самый простой способ управления кластером — это просто заплатить провайдеру, чтобы он сделал все за вас. Точно так же, как существует ряд поставщиков вычислительных сервисов, есть также поставщики, которые будут рады выполнять у себя большинство ваших операционных решений и, возможно, управлять ими. Это, как правило, самый дорогой вариант с точки зрения потраченных денег по сравнению с проектируемыми затратами на создание собственного кластера, но он значительно снижает операционные накладные расходы и устраняет необходимость в собственном опыте. Например, Amazon предлагает управляемые сервисы Flink и Spark. Google, Databricks (<https://databricks.com>), Microsoft — свои собственные пакеты Spark, а Google предлагает Dataflow — собственную реализацию исполнителя Apache Beam.

Еще один момент, который следует учесть, рассматривая вопрос об этих сервисах, заключается в том, что они, похоже, постоянно движутся к полному бессерверному подходу, когда весь физический кластер будет невидим для вас как для подписчика. Это может быть приемлемым или неприемлемым вариантом в зависимости от ваших потребностей в безопасности, производительности и изоляции данных. Убедитесь, что вы понимаете, что предлагают и чего не предлагают эти поставщики сервисов, поскольку они могут не включать в себя всю функциональность независимо управляемого кластера.

Постройте свой собственный полный кластер

«Тяжеловесный» фреймворк может иметь свой собственный выделенный масштабируемый кластер ресурсов, независимый от CMS. Такое развертывание является исторической нормой для «тяжеловесных» кластеров, поскольку оно тесно моделирует исходные дистрибутивы Hadoop. Это развертывание часто встречается в ситуациях, когда «тяжеловесный» фреймворк будет использоваться сервисами, требующими значительного числа (сотен или тысяч) рабочих узлов.

Создайте кластер, интегрированный с CMS

Вы также можете создать кластер, интегрированный с системой управления контейнерами (Container Management System, CMS). Первый режим определяет простое развертывание кластера на ресурсах, предоставляемых CMS, тогда как второй режим предусматривает использование самой CMS в качестве средства масштабирования и развертывания отдельных заданий. Некоторые из главных преимуществ развертывания кластера на CMS заключаются в том, что вы получаете мониторинг, журналирование и управление ресурсами, которые он предоставляет. Масштабирование кластера тогда становится вопросом простого добавления или удаления необходимых типов узлов.

Развертывание и запуск кластера с использованием CMS

Развертывание «тяжеловесного» кластера с использованием CMS имеет много преимуществ. Ведущие узлы, рабочие узлы и узлы Zookeeper (если они задействуются)

создаются в их собственном контейнере или в виртуальных машинах. Эти контейнеры управляются и мониторятся, как и любой другой контейнер, обеспечивая видимость отказов, а также средства автоматического их перезапуска.



Вы можете обеспечить выполнение статического назначения ведущих узлов и любых других сервисов, которые должны быть высокодоступными, чтобы CMS не перераспределяла их при масштабировании опорных вычислительных ресурсов. Это предотвращает чрезмерно частое появление оповещений из кластерного монитора об отсутствующих ведущих узлах.

Указание ресурсов для одного задания с использованием CMS

Исторически сложилось так, что «тяжеловесный» кластер отвечал за распределение и управление ресурсами для каждого приложения. Внедрение CMS в последние годы дает вам возможность делать то же самое, но также управлять всеми другими реализациями с использованием микросервисов. Когда «тяжеловесному» кластеру для масштабирования требуется больше ресурсов, он должен сначала запросить и получить ресурсы у CMS. Затем они могут быть добавлены в пул ресурсов кластера и, наконец, назначаться приложениям по мере необходимости.

Фреймворки Spark и Flink позволяют напрямую задействовать CMS Kubernetes для масштабируемого развертывания приложений за пределами первоначальной конфигурации их выделенного кластера, где каждое приложение имеет свой собственный набор выделенных рабочих узлов. Например, Apache Flink с помощью Kubernetes позволяет приложениям работать независимо в пределах их собственно-го изолированного сеансового кластера (<https://oreil.ly/bAydL>). Apache Spark предлагает аналогичную возможность (<https://oreil.ly/sQGkQ>), что позволяет Kubernetes играть роль ведущего узла и поддерживать изолированные рабочие ре-сурсы для каждого приложения. Общая схема того, как это работает, приведена на рис. 11.3.

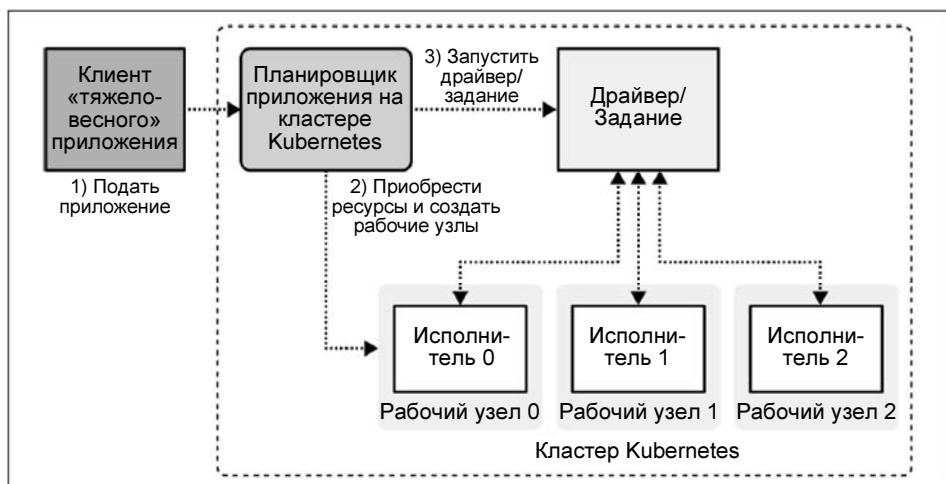


Рис. 11.3. Одно задание, развернутое в кластере Kubernetes и управляемое им



Этот режим развертывания почти идентичен тому, как вы развертываете «нетяжеловесные» микросервисы, и объединяет стратегии «легковесного» развертывания и развертывания ВРС.

Показанный здесь шаблон развертывания имеет несколько преимуществ:

- ◆ в нем используется модель приобретения ресурсов CMS, включая потребности в масштабировании;
- ◆ существует полная изоляция между заданиями;
- ◆ вы можете использовать различные фреймворки и их версии;
- ◆ «тяжеловесные» потоковые приложения можно рассматривать так же, как и микросервисы, и использовать те же процессы развертывания.

И конечно же, есть еще несколько недостатков:

- ◆ поддержка доступна не для всех ведущих «тяжеловесных» потоковых фреймворков;
- ◆ интеграция доступна не для всех ведущих CMS;
- ◆ функциональности, доступные в режиме полного кластера, такие как автоматическое масштабирование, могут еще не поддерживаться.

Режимы передачи приложений в кластер

Передача приложений для обработки в «тяжеловесный» кластер может осуществляться одним из двух главных способов: в драйверном режиме или в кластерном режиме.

Драйверный режим

Драйверный режим (*driver mode*) поддерживается фреймворками Spark и Flink. *Драйвер* — это просто отдельное локальное автономное приложение, которое помогает координировать и выполнять приложение, хотя само приложение по-прежнему выполняется в ресурсах кластера. Драйвер координируется с кластером, чтобы гарантировать выполнение приложения, и может использоваться для сообщения об ошибках, ведения журнала и других операций. В частности, завершение работы драйвера приведет к завершению работы приложения, что обеспечивает простой механизм для развертывания и завершения «тяжеловесных» потоковых приложений. Драйвер приложения можно развернуть как микросервис с помощью CMS, а рабочие ресурсы получать при этом из «тяжеловесного» кластера. Чтобы завершить работу драйвера, просто остановите его, как если бы это был любой другой микросервис.

Кластерный режим

Кластерный режим (*cluster mode*) поддерживается фреймворками Spark и Flink и является режимом развертывания, принятым по умолчанию для заданий Storm и

Heron. В кластерном режиме все приложение передается в кластер для управления и выполнения, после чего вызывающей функции возвращается уникальный идентификатор (ID). Этот уникальный идентификатор необходим для идентификации приложения и отправки ему заказов через API кластера. В кластерном режиме развертывания команды должны напрямую передаваться в кластер для развертывания и остановки приложений, что может не подходить для вашего конвейера развертывания микросервисов.

Обработка состояния и использование контрольных точек

Операции с поддержкой состояния могут выполняться с использованием внутреннего или внешнего состояния (см. главу 7), хотя большинство «тяжеловесных» фреймворков отдают предпочтение внутреннему состоянию из-за его высокой производительности и масштабируемости. Записи с поддержкой состояния хранятся в памяти для быстрого доступа, но также переносятся на диск для обеспечения уверенного их хранения, а также когда состояние выходит за пределы доступной памяти. Использование внутреннего состояния действительно сопряжено с некоторыми рисками — такими как потеря состояния из-за сбоя диска, отказов узлов и временных отключений состояния из-за агрессивного масштабирования с помощью CMS. Однако выгоды от повышения производительности, как правило, намного превышают потенциальные риски, которые можно уменьшить с помощью тщательного планирования.

Контрольные точки (checkpoints) — это моментальные снимки текущего внутреннего состояния приложения, которые используются для восстановления состояния после масштабирования или отказов узлов. Контрольная точка сохраняется в долговременном хранилище вне рабочих узлов приложения, что защищает ее от потери данных. Установка контрольных точек может выполняться с помощью любого хранилища, совместимого с фреймворком, такого как распределенная файловая система Hadoop (Hadoop Distributed File System, HDFS), что представляет собой весьма распространенный вариант, или высокодоступное внешнее хранилище данных. Каждое подразделенное хранилище состояний может затем восстановить себя из контрольной точки, обеспечивая возможности как полного восстановления в случае общего отказа приложения, так и частичного восстановления в случае отказов масштабирования и рабочего узла.

Существуют два главных состояния, которые механизм контрольных точек должен учитывать при потреблении и обработке подразделенных потоков событий:

◆ *Операторное состояние.*

Пары `<partitionId, offset>` (ключ, смещение). Контрольная точка должна обеспечивать, чтобы внутреннее состояние ключа (см. следующий пункт) совпадало со смещениями потребителя каждого раздела. Каждый параметр `partitionId` является уникальным среди всех входных тем.

◆ Состояние по ключу.

Пары $\langle \text{key}, \text{state} \rangle$ (ключ, значение). Это состояние, относящееся к объекту с ключом, — например, во время агрегирования, восстановления, работы с окнами, соединений и других операций с отслеживанием состояния.

Как операторное состояние, так и состояние по ключу, должны синхронно записываться таким образом, чтобы состояние по ключу точно представляло обработку всех событий, помеченных как потребляемые операторным состоянием. Невыполнение этого требования может привести к тому, что события либо вообще не будут обработаны, либо будут обработаны многократно. Пример обоих состояний, записанных в контрольную точку, показан на рис. 11.4.

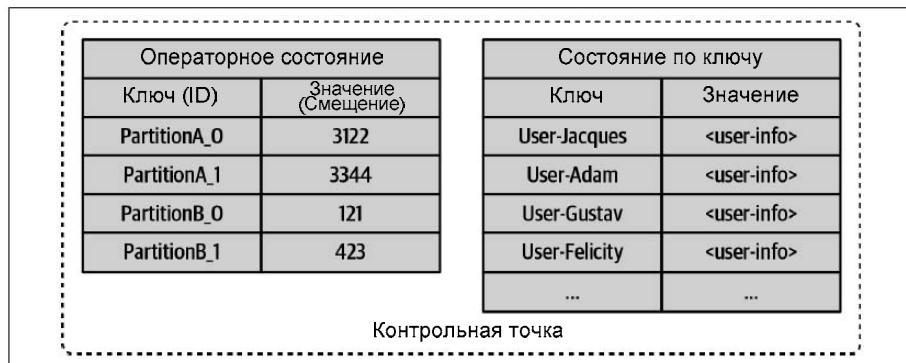


Рис. 11.4. Контрольная точка с операторным состоянием (слева) и состоянием по ключу (справа)



Восстановление из состояния, помеченного контрольной точкой функционально эквивалентно использованию моментальных снимков для восстановления внешних хранилищ состояний, как описано в разд. «Использование моментальных снимков» главы 7.

Состояние, ассоциированное с прикладной задачей, должно быть полностью загружено из контрольной точки, прежде чем вы сможете обработать какие-либо новые данные. «Тяжеловесный» фреймворк должен также верифицировать, что операторное состояние и ассоциированное с ним состояние по ключу совпадают для каждой задачи, обеспечивая правильное назначение разделов между задачами. В каждом «тяжеловесном» фреймворке, упомянутом в начале этой главы, контрольные точки реализованы по-своему, поэтому за подробностями обязательно обратитесь к соответствующей документации.

Масштабирование приложений и обработка разделов потока событий

Максимальный параллелизм «тяжеловесного» приложения ограничен теми же факторами, которые были рассмотрены в главе 5. Типичный потоковый обработчик с поддержкой состояния будет ограничен количеством входных данных самого

низкого подразделенного потока. Поскольку «тяжеловесные» обрабатывающие фреймворки особенно хорошо подходят для вычисления больших объемов генерируемых пользователем данных, весьма часто можно увидеть циклические шаблоны с серьезными вычислительными потребностями в течение дня и очень малыми — в середине ночи. Пример суточного циклического шаблона показан на рис. 11.5.

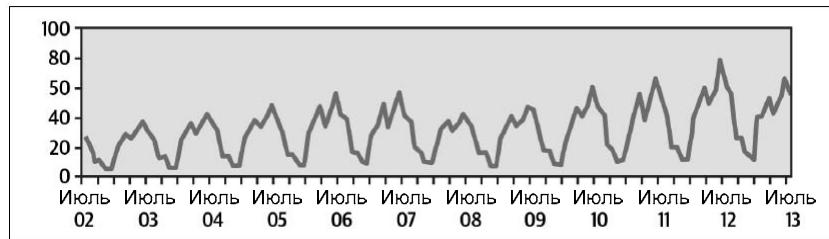


Рис. 11.5. Выборка ежедневного циклического объема данных

Приложения, обрабатывающие такие данные, в значительной степени выигрывают от возможности масштабирования вверх вместе с увеличивающимся спросом и вниз вместе с уменьшающимся спросом. Правильное масштабирование может обеспечить, чтобы приложение имело достаточную емкость для своевременной обработки всех событий, не тратя ресурсы на чрезмерную оптимизацию. В идеале задержка между временем получения события и временем его полной обработки должна быть сведена к минимуму, хотя многие приложения не так чувствительны к временно увеличенной задержке.



Масштабирование приложения отличается от масштабирования кластера. Все обсуждаемое здесь масштабирование исходит из наличия достаточного количества кластерных ресурсов для повышения параллелизма приложения. Обратитесь к документации вашего фреймворка для учета возможностей масштабирования ресурсов кластера.

Потоковые приложения без поддержки состояния очень легко масштабируются вверх или вниз. Новые необходимые для приложения обрабатывающие ресурсы могут просто присоединяться к группе потребителей или покидать ее, после чего происходит перебалансировка ресурсов и возобновление потоковой передачи. Приложения с поддержкой состояния могут быть сложнее в обработке — не только состояние должно быть загружено в назначенные приложению рабочие узлы, но и загруженное состояние должно соответствовать назначениям разделов входного потока событий.

Существуют две главные стратегии масштабирования приложений с поддержкой состояния, и хотя их специфика варьируется в зависимости от технологий, они имеют общую цель — минимизировать время простоя приложений.

Масштабирование приложения во время его работы

Первая стратегия позволяет удалять, добавлять или переназначать экземпляры приложения, не останавливая его и не влияя на точность обработки. Она доступна

только в некоторых «тяжеловесных» потоковых фреймворках, т. к. требует тщательной обработки как состояний, так и перераспределенных (shuffle) событий. Добавление и удаление экземпляров требует перераспределения всех назначенных разделов потока и перезагрузки состояния с последней контрольной точки. На рис. 11.6 показано обычное перераспределение, где каждая нижестоящая операция `reduce` получает свои перераспределенные события из вышестоящих операций `groupByKey`. Если бы один из экземпляров был внезапно прерван, то узлы `reduce` больше не знали бы, где брать источник перераспределенных событий, что привело бы к фатальному сбою.

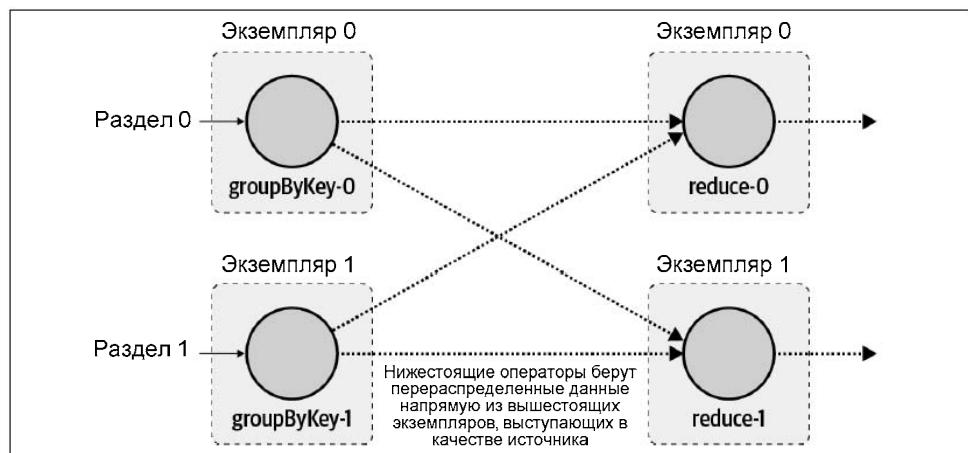


Рис. 11.6. Логическое представление обычного перераспределения

Такая стратегия масштабирования реализована в системе динамического выделения ресурсов фреймворка Spark (<https://oreil.ly/RvBg2>). Однако это требует использования так называемого крупнозернистого режима (coarse-grained mode) для кластерного развертывания и использования внешнего сервиса перераспределения (External Shuffle Service, ESS) в качестве изолирующего слоя. ESS получает перераспределенные события из вышестоящих задач и сохраняет их для потребления нижестоящими задачами, как показано на рис. 11.7. Нижестоящие потребители получают доступ к событиям, запрашивая у ESS данные, которые им назначены.

Экземпляры задач (исполнители) теперь могут завершаться, т. к. нижестоящие операции больше не зависят от конкретного вышестоящего экземпляра. Перераспределенные данные остаются в ESS, и масштабированный вниз сервис, как показано на рис. 11.8, может возобновить обработку. В этом примере экземпляр 0 является единственным оставшимся обработчиком и принимает на себя оба раздела, в то время как последующие операции бесшовно продолжают обработку через интерфейс с ESS.



Перераспределения в потоке событий в режиме реального времени по-прежнему остаются областью, где «тяжеловесным» фреймворкам еще предстоит дальнейшая доработка. В следующей главе мы рассмотрим, как «легковесные» фреймворки напрямую используют брокер событий, чтобы играть роль внешнего сервиса перемешивания.

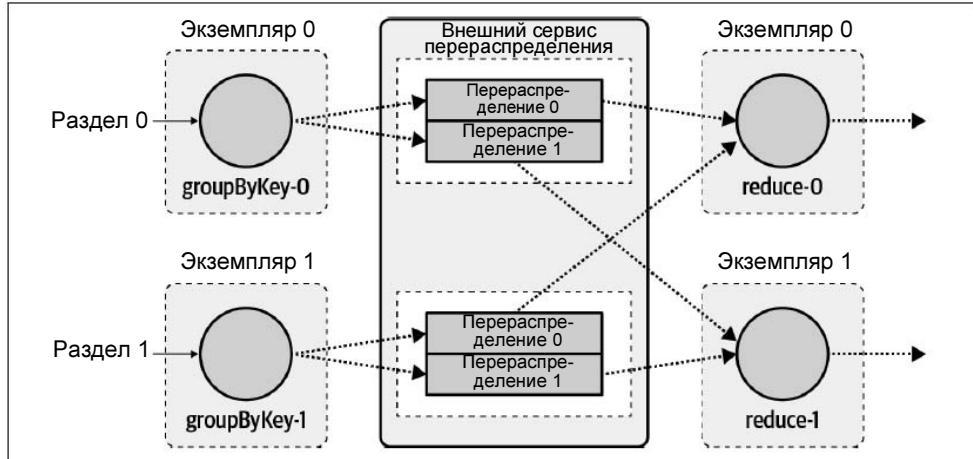


Рис. 11.7. Логическое представление перераспределения с использованием внешнего сервиса перераспределения

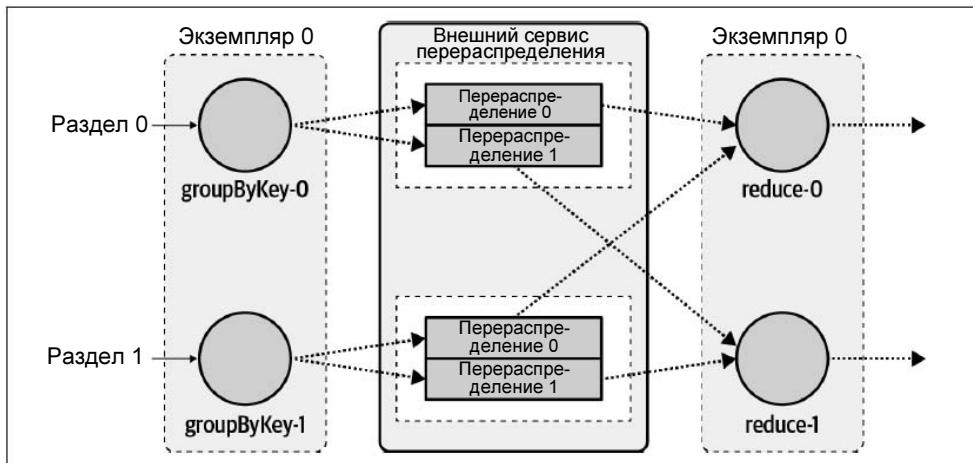


Рис. 11.8. Масштабированное вниз приложение с использованием внешнего сервиса перераспределения (обратите внимание, что экземпляра 1 исчез)

Решение Google Dataflow, которое выполняет приложения, написанные с помощью Beam API, обеспечивает встроенное масштабирование (https://oreil.ly/T_rsl) как ресурсов, так и рабочих экземпляров. Решение Neron предоставляет (в настоящее время экспериментальный) менеджер Health Manager (<https://oreil.ly/-E6bm>), который может сделать топологию динамической и саморегулирующейся. Эта функциональность все еще находится в стадии разработки, но предназначена для обеспечения масштабирования топологий с отслеживанием состояния в режиме реального времени.

Продолжающееся усовершенствование тяжеловесных фреймворков

Незадолго до того, как эта книга вышла в печать¹, было объявлено о выпуске Apache Spark 3.0.0. Одним из главных изменений этого выпуска является возможность динамического масштабирования количества экземпляров без использования ESS.

Этот режим работает, отслеживая этапы, которые генерируют файлы в случайном порядке, и поддерживая исполнителей, которые генерируют эти данные, в жизнеспособном состоянии, в то время как нижестоящие задания, которые их используют, все еще активны. По сути, источники играют роль собственного внешнего сервиса перераспределения. Они также позволяют очищать себя, как только все нижестоящие задания больше не требуют их перераспределенных файлов.

Наличие выделенного и постоянного отдельного хранилища для ESS не позволяет CMS полностью масштабировать задание, поскольку она всегда должна выделять достаточно ресурсов для обеспечения доступности ESS. Эта новая возможность динамического масштабирования в Spark иллюстрирует дальнейшую интеграцию «тяжеловесных» фреймворков с CMS, такими как Kubernetes, и фактически является одним из главных вариантов использования, упомянутых в заявке JIRA, описывающем новую функциональность (<https://oreil.ly/Bvygj>).

Масштабирование приложения путем его перезапуска

Вторая стратегия — масштабирование приложения путем его перезапуска — поддерживается всеми «тяжеловесными» потоковыми фреймворками. Потребление потоков приостанавливается, состояние приложения копируется в контрольную точку, после чего оно останавливается. Затем приложение повторно инициализируется с новыми ресурсами и параллелизмом, а данные о состоянии перезагружаются из контрольных точек по мере необходимости. Например, фреймворт Flink для этих целей обеспечивает простой механизм REST (<https://oreil.ly/ivlYc>), в то время как фреймворт Storm предоставляет свою собственную команду перебалансировки (<https://oreil.ly/g6Q3y>).

Автоматическое масштабирование приложений

Автомасштабирование — это процесс автоматического масштабирования приложений в ответ на специфические метрики. Такие метрики могут включать в себя задержку обработки, задержку потребителя, использование памяти и процессора и другие сходные ситуации. Некоторые фреймворки имеют собственные возможности автомасштабирования, такие как у механизма Google Dataflow, управляющего механизма Health Manager фреймворка Neron и у встроенной в фреймворт Spark функциональности динамического выделения ресурсов Spark Streaming. Другие фреймворки могут потребовать, чтобы вы собирали свои собственные метрики производительности и использования ресурсов и подключали их к механизму масштабирования своего фреймворка — например, к инструментарию монитора задержек, описанному в разд. «Мониторинг задержек смещения потребителей» главы 14.

¹ Имеется в виду ее исходное издание на английском языке. — Прим. ред.

Восстановление после отказов

«Тяжеловесные» кластеры предназначены быть высокоустойчивыми к неизбежным отказам длительных заданий. Отказы ведущих узлов, рабочих узлов и узлов Zookeeper (если они используются) — все это можно смягчить, чтобы приложения могли работать практически без перебоев. Такие отказоустойчивые функциональности встроены в кластерный фреймворк, но могут потребовать настройки дополнительных шагов при развертывании кластера.

В случае отказа ведущего узла исполнявшиеся на этом узле задачи перемещаются на другой доступный ведущий узел. Любое требуемое внутреннее состояние перезагружается из самой последней контрольной точки вместе с назначениями разделов. Отказы ведущего узла должны быть прозрачными для уже исполняемых приложений, но, в зависимости от конфигурации кластера, вы можете оказаться не в состоянии развертывать новые задания во время простоя ведущего узла. Режим высокой доступности, поддерживаемый Zookeeper (или аналогичной технологией), может смягчить потерю ведущего узла.



Убедитесь, что ваша система имеет надлежащий мониторинг и оповещения для ведущих и рабочего узлов. Хотя отказ одного узла кластера не обязательно приведет к остановке обработки, он все же может снизить производительность и помешать приложениям восстанавливаться после отказов, следующих один за другим.

Рекомендации по части мультитенантности

Помимо накладных расходов на управление кластером вы должны учитывать проблемы с мультитенантностью² (multitenancy), возникающие по мере роста в кластере числа приложений. В частности, вам надо не упускать из виду приоритет приобретения ресурсов, соотношение свободных и выделенных ресурсов и скорость, с которой приложения могут претендовать на ресурсы (т. е. масштабирование).

Например, новое потоковое приложение, ранее начавшее работу по своим входным темам, может запрашивать и получать большую часть свободных ресурсов кластера, ограничивая любые текущие приложения от получения ресурсов, требуемых им. Это может привести к тому, что приложения не смогут достичь своих целевых уровней обслуживания (Service-Level Objectives, SLO), и создаст проблемы для бизнеса.

Вот несколько методов для смягчения этих проблем:

- ◆ *Запуск нескольких малых кластеров.*

Каждая команда или бизнес-единица может иметь свой собственный кластер, и они могут быть полностью отделены друг от друга. Такой подход лучше всего

² Мультитенантность (мультиарендуность) — особенность архитектуры ПО, которая позволяет приложению обслуживать несколько независимых арендаторов. Пользователи не мешают друг другу, их данные хранятся независимо и безопасно, а разработчики могут быстро запускать версии продукта с разными техническими возможностями. — Прим. ред.

работает, когда вы можете запрашивать кластеры программно, чтобы снизить операционные накладные расходы, — либо с помощью собственных разработок, либо от стороннего поставщика услуг. Это может повлечь за собой более высокие финансовые затраты из-за накладных расходов на работу кластера, как с точки зрения координирующих узлов (например, ведущих узлов и узлов Zookeeper), так и мониторинга и управления кластерами.

◆ *Организация пространства имен.*

Отдельный кластер можно разделить на пространства имен с определенным назначением ресурсов. При этом каждой команде или бизнес-группе в их собственном пространстве имен могут быть предоставлены собственные ресурсы. Приложения, выполняемые в своем пространстве имен, могут захватывать только эти ресурсы, что предотвращает нехватку ресурсов для приложений за пределами этого пространства имен, как было бы в случае разрешения их агрессивного захвата. Недостатком такого варианта является то, что свободные ресурсы должны быть выделены каждому пространству имен, даже если они ему не нужны, что может привести к увеличению фрагментированного пула неиспользуемых ресурсов.

ЯЗЫКИ И СИНТАКСИС

«Тяжеловесные» фреймворки потоковой обработки основываются на языках группы JVM их предшественников, причем наиболее распространенным является Java, за которым следует Scala. Python также широко представлен, поскольку этот язык весьма популярен среди исследователей данных и специалистов по машинному обучению, которые составляют большую часть традиционных пользователей этих фреймворков. Обычно используются API в стиле MapReduce, где неизменяемые операции над наборами данных соединяются в цепочку. Так что «тяжеловесные» фреймворки довольно-таки ограничены в языках, которые поддерживают их API.

SQL-подобные языки также приобретают все большее распространение. Они позволяют выражать топологии в терминах трансформаций SQL и снижают издержки, связанные с изучением конкретного API для нового фреймворка. Spark (<https://spark.apache.org/sql>), Flink (<https://oreil.ly/jskNu>), Storm (https://oreil.ly/kKg_M) и Beam (<https://oreil.ly/ijfYX>) — все эти фреймворки предоставляют SQL-подобные языки, хотя они и различаются по функциональности и синтаксису, и не все операции в них поддерживаются.

Выбор фреймворка

Выбор «тяжеловесного» фреймворка потоковой обработки очень похож на выбор CMS и брокера событий. Вы должны определиться с тем, сколько операционных накладных расходов ваша организация может себе позволить и достаточно ли этого для запуска полного производственного кластера в крупном масштабе. Накладные расходы включают в себя регулярные операционные обязанности, такие как

мониторинг, масштабирование, устранение неполадок, отладка и распределение затрат, — все это сопутствует реализации и развертыванию актуальных приложений.

Поставщики программного обеспечения могут предлагать свои платформы в качестве сервисов, хотя их возможности, как правило, более ограничены, чем выбор поставщиков для вашей CMS и брокера событий. Оцените доступные вам варианты и выберите соответствующий.

Не исключено, что ваше решение будет определять также и популярность фреймворка. Например, чрезвычайно популярен Spark, тогда как Flink и Storm менее популярны, но все еще активно используются. Приложения могут быть написаны независимо от среды выполнения «тяжеловесного» фреймворка через Apache Beam (https://oreil.ly/_HDF), хотя такой поход может не соответствовать политике вашей организации. Фреймворк Heron (переработанная форма Storm), предлагающий более продвинутые функции, похоже, является наименее популярным из всех имеющихся вариантов. Так что опирайтесь при выборе «тяжеловесного» фреймворка или воздержания от его использования на те же соображения, которым вы следовали при выборе ваших CMS и брокера событий.



Имейте в виду, что «тяжеловесный» потоковый фреймворк не способен разумно реализовать все событийно-управляемые микросервисы. Убедитесь, что выбор его — это правильное решение для вашей проблемной области, прежде чем на нем остановиться.

Пример: обработка щелчков и просмотров с помощью сеансовых окон

Представьте себе, что вы управляете простой рекламной компанией. Вы покупаете в Интернете рекламное место и перепродааете его своим клиентам. Эти клиенты хотят видеть рентабельность своих инвестиций, которая в рассматриваемом случае измеряется рейтингом щелчков («кликов») пользователей, которым показывают рекламу. Кроме того, клиентам может начисляться оплата за каждый сеанс, который определяется как непрерывная активность пользователя с перерывами не более 30 минут.

В этом примере имеются два потока событий: просмотры пользователями рекламы и их щелчки на рекламе. Цель состоит в том, чтобы объединить эти два потока в сеансовые окна и выдать их, когда *время события* (не текущее фактическое время) продолжительностью 30 минут прошло без выполнения пользователем каких-либо новых действий. Обратитесь к главе 6, чтобы вспомнить о времени потока (stream time) и водяных знаках (watermarks).

Обычно при объединении этих поведенческих событий вы можете ожидать получения дополнительной информации в поле значений — например, о месте, где была опубликована реклама, информацию о веб-браузере или устройстве пользователя или другие различные контексты или метаданные. Для этого примера потоки

событий просмотра и щелчка были упрощены до следующего формата базовой схемы:

Ключ	Значение	Метка времени
String userId	Long advertisementId	ALong createdEventTime (местное время создания события)

Здесь вам необходимо выполнить следующие операции:

1. Сгруппировать все ключи таким образом, чтобы все события для конкретного пользователя были локальными для обрабатывающего экземпляра.
2. Агрегировать события, используя окно с 30-минутным тайм-аутом.
3. Создать окно событий, как только будет достигнут 30-минутный лимит.

Выходной поток соответствует следующему формату:

Ключ	Значение
<Window windowID, String userId>	Action[] sequentialUserActions

Объект Window здесь указывает время начала и окончания работы окна. Он является частью составного ключа, т. к. с течением времени пользователи будут иметь по несколько сеансовых окон, которые могут также дублироваться между пользователями. Наличие же составного ключа обеспечивает уникальность. Массив объектов Action в столбце Значение используется для хранения действий в последовательном порядке и позволяет микросервису вычислять то, какие рекламные просмотры приводят его пользователей к оплачиваемым щелчкам. Класс Action может быть представлен следующим образом:

```
Action {
    Long eventTime;
    Long advertisementId;
    Enum action; //одно действие из Click и View
}
```

Приведенный далее сокращенный исходный код Apache Flink показывает топологию с использованием API в стиле MapReduce:

```
DataStream clickStream = ... //Создать поток событий щелчка
DataStream viewStream = ... //Создать поток событий просмотра

clickStream
    .union(viewStream)
    .keyBy(<key selector>)
    .window(EventTimeSessionWindows.withGap(Time.minutes(30)))
    .aggregate(<aggregator function>)
    .addSink(<producer to output stream>)
```

Визуальное представление этой топологии показано на рис. 11.9 с параллелизмом 2 (обратите внимание на два отдельных экземпляра).

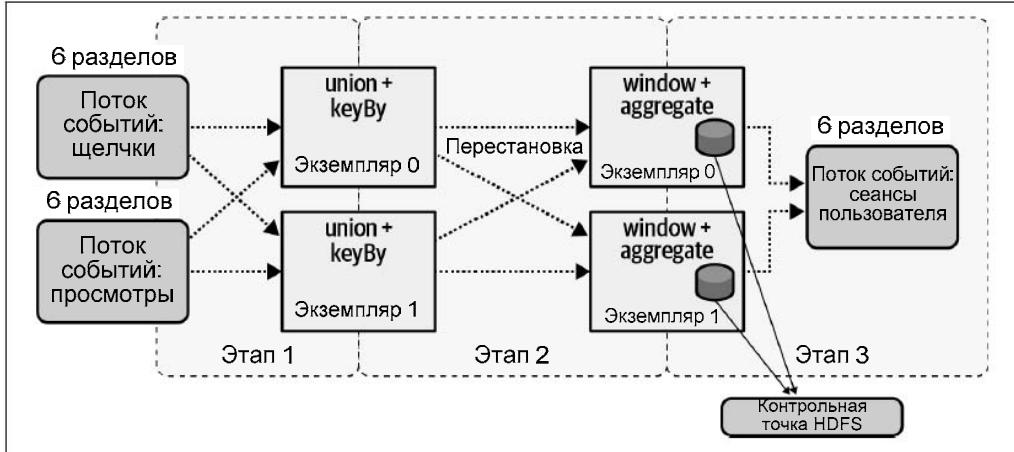


Рис. 11.9. Топология обработки, генерирующая сеанс на основе пользовательских просмотров и щелчков мышью

- ◆ **Этап 1** — исполнителям для каждого экземпляра назначаются свои задачи, которым, в свою очередь, назначаются разделы обрабатываемого входного потока событий. Поток щелчков и поток просмотров объединяются в один логический поток, который затем группируется по ключу `userId`.
- ◆ **Этап 2** — оператор `keyBy` в сочетании с нижестоящими операторами `window` и `aggregate` требует перераспределения теперь объединенных событий в правильные нижестоящие экземпляры. Все события для этого ключа потребляются в одном экземпляре, обеспечивая необходимую локальность данных для остальных операций.
- ◆ **Этап 3** — теперь, когда события каждого пользователя стали локальными для одного экземпляра, могут быть сгенерированы сеансовые окна для каждого пользователя. События добавляются в локальное хранилище состояний в последовательном порядке временных меток, причем функция агрегации применяется к каждому событию до тех пор, пока не будет обнаружен перерыв в 30 минут или более. На текущий момент хранилище событий выталкивает завершившийся сеанс и очищает память от ключа и значения `<windowId, userId>`.



Ваш фреймворк может обеспечивать дополнительный контроль над оконными и временными агрегациями. Сюда может входить удержание сеансов и окон, которые были закрыты в течение определенного периода времени, чтобы можно было применять запоздавшие события и передавать обновление в выходной поток. Обратитесь к документации вашего фреймворка для получения дополнительной информации на этот счет.

На рис. 11.10 показаны эффекты масштабирования вниз до единственной степени параллелизма. Предполагая, что динамическое масштабирование отсутствует, вам нужно будет остановить потоковый процессор перед его восстановлением из контрольной точки с новой настройкой параллелизма. После запуска служба считывает данные с поддержкой состояния по ключу из последней заведомо исправной кон-

трольной точки и восстанавливает состояние оператора в назначенных разделах. После восстановления состояния служба может возобновить нормальную обработку потока.

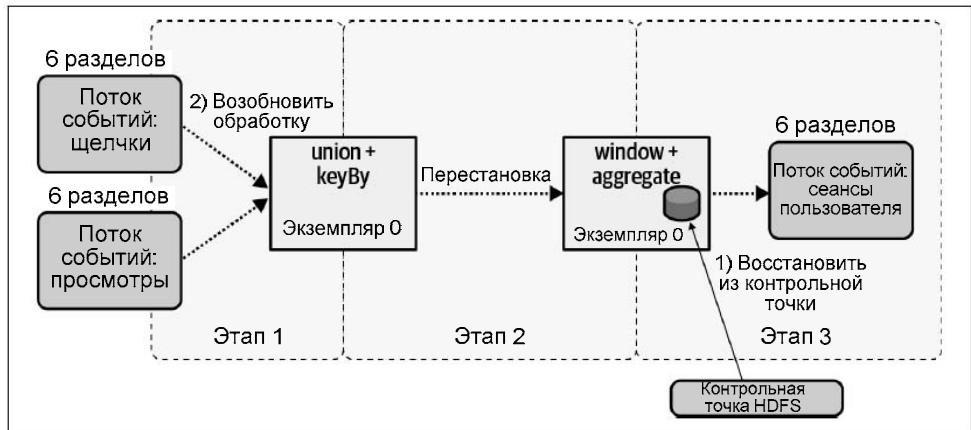


Рис. 11.10. Топология обработки, генерирующая сеанс без параллелизма

Этап 1 работает здесь так же, как и раньше, хотя в рассматриваемом случае все разделы назначаются для потребления задачами в экземпляре 0. Группировка и перераспределение по-прежнему выполняются, хотя источник и обработчик остаются тем же экземпляром, что и на этапе 2. Имейте в виду, что отдельные задачи, выполняемые на экземпляре 0, должны потреблять назначенные им перераспределенные события, хотя весь обмен информацией здесь является полностью локальным. Последний этап топологии, этап 3, обрабатывает события в окне и агрегирует их, как обычно.

Резюме

В этой главе мы познакомились с «тяжеловесными» фреймворками потоковой обработки, краткой историей их развития и задачами, для решения которых они были созданы. Эти системы обладают высокой масштабируемостью и позволяют обрабатывать потоки в соответствии с различными аналитическими шаблонами, но их возможностей бывает недостаточно для удовлетворения требований некоторых шаблонов приложений с поддержкой состояния и работающих на событийно-управляемых микросервисах.

«Тяжеловесные» фреймворки работают с использованием централизованных кластеров ресурсов, что может потребовать дополнительных операционных накладных расходов, мониторинга и координации для успешной интеграции в структуру микросервисов. Последние инновации в моделях развертывания кластеров и приложений обеспечили лучшую интеграцию с решениями по управлению контейнерами, такими как Kubernetes, что позволило более успешно развертывать «тяжеловесные» потоковые обработчики, аналогичные полностью независимым микросервисам.

«Легковесные» фреймворки для микросервисов

«Легковесные» фреймворки обеспечивают функциональность, аналогичную «тяжеловесным» фреймворкам, но таким образом, чтобы в значительной степени использовать брокер событий и систему управления контейнерами (CMS). В отличие от «тяжеловесных» фреймворков, «легковесные» фреймворки не имеют дополнительного выделенного кластера ресурсов для управления специфичными для фреймворка ресурсами. Горизонтальное масштабирование, управление состоянием и восстановление после отказов обеспечиваются брокером событий и CMS. Приложения развертываются как отдельные микросервисы, так же, как и любой микросервис ВРС. Параллелизм контролируется членством в группе потребителей и владением разделом. Разделы перераспределяются по мере того, как новые экземпляры приложений присоединяются к группе потребителей и покидают ее, включая соподразделенные назначения.

Выгоды и ограничения

«Легковесные» фреймворки предлагают функциональности потоковой обработки, которые соперничают с функциональностями «тяжеловесных» фреймворков, а в ряде случаев их превосходят.

Материализация потоков в таблицы, наряду с простой готовой к использованию функцией объединения, упрощает обработку потоков и реляционных данных, которые неизбежно в них попадают. Обратите внимание, что хотя функциональность материализации таблиц не является уникальной для «легковесных» фреймворков, ее готовность к включению и простота использования показывает, что «легковесные» фреймворки способны решать сложные проблемы с отслеживанием состояния.

«Легковесная» модель полагается на брокер событий, который предоставляет механизмы определения местоположения данных и совместного разделения за счет использования внутренних потоков событий. Брокер событий также действует как механизм долговременного хранения внутреннего состояния микросервиса с помощью журналов изменений, как показано в разд. «Запись состояния в поток событий журнала изменений» главы 7. А задействуя CMS, вы развертываете «легковесные» микросервисы, как и любое другое событийно-управляемое приложение. При этом вы регулируете параллелизм приложений, просто добавляя и удаляя экземпляры и привлекая возможности CMS для обеспечения механизмов масштабирования и управления отказами. На рис. 12.1 показана базовая «легковесная»

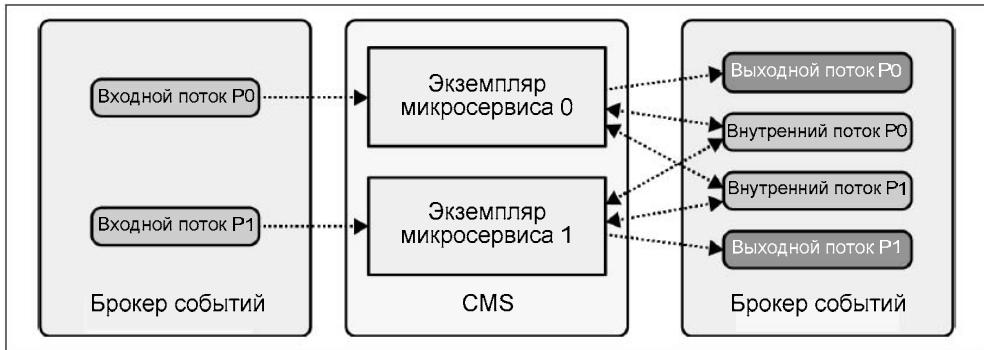


Рис. 12.1. Модель «легковесного» фреймворка, демонстрирующая использование внутренних потоков событий для репартизации данных

модель, включающая внутренний поток событий, используемый для обмена информацией между экземплярами.

Главные ограничения «легковесных» фреймворков связаны в основном с доступностью имеющихся в настоящее время возможностей, рассмотренных далее в этой главе.

«Легковесная» обработка

«Легковесный» фреймворк точно повторяет методологию обработки, свойственную «тяжеловесным» фреймворкам. Отдельные экземпляры обрабатывают события в соответствии с топологией, при этом брокер событий обеспечивает уровень связи между экземплярами для масштабируемости за пределами одного экземпляра.

Данные одного и того же ключа должны быть локальными для конкретного обрабатывающего экземпляра при любых операциях на основе ключей, таких как `join` (соединение) или `groupByKey`, за которыми идет последующее `reduce/aggregate` (сокращение/агрегирование). Эти перераспределения предусматривают отправку событий через внутренний поток событий, при этом каждое событие того или иного ключа записывается в один раздел (см. разд. «Соподразделение потоков событий» главы 5), вместо использования прямого обмена данными между экземплярами.

«Легковесный» фреймворк задействует брокер событий для обеспечения этого пути обмена информацией и демонстрирует более глубокую интеграцию «легковесного» приложения с брокером событий. Сравните это с «тяжеловесным» фреймворком, где перераспределение требует обширной координации непосредственно между узлами. В сочетании с возможностями управления приложениями, предоставляемыми CMS, «легковесные» фреймворки гораздо лучше согласованы с развертыванием приложений и управлением, требуемыми для современных микросервисов, чем фреймворки «тяжеловесные».

Обработка состояния и использование журналов изменений

По умолчанию «легковесные» фреймворки работают в режиме использования внутреннего состояния, поддерживаемого журналами изменений, хранящимися в брокере событий. Использование внутреннего состояния позволяет каждому микросервису контролировать ресурсы, которые он приобретает с помощью конфигураций развертывания.



Поскольку каждое «легковесное» приложение полностью независимо от других, одно приложение может запрашивать запуск на экземплярах с очень высокопроизводительным локальным диском, а другое — на экземплярах с очень большими, хотя, возможно, гораздо более медленными жесткими дисками.

Кроме того, могут быть подключены разные механизмы хранения, что позволяет использовать внешние хранилища состояний с альтернативными моделями и механизмами запросов. Тем самым достигаются все выгоды от функциональности «легковесного» фреймворка, а также задействуются такие возможности, как графовые базы данных и документные хранилища.

В отличие от модели контрольных точек, принятой в «тяжеловесных» фреймворках, «легковесные» фреймворки, использующие внутренние хранилища состояний, задействуют для хранения своих журналов изменений брокер событий. Эти журналы изменений обеспечивают надежность хранения данных, необходимую как для масштабирования, так и для восстановления после отказов.

Масштабирование приложений и восстановление после отказов

Масштабирование микросервиса и его восстановление после отказов — это фактически один и тот же процесс. Добавление экземпляра приложения из-за предна меренного масштабирования длительного процесса или из-за отказавшего восста новления экземпляра требует, чтобы разделы были правильно назначены вместе с любым сопутствующим состоянием. Точно так же удаление экземпляра, предна меренное или из-за отказа, требует переназначения разделов и состояния другому жизнеспособному экземпляру, чтобы обработка могла продолжаться непрерывно.

Одной из главных выгод модели «легковесного» фреймворка является возможность динамического масштабирования приложений по мере их исполнения. Нет необходимости перезапускать приложение только для изменения параллелизма, хотя может возникнуть задержка в обработке, вызванная перебалансированием группы потребителей и рематериализацией состояния из журнала изменений. На рис. 12.2 показан процесс масштабирования приложения. Назначенные входные разделы перебалансируются (включая любые внутренние потоки), и состояние, которое было до момента продолжения работы, восстанавливается из журналов изменений.

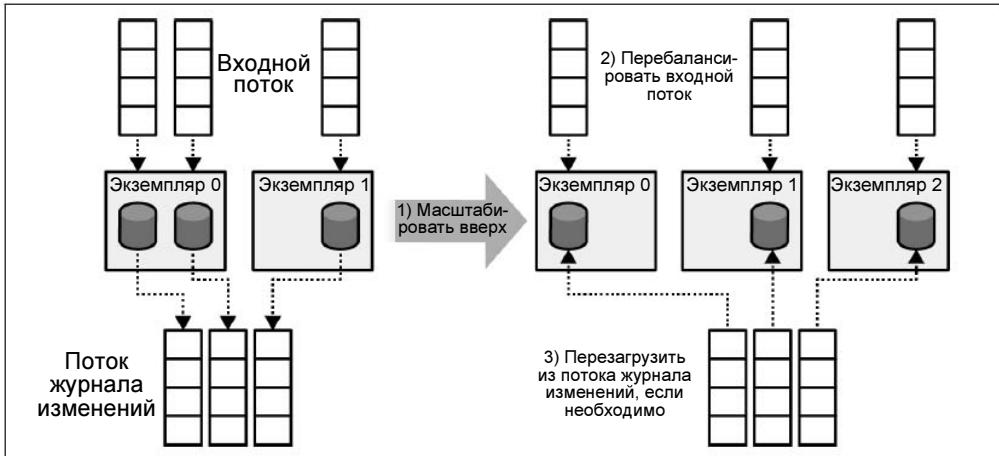


Рис. 12.2. Масштабирование «легковесного» микросервиса

Давайте рассмотрим основные аспекты масштабирования «легковесного» приложения.

Перераспределение событий

Перераспределение событий (event shuffling) в микросервисах «легковесного» фреймворка выполняется просто, поскольку события для нижестоящего потребления переподразделяются на внутренний поток событий. Этот внутренний поток событий изолирует вышестоящие экземпляры, создающие перераспределенные события, от нижестоящих, которые их потребляют, действуя как сервис перераспределения, аналогичный тому, который нужен «тяжеловесным» фреймворкам для выполнения динамического масштабирования. Любое динамическое масштабирование требует только независимого от производителей переназначения потребителей во внутренний поток событий.

Назначение состояния

После масштабирования экземпляр с новыми назначениями внутренних состояний перед обработкой любых новых событий должен загрузить данные из журнала изменений. Этот процесс аналогичен тому, как в «тяжеловесных» решениях из долговременного хранилища загружаются контрольные точки. *Операторное состояние* (сопоставления $\langle\text{partitionId}, \text{offset}\rangle$) для всех разделов потока событий, как входных, так и внутренних, хранится в группе потребителей индивидуального приложения. *Состояние по ключу* (пары $\langle\text{key}, \text{state}\rangle$) хранится внутри журнала изменений для каждого хранилища состояний в приложении.

Во время перезагрузки из журнала изменений экземпляр приложения должен перед обработкой любых новых событий определить приоритет потребления и загрузки всех внутренних данных с поддержкой состояния. В этом заключается фаза восстановления состояния, и любая обработка событий перед полным восстановлением

состояния рискует привести к неопределенным результатам. И только после полного восстановления состояния для каждого хранилища состояний в топологии приложения потребление как входных, так и внутренних потоков может быть безопасно возобновлено.

Репликация состояния и «горячие реплики»

«Горячая реплика», представленная в разд. «*Использование “горячих реплик”*» главы 7, является копией хранилища состояний, материализованной из журнала изменений. Она обеспечивает резервный откат на случай отказа первичного экземпляра, обслуживающего эти данные, но также может использоваться для масштабирования вниз приложений с поддержкой состояния. После завершения работы экземпляра и перебалансировки группы потребителей можно назначить разделы для использования состояния «горячей реплики» и продолжения обработки без прерывания. «Горячие реплики» позволяют поддерживать высокую доступность при масштабировании и отказах, но их использование требует дополнительных ресурсов диска и процессора.

Точно так же вы можете использовать «горячие реплики» для плавного увеличения количества экземпляров без задержек от пауз обработки из-за рематериализации состояния на новом узле. Одна из текущих проблем, с которыми сталкиваются «легковесные» фреймворки, заключается в том, что масштабирование экземпляра вверх обычно подчиняется текущему рабочему процессу:

1. Запустить новый экземпляр.
2. Присоединиться к группе потребителей и перебалансировать владение разделами.
3. Сделать паузу, пока состояние материализуется из журнала изменений (это может занять некоторое время).
4. Возобновить обработку.

Один из вариантов состоит в том, чтобы заполнить реплику состояния на новом экземпляре, подождать, пока он не догонит заголовок журнала изменений, а затем перебалансировать его, чтобы назначить ему владение входными разделами. Такой режим уменьшает простои из-за материализации потоков журнала изменений и требует только дополнительной полосы пропускания от брокера событий. Эта функциональность сейчас разрабатывается для фреймворка Kafka Streams (<https://oreil.ly/pGqFs>).

Выбор «легковесного» фреймворка

В настоящее время существуют два основных варианта, которые соответствуют модели «легковесного» фреймворка, и оба требуют использования брокера событий Apache Kafka. Они обеспечивают неограниченное сохранение материализованных потоков в своих высокуровневых API, открывая такие возможности, как соединения по первичному ключу (primary-key joins) и соединения по внешнему ключу

(foreign-key joins). Эти шаблоны соединений позволяют обрабатывать реляционные данные без необходимости материализоваться во внешних хранилищах состояний и, следовательно, снижают издержки на подготовку персонала для создания приложений на основе соединений.

Apache Kafka Streams

Kafka Streams — это насыщенная по функциональности библиотека потоковой обработки, которая встраивается в отдельное приложение, где входные и выходные события хранятся в кластере Kafka. Она сочетает в себе простоту написания и развертывания стандартных приложений на базе JVM с мощным фреймворком потоковой обработки, обеспечивающим глубокую интеграцию с кластером Kafka.

Apache Samza: режим встраивания

Samza предлагает многие из тех же функциональностей, что и Kafka Streams, хотя и отстает в некоторых из них, связанных с независимым развертыванием. Samza предшествует Kafka Streams, и ее первоначальная модель развертывания основана на использовании «тяжеловесного» кластера. Только относительно недавно Samza выпустила встроенный режим, который полностью отражает жизненный цикл написания, развертывания и управления приложениями Kafka Streams.

Режим встраивания Samza позволяет добавлять ее функциональность в отдельные приложения (<https://oreil.ly/Dv6Ov>), как и любая другая библиотека Java. Этот режим развертывания устраняет необходимость в выделенном «тяжеловесном» кластере, вместо этого полагаясь на модель «легковесного» фреймворка, рассмотренную в предыдущем разделе. По умолчанию Samza использует Apache Zookeeper для координации между отдельными экземплярами, но вы можете изменить это, чтобы использовать другие механизмы координации наподобие Kubernetes.



Режим встраивания Apache Samza может не обеспечивать всю ту функциональность, которую он предоставляет в кластерном режиме.

«Легковесные» фреймворки не так распространены, как «тяжеловесные» фреймворки или библиотеки потребитель/производитель для шаблона базового потребителя/производителя. «Легковесные» фреймворки во многом полагаются на интеграцию с брокером событий, что ограничивает их переносимость на другие технологии брокера событий. Предметная область «легковесных» фреймворков еще весьма молода, но она обязательно будет расти и развиваться по мере взросления пространства EDM.

ЯЗЫКИ И СИНТАКСИС

И Kafka Streams, и Samza основаны на Java, что ограничивает их использование с применением языков на основе JVM. Высокоуровневые API выражаются в форме синтаксиса MapReduce, как это имеет место в языках «тяжеловесных» фреймвор-

ков. Те, кто имеет опыт работы с функциональным программированием или с любым из «тяжеловесных» фреймворков, рассмотренных в предыдущей главе, будут чувствовать себя как дома, используя любой из этих фреймворков.

Apache Samza поддерживает SQL-подобный язык прямо «из коробки» (<https://oreil.ly/qeLIK>), хотя его функциональность в настоящее время ограничена простыми запросами без поддержки состояния. Kafka Streams не имеет сторонней поддержки SQL, хотя ее корпоративный спонсор Confluent (<https://oreil.ly/LJVOv>), предоставляет KSQL под лицензией собственного сообщества. Как и «тяжеловесные» решения, эти SQL-решения являются обертками поверх опорных потоковых библиотек и могут не предоставлять всю полноту функций и функциональных возможностей, которые в противном случае были бы доступны непосредственно из потоковых библиотек.

Операция соединения «поток-таблица-таблица»: шаблон обогащения

Допустим, вы работаете в той же крупной рекламной компании, о которой шла речь в разд. «Пример: обработка щелчков и просмотров с помощью сеансовых окон» главы 11, но являетесь нижестоящим потребителем сеансовых окон. В качестве краткого напоминания формат событий оконных сеансов показан в табл. 12.1.

Таблица 12.1. Определение ключей и значений потока рекламных сеансов (Advertisement-Sessions)

Ключ	Значение
WindowKey<Window windowId, String userId>	Action[] sequentialUserActions

Здесь представлено следующее действие:

```
Action {
    Long eventTime;
    Long advertisementId;
    Enum action; // одно действие из Click или View
}
```

Цель вашей команды состоит в том, чтобы взять поток рекламных сеансов Advertisement-Sessions и сделать следующее:

1. Для каждого действия по просмотру рекламы определить, есть ли последующий щелчок. Просуммировать каждую пару «просмотр-щелчок» и вывести результат в качестве события конверсии, как показано в табл. 12.2.

Таблица 12.2. Определение ключей и значений потока рекламных конверсий (Advertisement-Conversions)

Ключ	Значение
Long advertisementId	Long conversionSum

2. Сгруппировать все события рекламной конверсии по значению advertisementId и сложить их значения в итоговую сумму.
3. Объединить все события конверсии по значению advertisementId в поток материализованного рекламного объекта, чтобы ваш сервис мог определить, какой клиент владеет рекламой для начисления оплаты (табл. 12.3).

Таблица 12.3. Определение ключей и значений потока рекламных объявлений

Ключ	Значение
Long advertisementId	Advertisement<String name, String address, ...>

Вот пример исходного кода Kafka Streams, который мог бы соответствовать этой операции. Здесь Kstream — это высокоуровневая абстракция потока, а Ktable — высокоуровневая абстракция таблицы, сгенерированная материализацией потока объекта рекламы.

```
KStream<WindowKey,Actions> userSessions = ...

// Трансформировать 1 сеанс пользователя (userSession)
// в конверсионные события от 1 до N,
// переназначить ключ на AdvertisingId
KTable<AdvertisingId,Long> conversations = userSessions
    .transform(...) // Трансформировать сеансы пользователя (userSessions)
        // в конверсионные события
    .groupByKey()
    .aggregate(...) // Создает агрегатную таблицу KTable

// Материализовать рекламные сущности
KTable<AdvertisingId,Advertisement> advertisements = ...

// Таблица автоматически соподразделяется путем включения
// операции соединения в топологию
conversations
    .join(advertisements, joinFunc) // Подробности см. в этапе 4.
    .to("AdvertisementEngagements")
```

Представленная этим кодом топология выглядит так, как показано на рис. 12.3¹.

◆ Этапы 1a и 2a.

Поток Advertisement-Sessions (рекламные сеансы) содержит слишком много событий для обработки одним экземпляром, поэтому код необходимо распараллить с использованием нескольких обрабатывающих экземпляров. В этом примере максимальный уровень параллелизации изначально равен 3, т. к. это число разделов потока сущностей Advertisements. В непиковые часы можно использо-

¹ Чтобы соответствие кода и топологии было более понятным, часть надписей на рис. 12.3 оставлена в исходном варианте, а их перевод приведен отдельно. — Прим. ред.

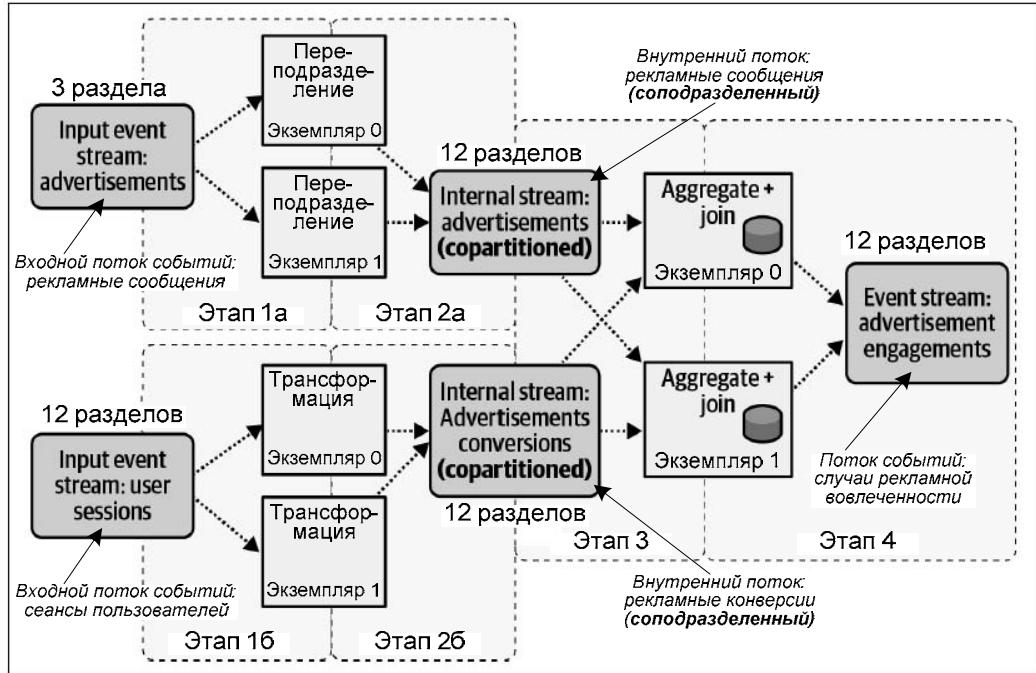


Рис. 12.3. Обрабатывающая топология для потока «рекламная вовлеченность-сеансы»

вать только один-два экземпляра, но в периоды интенсивной активности пользователей приложение будет отставать.

К счастью, поток сущности `Advertisements` может быть переподразделен до согласованных 12 разделов с помощью внутреннего потока. События просто потребляются и переподразделяются в новый внутренний поток из 12 разделов. Рекламные объекты сорасполагаются на основе `AdvertisingId` с событиями конверсии из этапов 1б и 2б.

◆ Этапы 1б и 2б.

События потребляются из потока событий `Advertisement-Sessions` (рекламные сеансы) с табулированием и отправкой событий конверсии (ключ = `Long advertisementId`, значение = `Long conversionSum`). Обратите внимание, что для такого сеансового события может создаваться несколько событий конверсии — по одному для каждой пары событий «просмотр-щелчок» на каждый `advertisementId`. Эти события сорасполагаются на основе `advertisementId` с рекламными объектами из этапов 1а и 2а.

◆ Этап 3.

События `Advertisement-Conversions` (рекламные конверсии) теперь должны быть агрегированы в формат материализованной таблицы, показанной в табл. 12.4, поскольку предприятие заинтересовано в том, чтобы вести постоянный учет взаимодействий с каждым рекламным событием `Advertisement`. Агрегация представляет собой простую сумму всех значений для каждого `advertisementId`.

Таблица 12.4. Определение ключа/значения потока Total-Advertisement-Conversions
(Итоговые рекламные конверсии)

Long advertisementId	Long conversionSum
AdKey1	402
AdKey2	600
AdKey3	38

Таким образом, новое событие Advertisement-Conversions (рекламные конверсии) с ключом (AdKey1, 15), обработанное этим оператором агрегации, увеличит значение внутреннего хранилища состояния AdKey1 с 402 до 417.

◆ Этап 4.

Последний шаг этой топологии состоит в том, чтобы соединить материализованную таблицу Total-Advertisement-Conversions (Итоговые рекламные конверсии), созданную на *этапе 3*, с переподразделенным потоком сущностей Advertisement. Вы уже заложили основу для этого объединения, соподразделив входные потоки на этапы 2a и 2b и обеспечив, чтобы все advertisementId были локальными для его экземпляра обработки. Сущности в рекламном потоке Advertisement материализуются в свои собственные локальные для раздела хранилища состояний и впоследствии соединяются с Total-Advertisement-Conversions.

Функция соединения (join) используется для указания желаемых результатов соединения, так же как инструкция select применяется в SQL для отбора только тех полей, которые требуются приложению. Функция соединения на языке Java для этого сценария может выглядеть следующим образом:

```
public EnrichedAd joinFunction (Long sum, Advertisement ad) {
    if (sum != null || ad != null)
        return new EnrichedAd(sum, ad.name, ad.type);
    else
        // Вернуть отметку об удалении, т. к. один из элементов
        // входных данных равен null, возможно, указывая на удаление.
        return null;
}
```

Присутствующая здесь функция joinFunction исходит из того, что любой вход может равняться null, что указывает на предшествующее удаление этого события. Соответственно, вам нужно обеспечить, чтобы ваш код выдавал отметки об удалении событий для своих потребителей. К счастью, большинство фреймворков (как «легковесных», так и «тяжеловесных») различают соединения внутренние, левые, правые, внешние и по внешнему ключу и выполняют некоторые закулисные операции, чтобы избавить вас от распространения отметок об удалении через ваши функции соединения. Однако важно учитывать значение отметок об удалении в топологии микросервисов.

Топология Kafka Streams и функция joinFunction идентичны выражению отбора полей в SQL:

```
SELECT adConversionSumTable.sum, adTable.name, adTable.type
FROM adConversionSumTable FULL OUTER JOIN adTable
ON adConversionSumTable.id = adTable.id
```

В этом случае материализованный вид потока событий *Enriched-Advertising-Engagements* будет выглядеть, как показано в табл. 12.5.

Таблица 12.5. Определение ключей и значений потока *Enriched-Advertising-Engagements*
(Расширенное рекламное взаимодействие)

AdvertisementId (ключ)	Enriched advertisements (значение)
AdKey1	sum=402, name="Josh's Gerbils", type="Pets"
AdKey2	sum=600, name="David's Ducks", type="Pets"
AdKey3	sum=38, name="Andrew's Anvils", type="Metalworking"
AdKey4	sum=10, name="Gary's Grahams", type="Food"
AdKey5	sum=10, name=null, type=null

Эта примерная таблица демонстрирует ожидаемые агрегации из *этапа 3*, соединенные с данными рекламной сущности *Advertising*. Ключи AdKey4 и AdKey5 показывают результаты полного внешнего соединения: для AdKey4 еще не произошло конверсий, а для AdKey5 еще нет данных о рекламных сущностях.



Обратитесь к документации на свой фреймворк, чтобы проверить, какие типы соединений для него доступны. Kafka Streams поддерживает соединения таблиц по внешнему ключу, которые могут быть чрезвычайно полезны для обработки реляционных событийных данных.

Резюме

В этой главе были представлены «легковесные» фреймворки потоковой обработки и рассмотрены их основные выгоды и компромиссы. Они представляют собой высокомасштабируемые обрабатывающие фреймворки, которые широко опираются на интеграцию с брокером событий для выполнения крупномасштабной обработки данных. Интенсивная интеграция с системой управления контейнерами обеспечивает реализацию требований к масштабируемости для каждого отдельного микросервиса.

«Легковесные» фреймворки все еще относительно новы по сравнению с «тяжеловесными». Тем не менее функциональности, которые они предоставляют, как правило, хорошо подходят для создания долгосрочных и независимых микросервисов с поддержкой состояния, и, безусловно, стоит обратить на них внимание для использования в своих бизнес-сценариях.

Интегрирование событийно-управляемых микросервисов с микросервисами типа «запрос-ответ»

Какими бы мощными ни были шаблоны событийно-управляемых микросервисов, они не способны удовлетворить все бизнес-требования организации. В то же время конечные точки «запрос-ответ» предоставляют средства для передачи важных данных в реальном времени, в особенности когда вы:

- ◆ собираете данные из внешних источников — таких как приложения на смартфоне пользователя или устройства Интернета вещей (IoT);
- ◆ интегрируетесь с существующими приложениями «запрос-ответ» — в особенностях со сторонними приложениями за пределами организации;
- ◆ раздаете контент в реальном времени пользователям веб-сайтов и мобильных устройств;
- ◆ раздаете динамические запросы, основываясь на информации реального времени — такой как местоположение, время и погода.

Событийно-управляемые шаблоны по-прежнему играют важную роль в этой предметной области, и интеграция их с решениями типа «запрос-ответ» поможет вам задействовать преимущества обоих.



В этой главе термин «сервис “запрос-ответ”» относится к сервисам, которые обмениваются друг с другом напрямую — обычно через синхронный API. Ярким примером обмена «запрос-ответ» являются два сервиса, обменивающиеся через HTTP.

Обработка внешних событий

Вы, конечно, можете предоставить внешнему клиенту доступ к брокеру событий и его потокам, однако это в значительной степени неразумно, поскольку вам тогда потребуется решить комплекс вопросов, связанных с доступом и безопасностью. Поэтому в силу целого ряда причин событийно-управляемые микросервисы преимущественно получают внешние события от внешних источников через API «запросов-ответов». API «запросов-ответов» прекрасно работают для таких сценариев, как и десятилетия назад. Существуют два основных типа внешне генерируемых событий.

Автономно генерируемые события

Первый тип событий — это события, которые *автономно* отсылаются от клиента на сервер вашими продуктами. Такие запросы обычно представляют собой информацию о действиях пользователя, периодически снимаемые показатели активности продукта или параметры от каких-либо датчиков. Вместе именуемые *аналитическими событиями*, они содержат данные конкретных измерений тех или иных величин и сведения о действиях с продуктом (приведенный в главе 3 «Пример: перегрузка определений событий» демонстрирует такое событие). Хорошим примером внешнего источника событий может служить и установленное на мобильном телефоне клиента приложения. Средства работы с потоковыми медиасервисами, такими как Netflix, также независимо отправляют аналитические события, чтобы учесть, какие фильмы вы начинали смотреть и сколько из них посмотрели. Любой запрос от внешнего продукта, основанный на действиях, исходящих от этого продукта, считается внешне сгенерированным событием.

Теперь вы можете задаться вопросом, считаются ли, скажем, запросы на загрузку следующих 60 секунд текущего фильма внешне сгенерированным событием. Безусловно, так оно и есть. Но реальный вопрос заключается в следующем: «Являются ли эти события достаточно важными для предприятия, чтобы они вошли в свой собственный поток событий для дополнительной обработки?» Во многих случаях ответ «нет», и вы не станете собирать и хранить эти события в потоке событий. Но в тех случаях, когда ответ «да», вы можете просто проанализировать и конвертировать такой запрос в событие и направить его в свой собственный поток событий.

Реактивно генерируемые события

Второй тип внешне генерируемого события — это *реактивное* событие, которое генерируется *в ответ* на запрос одного из ваших сервисов. Ваш сервис составляет запрос, отправляет его в конечную точку и ожидает ответа. В некоторых случаях действительно важно лишь убедиться, что запрос получен и запрашивающему клиенту не нужны никакие другие детали из ответа. Например, если вам надо выдавать запросы на отправку рекламных писем, то сбор ответов от сторонних сервисов, которые обрабатывают эти запросы, и создание событий на основе таких ответов не принесут никакой пользы. Как только запрос будет успешно выдан (ответ HTTP 202), вы можете допустить, что стороннее почтовое приложение его выполнит. Сбор ответов и их конвертирование в события могут не понадобиться, если нет никаких действий, которые можно выполнить на основе результатов этих событий.

С другой стороны, ваши бизнес-требования могут предполагать значительную детализацию от ответа на запрос. Ярким примером тому является использование стороннего платежного сервиса, где во входном событии содержится сумма, которая должна быть выплачена клиентом. Полезная нагрузка ответа от стороннего API чрезвычайно важна, поскольку она информирует, удался платеж или нет, а также содержит любые сообщения об ошибках и любые дополнительные подробности — такие как уникальный отслеживаемый номер с указанием платежной информации.

Эти данные важно поместить в поток событий, поскольку они позволяют нижестоящим бухгалтерским сервисам сверять кредиторскую задолженность с полученными платежами.

Обработка автономно генерируемых аналитических событий

Аналитические события можно объединять вместе и периодически отправлять в пакете, могут они также отправляться и по мере их возникновения. В любом случае они будут отправлены в API «запроса-ответа», откуда затем их можно будет перенаправить в соответствующие потоки событий. Это показано на рис. 13.1, где внешнее клиентское приложение отправляет аналитические события сервису-приемнику событий, который направляет их в правильный выходной поток событий.



Рис. 13.1. Сбор аналитических событий из внешнего источника



Используйте схемы для кодирования событий при их создании на стороне клиента. Это обеспечивает высокоточный источник, уменьшающий вероятность неправильного их истолкования нижестоящими потребителями, в то же время предоставляя производителям необходимые условия для создания и заполнения своих событий.

Представленные в виде схемы события важны для массового использования аналитических событий. Схема подробно описывает то, что собирается, чтобы пользователи позже могли понять смысл события. Схемы также предоставляют механизм для контроля версий и развития и возлагают бремя заполнения, проверки и тестирования событий на их производителей (разработчиков приложения), а не на потребителей (внутренних получателей и аналитиков). Обеспечение соответствия события схеме *во время его создания* означает, что сервису-приемнику более не нужно интерпретировать и анализировать событие, как это могло бы быть в случае с таким форматом, как простой текст.

Существует ряд ограничений, которые необходимо учитывать при приеме аналитических событий с устройств, на которых могут выполняться несколько версий кода. Например, это особенно распространенный сценарий для любого приложения, работающего на мобильном устройстве конечного пользователя. Добавление новых полей для сбора новых данных или прекращение сбора других данных о событиях, безусловно, разумно. Однако, хотя вы можете заставить пользователей обновлять

свои приложения, блокируя старые версии, нереально заставить их обновлять приложение для каждого небольшого изменения. Планируйте несколько версий аналитических событий, как показано на рис. 13.2.



Рис. 13.2. Внешние источники, генерирующие аналитические события с разными версиями



Представляйте себе внешние источники событий как набор экземпляров микросервисов. Каждый экземпляр создает события по схеме в поток событий через сервис-приемник событий.

Наконец, важно отсортировать входящие события в их собственные определенные потоки событий на основе их схем и определений событий. Разделяйте эти события в соответствии с бизнес-целями точно так же, как вы разделили бы потоки событий любого другого микросервиса.

Интегрирование со сторонними API «запрос-ответ»

Событийно-управляемые микросервисы часто должны обмениваться информацией со сторонними API через протоколы «запросов-ответов». Шаблон «запрос-ответ» прекрасно вписывается в событийно-управляемую обработку — запрос и ответ рассматриваются просто как вызов удаленной функции. Микросервис вызывает API на основе событийно-управляемой логики и ожидает ответа. Получив ответ, микросервис анализирует его, обеспечивает, чтобы он соответствовал схеме, и продолжает применять бизнес-логику, как если бы это было любое другое событие. Обобщенный пример этого процесса показан на рис. 13.3.

Следующий фрагмент исходного кода иллюстрирует логическую операцию микросервиса с использованием блокирующего вызова:

```

while (true) { // Бесконечный цикл обработки
    Event[] eventsToProcess = Consumer.consume("input-event-stream");
    for (Event event: eventsToProcess) {
        // Применить бизнес-логику к текущему событию для генерирования
        // любого необходимого запроса
        Request request = generateRequest(event, ...);

        // Выполнить запрос к внешней конечной точке. Указать тайм-ауты,
        // повторные попытки и т. д.
    }
}
  
```

```

// В этом коде используется блокирующий вызов в ожидании ответа.
Response response =
    RequestService.makeBlockingRequest(request, timeout, retries, ...);

// HTTP-ответ. Если успешно,
// то разобрать + применить бизнес-логику.
if (response.code == 200) {
    // Разобрать результаты в объект
    // для использования в приложении
    <Class Type> parsedObj = parseResponseToObject(response);

    // Применить любую дополнительную бизнес-логику,
    // если необходимо.
    OutputEvent outEvent = applyBusinessLogic(parsedObj, event, ...);
    // Записать результаты в выходной поток событий.
    Producer.produce("output-stream-name", outEvent);
} else {
    // Ответ не равен 200.
    // Вы должны решить, как урегулировать эти условия.
    // Повторная попытка, отказ, журналирование, пропуск и пр.
}
}

// Зафиксировать смещения только тогда,
// когда вы удовлетворены результатами обработки.
consumer.commitOffsets();
}

```

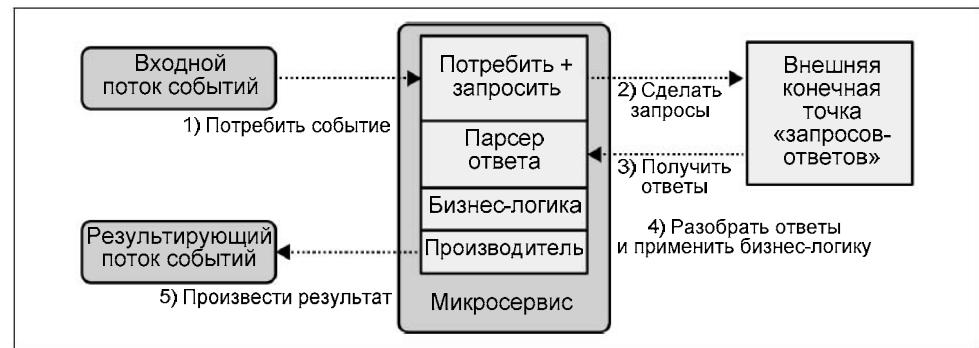


Рис. 13.3. Интегрирование API «запросов-ответов» в событийно-управляемые рабочие процессы

Использование шаблона «запрос-ответ» имеет ряд выгод. Во-первых, он позволяет смешивать обработку событий с API «запросов-ответов» при применении бизнес-логики. Во-вторых, ваш сервис может вызывать любые внешние API, которые ему нужны, для чего бы это ни потребовалось. Вы также можете обрабатывать события параллельно, делая много неблокирующих запросов к конечным точкам. Только после того, как каждый запрос будет отправлен, сервис ожидает результатов, и как

только их получает, обновляет смещения и переходит к следующему пакету событий. Обратите внимание, что параллельная обработка допустима только для потоков в стиле очереди, поскольку порядок обработки не сохраняется.

У этого подхода есть и ряд недостатков. Как обсуждалось в главе 6, выполнение запросов к внешнему сервису вводит в рабочий процесс неопределенные элементы. События переработки, даже просто отказавший пакет, могут дать иные результаты, чем вызов, выполненный во время первоначальной обработки. Обязательно учитывайте это при разработке приложения. В тех случаях, когда конечная точка «запросов-ответов» контролируется третьей стороной, внешней по отношению к вашей организации, внесение изменений в API или в формат ответа может привести к отказу микросервиса.

Наконец, подумайте о частоте, с которой вы делаете запросы к конечной точке. Например, предположим, что вы обнаружили в своем микросервисе ошибку и вам нужно перемотать входной поток для переработки. Событийно-управляемые микросервисы обычно потребляют и обрабатывают события так же быстро, как они способны выполнять код, что способно вызвать массовый всплеск запросов, направленных к внешнему API. Это может привести к отказу удаленного сервиса или, возможно, к ответному блокированию трафика, поступающего с ваших IP-адресов, и в результате к многочисленным отказывающим запросам и многократным циклам повторных попыток вашего микросервиса. Вы можете решить эту проблему, используя квоты (см. раздел «Квоты» главы 14), чтобы ограничить потребление и скорость обработки, но это также потребует жесткого регулирования со стороны обрабатывающего запросы микросервиса. В случае использования внешнего API, не зависящего от вашей организации, ответственность за такое регулирование, скорее всего, ляжет на вас и, возможно, должна быть реализована в вашем микросервисе. Это особенно часто встречается, когда внешний API способен предоставлять пакетную услугу значительного объема, но взимает непропорционально большую плату за объем, превышающий базовый уровень, как это бывает в случае с некоторыми сервисами ведения журналов и метрических данных.

Обработка и обслуживание данных с поддержкой состояния

Вы также можете создавать событийно-управляемые микросервисы, которые обеспечивают конечную точку «запросов-ответов» для произвольного доступа к состоянию, используя принципы EDM (Enterprise Data Management, Управление данными предприятия), рассмотренные в этой книге ранее (см. главу 8). Микросервис потребляет события из входных потоков событий, обрабатывает их, применяет любую бизнес-логику и сохраняет состояние внутри или снаружи в соответствии с потребностями приложения. API «запросов-ответов», который часто содержится в приложении (подробнее об этом далее), обеспечивает доступ к этим опорным хранилищам состояний. Такой подход можно разбить на два основных раздела: обслуживание состояния из внутренних хранилищ состояний и обслуживание состояния из внешних хранилищ состояний.

Обслуживание запросов реального времени с помощью внутренних хранилищ состояний

Микросервисы могут выдавать результаты, полученные из их внутреннего состояния (рис. 13.4). Запрос клиента передается балансировщику нагрузки, который переправляет запрос на один из опорных экземпляров микросервиса. В рассматриваемом случае имеется только один экземпляр микросервиса, и поскольку он материализует все данные состояния для этого приложения, то все его прикладные данные доступны в этом экземпляре. Указанное состояние материализуется через потребление двух входных потоков событий (A и B), а журнал изменений резервно копируется в брокер событий.

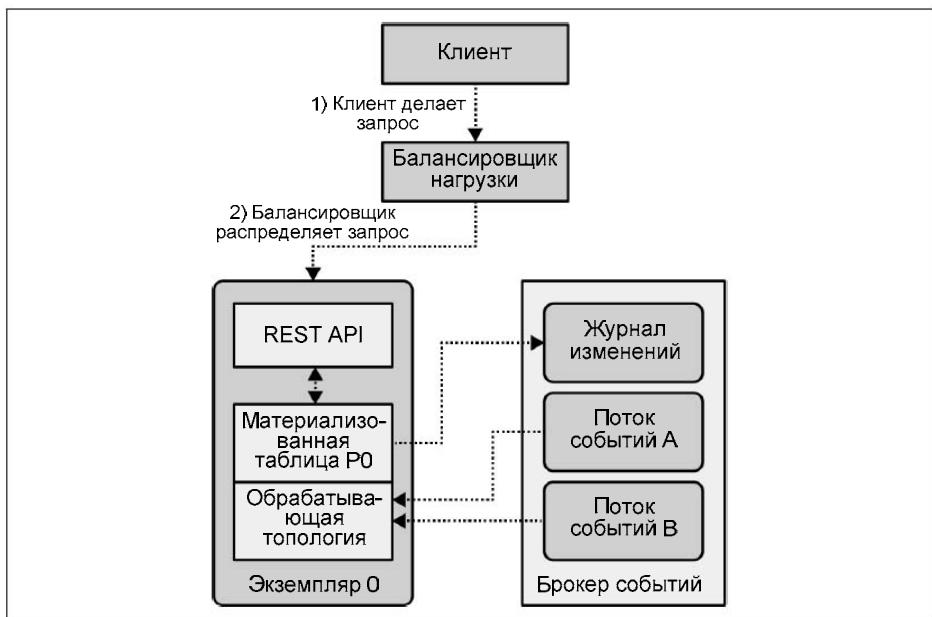


Рис. 13.4. Общий вид системы EDM с REST API, передающим контент клиенту

Теперь довольно часто для обработки нагрузки требуется несколько экземпляров микросервисов, и это внутреннее состояние может быть разделено между экземплярами. При использовании нескольких экземпляров микросервисов вы должны направлять запросы на получение состояния к нужному экземпляру, содержащему эти данные, т. к. все внутреннее состояние шардируется (сегментируется) в соответствии с ключом, а значение ключа может быть назначено только одному разделу. На рис. 13.5 показан клиент, делающий запрос, который затем пересыпается в нужный экземпляр, содержащий необходимое состояние.



«Горячие реплики» хранилищ состояний (см. разд. «Использование горячих реплик» главы 7) также могут использоваться для обслуживания запросов — в случае, если ваш фреймворк поддерживает их использование. Имейте в виду, что данные «горячей реплики» могут быть устаревшими настолько, насколько велика задержка репликации из первичного хранилища состояний.

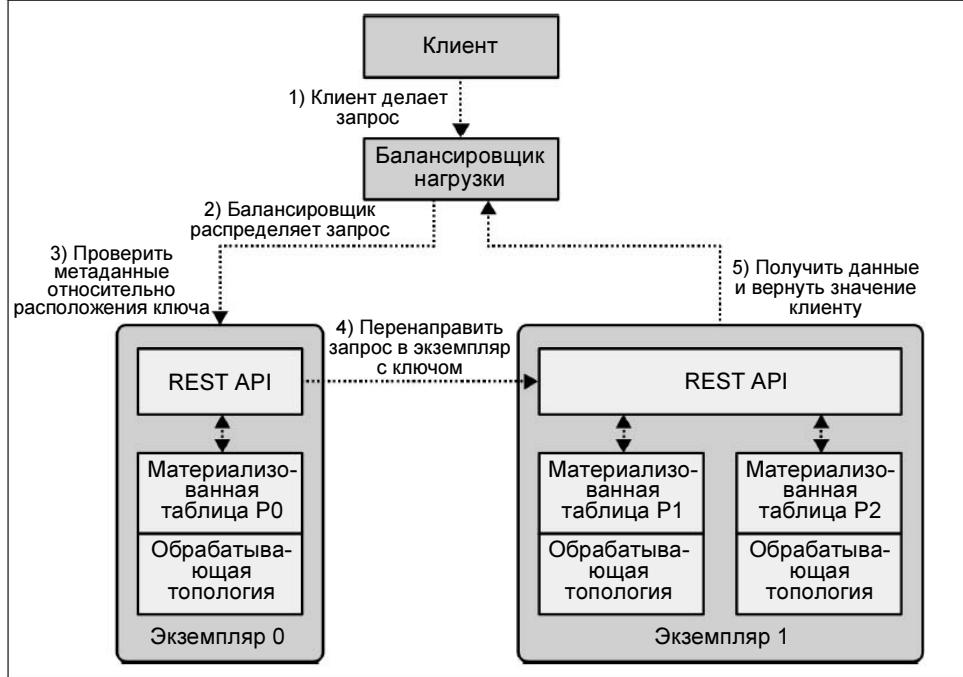


Рис. 13.5. Использование назначения разделов для определения месторасположения материализованного состояния для заданного ключа

Существует два свойства событийно-управляемой обработки, на которые можно положиться в определении того, какой экземпляр содержит конкретную пару ключ/значение:

- ◆ ключ может соотноситься только с одним разделом (см. разд. «Переподразделение потоков событий» главы 5);
- ◆ раздел может назначаться только одному экземпляру потребителя (см. разд. «Потребление в качестве потока событий» главы 2).

Экземпляр микросервиса в группе потребителей знает назначения своих разделов и назначения своих аналогов. Для материализации потока событий все события заданного ключа должны находиться внутри одного раздела. Применяя логику разделителя к ключу запроса, микросервис может сгенерировать назначение ID раздела этому ключу. Затем он может перекрестно ссылаться на этот ID раздела с назначениями разделов группы потребителей, чтобы определять то, какой экземпляр содержит материализованные данные, ассоциированные с ключом (если данные для этого ключа вообще существуют).

Например, на рис. 13.6 показано использование свойств выполняемого разделителем назначения с целью маршрутизации запроса REST GET.

Разделитель указывает на то, что ключ находится в P1, который назначен экземпляру 1. Если добавлен новый экземпляр и разделы перебалансираны, то последующей маршрутизации может потребоваться перейти к другому экземпляру,

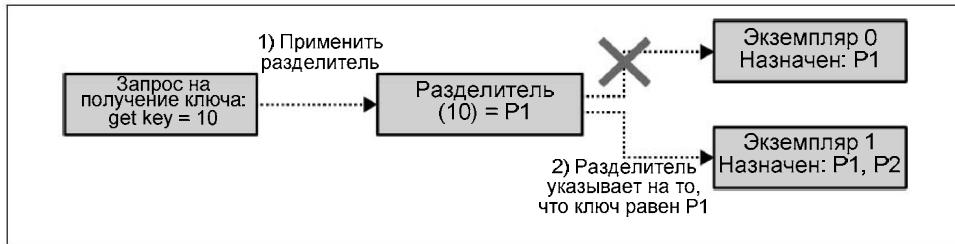


Рис. 13.6. Рабочий процесс, иллюстрирующий маршрутизацию запроса к правильному экземпляру

поэтому назначения группы потребителей играют важную роль в определении местоположения назначений разделов.

Одним из недостатков раздачи шардированного внутреннего состояния является то, что чем больше количество экземпляров микросервиса, тем больше разбросано состояние между отдельными экземплярами. Это уменьшает вероятность того, что запрос попадет в правильный экземпляр с первой попытки, не нуждаясь в перенаправлении. Если балансировщик нагрузки просто работает по круговой схеме распределения и предполагает равномерное распределение ключей, то вероятность того, что запрос окажется успешным с первой попытки, может быть выражена так:

$$\text{уровень успеха} = 1 / (\text{число экземпляров})$$

На самом деле для очень большого количества экземпляров почти все запросы окажутся неуспешными, и за ними последует перенаправление, что увеличит задержку ответа и нагрузку на приложение (поскольку для обработки каждого запроса, скорее всего, потребуется до двух сетевых вызовов вместо одного). К счастью, интеллектуальный балансировщик нагрузки может выполнить логику маршрутизации *перед* отправкой первоначального запроса к микросервисам, как показано на рис. 13.7.

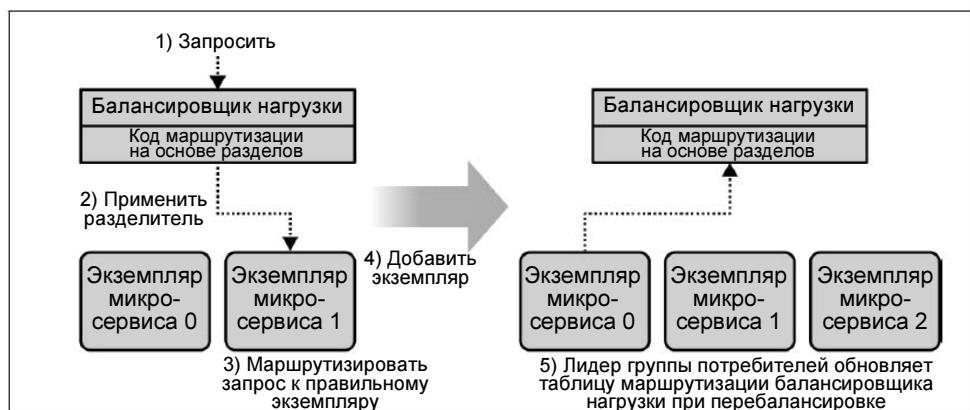


Рис. 13.7. Использование балансировщика нагрузки для правильной переадресации запросов на основе владения группой потребителей и логики разделителя

Интеллектуальный балансировщик нагрузки применяет логику разделителя для получения ID раздела, сравнивает его со своей внутренней таблицей назначений группы потребителей и затем перенаправляет запрос соответствующим образом. Назначения разделов должны логически выводиться из внутренних потоков переподразделения или потоков журнала изменений для того или иного хранилища состояний. Этот подход настолько запутывает логику вашего приложения балансировщиком нагрузки, что переименование хранилищ состояний или изменение топологии приведет к отказу пересылки. Лучше всего, когда любые интеллектуальные балансировщики нагрузки являются частью единого развертываемого и тестируемого процесса вашего микросервиса, в результате чего можно было бы отлавливать эти ошибки до начала производственного развертывания.



Использование интеллектуального балансировщика нагрузки — лучшее средство сократить задержку. Из-за конфликтных ситуаций и динамической перебалансировки внутренних хранилищ состояний каждый экземпляр микросервиса все равно должен иметь возможность перенаправлять неправильно переадресованные запросы.

Обслуживание запросов реального времени с помощью внешних хранилищ состояний

Обслуживание из внешнего хранилища состояний имеет два преимущества по сравнению с подходом на основе внутреннего хранилища состояний. Во-первых, все состояние доступно каждому экземпляру, а это означает, что запрос не нужно перенаправлять на экземпляр микросервиса, который служит хостом для данных в соответствии с моделью внутреннего хранилища. Во-вторых, перебалансировкам группы потребителей также не требуется, чтобы микросервис рематериализовывал внутреннее состояние в новом экземпляре, поскольку, опять же, все состояние поддерживается внешним по отношению к экземпляру. Это позволяет микросервису поддерживать бесшовное масштабирование и нулевое время простоя, что бывает трудно обеспечивать с помощью внутренних хранилищ состояний.



Убедитесь, что доступ к состоянию осуществляется через API «запрос-ответ» микросервиса, а не через прямую связь с хранилищем состояний. В противном случае создается общее хранилище данных, что приводит к тесной сцепке между сервисами, затрудняет внесение изменений и делает их рискованными.

Обслуживание запросов путем материализации событийно-управляемого микросервиса

Каждый экземпляр микросервиса потребляет и обрабатывает события из своих входных потоков событий и материализует данные во внешнее хранилище состояний. Каждый экземпляр также предоставляет API «запрос-ответ» для возврата материализованных данных обратно запрашивающему клиенту. Эта схема, показанная на рис. 13.8, отражает шаблон раздачи состояния из внутреннего хранилища состояний. Обратите внимание, что каждый экземпляр микросервиса может обслужи-



Рис. 13.8. Микросервис «все в одном», обслуживаемый из внешнего хранилища состояний.
Обратите внимание, что обслуживать запрос могут оба экземпляра

вать всю предметную область организованных по ключу данных из хранилища состояний и, таким образом, может справляться с любым переданным ему запросом.

Как обработка потока входных событий, так и емкость обслуживания «запросов-ответов» масштабируются за счет увеличения или уменьшения числа экземпляров, как и в случае с микросервисами с использованием внутреннего хранилища состояний. Число экземпляров может масштабироваться за пределы количества разделов потока событий. Эти дополнительные экземпляры не будут назначены на разделя, предназначенные для обработки, но они все равно могут обрабатывать внешние запросы из API «запрос-ответ». Более того, они существуют как резервные экземпляры, готовые к тому, чтобы им был назначен раздел в случае потери одного из других экземпляров.

Одно из главных преимуществ этого шаблона заключается в том, что он не требует большой координации при развертывании. Он представляет собой единый микросервис «все в одном», который может продолжать раздавать состояние из внешнего хранилища состояний независимо от текущего количества экземпляров.

Обслуживание запросов через отдельный микросервис

API «запрос-ответ» в этом шаблоне полностью отделен от функциональности событийно-управляемого микросервиса, который материализует состояние во внешнее хранилище состояний. Этот API остается независимым от обработчика событий, хотя оба имеют один и тот же ограниченный контекст и шаблоны развертывания. Такой шаблон показан на рис. 13.9. Здесь можно видеть, что запросы подаются через одну конечную точку REST API, в то время как события обрабатываются с помощью двух экземпляров обработки событий.



Хотя этот шаблон имеет два микросервиса, работающих на одном хранилище данных, существует только один ограниченный контекст. Так что эти два микросервиса рассматриваются как единый составной сервис. Они располагаются в одном хранилище кода и тестируются, строятся и развертываются вместе.

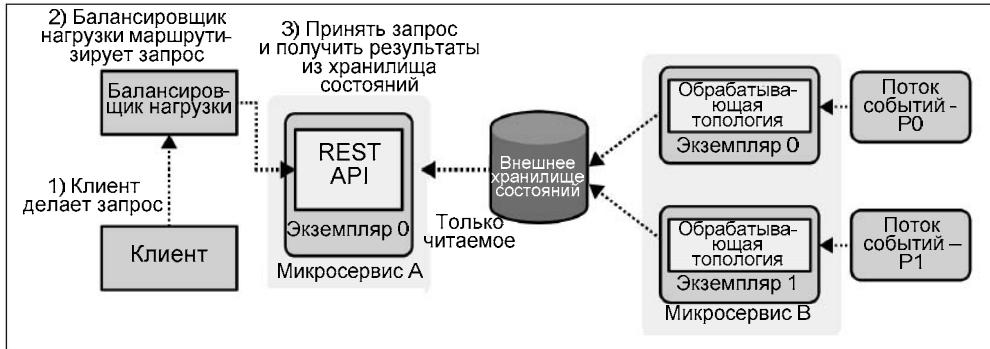


Рис. 13.9. Микросервис, состоящий из отдельных исполнимых элементов: один — для обслуживания запросов, другой — для обработки событий

Одним из главных преимуществ этой модели является то, что, поскольку API «запрос-ответ» полностью независим от обработчика событий, вы можете самостоятельно выбирать языки реализации и потребности в масштабировании. Например, использовать «легковесный» потоковый фреймворк для заполнения материализованного состояния, но задействовать язык и ассоциированные библиотеки, которые уже нашли широкое применение в вашей организации, для предоставления клиентам согласованного веб-решения. Этот подход может дать вам лучшее из обоих вариантов, хотя он и сопряжен с дополнительными накладными расходами на управление многочисленными компонентами в вашей кодовой базе.

Второе важное преимущество рассматриваемого шаблона заключается в том, что он изолирует любые отказы в логике обработки событий от приложения обработки «запросов-ответов». Это исключает вероятность того, что какие-либо ошибки или проблемы, связанные с данными в коде обработки событий, могут привести к отказу экземпляра обработки «запросов-ответов», тем самым сокращая время простоя (обратите внимание, что состояние станет устаревшим).

Главными недостатками этого шаблона являются сложность и риск. Координация изменений между двумя в остальном независимыми приложениями сопряжена с риском, поскольку изменение структур данных, топологий и шаблонов запросов может потребовать зависимых изменений в обоих сервисах. Кроме того, сцепленность сервисов делает нереализуемыми некоторые принципы EDM — такие как отказ от совместного использования состояния через общие хранилища данных и использование одинарных развертываемых элементов для ограниченного контекста.

Тем не менее рассмотренный шаблон по-прежнему остается полезным для обработки данных в реальном времени, и он часто успешно используется в производстве. Тщательное управление развертываниями и комплексное интеграционное тестирование являются ключевыми факторами обеспечения его успеха.

Обработка запросов внутри событийно-управляемого рабочего процесса

API типа «запрос-ответ» образуют основу взаимодействия между многими системами, поэтому вам необходимо убедиться, что ваши приложения могут обрабатывать входные данные таким образом, чтобы это соответствовало принципам событийно-управляемых микросервисов. Один из способов обработки запросов — точно такой же, как и в любой не событийно-управляемой системе, — немедленно выполнить операцию запроса и вернуть ответ клиенту. В качестве альтернативы вы также можете **конвертировать** запрос в событие, ввести его в свой собственный поток событий и обработать его так же, как и любое другое событие в системе. Наконец, микросервис может также комбинировать оба этих подхода, превращая в события только важные для предприятия запросы (те, что могут совместно использоваться за пределами ограниченного контекста), параллельно обрабатывая при этом прочие. На рис. 13.10 приведена схема работы на основе такой концепции, которую чуть позже мы подробно рассмотрим в разд. «Пример: рабочий процесс публикации газет (шаблон одобрения)».

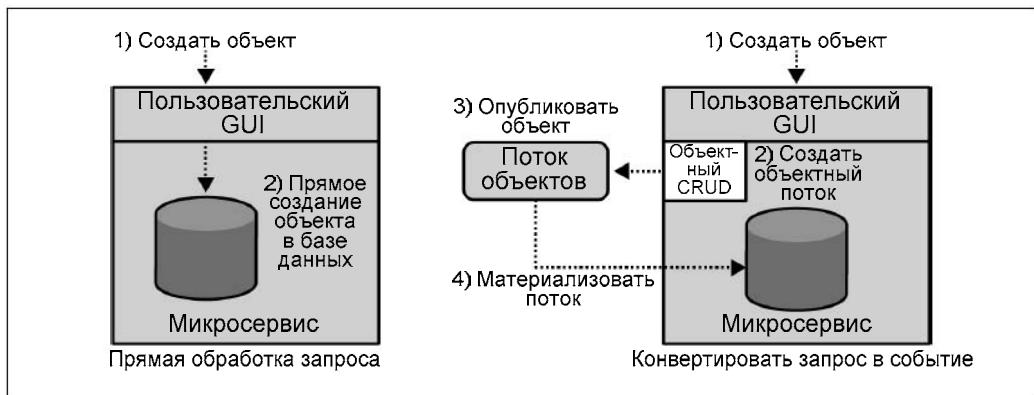


Рис. 13.10. Обработка запросов напрямую в сопоставлении с конвертацией их сначала в события¹

Слева на этой схеме показано выполнение традиционной операции создания объекта, результаты которой записываются непосредственно в базу данных. В качестве альтернативы правая часть схемы показывает событийно-управляемое решение, где запрос анализируется, конвертируется в событие и публикуется в соответствующем потоке событий прежде чем событийно-управляемый рабочий процесс потребит его, применит бизнес-логику и сохранит в базе данных.

Основное преимущество от записи сообщения в поток событий заключается в том, что это обеспечивает надежное его сохранение и позволяет любому сервису мате-

¹ Аббревиатура CRUD, присутствующая на рис. 13.10, представляет собой акроним, обозначающий четыре основные функции, используемые при работе с базами данных: создание (Create), чтение (Read), модификация (Update), удаление (Delete). — Прим. ред.

риализовываться на основе содержащихся в нем данных. Компромисс, однако, заключается в возникающей задержке, поскольку сервис должен ждать до тех пор, пока результат не окажется материализован в хранилище данных, прежде чем будет использован (согласованное чтение с последующей записью). Один из способов уменьшить эту задержку — сохранить значение в памяти после его успешной записи в поток объектов, что позволяет использовать его в операциях на стороне приложения. Однако это не будет работать для операций, требующих присутствия данных в базе данных (например, операций объединения — *joins*), поскольку событие должно быть сначала материализовано.

Обработка событий для пользовательских интерфейсов

Пользовательский интерфейс (UI) — это средство, с помощью которого люди взаимодействуют с ограниченным контекстом сервиса. Существующие фреймворки запросов и ответов широко применяются для разработки приложений с пользовательским интерфейсом, предлагая множество опций и языков, удовлетворяющих потребности пользователей. Интеграция этих фреймворков в событийно-управляемую предметную область весьма важна для раскрытия их внутренней ценности.

При конвертации вводимых пользователем данных в поток событий необходимо решить ряд проблем. Дизайн приложений, которые обрабатывают запросы как события, должен предусматривать *асинхронный* пользовательский интерфейс. Вам также следует убедиться, что поведение приложения соответствует ожиданиям пользователей. Например, в синхронной системе пользователь, который нажимает на кнопку, может ожидать получения успешного или неуспешного ответа в очень короткие сроки — возможно, через 100 мс или меньше. В системе асинхронной обработки событий обрабатывающему сервису для обработки и ответа может потребоваться более 100 мс, в особенности если поток событий содержит большое число обрабатываемых записей.



Представляя вводимые пользователем данные в качестве событий, следует изучить и реализовывать лучшие образцы решений в области асинхронных UI. Правильный дизайн пользовательского интерфейса готовит пользователя к ожиданию асинхронных результатов.

Для управления ожиданиями пользователей можно использовать некоторые асинхронные технические приемы построения пользовательского интерфейса. Например, вы можете обновлять пользовательский интерфейс, сообщая при этом пользователю, что его запрос был отправлен, и одновременно отговаривая его от выполнения каких-либо дополнительных действий до тех пор, пока запрос не будет выполнен. Веб-сайты бронирования авиабилетов и проката автомобилей часто выводят на экран сообщение «Пожалуйста, подождите» с символом вращающегося колеса, блокируя остальную часть веб-страницы от ввода пользователем данных. Тем самым пользователи информируются о том, что бэкендовый сервис обрабатывает событие и что они не смогут делать ничего другого до тех пор, пока эта работа не завершится.

Еще один фактор, который следует учитывать, заключается в том, что микросервису может потребоваться постоянно обрабатывать входящие непользовательские события, ожидая дальнейшего ввода данных пользователем. Вы должны поймать тот момент, когда обработка событий сервисом продвинулась в достаточной мере, чтобы обновление могло быть отправлено в пользовательский интерфейс. Фактически вы также должны принять решение о том, когда начавшаяся обработка событий догнала настоящее время, учитывая и текущие обновления, которые должны обрабатывать большинство сервисов EDM.

Нет единых правил, диктующих, когда вы должны обновлять свой интерфейс. Разве что бизнес-правила ограниченного контекста помогут вам определиться с тем, какие последствия могут настичь пользователей, когда они принимают решения на основе текущего состояния. Ответы на следующие вопросы помогут вам решить, как и когда обновлять пользовательский интерфейс:

- ◆ каковы последствия для пользователя, когда он принимает решение на основе устаревшего состояния;
- ◆ каковы последствия для производительности и/или для пользователя при инициировании обновления пользовательского интерфейса?



Кратковременные сетевые отказы, вызывающие повторные попытки сделать запрос, могут привести к дублированию событий. Убедитесь, что ваши потребители способны обрабатывать дубликаты идемпотентно, как описано в разд. «Генерация дублирующихся событий» главы 7.

Приведенный далее пример демонстрирует некоторые выгоды от конвертирования запросов непосредственно в события перед их обработкой.

Пример: рабочий процесс публикации газет (шаблон одобрения)

У издателя газеты есть приложение, которое управляет макетом ее публикаций. Каждая публикация опирается на индивидуально настраиваемые шаблоны, определяющие, как и где размещаются статьи и рекламные объявления..

Графический пользовательский интерфейс (GUI) позволяет сотрудникам редакции упорядочивать и размещать статьи в соответствии с бизнес-логикой издательства. Самые «горячие» новости помещаются на первых полосах, а менее важные статьи сдвигаются на следующие. Реклама также позиционируется в соответствии со своими собственными правилами, обычно зависящими от ее размера, содержания, бюджета и договоренностей о ее размещении. Например, некоторые рекламодатели могут не захотеть, чтобы их реклама размещалась рядом с определенными типами сообщений (так, компания-производитель детских игрушек хотела бы избежать размещения своей рекламы рядом со статьей о похищении детей).

Дизайнер газеты несет ответственность за размещение статей и рекламных объявлений в соответствии с заготовкой макета. Редактор газеты отвечает за то, чтобы газета выглядела организованно, статьи были упорядочены по категориям и предполагаемой важности для читателя, а реклама размещалась в соответствии с кон-

трактами. При этом он должен одобрить работу, выполненную дизайнером, прежде чем газета будет отправлена в печать, или отклонить ее в случае, если, на его взгляд, материал требует перекомпоновки. Рис. 13.11 иллюстрирует этот рабочий процесс.

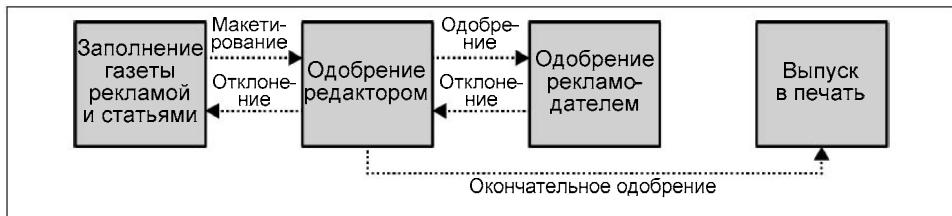


Рис 13.11. Рабочий процесс заполнения газеты материалами с выпуском ее в печать на основе одобрения редактором и рекламодателем

И редактор, и рекламодатель могут отклонить предложенный дизайнером макет газеты, хотя рекламодатель получит возможность сделать это только в том случае, если редактор уже одобрил макет. Более того, газета заинтересована в получении одобрения только от наиболее важных рекламодателей, чьи рекламные расходы являются значительным источником дохода газетного издательства.

Заполнение газеты и ее одобрение — это два отдельных ограниченных контекста, каждый из которых связан со своей собственной бизнес-функциональностью, что может быть представлено двумя микросервисами, как показано на рис. 13.12 (для простоты здесь не указаны учетные записи, управление учетными записями, аутентификация и данные для входа).

В этом примере кое-что необходимо разъяснить, поэтому давайте начнем с микросервиса заполнения газеты. Этот сервис потребляет потоки заготовок макетов, рекламных объявлений и статей в реляционную базу данных. При этом ответственный

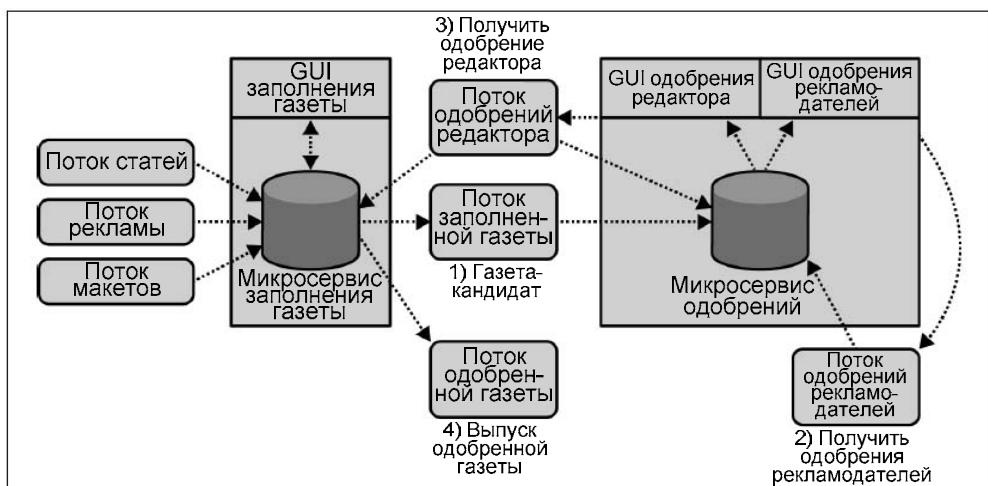


Рис. 13.12. Рабочий процесс заполнения и одобрения газеты на основе микросервисов

за макетирование сотрудник выполняет свои задачи, и когда газета готова к передаче на одобрение, он компилирует заполненную газету в PDF-файл, сохраняет его во внешнем хранилище и выпускает в поток событий заполненной газеты. Формат события заполненной газеты выглядит следующим образом:

```
// Событие заполненной газеты

Key: String pn_key          // Ключ заполненной газеты
Value: {
    String pdf_uri          // Местоположение сохраненного PDF-файла
    int version              // Версия заполненной газеты
    Pages[] page_metadata    // Метаданные о том, что находится
                             // на каждой странице
    - int page_number
    - Enum content           //Реклама, статья
    - String id               //ID рекламы или статьи
}
}
```



Поскольку PDF-файл может быть слишком крупным для хранения в событии, его можно хранить во внешнем хранилище файлов с доступом через URI — универсальный идентификатор ресурса (см. разд. «Минимизируйте размер событий» главы 3).

Возможно, вы заметили, что этот микросервис не переводит взаимодействия человека с графическим интерфейсом пользователя (GUI) при заполнении газеты в события — почему так? Несмотря на то что «взаимодействия человеком с GUI как события» являются одной из главных тем этого примера, нет необходимости превращать *все* взаимодействия человека с GUI в события. Рассматриваемый конкретный ограниченный контекст на самом деле связан только с созданием события окончательно заполненной газеты, но не особенно важно, *как* оно возникло. Такая инкапсуляция ответственности позволяет вам действовать для создания этого микросервиса монолитный фреймворк с шаблонами синхронных GUI, а также использовать шаблоны и программные технологии, с которыми вы или ваши разработчики уже знакомы.



Поток заполненной газеты может выйти из синхронизации с состоянием внутри микросервиса заполнения газеты. Подробнее об атомарном производстве из монолита, в частности с использованием шаблона таблицы исходящих данных или журналов захвата изменений в данных, рассказано в разд. «Шаблоны освобождения данных» главы 4.

Одобрения регулируются отдельным микросервисом, куда событие заполненной газеты загружается редактором для просмотра и одобрения. По мере необходимости редактор может пометить копию PDF-файла, добавить комментарии и предоставить предварительное одобрение, чтобы перейти к следующему шагу: получению одобрения рекламодателем. Редактор также может отклонить событие заполненной газеты в любой момент рабочего процесса — до, во время или после получения отзыва рекламодателя. Структура такого события выглядит следующим образом:

```
// Событие одобрения редактором

Key: String pn_key           // Ключ заполненной газеты
Value: {
    String marked_up_pdf_uri // Опциональный URI размеченного PDF-файла
    int version              // Версия заполненной газеты
    Enum status               // ожидает одобрения, одобрено, отклонено
    String editor_id
    String editor_comments
    RejectedAdvertisements[] rejectedAds // Опционально, если отклонено
    - int page_number
    - String advertisement_id
    - String advertiser_id
    - String advertiser_comments
}
}
```

Рекламодателям также предоставляется пользовательский интерфейс для одобрения размера рекламы и места ее размещения. Этот сервис отвечает за определение того, *какие* рекламные объявления требуют одобрения, а какие нет, и за разрезание PDF-файла на соответствующие части для просмотра рекламодателем. Важно не допустить утечки информации о новостях или рекламе конкурентов. События одобрения записываются в поток одобрения рекламодателем, аналогичный потоку одобрения редактором:

```
// Событие одобрения рекламодателем
```

```
Key: String pn_key           // Ключ заполненной газеты
Value: {
    String advertiser_pdf_uri // Часть PDF-файла, показываемая рекламодателю
    int version              // Версия заполненной газеты
    int page_number
    boolean approved          // Одобрено или нет
    String advertisement_id
    String advertiser_id       // ID одобряющего
    String advertiser_comments
}
}
```

Возможно, вы заметили, что доступ к одобрениям рекламодателей осуществляется по ключу `pn_key` и что в каждой газете будет несколько событий рекламодателя с этим же ключом. В таком случае одобрения рекламодателя рассматриваются как *события*, а не как *объекты*, и именно *совокупность* этих событий определяет полное одобрение рекламодателем газеты. Имейте в виду, что каждый рекламодатель входит в свой графический интерфейс и одобряет свои объявления отдельно, и только после того, как все они ответят (или, возможно, не ответят в установленный срок), процесс может перейти к окончательному одобрению. Если вы посмотрите на определение события одобрения редактором, то увидите, что агрегация отклоненных событий представлена в форме массива объектов `RejectedAdvertisements`.

Одно из преимуществ того, что наличие заполненной газеты, одобрения редактора и одобрения рекламодателя трактуются как события, состоит в том, что вместе они образуют общий блок информации о газете, отклонениях, комментариях и одобрениях. Вы можете выполнить аудит этого блока в любой момент времени, чтобы увидеть историю представлений и одобрений и точно засечь место, где что-то пошло не так. Еще одно преимущество заключается в том, что, записывая данные непосредственно в события, микросервис одобрения может использовать чистую библиотеку потоковой обработки, такую как Apache Kafka или Samza, для материализации состояния непосредственно из потока событий при каждом запуске приложения. Нет необходимости создавать внешнее хранилище состояний для управления этими данными.

Разделение сервисов одобрения редактором и рекламодателем

Бизнес-требования диктуют, что сервис одобрения редактором и сервис одобрения рекламодателем должны быть разделены. Каждый из них служит родственному, хотя ициальному, бизнес-контексту. В частности, компоненты рекламодателя объединенного в настоящее время сервиса (см. рис. 13.12) отвечают за:

- ◆ определение того, к каким рекламодателям обращаться за одобрением;
- ◆ нарезку PDF-файла на части;
- ◆ управление ориентированными на рекламодателя компонентами, элементами управления и брендингом;
- ◆ ориентированное на пользователей выставление данных в Интернет — в особенности в связи с практикой обеспечения безопасности и доработок программного обеспечения.

С другой стороны, редакторские компоненты объединенного сервиса не нуждаются в решении ориентированных на публику задач — таких как имидж, брендинг и безопасность. В первую очередь они касаются:

- ◆ одобрения заполненного макета, его дизайна и компоновки;
- ◆ оценки сводки ответов розничных продавцов (не каждого в отдельности);
- ◆ предоставления рекомендаций дизайннеру газеты о том, как учесть отказы рекламодателей.

Схема взаимодействия разделенных микросервисов одобрения газеты редактором и рекламодателем показана на рис. 13.13.

Здесь мы видим два новых потока событий. Первый — это *поток одобренной редактором заполненной газеты* (шаг 2). Формат этого потока идентичен формату потока заполненной газеты, но это событие производится только после того, как редактор будет полностью удовлетворен газетой и выпустит ее на одобрение рекламодателем.

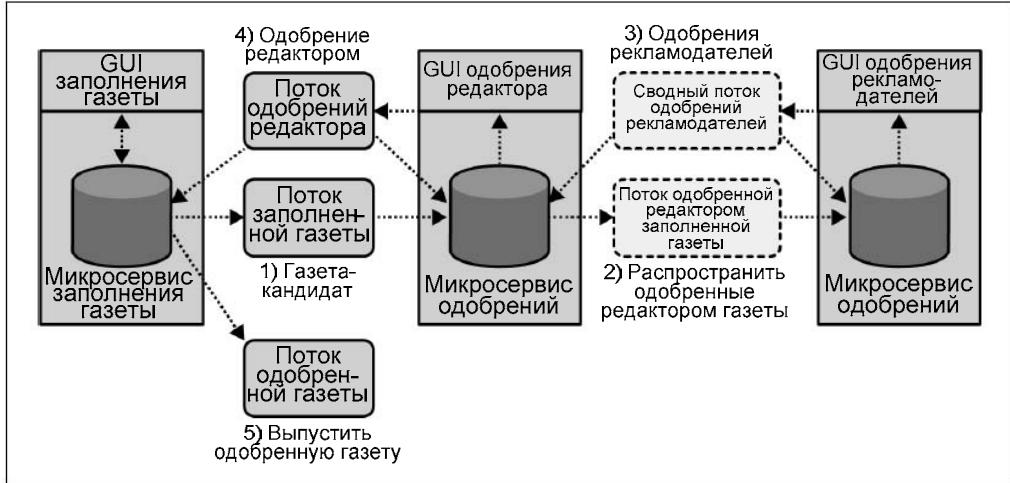


Рис. 13.13. Разделенные микросервисы одобрения газеты редактором и рекламодателем



Многопользовательский поток газеты является единственным источником истины для всех газет-кандидатов. Поток одобренной редактором заполненной газеты — это единственный источник истины только для газет, одобренных для просмотра рекламодателями, которые перед этим были отфильтрованы логикой редакционной системы. Эти два потока событий имеют разные бизнес-значения.

Главным преимуществом этой схемы является то, что *вся* редакторская логика выборочного отбора полностью остается в сервисе одобрения редактором. Обратите внимание, что обновления потока заполненной газеты *не* перенаправляются далее автоматически, а ожидают, пока редактор не выпустит их для одобрения. Все версии одной и той же газеты (`pn_key`) полностью остаются внутри редакторского сервиса. Такое их расположение позволяет редактору управлять тем, какие версии отправляются на одобрение, а также не допускать прохождения любых дальнейших изменений до тех пор, пока они не будут одобрены по обратной связи со стороны рекламодателя.

Второй новый поток событий — это *сводный поток одобрений рекламодателей* (шаг 3). Он содержит сводку результатов, полученных от сервиса одобрения рекламодателями, и предназначенную как для последующего хранения, так и для определения текущего статуса каждого из ответов для создаваемой газеты. Имейте в виду, что сервис одобрения редактором — это отдельный сервис, и он не имеет возможности знать, каким рекламодателям были отправлены предложения по одобрению их рекламных объявлений. Эта информация является строго прерогативой системы одобрения рекламы, хотя и может передаваться редактору в составе сводки результатов.

Формат сводного события одобрения рекламы выглядит следующим образом:

```
// Сводное событие одобрения рекламы
```

Key: String `pn_key`

```

Value: {
    int version           // Версия заполненной газеты
    AdApprovalStatus[] ad_app_status
        - Enum status      // Ожидает, Одобрено, Отклонено, Тайм-аут
        - int page_number
        - String advertisement_id
        - String advertiser_id
        - String advertiser_comments
}

```

Это определение сводного события одобрения рекламы демонстрирует инкапсуляцию состояния одобрения рекламодателем в сервис одобрения рекламодателем. Редактор может принимать решение об одобрении газеты на основе статуса сводного события одобрения рекламы, не имея надобности управлять какой-либо работой по получению этих результатов или обрабатывать ее.

Микрофронтенды в приложениях на основе запросов-ответов

Фронтенд- и бэкэнд-сервисы координируются по трем основным направлениям, принося пользу их пользователям. Монолитные серверные части распространены во многих организациях любого размера. Бэкенды микросервисов также становятся все более популярными в связи с растущим применением микросервисов — как синхронных, так и событийно-управляемых. В обоих этих двух подходах интерфейсные и серверные службы принадлежат и управляются отдельными командами, так что сквозные бизнес-функции выходят за рамки этих команд. В отличие от этого, подход на основе микрофронтендов полностью согласовывает свои реализации с бизнес-задачами — от бэкенда до внешнего интерфейса. Эти три подхода показаны на рис. 13.14.

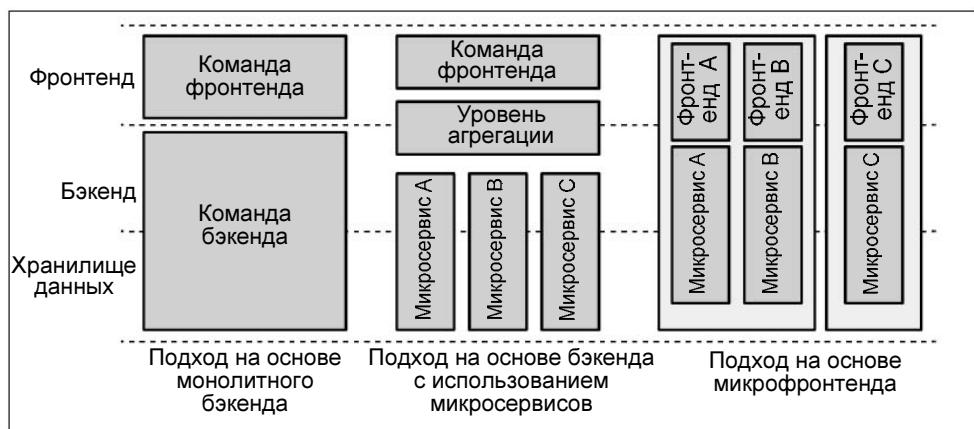


Рис. 13.14. Три основных подхода к организации продуктов и команд для работы с контентом, ориентированным на клиентов

Вариант на основе монолитного бэкенда — это подход, с которым большинство разработчиков программного обеспечения как минимум до некоторой степени знакомы. Во многих случаях большую часть работы на монолите выполняет специальная бэкендовая команда, обычно состоящая — в случае очень крупных монолитов — из многочисленных подкоманд. Количество персонала увеличивается по мере роста монолита.

Фронтендовый коллектив полностью отделен от бэкенда, и они обмениваются информацией через API «запрос-ответ», чтобы получать необходимые данные для визуализации пользовательского интерфейса клиента. Продукт, реализованный в этой архитектуре, требует координации усилий между командами и между различными техническими воплощениями, что делает его потенциально одним из самых дорогих способов предоставления функциональности.

Бэкендовый вариант с использованием микросервисов — это тот подход, при котором многие команды мигрируют на микросервисы, на чем и останавливаются. Основное преимущество этого подхода заключается в том, что бэкенд теперь состоит из независимых, *ориентированных на продукт* микросервисов, причем каждый микросервис (или набор микросервисов, поддерживающих продукт) независимо принадлежит одной команде. Каждый микросервис материализует необходимые данные, реализует свою бизнес-логику и предоставляет все необходимые API «запрос-ответ» и потоки событий вплоть до уровня агрегации.

Основным недостатком бэкендового подхода с использованием микросервисов является то, что он по-прежнему сильно зависит от уровня агрегации, где могут возникать многочисленные проблемы. Так, в этот уровень может проникать бизнес-логика — из-за попыток решать проблемы с границами продукта или добиваться «быстрых побед» за счет объединения функциональности отдельных продуктов. Уровень агрегации также часто страдает от того, что все полагаются на него, но никто не несет за него ответственности. Хотя это может быть до некоторой степени решено с помощью строгой модели управления, скопление незначительных, на первый взгляд, невинных изменений может все же позволить просочиться в него неприемлемому количеству бизнес-логики.

Третий подход — на основе микрофронтендов — разбивает монолитный фронтенд на серию независимых компонентов, каждый из которых подкрепляется поддерживающими бэкендовыми микросервисами.

Выгоды от микрофронтендов

Микрофронтендинговые шаблоны очень хорошо сочетаются с бэкендовыми событийно-управляемыми микросервисами и наследуют многие из их преимуществ — такие как модульность, разделение бизнес-задач, автономные команды, независимость от развертывания, языка и кодовой базы.

Давайте рассмотрим некоторые другие выгоды от микрофронтендов.

Композиционные микросервисы

Микрофронтиенды предоставляют возможности композиционного шаблона, т. е. вы можете по мере необходимости добавлять сервисы в существующий пользовательский интерфейс. Примечательно, что микрофронтиенды очень хорошо сочетаются с событийно-управляемыми бэкендами, которые также внутренне основаны на композиции. Потоки событий обеспечивают микросервисы возможностью получать события и сущности, необходимые для поддержки бэкендового ограниченного контекста. Бэкендовый сервис может создавать необходимое состояние и применять бизнес-логику, ориентированную специально на бизнес-требования продукта, предоставляемого микрофронтиенном. Реализацию хранилища состояний можно выбрать в соответствии с требованиями сервисов. Эта форма композиции обеспечивает огромную гибкость в том, как строятся фронтеневые сервисы, что и будет показано далее в разд. «Пример: приложение поиска источников впечатлений и отзывов о них».

Простота привязки к бизнес-требованиям

Привязывая микрофронтиенды строго к бизнес-ограниченным контекстам, как и в случае с другими микросервисами, функционирующими в бэкенде, вы получаете возможность отслеживать конкретные бизнес-требования непосредственно до их реализации. Благодаря этому вы можете легко внедрять в приложение экспериментальные продукты, не оказывая негативного влияния на кодовую базу существующих основных сервисов. И если их производительность или восприятие пользователями не будут соответствовать ожиданиям, вы сможете так же легко их удалить. Такая привязка и изоляция гарантируют, что требования к продукту из различных рабочих процессов не переходят друг в друга.

Недостатки микрофронтиендов

Хотя микрофронтиенды позволяют разделить бизнес-задачи, вы должны предусматривать функциональности, которые обычно считаются само собой разумеющимися в монолитном интерфейсе, такие как согласованные элементы пользовательского интерфейса и полный контроль над размещением каждого элемента. Микрофронтиенды также наследуют некоторые общие для всех микросервисов проблемы, связанные с возможностью дублирования кода, и эксплуатационные сложности, связанные с развертыванием микросервисов и управлением ими. В этом разделе рассматриваются некоторые особенности, специфичные для микрофронтиендов.

Потенциальное отсутствие единообразия элементов пользовательского интерфейса и стилизации

Важно, чтобы визуальный стиль приложений оставался единообразным, а этого достичь непросто, когда фронтенд состоит из многих независимых микрофронтиендов. Каждый микрофронтен — это еще одна потенциальная точка отказа, т. е. ме-

сто, где дизайн пользовательского интерфейса может оказаться несовместимым с желаемым пользовательским впечатлением. Один из способов это исправить — обеспечить строгое стилевое руководство в сочетании с единой библиотекой общих элементов пользовательского интерфейса, которые будут применяться в каждом микрофронтенде.

Недостаток этого подхода заключается в том, что он требует пристального внимания как к стилевому руководству, так и к элементам интерфейса. Координировать добавление в интерфейс приложения новых элементов и модифицирование существующих между несколькими командами, использующими библиотеку элементов в своих продуктах, бывает трудно. Впрочем, такая координация на основе модели управления, аналогичной той, которая используется во многих популярных проектах с открытым исходным кодом, может помочь обеспечить взвешенное и осознанное внесение планируемых изменений. Однако это требует совместной работы и диалога между командами разработчиков и, как следствие, влечет за собой накладные расходы.



Обеспечьте, чтобы общие библиотеки элементов пользовательского интерфейса были свободны от любой бизнес-логики, зависящей от ограниченного контекста. Держите всю бизнес-логику инкапсулированной в ее собственном надлежащем ограниченном контексте.

Наконец, внесение изменений в общие элементы пользовательского интерфейса приложения может потребовать перекомпиляции и переразвертывания каждого микрофронтенда. Это весьма затратный процесс, поскольку каждая микрофронтендовая команда должна будет обновить свое приложение, тестировать его на соответствие его пользовательского интерфейса новым требованиям и убедиться, что он должным образом интегрируется с уровнем общего пользовательского интерфейса приложения (подробнее об этом далее). Эти расходы несколько смягчаются редкостью радикальных изменений пользовательского интерфейса.

Различающаяся производительность микрофронтендов

Микрофронтенды, являющиеся частями композиционного фреймворка, иногда могут создавать связанные с этим их положением проблемы. Так, отдельные фронтенды могут загружаться с разной скоростью или, что еще хуже, вообще не загружаться во время сбоя.

Вы должны убедиться, что композиционный фронтенд способен корректно обрабатывать эти сценарии и уверенно обеспечивать согласованное взаимодействие тех его частей, которые все еще работают. Например, вы можете использовать врачающиеся знаки «загрузки» для элементов, которые ожидают результатов от медленных микрофронтендов. Объединение всех микрофронтендов приложения в единое целое — прекрасное упражнение в правильном дизайне пользовательского интерфейса, но более глубокие детали и нюансы такого процесса выходят за рамки этой книги.

Пример: приложение поиска источников впечатлений и отзывов о них

«Впечатление — это то, что вы никогда не забудете!» — утверждают создатели приложения, связывающего отдыхающих с путеводителями, достопримечательностями, развлечениями и кулинарными изысками того места, которое они посещают. Пользователи могут искать в нем местные источники впечатлений, получать подробную информацию о них и контактные сведения, а также оставлять отзывы.

Первая версия этого приложения имеет единый сервис, который материализует как объекты впечатлений, так и отзывы клиентов в единую конечную точку. Пользователи могут вводить название того или иного города, чтобы получать на экране список доступных источников впечатлений о его регионе. После выбора ими варианта информация об источнике впечатлений вместе с любыми ассоциированными отзывами выводится на экран, как показано в простой имитации этого процесса на рис. 13.15.



Рис. 13.15. Приложение поиска источников впечатлений и отзывов о них: имитация GUI в версии 1 с монолитным фронтеном

В первой версии приложения данные хранятся в базовом хранилище состояний в формате ключ/значение, и она предлагает только ограниченные возможности поиска. Не доступен даже поиск на основе геолокации пользователей, хотя это то, чего бы они хотели. Кроме того, было бы неплохо, если бы версия 2 вынесла отзывы в свои собственные отдельные микросервисы, поскольку у них есть достаточно четкие бизнес-обязанности, чтобы сформировать свой собственный ограниченный контекст. Наконец, вы должны создать микрофронтенд, объединяющий оба эти сервиса вместе и действующий для них как уровень агрегации. Каждый из этих трех микрофронтендов может принадлежать и управляться своей собственной командой или одной сводной командой, хотя разделение обязанностей позволяет масштабировать владение так же, как и в случае бэкендовых микросервисов. Новый макет графического интерфейса, показывающий разделенные фронтенды обязанности, показан на рис. 13.16.

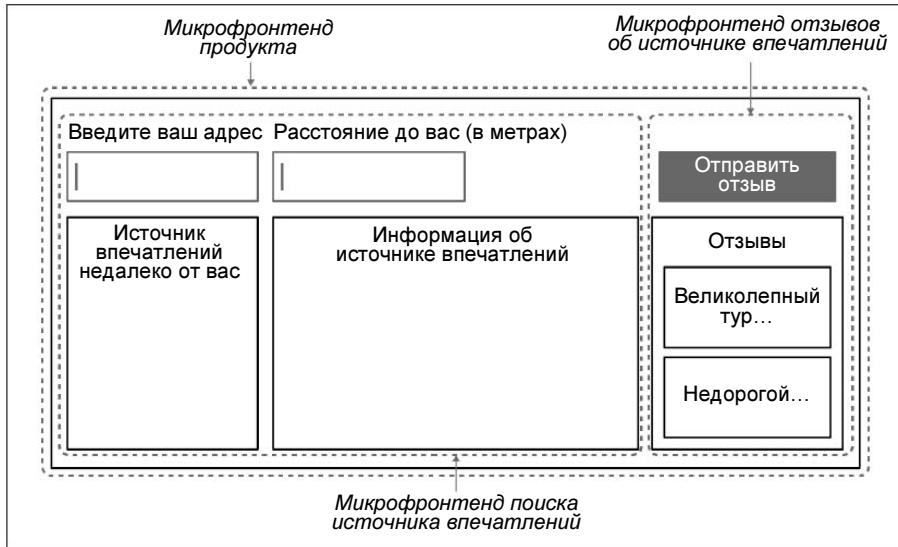


Рис. 13.16. Приложение поиска источников впечатлений и отзывов о них: имитация GUI в версии 2 с микрофронтендами

Теперь продукт инкапсулирует микрофронтенды как поиска, так и отзывов, и включает всю логику, необходимую для объединения этих двух сервисов вместе. Однако он пока не содержит никакой бизнес-логики, относящейся непосредственно к самим сервисам. Обновленный пользовательский интерфейс также иллюстрирует характер изменений обязанностей микрофронтенда, поскольку теперь он должен поддерживать функцию поиска по геолокации. Адрес пользователя при этом переводится в координаты широты и долготы, которые можно использовать для вычисления расстояния до ближайших источников впечатлений. В то же время обязанности микрофронтенда отзывов остаются прежними, но он освобождается от сцепки с сервисом поиска.

В схеме, приведенной на рис. 13.17, показано, как могла бы выглядеть миграция всего приложения в микрофронтенды. Здесь есть несколько примечательных моментов. Во-первых, как обсуждалось ранее в этой главе, отзывы о полученных впечатлениях публикуются *сначала* в качестве событий в потоке событий отзывов, а *затем* возвращаются обратно в хранилище данных. Это верно для обеих версий сервисов и иллюстрирует важность поддержания основных бизнес-данных внешними по отношению к реализации. Благодаря этому вы можете легко разбить сервис отзывов на самостоятельные микросервисы, не выполняя ненужных и подверженных ошибкам операций с синхронизацией данных.

Если бы обзоры хранились внутри хранилища данных версии 1, вам вместо этого пришлось бы рассмотреть возможность их освобождения для использования в версии 2 (см. главу 4), а затем разработать план миграции для их долгосрочного хранения в потоке событий.

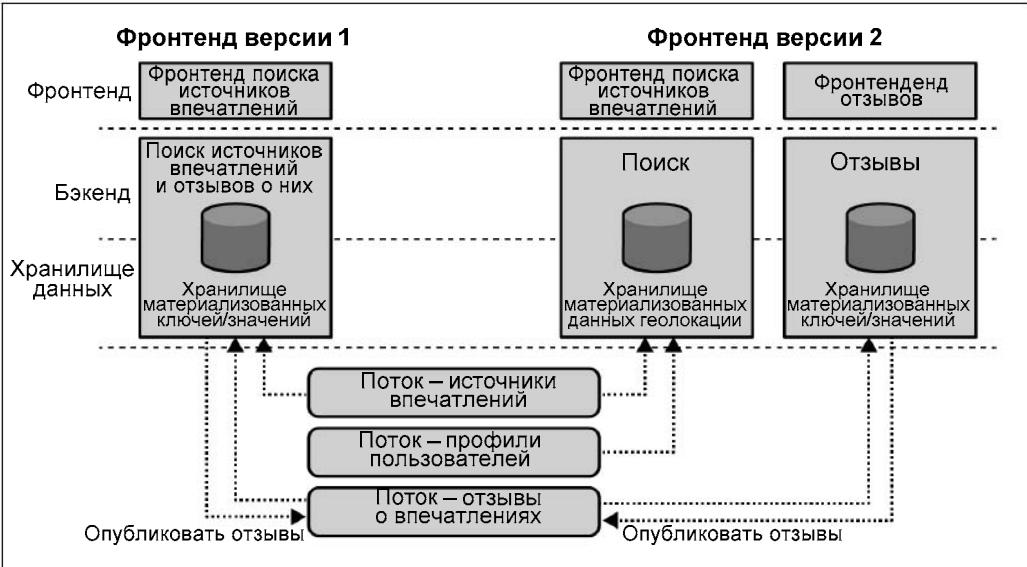


Рис. 13.17. Гибкость микрофронтендов
в сочетании с бэкендовыми событийно-управляемыми микросервисами



Способность материализовывать и потреблять любой поток бизнес-событий, независимо от того, насколько они нужны сервису, делает сочетание бэкендов на событийно-управляемых микросервисах с микрофронтендами столь эффективным.

Во-вторых, разбиение сервиса отзывов на самостоятельные микросервисы полностью отделило его ограниченный контекст и реализацию от ограниченного контекста и реализации поиска. В-третьих, сервис поиска заменил свое хранилище состояний на то, которое способно выполнять функциональность как обычного текстового, так и геолокационного поиска. Это изменение поддерживает бизнес-требования сервиса поиска, которые теперь могут обеспечиваться независимо от бизнес-требований сервиса отзывов. Такое решение показывает, как композиционные бэкенды дают командам разработчиков возможность использовать наилучшие инструменты для поддержки микрофронтендового продукта.

В этой новой версии микросервис поиска использует события из потока сущностей профиля пользователя для персонализации результатов поиска. Хотя версия 1 бэкендового сервиса, безусловно, может также потреблять и использовать эти данные, повышенная степень детализации сервисов в версии 2 проясняет, какие бизнес-функции используют данные пользователя. Наблюдатель может различить, какие потоки потребляются и используются для каждой части фронтенда, просто посмотрев на входные потоки для ограниченных контекстов. И наоборот, в версии 1, не углубившись в код, наблюдатель не имел бы представления о том, является ли использование пользовательских событий частью поиска или отзывов.

Наконец, обратите внимание, что все необходимые данные как для старой, так и для новой версии получены из одних и тех же потоков событий. Поскольку эти по-

токи событий являются единственным источником истины, вы можете изменять бэкенды приложений, не беспокоясь о поддержке конкретной реализации хранилища состояний или о переносе данных. Это резко контрастирует с монолитным бэкендом, где база данных также играет роль уровня обмена данными и не может быть легко заменена. Комбинация событийно-управляемого бэкенда с микрофронтиром ограничена только масштабом и степенью детализации имеющихся событийных данных.

Резюме

В этой главе рассмотрена интеграция событийно-управляемых микросервисов с API «запросов-ответов». Внешние системы преимущественно обмениваются данными через API типа «запрос-ответ», управляемые человеком или машиной, и их «запросы-ответы», возможно, придется преобразовать в события. Машинный ввод может быть описан схемой заранее, чтобы генерировать события, которые будут собираться на стороне сервера через API «запрос-ответ». Сторонние API обычно требуют синтаксического анализа и включения ответов в собственное определение события и, как правило, более уязвимы при изменении.

Взаимодействие с пользователями также может быть преобразовано в события, которые будут асинхронно обрабатываться потребляющим событийно-управляемым микросервисом. Для этого требуется интегрированный дизайн, в котором пользовательский интерфейс сообщает пользователю, что его запрос обрабатывается асинхронно. Трактуя все вводимые пользователем данные как потоки событий, реализация ограниченного контекста эффективно отвязывается от пользовательских данных. Это обеспечивает значительную гибкость в архитектурном развитии проекта и позволяет изменять компоненты без излишних трудностей.

Наконец, микрофронтиены предоставляют архитектуру для полнофункциональной разработки продуктов на основе событийно-управляемых микросервисов. Бэкендовые событийно-управляемые микросервисы по своей природе являются композиционными, объединяя события и сущности для применения бизнес-логики. Этот шаблон распространяется и на фронтенд, где пользовательский интерфейс не обязательно должен обеспечиваться одним большим монолитным приложением, а может в качестве компромисса быть реализован в виде ряда специально созданных микрофронтиендов. Каждый микрофронтенд обслуживает свою конкретную бизнес-логику и функциональность, а его общий композиционный уровень объединяет различные приложения вместе. Такой архитектурный стиль отражает модели автономности и развертывания бэкендовых микросервисов, обеспечивая полное выравнивание продукта и предлагая гибкие варианты фронтенда для экспериментов, сегментации и предоставления удобного пользовательского интерфейса.

Вспомогательные инструменты

Вспомогательные инструменты позволяют эффективно управлять событийно-управляемыми микросервисами в любом масштабе. Хотя многие из этих инструментов могут быть представлены интерфейсами командной строки, с которыми работают администраторы, все же лучше иметь набор инструментов, которые вы могли бы задействовать самостоятельно. Такие инструменты предоставляют возможности DevOps¹ по разработке и сопровождению продукта, необходимые для обеспечения масштабируемой и гибкой бизнес-структурой. Описанные в этой главе инструменты ни в коем случае не единственно доступные, но именно их я и многие другие сочли полезными на собственном опыте. Вашей организации нужно будет решить, что из них взять на вооружение.

К сожалению, сейчас не так много свободно доступных инструментов с открытым исходным кодом, пригодных для управления событийно-управляемыми микросервисами. Там, где это было возможно, я представил конкретные доступные реализации, но многие из них были созданы для компаний, в которых я работал, и права на их использование принадлежат им. Скорее всего, вам придется написать свои собственные инструменты, но я рекомендую вам задействовать доступные инструменты с открытым исходным кодом и по возможности вносить свой вклад в их развитие.

Система закрепления микросервисов за командами

Когда у компании небольшое количество систем, легко использовать основанные на предыдущем опыте «племенные знания» или неформальные методы, чтобы узнать, кто какими системами владеет. Однако в мире микросервисов важно явно отслеживать владение реализациами микросервисов и потоками событий. Следуя принципу единственного источника событий (см. разд. «Принцип единственного источника событий» главы 2), вы можете приписывать право владения потоком событий микросервису, у которого есть разрешения на запись.

Для отслеживания всех зависимостей между людьми, командами и микросервисами и управления ими можно воспользоваться простым микросервисом, разработанным внутри компании. Такая система является фундаментом для многих других

¹ DevOps — методология активного взаимодействия специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимная интеграция их рабочих процессов друг в друга для обеспечения качества продукта. — Прим. ред.

инструментов, описанных в этой главе, поэтому я настоятельно рекомендую вам заняться ее поиском или разработкой. Определение права владения микросервисом на ее основе помогает обеспечить правильное назначение разрешений DevOps командам, которые в них нуждаются.

Создание и модификация потока событий

Командам требуется возможность создавать новые потоки событий и соответствующим образом изменять их. Микросервисы также должны иметь право автоматически создавать свои собственные внутренние потоки событий и иметь полный контроль над такими важными их свойствами, как количество разделов, политика хранения и коэффициент репликации.

Например, поток, содержащий очень важные и чрезвычайно чувствительные данные, которые не должны быть потеряны ни при каких обстоятельствах, может иметь политику постоянного хранения и высокий коэффициент репликации. С другой стороны, поток, содержащий большой объем обновлений, не так уже в целом и важных, может включать значительное количество разделов с низким коэффициентом репликации и политикой короткого хранения. При создании потока событий принято назначать право владения потоком тому или иному микросервису или даже внешней системе. Об этом мы поговорим в следующем разделе.

Разметка потока событий метаданными

Одним из полезных технических приемов назначения прав владения является разметка (тегирование) потоков метаданными. При этом только те команды, которые владеют правами на поток, могут добавлять, изменять или удалять теги метаданных. Далее приведен ряд примеров полезных метаданных (разумеется, перечень этот не исчерпывающий):

- ◆ *Владелец потока (сервис).*

Сервис, владеющий потоком. Эти метаданные регулярно используются при передаче запросов на изменение или аудите того, какие потоки каким сервисам принадлежат, что добавляет ясность в права владения и структуру обмена бизнес- информацией любого микросервиса или потока событий в вашей организации.

- ◆ *Персонально идентифицирующая информация (PII, Personally Identifiable Information).*

Информация, которая требует более строгого обеспечения безопасности, поскольку может идентифицировать пользователей прямо или косвенно. Одним из базовых вариантов использования этих метаданных является ограничение доступа к любому потоку событий, помеченному как PII, если команда, владеющая данными, явно не дает разрешение.

- ◆ *Финансовая информация.*

Все, что связано с деньгами, выставлением счетов или другими важными событиями, приносящими доход. Этот тег похож, но не идентичен тегу PII.

◆ *Пространство имен.*

Дескриптор, соответствующий ограниченному контексту бизнеса. Поток с назначенным пространством имен может быть скрыт от сервисов за пределами пространства имен, но доступен для сервисов внутри него. Это помогает уменьшать перегрузку обнаружения данных, скрывая недоступные потоки событий для пользователя, просматривающего доступные потоки событий.

◆ *Сведения об устаревании формата данных в потоке.*

Способ указывать, что поток устарел (*deprecation*) или по какой-то причине был заменен. Разметка потока событий как устаревшего позволяет существующим системам продолжать его использовать, тогда как новые микросервисы уже не могут запрашивать подписку на него. Этот тег обычно задействуется, когда необходимо внести критические изменения в формат данных существующего потока событий. Новые события помещаются в новый поток, а старый поток поддерживается до тех пор, пока зависимые микросервисы не будут с него мигрированы. Наконец, владелец устаревшего потока событий может получить оповещение, что зарегистрированных потребителей этого потока более не осталось, после чего его можно будет безопасно удалить.

◆ *Индивидуально настраиваемые теги.*

Любые другие метаданные, подходящие для вашего бизнеса, можно и нужно отслеживать с помощью этого инструмента. Подумайте, какие теги могут быть важны для вашей организации, и убедитесь, что они доступны.

Квоты

Квоты, как правило, устанавливаются брокером событий на универсальном уровне. Например, брокер событий может быть настроен так, чтобы только 20% времени работы процессора уходило на обслуживание одного производителя или группы потребителей. Такая квота предотвращает случайный отказ в обслуживании из-за неожиданно возросшего объема событий от какого-либо производителя или сильно запараллельной группы потребителей, начиная со старта очень крупного потока событий. В общем, квоты применяются тогда, когда вы не хотите, чтобы весь ваш кластер был перегружен запросами ввода/вывода от одного сервиса. При этом вы можете просто установить лимит на объем ресурсов, который потребитель или производитель может использовать, и у него не получится выйти за его рамки.

Не исключено, что вам придется устанавливать квоты более точно, чтобы предотвратить перегрузку систем, подверженных повышенной нагрузке, и при этом обеспечить необходимый объем вычислительной мощности и сетевого ввода/вывода для постоянных потребителей. Возможно, вы также захотите установить разные квоты или полностью снять их для производителей, обрабатывающих данные из источников, находящихся за пределами кластера брокера событий. Например, производитель, публикующий события на основе сторонних входных потоков или внешних синхронных запросов, может просто потерять данные или дать сбой, если

его производительность будет установлена ниже скорости поступления входящих сообщений.

Реестр схем

Подробные схемы событий создают прочную основу для их моделирования. Точные определения данных, включающие имена, типы, значения по умолчанию и документацию, обеспечивают ясность как для производителей, так и для потребителей события. *Реестр схем* — это сервис, который позволяет вашим производителям сохранять схемы, которые они использовали для записи события. Его применение дает несколько явных выгод:

- ◆ схема события не нуждается в переносе вместе с событием. Можно передавать только идентификатор (ID) схемы, что значительно сократит загрузку полосы пропускания;
- ◆ реестр схем обеспечивает единую отправную точку при получении схемы для события;
- ◆ схемы позволяют обнаруживать данные, в особенности при свободном текстовом поиске.

Рабочий процесс реестра схем показан на рис. 14.1.

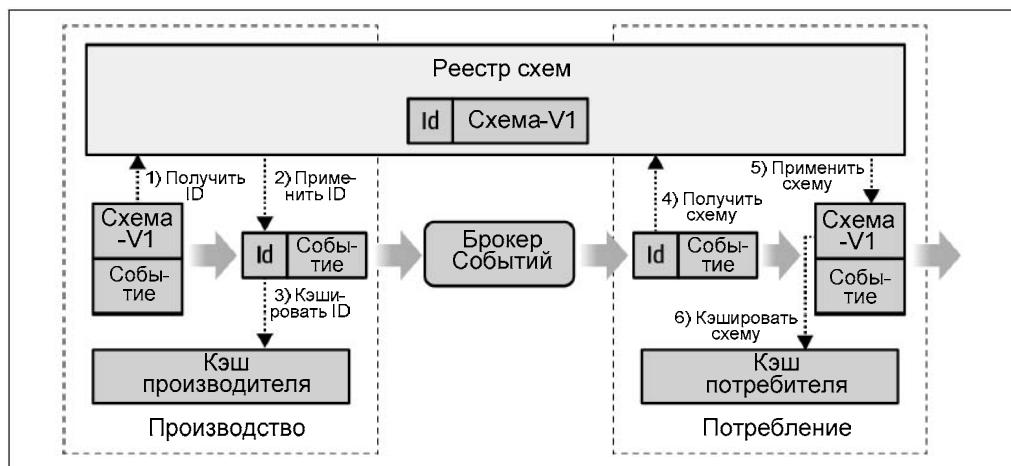


Рис. 14.1. Рабочий процесс реестра схем при производстве и потреблении события

Производитель (после сериализации события перед началом производства) регистрирует схему в реестре схем, чтобы получить ее ID (шаг 1). Затем он добавляет этот ID в сериализованное событие (шаг 2) и кэширует информацию в кэше производителя (шаг 3) во избежание повторного запроса реестра для этой схемы. Помните, что производитель должен завершить этот процесс для каждого события, поэтому очень важно исключить внешний запрос для известных форматов событий.

Потребитель получает событие и схему (шаг 4) для этого конкретного ID из своего кэша либо из реестра схем. Затем он меняет этот ID на схему (шаг 5) и десериали-

зует событие в известный формат. Схема кэшируется, если она новая (шаг 6), и десериализованное событие теперь может использоваться бизнес-логикой потребителя. На этом этапе к событию также может быть применено некоторое изменение схемы.

Библиотека Confluent обеспечила отличную реализацию реестра схем для Apache Kafka (<https://oreil.ly/5HT00>). Она поддерживает форматы Apache Avro, Protobuf и JSON и свободно доступна для использования.



Регистрация схем в выделенном потоке событий освобождает реализацию реестра схем от необходимости обеспечения надежного хранилища. Именно такой вариант организации реестра схем и предусматривает библиотека Confluen.

Оповещение о создании и модификации схем

Схемы потоков событий важны с точки зрения стандартизации обмена информацией. Но одна из проблем, которая может возникнуть, в особенности при большом числе потоков событий, заключается в том, что не всегда удается вовремя оповестить другие команды о том, что схема, от которой они зависят, изменилась (или будет изменяться). Именно здесь в игру вступают оповещения о создании и модификации схем.

Цель системы оповещений состоит в том, чтобы просто предупреждать потребителей об изменении их схем ввода. Списки управления доступом (Access Control List, ACL), рассматриваемые далее в этой главе, — отличный способ определять, какой микросервис из какого потока событий потребляет и, по ассоциации, от каких схем он зависит.

Обновления схемы можно получать из потока схемы (если вы используете реестр схем Confluent) и давать перекрестные ссылки на связанные с ними потоки событий. Учитывая все это, ACL-списки предоставляют информацию о том, какие службы какие потоки событий используют, а затем — через систему закрепления микросервисов за командами — уведомляют об этом соответствующие команды, владеющие сервисами.

Система оповещений имеет ряд преимуществ. Хотя в идеальном мире каждый потребитель мог бы полностью просматривать каждое изменение схемы, система оповещений обеспечивает страховочную сетку для выявления вредных или критических изменений до того, как они перерастут в кризис. Наконец, потребитель может захотеть просто следить за всеми общедоступными изменениями схемы в компании, что позволит ему лучше понимать данные по мере поступления новых потоков событий в оперативном режиме.

Управление смещением

Событийно-управляемые микросервисы требуют, чтобы вы управляли смещениями до того, как они перейдут к обработке данных. В условиях нормальной работы

микросервис будет продвигать смещение своего потребителя вперед по мере обработки сообщений. Однако бывают случаи, когда вам придется регулировать смещение вручную:

◆ *Сброс приложения: сброс смещения.*

Изменение логики микросервиса может потребовать повторной обработки событий из предыдущего момента времени. Обычно повторная обработка требует запуска в начале потока, но ваша точка выбора может варьироваться в зависимости от потребностей вашего сервиса.

◆ *Сброс приложения: продвижение смещения.*

С другой стороны, возможно, ваш микросервис не нуждается в старых данных и должен содержать только самые новые. Вы можете сбросить смещение приложения, чтобы оно стало самым последним смещением, а не самым ранним.

◆ *Восстановление приложения: указание смещения.*

Вы можете сбросить смещение на определенный момент времени. Это часто используется при многокластерном восстановлении после отказа, когда вы хотите убедиться, что не пропустили ни одного сообщения, но не хотите начинать с самого начала. Одна из стратегий включает сброс смещения на время N минут до сбоя, гарантируя, что ни одно реплицированное сообщение не будет пропущено.

В случае производственных операций DevOps команда должна владеть микросервисом, чтобы изменять его смещения. Эта функциональность предоставляется системой закрепления микросервисов за командами.

Разрешения и списки контроля доступа для потоков событий

Контроль доступа к данным важен не только с точки зрения безопасности бизнеса, но и как средство обеспечения принципа единственного источника событий. Списки разрешений и контроля доступа гарантируют, что ограниченные контексты могут не беспокоиться за соблюдение своих границ. Разрешения на доступ к заданному потоку событий должны предоставляться только той командой, которая владеет производящим микросервисом, — это ограничение вы можете установить, используя систему закрепления микросервисов за командами. Разрешения обычно попадают в следующие часто встречающиеся категории (в зависимости, конечно, от реализации брокера событий): READ (ЧТЕНИЕ), WRITE (ЗАПИСЬ), CREATE (СОЗДАНИЕ), DELETE (УДАЛЕНИЕ), MODIFY (МОДИФИКАЦИЯ) и DESCRIBE (ОПИСАНИЕ).



ACL-списки основаны на индивидуальной идентификации каждого потребителя и производителя. Убедитесь, что вы как можно раньше включили идентификацию для вашего брокера событий и сервисов — желательно с первого дня. Добавление идентификации постфактум чрезвычайно болезненно, поскольку требует обновления и проверки каждого отдельного сервиса, который подключается к брокеру событий.

ACL-списки обеспечивают соблюдение ограниченных контекстов. Так, микросервис должен быть единственным владельцем разрешений на СОЗДАНИЕ, ЗАПИСЬ и ЧТЕНИЕ для своих внутренних потоков событий и потоков журнала изменений. И ни в коем случае микросервис не должен связываться с внутренними потоками событий другого микросервиса. Кроме того, этот микросервис — в соответствии с принципом единственного источника событий — должен быть единственным сервисом, которому назначены разрешения на ЗАПИСЬ в его выходной поток. Выходные потоки могут быть сделаны как публично доступными, вследствие чего любая другая система сможет потреблять их данные, так и получить ограниченный доступ, если содержат чувствительные финансовые или РП-данные или являются частью вложенного ограниченного контекста.

Типичному микросервису может индивидуально назначаться набор разрешений в соответствии с форматом, показанным в табл. 14.1.

Таблица 14.1. Типичные разрешения потока событий для данного микросервиса

Компонент	Разрешения для микросервиса
Входные потоки событий	ЧТЕНИЕ
Выходные потоки событий	СОЗДАНИЕ, ЗАПИСЬ (и, возможно, ЧТЕНИЕ, если используются внутренние)
Внутренние потоки и потоки журнала изменений	СОЗДАНИЕ, ЗАПИСЬ, ЧТЕНИЕ

Особенно полезной функциональностью является предоставление отдельным командам средств запроса доступа потребителей к конкретному микросервису — с перекладыванием на них обязанности по обеспечению такого контроля доступа. Впрочем, в зависимости от бизнес-требований и тегов метаданных вы можете централизовать этот процесс, чтобы команды проходили проверку безопасности при каждом запросе доступа к конфиденциальной информации. Предоставление и отзыв разрешений могут храниться в виде собственного потока событий, что обеспечит длительную и неизменяемую запись доступа к данным для целей аудита.

Обнаружение неиспользуемых потоков и микросервисов

В ходе нормального роста бизнеса будут создаваться новые микросервисы и потоки, а устаревшие — удаляться. Перекрестные ссылки на список разрешений доступа к существующим потокам и микросервисам могут помочь в обнаружении возможных «сирот». Если у потока нет потребителей, он может быть помечен для удаления, если микросервис этого потока событий не производит никаких других данных в других потоках событий при активном потреблении, то он также может быть удален. Таким образом, вы можете использовать список разрешений для поддержания работоспособности и актуальности потока событий и бизнес-топологии.

Управление состоянием и сброс приложения

Обычно при изменении реализации приложения с поддержкой состояния выполняется сброс внутреннего состояния приложения. Любые изменения в структурах данных, хранящихся во внутренних потоках событий и потоках журнала измене-

ний, а также любые изменения в бизнес-процессе топологии потребуют удаления и повторного создания потоков в соответствии с новым приложением.

В некоторых из рассмотренных в главе 7 шаблонов микросервисов с поддержкой состояния используются хранилища состояний, внешние по отношению к обрабатывающему узлу. В зависимости от возможностей, поддерживаемых организацией платформы микросервисов в вашей компании, бывает возможным (и, безусловно, желательным) сбрасывать эти внешние хранилища состояний по запросу владельца микросервиса. Например, если микросервис использует внешнее хранилище состояний, такое как Amazon DynamoDB или Google Bigtable, то было бы неплохо очищать ассоциированное состояние при сбросе приложения. Это уменьшает операционные издержки и обеспечивает, чтобы любые устаревшие или ошибочные данные автоматически удалялись. Любые внешние сервисы с поддержкой состояния вне предметной области «официально поддерживаемых возможностей», скорее всего, должны быть автоматически сброшены.

Важно отметить следующий момент: хотя инструмент сброса внутреннего состояния приложения должен работать в «режиме самообслуживания», ни в коем случае ни одна команда не должна иметь возможность удалять потоки событий и состояния, принадлежащие другой команде. Опять же, я рекомендую использовать обсуждаемую в этой главе систему закрепления микросервисов за командами, чтобы гарантировать, что приложение может быть сброшено только его владельцем или администратором.

Таким образом, этот инструмент должен:

- ◆ удалять внутренние потоки микросервиса и потоки журнала изменений;
- ◆ удалять все внешние материализации хранилища состояний (если это применимо);
- ◆ сбрасывать смещения группы потребителей в начало для каждого входного потока.

Мониторинг запаздывания смещения потребителей

Отставание, или задержка, потребителей (*consumer lag*) — это один из самых лучших показателей, необходимых для масштабирования событийно-управляемого микросервиса. Вы можете отслеживать его с помощью инструмента, который периодически вычисляет отставание рассматриваемых групп потребителей. Хотя механизм этого инструмента может варьироваться в разных реализациях брокера, определение отставания (задержки) одно и то же: разница в количестве событий между самым последним событием и последним обработанным событием для той или иной группы потребителей микросервиса. Базовые показатели отставания, такие как пороговый размер, достаточно просты и легко реализуемы. Например, если задержка смещения потребителей больше, чем N событий в течение M минут, то следует удвоить количество обработчиков потребителей и перебалансировать рабочую нагрузку. Если задержка урегулирована и число выполняемых обработчи-

ков в настоящее время превышает минимально необходимое, то масштабировать количество обработчиков вниз.

В некоторых системах мониторинга, таких как Burrow для Apache Kafka, при вычислении состояния задержки смещения учитывается ее история. Этот подход может быть полезен в тех случаях, когда в поток входит крупный объем событий, в результате чего величина задержки всегда будет равна нулю в течение доли секунды до поступления следующего события. Поскольку размеры задержки, как правило, носят периодический характер, вполне возможно, что система при обычном измерении всегда будет казаться запаздывающей. Поэтому полезным механизмом определения того, отстает ли система или догоняет, является учет величины отклонения от исторических норм.

Помните, что хотя микросервисы должны быть свободны для масштабирования вверх и вниз по мере необходимости, обычно некоторую форму гистерезиса — порога допуска — следует предусматривать для предотвращения бесконечного колебания системы туда-сюда. Эта петля гистерезиса должна быть частью логики, которая оценивает сигнал, и часто может быть адаптирована современными облачными платформами, такими как AWS CloudWatch и Google Cloud Operations (ранее Stackdriver).

Оптимизированный процесс создания микросервисов

Создание репозитория кода для нового бизнес-требования — типичная задача в среде микросервисов. Автоматизация этой задачи на основе оптимизированного процесса обеспечивает, чтобы все друг с другом сочеталось и интегрировалось в общий инструментарий, предоставляемый командами по функциональным возможностям.

Вот типичный процесс создания микросервиса:

1. Создать репозиторий.
2. Создать все необходимые интеграции с помощью конвейера непрерывной интеграции (см. разд. «Системы непрерывной интеграции, доставки и развертывания» главы 16).
3. Настроить любые веб-перехватчики (web-hook) или другие зависимости.
4. Назначить команде разрешения владеть ресурсами с помощью системы закрепления микросервисов за командами.
5. Зарегистрировать получение разрешений на доступ изнутри входных потоков.
6. Создать необходимые выходные потоки и применить разрешения на владение ресурсами.
7. Предоставить возможность применения шаблона или генератора кода для создания скелета микросервиса.

Команды будут выполнять этот процесс много раз, поэтому оптимизация его таким образом сэкономит значительное время и усилия. Новый автоматизированный

рабочий процесс включает в себя точку входа современных шаблонов и генераторов кода, обеспечивая тем самым, чтобы новые проекты не просто копировали старый проект, а включали в себя новейший поддерживаемый код и инструменты.

Элементы управления контейнерами

Управление контейнерами осуществляется с помощью различных специализированных программ, именуемых *системами управления контейнерами* (Container Management System, CMS), уже упомянутых в главе 2. Я рекомендую раскрыть некоторые аспекты CMS, чтобы команды могли предоставлять свои собственные возможности DevOps, такие как:

- ◆ настройка переменных окружения для своих микросервисов;
- ◆ указание, на каком кластере запускать микросервис (например, тестирование, интеграция, производство);
- ◆ управление процессором, памятью и дисковыми ресурсами в зависимости от потребностей их микросервисов;
- ◆ увеличение и уменьшение количества сервисов вручную или в зависимости от соглашений об уровне обслуживания и задержки обработки;
- ◆ автоматическое масштабирование работы процессора, памяти, диска или задержки.

Бизнесу необходимо будет определить, сколько вариантов управления контейнерами должно быть доступно разработчикам, а сколько — управляться специальной операционной группой. Обычно это зависит от культуры DevOps в организации.

Создание кластеров и управление ими

Необходимость в создании кластеров и управлении ими, как правило, возникает по мере масштабирования компании на основе событийно-управляемых микросервисов. Вообще говоря, малые и средние компании часто могут обойтись без использования единого кластера брокеров событий для всех своих потребностей в обслуживании. Однако более крупные компании часто оказываются под давлением необходимости создания нескольких кластеров по различным техническим и юридическим причинам. Международным компаниям, возможно, потребуется хранить некоторые данные в пределах страны их происхождения. Объемы данных могут вырасти настолько, что все их практически невозможно будет держать в одном единственном кластере, несмотря на превосходные качества горизонтального масштабирования современных брокеров событий. Различным бизнес-единицам в организации могут потребоваться собственные кластеры в целях изоляции друг от друга. Возможно, наиболее часто данные должны реплицироваться в масштабах региона в нескольких кластерах, чтобы обеспечить резервирование в случае полного сбоя какого-либо из них.

Многокластерное управление, включающее динамический межрегиональный обмен информацией и восстановление данных, — сложная тема, которая вполне мог-

ла бы стать предметом отдельной книги. Все это также сильно зависит от соответствующих сервисов и используемых стратегий профилактики и восстановления. Некоторые компании, такие как Capital One, имеют значительные пользовательские библиотеки и код, построенный на их реализациях Apache Kafka, что обеспечивает нативную многокластерную репликацию. Будучи банком, эта компания не может позволить себе потерять какие-либо события финансовых транзакций. Однако ваши потребности могут быть иными, и по этой причине здесь не рассматриваются стратегии многокластерного управления сервисами и данными.

Программная доводка брокеров событий

Команда, ответственная за управление кластерами брокеров событий, часто также предоставляет инструменты для создания новых кластеров и управления ими. Тем не менее коммерческие облачные провайдеры также перемещаются в эту область — например, кластеры Apache Kafka теперь могут по требованию создаваться в облачном сервисе AWS (по состоянию на ноябрь 2018 года), присоединившемуся к ряду других поставщиков облачных сервисов. Различные технологии брокеров событий могут потребовать разного объема работы для поддержки и должны быть тщательно изучены экспертами в вашей предметной области. В любом случае цель состоит в том, чтобы иметь инструмент управления кластером брокеров событий, который вся организация может использовать для простого создания и масштабирования таких брокеров.

Программная доводка вычислительных ресурсов

Вам часто придется задействовать набор вычислительных ресурсов, которые не зависят от всех других ресурсов. При этом не всегда необходимо создавать совершенно новый сервис управления контейнерами, поскольку существующий сервис обычно может обслуживать несколько пространств имен. Как и в случае с брокерами событий, провайдеры облачных вычислений нередко предоставляют размещенные на своих хостах сервисы — такие как решения Google и Amazon Kubernetes — которые смогут обеспечить вам требуемые возможности по запросу.

Те же технические и юридические требования, которые относятся к событийно-управляемым брокерам, распространяются и на вычислительные ресурсы. Избегайте региональных сбоев, распределяя обработку по центрам обработки данных, обрабатывайте данные локально, если они не могут покинуть страну происхождения, и экономьте деньги, оперативно перекладывая тяжелые вычислительные нагрузки на более дешевых поставщиков сервисов.

Обычно для выполнения этой задачи можно использовать одни и те же инструменты *непрерывной интеграции* (Continuous Integration, CI) и *непрерывной доставки* (Continuous Delivery, CD), но вам понадобится механизм выбора, чтобы определить, где развернуть микросервисы. Кроме того, вам нужно будет убедиться, что необходимые событийные данные доступны вычислительным ресурсам, что, как правило, обеспечивается за счет совместного их размещения в одном регионе или

в одной зоне доступности. Хотя межрегиональный обмен информацией всегда возможен, но он зачастую дорогой и медленный.

МежклUSTERНАЯ РЕПЛИКАЦИЯ СОБЫТИЙНЫХ ДАННЫХ

Репликация событийных данных между кластерами важна для масштабирования событийно-управляемых микросервисов за пределами границ одного кластера — например, для целей аварийного восстановления, регулярного межклUSTERного обмена информацией и программно генерируемых сред тестирования.

Специфика репликации данных между кластерами варьируется в зависимости от реализаций брокера событий и инструмента репликации. При отборе варианта реализации средства репликации учитывайте следующее:

- ◆ реплицирует ли он вновь добавляемые потоки событий автоматически;
- ◆ как он обрабатывает репликацию удаленных или модифицированных потоков событий;
- ◆ реплицируются ли данные точно, с одинаковыми смещениями, разделами и временными метками или же приблизительно;
- ◆ какова задержка репликации, приемлемо ли это для бизнес-потребностей;
- ◆ каковы его эксплуатационные характеристики, может ли он масштабироваться в соответствии с бизнес-требованиями?

ПРОГРАММНАЯ ДОВОДКА ИНСТРУМЕНТАРИЯ

И последнее, но не менее важное: те же самые наборы инструментов, которые обсуждались до сих пор, также следует программно задействовать и для новых кластеров. Это обеспечит наличие общего набора инструментов, которые можно использовать в любом кластерном развертывании, не полагаясь на какие-либо иные хранилища данных, кроме самого брокера событий. Такой способ обращения с инструментами имеет ряд преимуществ. Во-первых, инструменты используются гораздо чаще, что помогает выявлять случающиеся в них ошибки и добавлять требуемые функциональности. Во-вторых, это снижает порог входа в новые кластеры, поскольку пользователи уже будут знакомы с интерфейсами инструментов. Наконец, когда работа кластера завершается, инструменты могут быть остановлены вместе с ним без дополнительной очистки.

Отслеживание зависимостей и визуализация топологии

Отслеживать зависимости данных между микросервисами чрезвычайно полезно для управления организацией, работающей на основе событийно-управляемых микросервисов. Единственное требование — организация должна знать, какие микросервисы читают и записывают в какие потоки событий. Для достижения этой цели она может задействовать систему самоотчетности, при которой потребители и производители отчитываются о своих собственных шаблонах потребления и произ-

водства. Однако проблема с любым решением на основе самоотчетности заключается в том, что она фактически является мероприятием добровольным, и всегда найдется ряд команд, которые забудут, откажутся или просто не захотят отчитываться. Определять зависимости без полного знания происходящих процессов не особенно полезно, т. к. пробелы в структуре обмена информацией и неполные топологии ограничивают их понимание. Именно здесь в игру вступают структура разрешений и ACL-списки, рассмотренные ранее в этой главе.

Использование структуры разрешений для определения зависимостей гарантирует обеспечение двух важных моментов. Во-первых, микросервис не может работать без регистрации своих требований к разрешениям, поскольку не сможет читать или записывать в какие-либо потоки событий. Во-вторых, если в структуру разрешений вносятся какие-либо изменения, соответствующие разрешения, используемые для определения зависимостей и генерации топологии, также обновляются. Никаких других изменений для обеспечения надлежащего отслеживания зависимостей не требуется.

Вот некоторые другие варианты использования такого инструмента:

◆ *Определять происхождение данных.*

Одна из проблем, с которой регулярно сталкиваются исследователи данных и специалисты по обработке данных, заключается в определении того, откуда пришли данные и каким маршрутом они прошли. Имея полный график структуры разрешений, они могут идентифицировать каждый предшествующий сервис и поток любого события. Это помогает им отслеживать сбои и ошибки вплоть до источника и определять все сервисы, участвующие в конкретной трансформации данных. Помните, что всегда можно вернуться назад в любой момент времени событийных потоков разрешений и событийных потоков закрепления микросервисов за командами, чтобы получить внешний вид топологии в тот момент времени. Это часто бывает очень полезно при аудите старых данных.

◆ *Исследовать границы команд.*

Команды, владеющие микросервисами и потоками, могут быть сопоставлены с топологией. Визуализация топологии с помощью соответствующего инструмента визуализации четко покажет, какие команды непосредственно отвечают за какие сервисы.

◆ *Обнаруживать источники данных.*

Визуализаторы представляют собой полезный инструмент обнаружения данных. Потенциальный потребитель может видеть, какие потоки доступны и кто их производители и потребители. Если потребуется дополнительная информация о потоковых данных, то потенциальный потребитель сможет связаться с их производителями.

◆ *Измерять взаимосвязанность и сложность.*

Для команд точно так же, как и для микросервисов, идеально быть сильно сцепленными и слабо связанными. С помощью инструмента визуализации топологии команда может измерить, сколько имеется у микросервисов внутренних соединений.

нений и сколько внешних. Общее эмпирическое правило состоит в том, что чем меньше внешних подключений, тем лучше, однако простой подсчет количества подключений — это не очень значимый показатель. Тем не менее его последовательное рассмотрение может выявить относительную взаимозависимость между командами.

◆ *Соотнесение бизнес-требований с микросервисами.*

Выравнивание микросервисов по бизнес-требованиям позволяет соотносить с бизнес-требованиями их реализацию. Разумно явно указать бизнес-требования каждого микросервиса вместе с его кодом, например, в репозитории README или в хранилище метаданных микросервиса. В свою очередь, их можно сопоставить с командами-владельцами.

Владелец бизнеса может взглянуть на данные этих исследований и спросить себя: «Соответствует ли реализованная структура его целям и приоритетам?» Таким образом, инструмент визуализации топологии — один из самых важных инструментов, которые может иметь в своем распоряжении бизнес, чтобы гарантировать, что его команды соответствуют структуре требуемого обмена бизнес-информацией.

Пример топологии

На рис. 14.2 показана топология с 25 микросервисами, наложенными на права владения четырех команд. Каждая стрелка здесь представляет производство данных в поток событий, а также потребление их потребляющим процессом. Например, микросервис 3 потребляет поток данных из микросервиса 4.

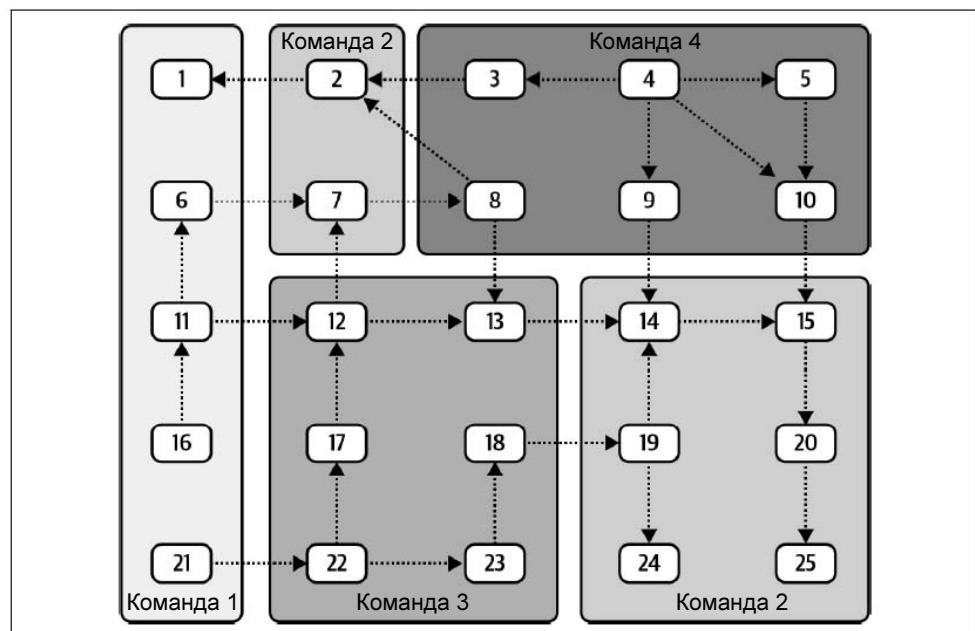


Рис. 14.2. Топологическая карта соединений сервисов

Приведенная топология показывает, что команда 2 отвечает за два микросервиса, которые не являются частью ее главного ограниченного контекста (внизу справа). Это может вызывать беспокойство, если бизнес-цели команды 2 не совпадают с функциями, которые обслуживаются микросервисами 2 и 7. Кроме того, микросервисы 2 и 7 имеют ряд зависимостей от команд 1, 3 и 4, что увеличивает «площадь поверхности», которую команда 2 демонстрирует внешнему миру. Мера взаимосвязанности этой топологии показана в табл. 14.2.

Таблица 14.2. Мера взаимосвязанности топологии, приведенной на рис. 14.2

	Входящие потоки	Исходящие потоки	Входящие соединения команды	Исходящие соединения команды	Владеемые сервисы
Команда 1	1	3	1 (Команда 2)	2 (Команды 2, 3)	5
Команда 2	8	2	3 (Команды 1, 3, 4)	2 (Команды 1, 4)	8
Команда 3	3	3	2 (Команды 1, 4)	1 (Команда 2)	6
Команда 4	1	5	1 (Команда 1)	2 (Команды 2, 3)	8

Давайте посмотрим, что произойдет, если мы уменьшим количество соединений между командами и число входящих и исходящих потоков на границах команд. Микросервисы 2 и 7 являются хорошими кандидатами на переназначение просто из-за их небольшой области владения в топологии и могут быть легко переназначены для уменьшения числа зависимостей между командами. Микросервис 7 можно назначить команде 1 (или команде 3), а микросервис 2 — команде 4. Теперь становится более очевидным и то, что микросервис 1 может быть назначен команде 4 для дальнейшего сокращения трансграничного обмена информацией. Результат этих переназначений показан на рис. 14.3 и в табл. 14.3.

Таблица 14.3. Новая мера взаимосвязанности топологии
(различия с табл. 14.2 показаны в скобках)

	Входящие потоки	Исходящие потоки	Входящие соединения команды	Исходящие соединения команды	Владеемые сервисы
Команда 1	1	3	1 (Команда 3)	2 (Команда 3) (-1)	5
Команда 2	4 (-4)	0 (-2)	3 (Команды 3, 4) (-1)	0 (-2)	8
Команда 3	3	3	2 (Команды 1, 4)	2 (Команды 1, 2) (+1)	6
Команда 4	1	3 (-2)	1 (Команда 1)	2 (Команды 3, 4)	8

Изучение трансграничных зависимостей дает положительный результат: количество входящих и исходящих потоков между командами сокращается, а общее количество соединений между ними уменьшается на три. Таким образом, сведение к минимуму количества трансграничных подключений помогает оптимизировать назначение командам тех или иных микросервисов. Однако для окончательного закрепления микросервисов за командами этого недостаточно. Вы также

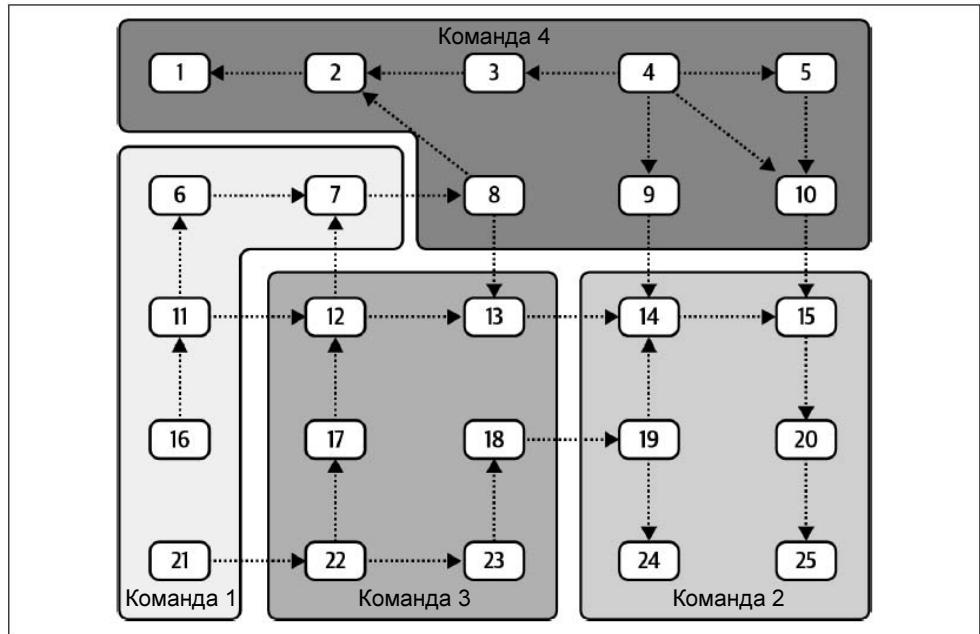


Рис. 14.3. Топологическая карта соединений после переназначения микросервисов

должны принять во внимание множество факторов — таких как количество сотрудников в команде, области их компетенции и сложности реализации.

Однако, что наиболее важно, вы должны учитывать бизнес-функции, которые выполняет микросервис. Рассмотрим сценарий, в котором одна команда производит целый ряд данных о событиях, возможно, полученных из нескольких внешних источников. Может случиться так, что бизнес-ответственность этой команды ограничивается простым поиском и организацией данных в события, а бизнес-логика выполняется ниже по потоку другими потребителями. В этом случае у команды будет много потоковых соединений и командных связей. Здесь полезно иметь возможность просматривать бизнес-функции, связанные с микросервисами, принадлежащими команде.

По приведенному примеру стоит задать несколько вопросов. Во-первых, соответствуют ли реализации бизнес-функций микросервиса 2 целям команды 2 или команды 4? А как насчет микросервиса 7: он ближе к целям команды 1, чем команды 2? И вообще, какие сервисы лучше всего подходят какой команде? Ответы на эти вопросы, как правило, носят качественный характер и должны быть тщательно оценены с точки зрения целей команд. Естественно, что цели команд меняются по мере развития потребностей бизнеса, и важно обеспечить, чтобы микросервисы, назначенные этим командам, соответствовали общим бизнес-целям. При этом инструмент отслеживания зависимостей и визуализации топологии дает владельцам бизнеса понимание производимых назначений и помогает обеспечить им ясность производимых изменений.

Резюме

Наличие в системе нескольких автономных сервисов требует тщательного обдумывания того, как вы собираетесь управлять такой системой. Описанные в этой главе инструменты предназначены для того, чтобы помочь вашей организации управлять своими сервисами.

По мере увеличения количества сервисов способность любого человека знать, как все работает и как каждый сервис и поток вписываются в общую картину, уменьшается. Явное отслеживание таких свойств, как владение сервисом и потоком, схемы и метаданные, позволяет организациям систематизировать и отслеживать изменения во времени. Прекращение следования «племенным знаниям» и формальная кодификация в остальном простых свойств потоков и сервисов для вашей организации должны стать приоритетными, поскольку любая неопределенность в свойствах сервиса или потока будет только усиливаться для каждого дополнительного созданного экземпляра.

Автономность и контроль над сервисами вашей команды — важные аспекты управления микросервисами в большом масштабе. В соответствии с принципами DevOps вы должны иметь возможность сбрасывать смещения групп потребителей и хранилища состояний микросервисов.

Схемы способствуют общему пониманию значения событий. Реестр схемы предоставляет механизм оптимизации управления схемами и может использоваться для уведомления заинтересованных сторон о любых изменениях в конкретной схеме.

Тестирование событийно-управляемых микросервисов

Одна из замечательных особенностей тестирования событийно-управляемых микросервисов заключается в том, что они очень модульные: ввод в сервис обеспечивается потоками событий или запросами из API «запрос-ответ», состояние материализуется в собственное независимое хранилище состояний, а выходные события записываются в выходные потоки сервиса. Небольшие размеры микросервисов и их специализированный характер значительно упрощают их тестирование по сравнению с более крупными и сложными сервисами. У микросервисов меньше связей, относительно стандартная методология обработки ввода/вывода и состояния, а также масса возможностей повторного использования инструментария тестирования с другими микросервисами. В этой главе рассматриваются принципы и стратегии тестирования событийно-управляемых микросервисов, включая их модульное тестирование, интеграционное тестирование и тестирование производительности.

Общие принципы тестирования

Для тестирования событийно-управляемых микросервисов используются его лучшие методы, общие для всех приложений. *Функциональное* тестирование, такое как модульное, интеграционное, системное и регрессионное, позволяет убедиться, что микросервис делает то, что должен, и не делает того, чего не должен. *Нефункциональное* тестирование, такое как тестирование производительности, нагрузки, стресса и восстановления, дает возможность понять, будет ли он себя вести так, как ожидалось, в различных сценариях среды исполнения.

Здесь, прежде чем идти дальше, важно отметить, что эта глава призвана стать лишь дополнением к более глубоким работам по принципам и методам тестирования. В конце концов, по теме тестирования написано множество книг, имеются соответствующие блоги и документы, и я, разумеется, не способен охватить тестирование в той степени, в какой это делается в них. В этой главе в первую очередь рассматриваются методологии и принципы тестирования, ориентированные на конкретные события, и то, как они интегрируются в общую картину тестирования. Поэтому, чтобы расширить свое представление о методах тестирования, обратитесь также к дополнительным источникам по языковым фреймворкам тестирования и лучшим его приемам.

Модульное тестирование функций топологии

Модульные тесты используются для тестирования мельчайших фрагментов кода приложения, чтобы убедиться, что они работают должным образом. Эти малые единицы тестирования обеспечивают основу, для которой могут быть написаны более крупные и комплексные тесты для тестирования общей функциональности приложения. Событийно-управляемые топологии часто применяют к событиям функции трансформации, агрегации, соотнесения (отображения) и редукции, что делает эти функции идеальными кандидатами для модульного тестирования.



Обязательно проверьте граничные условия, такие как нулевые и максимальные значения, для каждой из функций вашего приложения.

Функции без поддержки состояния

Функции без поддержки состояния не требуют какой-либо устойчивой взаимосвязи с предыдущими вызовами функций, поэтому их довольно легко протестировать независимо. В следующем коде показан пример топологии EDM, аналогичной топологии, которую можно найти во фреймворке в стиле MapReduce:

```
myInputStream
    .filter(myFilterFunction)
    .map(myMapFunction)
    .toOutputStream()
```

Здесь `myMapFunction` и `myFilterFunction` являются независимыми функциями, ни одна из которых не поддерживает состояние. Каждая из них должна быть подвергнута модульному тестированию, позволяющему убедиться, что она правильно обрабатывает диапазон ожидаемых входных данных и в особенности граничные случаи.

Функции с поддержкой состояния

Функции с поддержкой состояния, как правило, сложнее тестировать, чем функции без поддержки состояния. Состояние может варьироваться как во времени, так и во входных событиях, поэтому вы должны позаботиться о том, чтобы проверить все необходимые граничные случаи таких функций. Модульное тестирование с поддержкой состояния также требует, чтобы устойчивое состояние, будь то в виде имитируемого внешнего хранилища данных или временного внутреннего хранилища данных, было доступно в течение всего теста.

Вот пример функции агрегации с поддержкой состояния, которую можно найти в реализации базового производителя/потребителя:

```
public Long addValueToAggregation(String key, Long eventValue) {
    // Хранилище данных должно быть доступно
    // среди модульного тестирования
    Long storedValue = datastore.getOrDefault(key, 0L);
```

```
// Просуммировать значения и загрузить их
// обратно в хранилище состояний
Long sum = storedValue + eventValue;
datastore.upsert(key, sum);
return sum;
}
```

Эта функция используется для суммирования всех `eventValue` по каждому ключу. Имитация (mocking) конечной точки — это один из способов обеспечения надежной реализации хранилища данных на протяжении всего теста. Еще один вариант состоит в создании локально доступной версии хранилища данных, хотя это больше похоже на интеграционное тестирование, которое будет рассмотрено подробнее чуть позже. В любом случае вы должны тщательно продумать вопрос о том, что должно делать это хранилище данных и как оно связано с фактической реализацией, используемой во время выполнения. Имитация, как правило, хорошо справляется с задачей, потому что она позволяет проводить очень производительное модульное тестирование, которое не обременено накладными расходами на запуск полной реализации хранилища данных.

Тестирование топологии

Полнофункциональные «легковесные» и «тяжеловесные» фреймворки обычно предоставляют средства для локального тестирования всей вашей топологии. Если ваш фреймворк этого не делает, то сообщество пользователей и участников, возможно, уже создало сторонний вариант, предоставляющий такую функциональность (это еще одна причина выбирать фреймворк с сильным сообществом). Например, для Apache Spark разработаны два отдельных сторонних варианта модульных тестов: `StreamingSuiteBase` (<https://oreil.ly/1e4lr>) и быстрые spark-тесты (<https://oreil.ly/jkoI5>) — в дополнение к предоставляемому им встроенному классу `MemoryStream` (<https://oreil.ly/5Ao0U>), обеспечивающему точное управление потоками ввода и вывода. Apache Flink (<https://oreil.ly/dHZb5>), как и Apache Beam (<https://oreil.ly/hnMRJ>), имеют собственные варианты тестирования топологии. Что касается «легковесных» потоковых фреймворков, то Kafka Streams предоставляет средства для тестирования топологий с помощью технологии `TopologyTestDriver` (<https://oreil.ly/R0fg9>), которая имитирует функциональность фреймворка, не требуя от вас настройки всего брокера событий.

Тест топологии является более сложным, чем единичный модульный тест, поскольку он проверяет всю топологию в соответствии с вашей бизнес-логикой. Вы можете представить себе свою топологию как одну большую сложную функцию с множеством связанных частей. Фреймворки тестирования топологии позволяют осуществлять полный контроль над тем, какие события продаются во входные потоки и когда они создаются. Вы можете генерировать события с конкретными значениями: события, которые находятся вне порядка, содержат недействительные временные метки или включают недействительные данные, а также события, которые служат для исполнения логики граничного случая. Благодаря этому вы можете

обеспечить, чтобы такие операции, как агрегирование по времени, планирование событий и функции с поддержкой состояния, выполнялись должным образом.

Рассмотрим, например, следующее определение топологии в стиле MapReduce:

```
myInputStream
    .map (myMapFunction)
    .groupByKey ()
    .reduce (myReduceFunction)
```

В этой топологии потребляемые события представлены переменной `myInputStream`. К ней применяется функция соотнесения, результаты группируются по ключам и в завершение сводятся к одному событию в расчете на ключ. Тем не менее модульные тесты хотя и могут быть реализованы для функций `myMapFunction` и `myReduceFunction`, но они не способны легко воспроизвести фреймворковые операции `map`, `groupByKey` и `reduce`, поскольку эти операции (среди прочих) по своей сути являются являющимися неотъемлемыми частями фреймворка.

И именно здесь в игру вступает тестирование топологии. Каждый потоковый фреймворк имеет разные уровни поддержки для тестирования топологии, и вы должны разобраться с доступными вам вариантами, учитывая, что такие тестовые фреймворки *не* требуют создания брокера событий для хранения входных событий или настройки кластера «тяжеловесного» фреймворка для их обработки.

Тестирование развития и совместимости схем

Чтобы убедиться, что любые выходные схемы совместимы с предыдущими схемами и в полной мере отвечают правилам развития схем потока событий (см. разд. «*Полнофункциональное развитие схемы*» главы 3), вы можете извлекать схемы из реестра схем и выполнять их проверку на соответствие правилам развития, которую можно рассматривать как часть процесса предъявления кода на рассмотрение. Некоторые приложения могут использовать инструменты генерации схем для автоматического создания схем из классов или структур, определенных в коде во время компиляции, в результате чего программно сгенерированная схема может сопоставляться с предыдущими версиями.

Интеграционное тестирование событийно-управляемых микросервисов

Существуют два основных типа интеграционного тестирования микросервисов: локальное интеграционное тестирование, когда тестирование выполняется на локализованной реплике производственного окружения, и удаленное интеграционное тестирование, когда микросервис исполняется на внешнем по отношению к локальной системе окружении. Каждое из этих решений имеет ряд преимуществ и недостатков, которые мы вскоре рассмотрим.

Есть и третий, гибридный, вариант — когда некоторые части вашего микросервиса и его тестового окружения размещаются на хосте или исполняются локально, а

другие части — удаленно. Поскольку технически невозможно оценить все комбинации и перестановки такого гибридного шаблона, я просто сосредоточусь на двух основных типах интеграционного тестирования и предоставлю вам самим определять свои собственные требования, если они будут отличаться.

Есть несколько общих вопросов, которые вам следует иметь в виду до конца этой главы:

- ◆ что вы надеетесь получить от интеграционного тестирования? Является ли оно таким же простым, как ответ на вопрос: «Работает это или нет?» Является ли оно «проверкой на дым» (smoke test)¹ производственными данными? Или есть еще какие-либо сложные рабочие процессы, которые необходимо протестировать и проверить;
- ◆ должен ли ваш микросервис поддерживать перезапуск с начала поступления входного потока, как это бывает необходимо в случае полной потери данных или повторной обработки из-за сбоя? Если да, то что вам нужно знать, чтобы протестировать работоспособность этой функциональности? Вам также может потребоваться проверить способность ваших входных потоков событий поддерживать это требование;
- ◆ какие данные вам нужны для определения успеха или неуспеха? Достаточно ли созданных вручную событийных данных? А созданных программно? Должны ли это быть реальные производственные данные? Если да, то в каком объеме;
- ◆ есть ли у вас какие-либо проблемы с производительностью, нагрузкой, пропускной способностью или масштабированием, которые необходимо протестировать;
- ◆ как вы будете следить за тем, чтобы каждый выстраиваемый вами микросервис не требовал полной собственной разработки среды для интеграционного тестирования?

Следующие разделы помогут вам разобраться с некоторыми из доступных вам вариантов, чтобы вы могли сформулировать свои собственные ответы на эти вопросы.

Локальное интеграционное тестирование

Локальное интеграционное тестирование позволяет проводить широкий спектр функциональных и нефункциональных тестов. В этом виде тестирования используется локальная копия производственной среды, в которой будет развернут микросервис. Как минимум это означает создание брокера событий, реестра схем, специфичных для микросервиса хранилищ данных, самого микросервиса и необходимого обрабатывающего фреймворка (например, когда вы используете «тяжелое

¹ «Проверка на дым» (smoke test) — в машиностроении испытание, включающее быструю подачу полной нагрузки на двигатель, работающий при постоянной частоте вращения, в ожидании того, задымит он или нет. — Прим. перев.

весный» фреймворк или FaaS). Вы также можете ввести контейнеризацию, журнализацию и даже систему управления контейнерами, но они не связаны строго с бизнес-логикой микросервиса и поэтому не являются абсолютно необходимыми.

Самая большая польза от разворачивания собственной локально управляемой среды заключается в том, что она предоставляет возможность управлять каждой системой независимо. Вы можете программно создавать сценарии, которые реплицируют фактические производственные ситуации, такие как кратковременные отказы, внепорядковые события и потеря доступа к сети. Вы также можете протестировать интеграцию фреймворка с вашей бизнес-логикой. Локальное интеграционное тестирование предоставляет средства и для тестирования базовой функциональности горизонтального масштабирования — в особенности когда речь идет о соподразделении и состоянии.

Еще одной существенной выгодой от локального интеграционного тестирования является то, что вы можете эффективно тестировать как событийно-управляемую логику, так и логику «запросов-ответов» одновременно в одинх и тех же рабочих процессах. Вы получаете полный контроль над тем, когда события записываются во входные потоки, и можете выдавать запросы в любой момент до, во время или после обработки событий. Возможно, имеет смысл подумать и об API «запрос-ответ» как о еще одном источнике событий для целей тестирования вашего микросервиса.

Давайте рассмотрим некоторые варианты, предоставляемые вам каждым компонентом системы.

◆ *Брокер событий.*

- создавать и удалять потоки событий;
- применять выборочное упорядочение событий для входных потоков с целью исполнения ориентированной на время логики, внепорядковых событий и отказов вышестоящего производителя;
- модифицировать количество разделов;
- вызывать сбои и восстановление брокера;
- вызывать сбои доступности потока событий и ее восстановление.

◆ *Регистр схем.*

- публиковать совместимые схемы для того или иного потока событий и использовать их для создания входных событий;
- вызывать сбои и восстановление.

◆ *Хранилища данных.*

- вносить изменения в схему существующих таблиц (если это применимо);
- вносить изменения в хранимые процедуры (если это применимо);
- перестраивать внутреннее состояние (если это применимо) при изменении количества экземпляров приложения;
- вызывать сбои и восстановление.

◆ *Обрабатывающий фреймворк (если это применимо).*

Приложение и обрабатывающий фреймворк обычно взаимосвязаны, и вам может потребоваться предоставить для тестирования полную реализацию фреймворка, как в случае решений на основе FaaS и «тяжеловесных» фреймворков. Фреймворк обеспечивает следующие функциональности:

- перераспределение (shuffling) данных посредством внутренних потоков событий («легковесные») или механизма перераспределения («тяжеловесные») для обеспечения их надлежащего разделения и локализации;
- копирование состояния в контрольных точках, сбои и восстановление контрольных точек;
- создание сбоев рабочего экземпляра для имитации потери экземпляра приложения («тяжеловесные» фреймворки).

◆ *Приложение.*

Управление на уровне приложений в основном предусматривает управление числом экземпляров, выполняющихся в любой момент времени. Интеграционное тестирование должно включать масштабирование количества экземпляров (динамически, если поддерживается) в целях обеспечения того, чтобы:

- перебалансировка происходила так, как и ожидалось;
- внутреннее состояние восстанавливалось из контрольных точек или потоков журнала изменений, а локальность данных сохранялась;
- доступ к внешнему состоянию не затрагивался;
- доступ по «запросу-ответу» к данным с поддержкой состояния не зависел от изменений в количестве экземпляров приложения.

Смысл полного контроля над всеми этими системами состоит в том, чтобы вы смогли убедиться, что ваш микросервис по-прежнему работает так, как задумано, в различных режимах сбоев, при неблагоприятных условиях и с различной производительностью обработки.

Существуют два главных способа выполнения локальных интеграционных тестов. Первый предусматривает встраивание библиотек тестирования, которые могут располагаться прямо в вашем коде. Они доступны не для всех решений с использованием микросервисов и, как правило, сильно зависят как от языка, так и от поддержки фреймворков. Второй вариант предполагает создание локальной среды, куда каждый необходимый компонент инсталлирован и может управляться по мере необходимости. Сначала мы рассмотрим эти варианты, а затем познакомимся с вариантами тестирования микросервисов, опирающимися на сервисы, размещенные на хосте.

Создание временной среды внутри среды выполнения тестируемого кода

Встраивание библиотек тестирования в ваш код — это, безусловно, самый сомнительный из вариантов, хотя он может работать очень хорошо в зависимости от того,

насколько совместимы ваш клиент, брокер и язык программирования фреймворка. При таком подходе тестовый код запускает необходимые компоненты из того же исполняемого файла, что и приложение.

Например, тестовый код приложения Kafka Streams запускает собственный брокер Kafka, реестр схем и экземпляры топологии микросервисов. Этот тестовый код может запускать и останавливать экземпляры топологии, публиковать события, ожидать ответов, вызывать отказы брокера и создавать другие режимы сбоев. После завершения все компоненты останавливаются и состояние очищается. Рассмотрим следующий псевдокод (объявления и экземпляры пропущены для краткости):

```
broker.start(brokerUrl, brokerPort, ...);  
schemaRegistry.start(schemaRegistryUrl, srPort, ...);  
  
// Первый экземпляр микросервиса  
topologyOne.start(brokerUrl, schemaRegistryUrl,  
    inputStreamOne, inputStreamTwo ...);  
// Второй экземпляр того же микросервиса  
topologyTwo.start(brokerUrl, schemaRegistryUrl,  
    inputStream, inputStreamTwo, ...);  
  
// Опубликовать немного тестовых данных во входной поток 1  
producer.publish(inputStreamOne, ...);  
// Опубликовать немного тестовых данных во входной поток 2  
producer.publish(inputStreamTwo, ...);  
  
// Подождать немного. Не самый лучший способ сделать это,  
// но вы понимаете, о чём идет речь  
Thread.sleep(5000);  
  
// Теперь сымитировать отказ топологии topologyOne  
topologyOne.stop();  
  
// Проверить выход из выходной темы. Все так, как ожидалось?  
event = consumer.consume(outputTopic, ...)  
  
// Выключить оставшиеся компоненты,  
// если тестирование более не требуется  
topologyTwo.stop()  
schemaRegistry.stop()  
broker.stop()  
  
if (event ...) // Проверить данные на выходе из потребителя  
    // Пройти тест, если правильно  
else  
    // Отказать в teste
```

Kafka Streams здесь представляется наиболее подходящим примером, поскольку иллюстрирует ограниченность этого подхода. Код приложения, брокер и реестр

схем Confluent основаны на JVM, поэтому вам нужно приложение на базе JVM, чтобы программно управлять всем этим в рамках одной и той же среды выполнения. Другие «тяжеловесные» фреймворки с открытым исходным кодом также могут работать на основе этого подхода, хотя для создания как ведущего экземпляра, так и рабочего, потребуются некоторые дополнительные накладные расходы. Имейте в виду, что раз эти «тяжеловесные» фреймворки также почти повсеместно базируются на JVM, на момент подготовки этой книги представленная в этом разделе стратегия в основном подходит только для JVM. Хотя, конечно, можно использовать и обходные пути для тестирования таким образом приложений, не основанных на JVM, однако этот процесс далеко не так прост.

Создание временной среды, внешней по отношению к тестируемому коду

Одним из вариантов подготовки для выполнения тестов такой среды является простое инсталлирование и конфигурирование всех необходимых систем локально. Этот подход отличается низкими накладными расходами, в особенности когда вы только начинаете работать с микросервисами, но если каждый член команды должен будет делать то же самое, да еще работать при этом с немного разными версиями, отлаживание становится дорогостоящим и сложным. Как и в большинстве случаев микросервисов, часто лучше избегать дублирования шагов и вместо этого пользоваться вспомогательными инструментами, устраниющими накладные расходы.

Более гибким вариантом является создание единого контейнера, в котором установлены и сконфигурированы все необходимые компоненты. Этим контейнером сможет воспользоваться любая команда, которая собирается протестировать свое приложение рассматриваемым способом. Вы можете при этом поддерживать модель участия на основе открытого исходного кода (даже если она является внутренней для организации), позволяющую добавлять в такой контейнер исправления, обновления и новые функциональности на благо всех команд. Эта модель достаточно гибка в применении с любым языком программирования, хотя ее гораздо

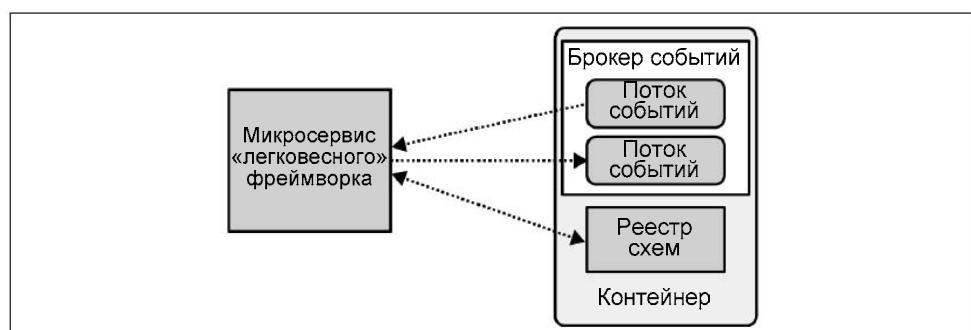


Рис. 15.1. «Легковесный» микросервис, использующий контейнеризированные зависимости для локального интеграционного тестирования

проще использовать с программным API, который позволяет легко обмениваться информацией при помощи компонентов системы, размещенных в контейнере. Пример такого «легковесного» обрабатывающего фреймворка показан на рис. 15.1, где реестр схем, брокер событий и необходимые разделы создаются внутренне по отношению к разработчику. Сам экземпляр микросервиса исполняется внешне по отношению к контейнеру и просто ссылается на адреса брокера и реестра схем из своего конфигурационного файла тестирования.

Интеграция размещенных на хосте сервисов с использованием возможностей имитации и симуляции

Локальная среда интеграционного тестирования также может нуждаться в использовании сервисов, размещенных на хосте, — таких как брокер событий, «тяжеловесный» фреймворк или платформа FaaS. Хотя некоторые размещенные на хосте сервисы могут обеспечивать функциональности на основе открытого исходного кода, которым вы можете воспользоваться вместо них (например, Kafka с открытым исходным кодом вместо Kafka на хосте), не все размещенные на хосте сервисы поддерживают такую альтернативу. Например, Microsoft Event Hubs, Google PubSub и Amazon Kinesis являются проприетарными и закрытыми, а полные их реализаций недоступны для скачивания. В такой ситуации лучшее, что вы можете сделать, — это задействовать любые эмуляторы, библиотеки или компоненты, доступные от этих компаний или инициатив с открытым исходным кодом.

Например, у Google PubSub есть эмулятор (<https://oreil.ly/pC5GC>), который может обеспечить достаточную для локального тестирования функциональность, как и версия Kinesis с открытым исходным кодом (и многие другие сервисы Amazon), предоставляемая LocalStack (<https://oreil.ly/PqA9b>). К сожалению, Microsoft Azure Event Hubs в настоящее время не имеет эмулятора, и его реализация недоступна в мире с открытым исходным кодом. Однако клиенты Azure Event Hub позволяют использовать вместо него Apache Kafka (<https://oreil.ly/Mkx2b>), хотя и не все функциональности им поддерживаются.

Для тестирования приложений на основе платформ FaaS можно задействовать локальные библиотеки тестирования, предоставляемые сервисом хостинга. Функции Google Cloud (<https://oreil.ly/sNCP4>) могут быть протестированы локально, как и функции Amazon Lambda (<https://oreil.ly/PmNyT>) и функции Microsoft Azure (<https://oreil.ly/G-MZz>). Решения с открытым исходным кодом OpenWhisk, OpenFaaS и Kubeless, описанные в главе 9, предоставляют аналогичные механизмы тестирования, которые можно найти с помощью быстрого веб-поиска. Эти варианты позволяют конфигурировать полную среду FaaS локально, благодаря чему вы сможете тестировать ее на платформе, сконфигурированной в виде, как можно более похожем на производственный.

Создание среды интеграционного тестирования для приложений, использующих «тяжеловесные» фреймворки, аналогично процессу создания среды для фреймворков FaaS. Каждый из них требует инсталляции и конфигурирования фреймворка,

при этом приложение подает задание обработки непосредственно в фреймворк. С «тяжеловесными» фреймворками типичная инсталляция в один контейнер просто должна запускать ведущий и рабочие экземпляры бок о бок вместе с брокером событий и любыми другими зависимостями. С настроенным «тяжеловесным» фреймворком вам лишь нужно будет отправить задание обработки ведущему экземпляру и дождаться результатов теста в выходных потоках событий. Пример такого подхода показан на рис. 15.2, где весь набор зависимостей контейнеризирован для удобства использования разработчиками.

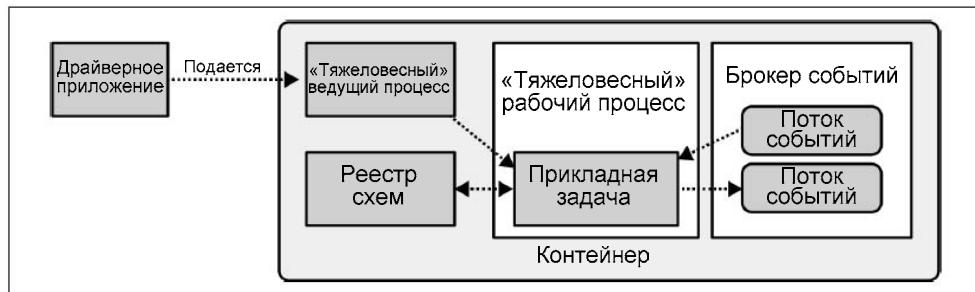


Рис. 15.2. «Тяжеловесный» микросервис, использующий контейнеризированные зависимости для локального интеграционного тестирования

Интеграция удаленных сервисов, не имеющих локальных возможностей

Некоторые используемые в производстве сервисы могут просто не иметь локально доступных возможностей, что является недостатком как для разработки, так и для интеграционного тестирования. Ближайший пример — отсутствие какого-либо эмулятора для Microsoft Azure Event Hub. Отсутствие локальной реализации означает, что для каждого разработчика приложений должны быть предусмотрены удаленные среды в дополнение к средам их интеграционного тестирования. Именно здесь линии интеграционного тестирования могут начать размываться, поскольку оно до этого момента было в основном связано с изоляцией одного экземпляра приложения в свободно располагаемой, легко управляемой локальной среде. Накладные расходы, которые вам придется понести при работе по такому сценарию, могут стать реальным препятствием для самостоятельной разработки и интеграционного тестирования ваших сервисов, поэтому, прежде чем двигаться в этом направлении, обязательно внимательно проанализируйте.

Для смягчения этой проблемы, как правило, требуется тесная координация с инфраструктурными командами, чтобы обеспечить поддержку независимых сред тестирования с помощью средств контроля доступа или создать крупную общую среду для всех (что также имеет свои собственные проблемы, обсуждаемые далее). При этом могут также возникать и вопросы с безопасностью, поскольку разработчикам придется подключать свою локальную промежуточную среду к удаленным ресурс-

сам. Очистка и управление удаленной промежуточной средой (средами) также может создавать проблемы. Решить все их можно многими способами, но проблемы, которые рассматриваемая ситуация может поднять, слишком велики, чтобы всесторонне решать их здесь.

Хорошая же новость заключается в том, что большинство крупнейших поставщиков сервисов с закрытым исходным кодом прилагают серьезные усилия, чтобы обеспечить локальные варианты разработки и тестирования, так что со временем все они станут доступными. В то же время будьте осторожны при выборе удаленных сервисов и подумайте о том, не доступен ли локальный вариант разработки и интегрированного тестирования.

Полное интеграционное тестирование

Полное интеграционное тестирование позволяет выполнять специальные тесты, которые трудно проводить в локальной среде. Например, тестирование производительности и нагрузки, необходимое, чтобы убедиться, что тестируемый микросервис соответствует заданным целям. Пропускная способность обработки событий, задержка «запросов-ответов», масштабирование экземпляра и восстановление после отказов — все это обеспечивается полным интеграционным тестированием.



Цель полного интеграционного тестирования — создать среду, максимально приближенную к рабочей, включая потоки событий, объем событийных данных, схемы событий и шаблоны запросов-ответов (если это применимо), в которой приложение будет выполняться.

Полное интеграционное тестирование обычно проводится одним из трех следующих способов: вы можете использовать временную среду интеграции и отказаться от нее после завершения тестирования, вы можете также использовать единую общую среду тестирования, которая сохраняется в промежутках между проведением интеграционных тестов и используется другими командами, и наконец, вы можете использовать для тестирования саму производственную среду.

Программное создание временной среды интеграционного тестирования

В разд. «*Создание кластеров и управление ими*» главы 14 рассматривались преимущества программного генерирования брокеров событий и менеджеров вычислительных ресурсов. Вы можете использовать эти инструменты для создания временных сред для интеграционного тестирования. Например, можно создать отдельный набор брокеров вместе с индивидуально зарезервированными вычислительными ресурсами и запускать тестируемые микросервисы в контейнерах. Дополнительным преимуществом использования этого подхода для полного интеграционного тестирования является то, что он регулярно воспроизводит процесс создания новых брокеров и вычислительных сред. Это гарантирует, что любые сбои, возникающие в скриптах, или любые ошибки в конфигурации будут выявлены при следующем интеграционном teste.

Другая проблема вновь созданной среды заключается в том, что в ней отсутствуют как потоки событий, так и данные о событиях. А для тестирования вашего микросервиса необходимы оба эти компонента. Вы можете получить имена создаваемых потоков событий, напрямую запросив пользователя или используя файл конфигурации в кодовой базе микросервиса, доступный с помощью соответствующего инструментария. Количество разделов также должно соответствовать количеству разделов производственной системы, чтобы гарантировать правильное выполнение логики масштабирования, соподразделения и переподразделения микросервиса.

Когда потоки событий будут созданы, на следующем шаге их надо заполнить событиями. Это может быть сделано с использованием реальных производственных данных, специально подобранных для тестирования наборов данных или данных, программно сгенерированных специально для тестирования на основе выбранной из реестра схемы.

Заполнение событиями с использованием реальных производственных данных

События могут быть скопированы из производственного кластера во вновь созданные потоки событий в тестовом кластере. Именно здесь вступает в игру инструмент репликации, описанный в разд. «МежклUSTERНАЯ РЕПЛИКАЦИЯ СОБЫТИЙНЫХ ДАННЫХ» главы 14, поскольку его можно использовать для репликации определенных потоков событий из производственного кластера и загрузки их в тестовый. При этом вы должны учитывать любые ограничения безопасности и доступа, которые могут помешать производству получать данные.

◆ *Преимущества этой стратегии:*

- она точно отражает производственные данные;
- вы можете скопировать столько событий, сколько потребуется;
- полностью изолированная среда предотвращает непреднамеренное влияние других тестируемых микросервисов на ваше тестирование.

◆ *Ее недостатки:*

- копирование данных может повлиять на производительность производства, если вы должным образом не запланировали и не установили брокерские квоты;
- она может потребовать копирования значительных объемов данных, в особенности в случае долгоживущих сущностей;
- вы должны учитывать, что некоторые потоки событий могут содержать конфиденциальную информацию;
- она требует значительных инвестиций в оптимизацию процесса создания и копирования, чтобы уменьшить препятствия для использования данных;
- она может открыть внешнему миру важные производственные события.

Заполнение событиями из подготовленного источника тестирования

Специально подготовленные для тестирования наборы данных позволяют использовать в интеграционном тестировании события с конкретными свойствами, значениями и отношениями с другими событиями. Эти события должны храниться в стабильном и безопасном месте, где они не могут быть случайно или непреднамеренно перезаписаны, искажены или потеряны. Такая стратегия часто используется в *единых общих средах тестирования* (подробнее об этом позже), но вы также можете применять ее и здесь, загрузив подготовленные для тестирования события из надежного хранилища данных в заданные пользователем потоки, аналогично только что рассмотренному копированию событий из производственной среды.

◆ Преимущества этой стратегии:

- она предусматривает меньший набор данных;
- она тщательно подготавливается, чтобы обеспечить конкретные значения и связи событий;
- она никак не влияет на производство.

◆ Ее недостатки:

- присутствуют значительные накладные расходы на обслуживание;
- данные могут стать устаревшими;
- необходимо обрабатывать новые потоки событий;
- необходимо учитывать изменения схемы;
- не часто используемые потоки событий могут оказаться недоступны.

Хотя многие из этих недостатков можно уменьшить с помощью строгого подбора данных для тестирования, с этими данными часто случается то же самое, что происходит во многих организациях с хранящейся у них документацией. Каковыми были быни были благие намерения, но подобранные данные все равно быстро устаревают, перестают быть актуальными, да и сама работа по подбору данных для тестирования часто имеет у персонала более низкий приоритет, чем любая другая.

Создание имитационных событий с использованием схем

Программное создание имитационных событий — это еще одна возможность для заполнения потоков событий. Вы можете получить схему из реестра схем и сгенерировать события, соответствующие ее определению. Вы даже можете взять более старые версии этой схемы и сгенерировать события для них тоже.

Сложность этого подхода заключается в обеспечении наличия событий с надлежащими отношениями с другими событиями, в особенности если какой-либо из сервисов выполняет соединения между потоками или агрегации между разными типами событий. Микросервис, который соединяет несколько событий вместе, потребует создания событий с совпадающими первичными/внешними ключами для правиль-

ного исполнения логики соединения, присущей сервису. Хотя это обычно не является существенной проблемой (в особенности потому, что код микросервиса выражает, *какие* связи требуются бизнес-логике), такой подход все-таки оставляет на усмотрение создателей данных для тестирования необходимость обеспечить, чтобы все они были надлежащим образом определены и попадали в ожидаемые диапазоны и значения.

◆ *Преимущества этой стратегии:*

- она не требует от производственного кластера предоставления каких-либо данных и не может негативно повлиять на производительность системы;
- вы можете использовать инструменты фаззинга² для создания событийных данных, тестирования граничных условий, а также и других потенциальных плохо сформулированных и наполовину сформулированных полей данных;
- вы можете создавать конкретные тестовые случаи, которые недоступны в производственных данных, обеспечивая тем самым охват граничных случаев;
- она позволяет использовать сторонние инструменты для программного создания тестовых данных — например, инструменты Confluent Avro (<https://oreil.ly/HTQhX>).

◆ *Ее недостатки:*

- она требует гораздо большего внимания к созданию реалистичных данных, чем другие варианты;
- распределение созданных данных все-таки не является совершенно точным по сравнению с производственным распределением. Например, производственные данные могут иметь серьезное несоответствие в объеме данных из-за распределения ключей, которое не отображается в имитационных данных;
- созданные данные могут неточно представлять некоторые поля. Например, разбор тем или иным способом строкового поля для бизнес-операций может свободно пройти с созданными тестовыми данными, но не сработать в подмножестве производственных данных.

Тестирование с использованием единой общей среды

Еще один вариант предусматривает создание единой общей среды тестирования с совместным пулом потоков событий, находящихся в одном и том же брокере событий. Эти потоки заполняются тестовыми данными, представляющими подмножество производственных данных, или тщательно обработанными тестовыми данными, как обсуждалось в предыдущем разделе. Этот вариант обеспечивает низкие накладные расходы на среду тестирования, но перекладывает управление потоками событий и данными на разработчиков приложений.

² Фаззинг (fuzzing) — техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных. — Прим. ред.

◆ *Преимущества этой стратегии:*

- ее легко начать;
- вам нужно поддерживать инфраструктуру только для одной тестовой среды;
- она изолирована от производственных нагрузок.

◆ *Ее недостатки:*

- она подвержена «трагедии общих ресурсов»: фрагментированные и заброшенные потоки событий могут затруднить понимание того, какие потоки являются пригодными для тестирования входных данных, а какие — просто выходными данными предыдущих тестов, которые не были своевременно очищены;
- тестируемые системы не обязательно изолированы. Например, сервисы, выполняющие одновременное крупномасштабное тестирование производительности, могут влиять на результаты друг друга;
- во входных потоках событий других сервисов могут создаваться несовместимые события;
- данные потока событий неизбежно устаревают и должны обновляться новыми событиями;
- она неточно представляет весь спектр событий, которые можно найти в производстве.



Эта стратегия является худшим вариантом с точки зрения удобства использования, поскольку брокер событий в конечном счете становится свалкой запутанных потоков событий и испорченных данных.

Изоляция от тестирования других приложений труднодостижима, в особенности потому, что выходной поток одного микросервиса обычно является входом для другого. Тщательная обработка потоков данных, строгие соглашения об именах и ограничения на запись в потоки событий помогают смягчить недостатки этого варианта, но специалисты по обслуживанию среды и ее пользователи должны проявлять в работе с ней особые усердие и дисциплину.

Тестирование с использованием производственной среды

Вы также можете протестировать микросервисы в производственной среде (примечание: будьте осторожны). Микросервис при этом может функционировать, потребляя входные потоки событий, применять бизнес-логику и производить выходные данные. Наиболее распространенный подход заключается в том, чтобы микросервис использовал свои собственные назначенные выходные потоки событий и хранилища состояний так, чтобы это не влияло на существующие производственные системы. Это особенно важно, когда предыдущая версия того же микросервиса работает вместе с новой, тестируемой.

◆ *Преимущества этой стратегии:*

- у вас есть полный доступ к производственным событиям;
- она задействует производственные модели обеспечения безопасности для поддержки соблюдения надлежащих протоколов доступа;
- она отлично подходит для тестирования приложения «на дымность»;
- вам не нужно поддерживать отдельную среду тестирования.

◆ *Ее недостатки:*

- она не подходит для тестирования нагрузки и производительности. Кроме того, существует риск влияния на производственные мощности, в особенности если рабочая нагрузка высока;
- вы должны тщательно очистить все ресурсы, созданные во время тестирования, — такие как потоки событий, группы потребителей, разрешения контроля доступа и хранилища состояний. Это похоже на требования к распространенному варианту с промежуточной средой;
- требуется инструментальная поддержка, чтобы тестируемые микросервисы и потоки событий были отделены от «истинных производственных» микросервисов, в особенности когда вы производите тестирование в течение длительного периода времени. Сюда входят ресурсы, используемые для управления и развертывания микросервисов, поскольку каждый наблюдатель в производственной среде должен быть в состоянии выявлять то, какие сервисы являются истинными производственными, а какие — тестируемыми.

Выбор стратегии полного интеграционного тестирования

Преимущество модульности микросервисов заключается в том, что вам не приходится выбирать только один способ выполнения тестов. Вы можете использовать любой вариант, при необходимости переключаясь на другой для других проектов, и обновлять свою методологию тестирования по мере изменения требований. Во многом ваши варианты тестирования определят также инвестиции во вспомогательные инструменты для брокеров мультиклusterных событий и возможности копирования событий.

Если у вас практически нет вспомогательных инструментов, вы, вероятно, в конечном итоге получите один общий брокер событий тестирования с мешаниной потоков событий, генерируемых различными командами и системами. Скорее всего, вы увидите смесь «хороших» потоков событий, которые можно использовать для тестирования, и потоков событий с суффиксами типа «-testing-01», «-testing-02», «-testing-02-final» и «-testing-02-final-v2». Данные о событиях в них могут быть или не быть надежными, актуальными или иметь правильный формат схемы. В этом мире большую роль играют «племенные знания», и бывает трудно обеспечить, что-

бы тестирование в достаточной степени отражало производственную среду вашего сервиса. Кроме того, в этом варианте гораздо выше затраты на постоянно доступный промежуточный кластер, который также должен обеспечивать тестирование производительности, загружать большие объемы данных и предоставлять хранилища событий с неопределенной продолжительностью хранения.

Однако при надлежащих инвестициях в инструменты каждый микросервис сможет создать свой собственный выделенный кластер, заполнить его потоками событий, скопировать в него некоторые производственные данные и запустить тест в почти идентичной производственной среде. После завершения тестирования этот кластер можно отключить, что устранит артефакты тестирования, которые в противном случае остались бы в общем кластере. Накладные расходы на переход к этому варианту значительны, но инвестиции откроют возможности для работы с несколькими кластерами, избыточности и методам аварийного восстановления, которые трудно получить иным способом (подробнее об этом см. в главе 14).

Это не значит, что один совместный тестовый кластер изначально плох. Важно тщательно отмечать чистые и надежные исходные потоки, а также удалять неиспользуемые артефакты тестирования. Конкретные, систематизированные обязанности могут гарантировать, что ожидания относительно надежности промежуточных событийных данных будут подтверждены командами, владеющими данными о производственных событиях. Команды также должны координировать производительность и нагружочное тестирование с тем, чтобы они не влияли на результаты друг друга. По мере того, как ваша команда будет совершенствовать свои инструменты многокластерного копирования событий, другие команды смогут начать миграцию на свои собственные динамически созданные кластеры тестирования.

Резюме

Событийно-управляемые микросервисы в основном получают входные данные из потоков событий. Вы можете создавать и заполнять эти потоки различными способами, включая копирование данных из производственной среды, создание специально подобранных для тестирования наборов данных и автоматическое генерирование событий для тестирования на основе схемы. Каждый метод имеет свои преимущества и недостатки, но все они опираются на вспомогательные инструменты для создания, заполнения и управления этими потоками событий.

Создание среды для тестирования микросервиса должно обеспечиваться общими усилиями. Другие разработчики и инженеры в вашей организации, несомненно, выиграют от общей платформы тестирования, поэтому вам следует подумать об инвестициях в инструменты для оптимизации процессов тестирования. Программная доводка сред, включая заполнение потоков событий, может значительно снизить накладные расходы на настройку среды для каждого тестируемого микросервиса.

Единая общая среда тестирования — это часто встречающаяся стратегия, которую можно использовать, когда инвестиции в инструменты невелики. Компромисс

в этом случае заключается в повышенной сложности управления событийными данными, обеспечении достоверности и уточнении прав владения. Среды на собственном выделенном кластере являются предпочтительной альтернативой варианту общей среды, поскольку обеспечивают изоляцию тестируемых сервисов и снижают риски и недостатки, вызванные многочисленными проблемами аренды. Но они, как правило, требуют больших вложений в общие вспомогательные инструменты, но в долгосрочной перспективе значительно экономят время и усилия. В качестве дополнительного преимущества использование программных средств доводки среды и копирования событий может лучше подготовить вашу организацию к аварийному восстановлению.

Развертывание событийно-управляемых микросервисов

Развертывание событийно-управляемых микросервисов может оказаться сложной задачей. По мере увеличения в организации количества микросервисов возрастает и важность наличия у нее стандартизованных процессов их развертывания. Организация, управляющая всего несколькими десятками микросервисов, может обойтись несколькими индивидуально настраиваемыми процессами развертывания, но любая организация, серьезно развивающая микросервисы, событийно-управляемые или иные, должна инвестировать в стандартизацию и оптимизацию процессов развертывания.

Принципы развертывания микросервисов

Существует ряд принципов управления процессами развертывания:

- ◆ *Предоставлять командам развертывания самостоятельность.*

Команды должны сами управлять своими процессами тестирования и развертывания и иметь право самостоятельно развертывать свои микросервисы по собственному усмотрению.

- ◆ *Реализовывать стандартизованный процесс развертывания.*

Процесс развертывания должен быть согласован между сервисами. Создавать новый микросервис следует, уже имея доступный для него процесс развертывания. Обычно это достигается с помощью фреймворка *непрерывной интеграции*, о чем мы вскоре поговорим.

- ◆ *Использовать необходимые вспомогательные инструменты.*

Развертывание может потребовать от команд, чтобы они сбрасывали смещения групп потребителей, очищали хранилища состояний, проверяли и обновляли версию схемы и удаляли внутренние потоки событий. Вспомогательные инструменты предоставляют эти функции, обеспечивая дальнейшую автоматизацию развертывания и самостоятельность команды поддержки.

- ◆ *Учитывать возможность повторной обработки потока событий.*

Повторное потребление входных потоков событий развертываемым микросервисом может занять много времени, что приведет к устареванию результатов для последующих потребителей. Кроме того, он может при этом генерировать большой объем выходных событий, вызывая дополнительную высокую нагрузку для последующих потребителей. Очень большие потоки событий и потоки

с большим количеством потребителей могут столкнуться с нетривиальными скачками требований к вычислительной мощности. Вы также должны учитывать побочные эффекты, особенно те, которые могут помешать клиентам (например, повторную отправку рекламных писем за несколько лет).

◆ *Соблюдать соглашения об уровне обслуживания (Service Level Agreements, SLA).*

Развертывание может нарушить работу других сервисов. Например, перестраивание хранилищ состояний способно привести к значительному времени простоя, а повторная обработка входных потоков событий вызвать значительное количество событий. Убедитесь, что в процессе развертывания соблюдаются все SLA.

◆ *Минимизировать зависимые изменения сервисов.*

Развертывание может потребовать, чтобы другие сервисы изменили свои API или модели данных — например, при взаимодействии с REST API или внесении изменений в схему домена. Такие изменения должны быть минимизированы, насколько это возможно, поскольку они нарушают самостоятельность других команд по развертыванию их сервисов, и допускать их следует только тогда, когда это требуется из-за изменения бизнес-требований.

◆ *Обсуждать критические изменения с нижестоящими потребителями.*

В некоторых случаях серьезные изменения схемы взаимодействия бывают неизбежными и требуют создания новых потоков событий и пересмотра контрактов на передачу данных с нижестоящими потребителями. Убедитесь, что обсуждение этих вопросов проводится перед любым развертыванием и что для потребителей существует план миграции.



Микросервисы должны развертываться независимо, и если это не так, то перед нами пример неправильной организации работы. Если развертывание какого-либо микросервиса требует, чтобы другие микросервисы синхронизировали с ним свои развертывания, вы можете быть уверены, что их ограниченные контексты плохо определены и должны быть пересмотрены.

Архитектурные компоненты развертывания микросервисов

Архитектура развертывания микросервисов состоит из нескольких основных элементов, каждый из которых играет ключевую роль. Эту архитектуру можно условно разбить на два основных компонента: систему, служащую для построения и развертывания кода, и вычислительные ресурсы, используемые микросервисами.

Системы непрерывной интеграции, доставки и развертывания

Системы непрерывной интеграции, доставки и развертывания позволяют создавать, тестировать и развертывать микросервисы по мере внесения изменений кода в хра-

нилище. Использование их — это часть «налога на микросервисы», который вы должны платить, чтобы успешно управлять микросервисами и их масштабным развертыванием. Эти системы позволяют владельцам микросервисов решать, когда развертывать свои микросервисы, что важно для увеличения количества микросервисов, используемых в организации.

- ◆ *Непрерывная интеграция* (Continuous Integration, CI) — это способ автоматизации интеграции изменений кода от нескольких участников в один проект программного обеспечения. Изменения кода интегрируются по усмотрению команды, управляющей микросервисом, с целью сокращения времени между внесением изменений в код и их развертыванием в производственной среде. Фреймворки CI позволяют автоматически выполнять процессы при объединении кода в главную ветвь, включая операции сборки, модульное тестирование и операции интеграционного тестирования. Другие процессы CI могут выполнять проверку стиля кода и версии схемы. Готовый к развертыванию контейнер или виртуальная машина — это конечный результат конвейера CI.
- ◆ *Непрерывная доставка* (Continuous delivery) — это способ обеспечения возможности развертывания вашей кодовой базы. Микросервисы, которые придерживаются принципов непрерывной доставки, используются для проверки готовности сборки к развертыванию *конвейером CI*. Однако само развертывание *не* автоматизировано и требует некоторого ручного вмешательства со стороны владельца сервиса.
- ◆ *Непрерывное развертывание* (Continuous deployment) — это автоматическое развертывание сборки. При сквозном непрерывном развертывании зафиксированное изменение кода распространяется по конвейеру CI, достигает состояния доставки и автоматически развертывается в производственной среде в соответствии с конфигурацией развертывания (рис. 16.1). Это способствует созданию плотного цикла разработки с коротким сроком выполнения, поскольку зафиксированные изменения быстро поступают в производство.

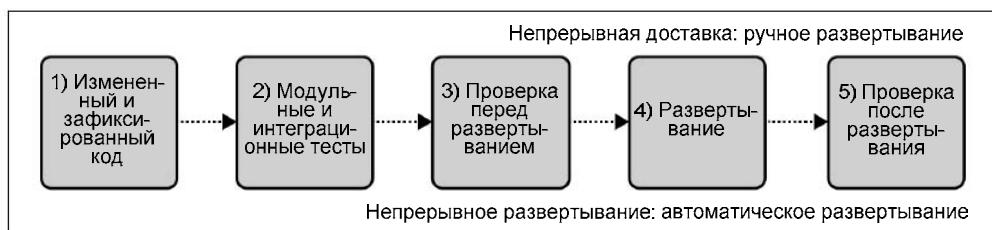


Рис. 16.1. Конвейер CI, демонстрирующий разницу между непрерывной доставкой и непрерывным развертыванием



Непрерывное развертывание трудно осуществлять на практике. Сервисы с поддержкой состояния особенно сложны, поскольку развертывание может потребовать перестройки хранилищ состояний и повторной обработки потоков событий, что особенно мешает работе зависимых сервисов.

Системы управления контейнерами и стандартное аппаратное обеспечение

Система управления контейнерами (Container Management System, CMS) предоставляет средства управления, развертывания и контроля использования ресурсов контейнеризированных приложений (см. разд. «Управление контейнерами и виртуальными машинами» главы 2). Контейнер, созданный в процессе CI, помещается в репозиторий, где он ожидает от CMS инструкций по развертыванию. Интеграция между конвейером CI и CMS важна для упрощенного процесса развертывания и обычно обеспечивается всеми ведущими поставщиками CMS, как и показано в главе 2.

Для развертывания событийно-управляемых микросервисов обычно используется стандартное оборудование, поскольку оно недорогое, надежно работает и обеспечивает горизонтальное масштабирование сервисов. Вы можете добавлять оборудование в пулы ресурсов и удалять из них по мере необходимости — при этом восстановление отказавших экземпляров микросервисов потребует лишь повторного развертывания их на новом оборудовании. Хотя ваши реализации микросервисов могут различаться, многие событийно-управляемые микросервисы не требуют для работы какого-либо специального оборудования. Для тех же, кто в этом нуждается, вы можете выделить специализированные ресурсы в их собственные независимые пулы, чтобы связанные микросервисы могли быть развернуты соответствующим образом. Примерами могут служить вычислительные экземпляры с интенсивным использованием памяти для целей кэширования или вычислительные экземпляры с интенсивным использованием процессора для приложений, требующих значительной вычислительной мощности.

Базовый шаблон полного развертывания

Базовый шаблон полного развертывания (с полной остановкой) является основой всех остальных шаблонов, и в этом разделе описаны его соответствующие шаги (см. рис. 16.1). В вашем конвейере могут присутствовать и дополнительные шаги — в зависимости от конкретных требований вашей предметной области, но для экономии места я здесь представлю вам именно эти шаги. Вы же, разумеется, можете ориентироваться на свои собственные суждения и знание предметной области, чтобы добавить в шаблон любые шаги, специфичные для ваших вариантов использования.

- 1. Зафиксировать код** — объединить новый код с основной веткой, запустив конвейер CI. Специфика зависит от вашего репозитория и конвейера CI, но обычно это делается с помощью обработчиков фиксации, способных выполнять произвольную логику, когда код фиксируется в репозитории.
- 2. Выполнить автоматизированные модульные и интеграционные тесты.** Этот шаг является частью конвейера CI — здесь принятый код проходит все модульные и интеграционные тесты, необходимые после объединения нового кода с основной веткой. Интеграционные тесты могут потребовать развертывания временных сред и заполнения их данными для выполнения более сложных тес-

тов. В таком случае понадобится интеграция конвейера CI с инструментами, описанными в разд. «Локальное интеграционное тестирование» главы 15, чтобы каждый сервис мог запускать свою собственную среду интеграционного тестирования.



Лучше всего иметь независимые среды интеграционного тестирования для любого вида автоматизированного тестирования, что позволяет запускать тесты для каждого сервиса изолированно от других сервисов. Это значительно сокращает количество проблем, связанных с мультитенантностью (см. главу 11), которые возникают из-за наличия длительной и общей тестовой среды интеграции.

3. Выполнить проверочные тесты перед развертыванием. Этот шаг гарантирует правильное развертывание микросервиса, проверяя наличие у него известных проблем. Проверка может включать в себя:

- *проверку потока событий* — здесь проверяется, что входные потоки событий существуют, выходные потоки событий также существуют (либо могут создаваться, если активировано автоматическое создание), и ваш микросервис имеет соответствующие разрешения на чтение/запись для доступа к ним;
- *проверку схемы* — здесь проверяется, соблюдаются ли входные и выходные схемы правила развития схем. Для этого существует простой традиционный способ: если ваши схемы ввода и вывода содержатся в структуре каталогов вместе с картой схем для потоков событий, то на этом шаге конвейера можно просто принять схемы и выполнить сравнение, обнаруживая любую несоставимость.

4. Развернуть микросервис. Текущий микросервис должен быть остановлен до того, как будет развернут новый. Этот процесс состоит из двух основных этапов:

- **остановить экземпляры и выполнить полную очистку перед развертыванием.**

Остановите экземпляры микросервиса. Выполните все необходимые сбросы хранилища состояний и/или сбросы групп потребителей и удалите все внутренние потоки. Если восстановление состояния в случае сбоя развертывания обходится дорого, вы можете оставить свое состояние, группу потребителей и внутренние темы в покое и вместо этого развернуть их как новую службу. Это позволит вам быстро откатиться в случае сбоя.

- **развернуть.**

Выполните фактическое развертывание. Разверните контейнерный код и запустите необходимое количество экземпляров микросервиса. Подождите, пока они загрузятся и дадут сигнал, что готовы, прежде чем переходить к следующему шагу. В случае сбоя откажитесь от этого шага и разверните предыдущую рабочую версию кода.

5. Выполнить проверочные тесты после развертывания. Убедитесь, что микросервис работает нормально, что задержка потребителя возвращается в нормальное состояние, что нет ошибок журнализации и что конечные точки работают так, как ожидалось.



Учитывайте влияние на все зависимые службы, включая SLA, время простоя, время наверстывания обработки потоковой обработки, загрузку выходных событий, новые потоки событий и критические изменения схемы. Общайтесь с зависимыми владельцами сервисов, чтобы убедиться, что воздействия приемлемы.

Шаблон непрерывного обновления

Шаблон непрерывного обновления можно использовать для поддержания работы сервиса при обновлении отдельных экземпляров микросервиса. Его предварительные условия таковы:

- ◆ никаких критических изменений в каких-либо хранилищах состояний;
- ◆ никаких критических изменений во внутренней топологии микросервисов (это особенно актуально для реализаций с использованием «тяжеловесных» фреймворков);
- ◆ никаких критических изменений во внутренних схемах событий.

До тех пор, пока соблюдаются указанные предварительные условия, этот шаблон развертывания хорошо работает в следующих сценариях:

- ◆ к событиям ввода были добавлены новые поля, которые необходимо отразить в бизнес-логике;
- ◆ должны потребляться новые входные потоки;
- ◆ необходимо исправить ошибки, но повторная обработка не потребуется.



Непреднамеренное изменение внутренней топологии микросервисов — одна из наиболее распространенных ошибок, которые люди допускают при использовании этого шаблона развертывания. Такое изменение является критическим и вместо непрерывного обновления потребует полного сброса приложения.

При использовании шаблона непрерывного обновления изменяется только шаг 4 из разд. «*Базовый шаблон полного развертывания*», рассмотренного ранее. Вместо того чтобы останавливать все экземпляры одновременно, производится остановка только одного экземпляра. Остановленный экземпляр затем обновляется и запускается снова, в результате чего в процессе такого развертывания получается смесь новых и старых экземпляров. Непрерывное обновление означает, что в течение короткого периода времени как старая, так и новая логика будут работать одновременно.



Интеллектуальные реализации будут запускать тест, проверяющий совместимость выпуска, чтобы уведомить вас, действительно ли непрерывное обновление. Делать такой тест вручную весьма рискованно, и этого следует избегать.

Главное преимущество этого шаблона заключается в том, что сервисы могут обновляться, а обработка данных в режиме, близком к реальному времени, продолжается непрерывно без какого-либо простоя. Главным недостатком этого шаблона

являются его предварительные условия, ограничивающие его использование конкретными сценариями.

Шаблон разрушительных изменений схемы

Разрушительное изменение схемы иногда бывает неизбежным (см. разд. «Разрушительное изменение схемы» главы 3). Развертывание, влекущее за собой разрушительное изменение схемы, должно учитывать ряд зависимостей, включая обязанности как потребителя, так и производителя, координацию усилий по миграции и время простоя при переработке.

Технически процесс развертывания разрушительного изменения схемы весьма прост. Трудная его часть заключается в пересмотре определения схемы, доведении его до сведения заинтересованных сторон и координации с ними планов развертывания и миграции. Каждый из этих шагов требует обмена четкой информацией между сторонами и строго определенных сроков действий.

Последствия разрушительного изменения схемы различаются в зависимости от типа события. Разрушительные изменения схемы *сущностей* сложнее, чем изменения *несущностных событий*, поскольку сущности требуют согласованного определения для материализации потребителя. Сущности также по определению являются *постоянными единицами* данных, которые будут потребляться повторно всякий раз, когда потребитель рематериализует поток сущностей из своего источника. Потоки сущностей должны воссоздаваться в соответствии с новой схемой, включающей как новую бизнес-логику, так и определения схемы.

Воссоздание сущностей для нового потока потребует переработки необходимых исходных данных для производителя, будь то пакетный источник или его собственные входные потоки событий. Эта логика может быть встроена в того же производителя, либо новый производитель может быть создан и развернут рядом с ним. Первый вариант сохраняет всю логику инкапсулированной в пределах своего собственного сервиса, в то время как второй вариант позволяет первоначальному производителю продолжать свою работу непрерывно, уменьшая воздействие на нижестоящих потребителей. Эти варианты показаны на рис. 16.2.

Разрушительные изменения схемы часто отражают фундаментальный сдвиг в сфере деятельности для сущности или события. Обычно это происходит не слишком часто, но когда все же происходит, изменения обычно оказываются достаточно значительными, и потребителям приходится обновлять информацию, чтобы отразить изменившееся бизнес-значение предметной области.

С другой стороны, разрушительные изменения для *несущностных событий* могут не требовать переработки. Это главным образом связано с тем, что многие приложения потоковой передачи событий перерабатывают потоки событий не регулярно, а с той периодичностью, какая нужна сервису для рематериализации его потоков сущностей. Потребители часто могут просто добавлять определение нового события в качестве нового потока событий и изменять свою бизнес-логику для обработки как старых, так и новых событий. По истечении срока действия старых событий старый поток событий может просто удаляться из бизнес-логики.

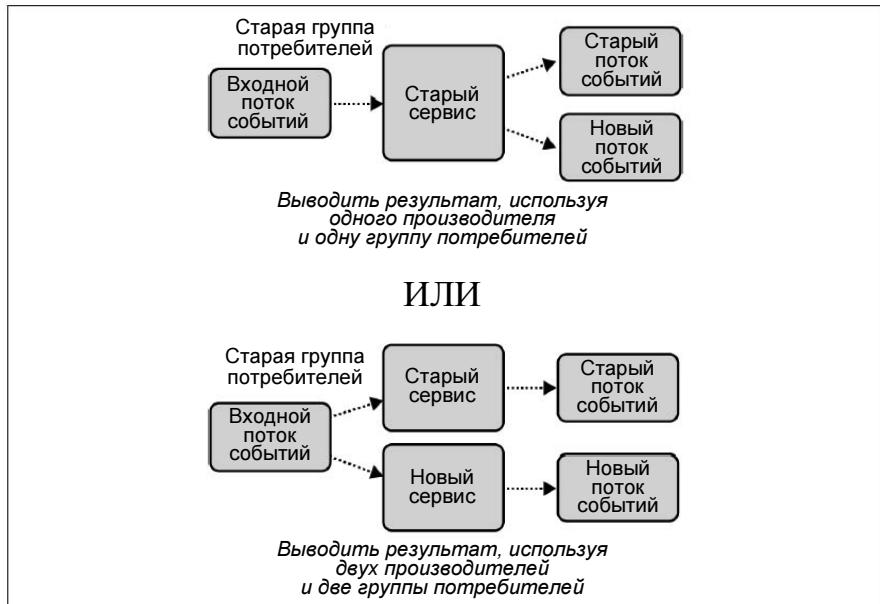


Рис. 16.2. Варианты действий по воссозданию потоков событий с новой схемой при разрушительном изменении старой

Существуют два главных варианта миграции разрушительного изменения схемы:

- ◆ продолженная миграция через два потока событий: один со старой схемой и другой с новой схемой;
- ◆ синхронизированная миграция в один новый поток с удалением старого.

Продолженная миграция через два потока событий

Продолженная (eventual) миграция через два потока событий требует, чтобы производитель записывал события как в старом, так и в новом формате — каждое в свой соответствующий поток. Старый поток помечается как устаревший, и его потребители в свое время мигрируют в новый поток. После миграции всех потребителей старый поток можно удалить или выгрузить в долговременное хранилище данных.

Эта стратегия предполагает ряд допущений:

- ◆ **события могут создаваться как для старого потока, так и для нового.**

Производитель должен иметь необходимые данные для создания событий как старого, так и нового формата. Предметная область созданного события изменится весьма значительно, чтобы потребовать разрушительного изменения, оставаясь при этом такой, чтобы сопоставление 1:1 старого формата события с новым все еще имело смысл. Возможно, так не будет происходить со всеми разрушительными изменениями схемы;

- ◆ **продолженная миграция не вызовет несогласованности в нижестоящих потоках.**

Нижестоящие сервисы будут по-прежнему потреблять два разных определения, не имея при этом никаких побочных эффектов, или эти эффекты окажутся невелики. Тем не менее разрушительное изменение схемы предполагает, что предметная область изменена в такой степени, что переопределение стало для организации необходимым. Редко бывает так, что разрушительные изменения были необходимы для бизнеса, но оказались в значительной степени несущественными для потребителей, которые используют события.



Один из главных рисков продолженной миграции заключается в том, что миграция никогда не заканчивается, и схожие, но все же разные, потоки данных остаются в использовании бесконечно. Вдобавок, новые сервисы, созданные в процессе такой миграции, могут непреднамеренно регистрировать себя в качестве потребителей в старом потоке вместо нового. Используйте тегирование метаданными (см. разд. «Разметка потока событий метаданными» главы 14), чтобы помечать потоки как устаревшие и поддерживать размер окон миграции малыми.

Синхронизированная миграция в новый поток событий

Еще один вариант состоит в обновлении производителя, чтобы он начал создавать события строго в новом формате и перестал предоставлять обновления старому потоку. Этот вариант проще — с технической точки зрения, — чем поддержание двух потоков событий, но он требует более интенсивного обмена информацией между производителем и потребителями данных. Потребители должны обновлять свои определения, чтобы учитывать разрушительные изменения, вносимые производителем.

Эта стратегия также предполагает ряд допущений:

- ◆ **изменение определения события оказывается достаточно существенным, чтобы старый формат больше не использовался.**
Предметная область сущности или события изменилась настолько, что старый и новый формат не могут поддерживаться одновременно;
- ◆ **миграция должна происходить синхронно, чтобы не вызвать несоответствия в исходящем направлении.**

Предметная область изменилась настолько значительно, что сервисы нуждаются в обновлении, чтобы обеспечить соответствие бизнес-требованиям. В противном случае у нижестоящих сервисов могут возникнуть серьезные несогласованности. Например, подумайте о сущности, в которой изменились критерии отбора для создания события.

Самый большой риск этого варианта развертывания заключается в том, что потребители могут потерпеть неудачу при миграции на новый поток событий, но не смогут корректно вернуться к старому источнику данных, как это было бы при использовании стратегии продолженной миграции. Интеграционное тестирование (предпочтительно на основе программно созданных сред и исходных данных) может снизить этот риск, предоставив среду, в которой можно полностью выполнить процесс миграции. В тестовой среде вы сможете создать и зарегистрировать и производителя, и потребителей, и полностью проверить в ней миграцию перед ее выполнением в производственной среде.



Синхронизированные миграции, как правило, встречаются на практике редко, поскольку они требуют значительных разрушительных изменений или даже разрушения предыдущей модели предметной области для события. Основные бизнес-сущности обычно имеют очень стабильные модели предметной области, но когда происходят серьезные разрушительные изменения, синхронная миграция может стать неизбежной.

Шаблон «сине-зеленого» развертывания

Главная цель «сине-зеленого» развертывания состоит в обеспечении нулевого времени простоя при развертывании новой функциональности. Этот шаблон в основном используется в развертываниях микросервисов синхронных «запросов-ответов», поскольку он позволяет синхронным запросам продолжаться во время обновления сервиса. Пример этого шаблона показан на рис. 16.3.

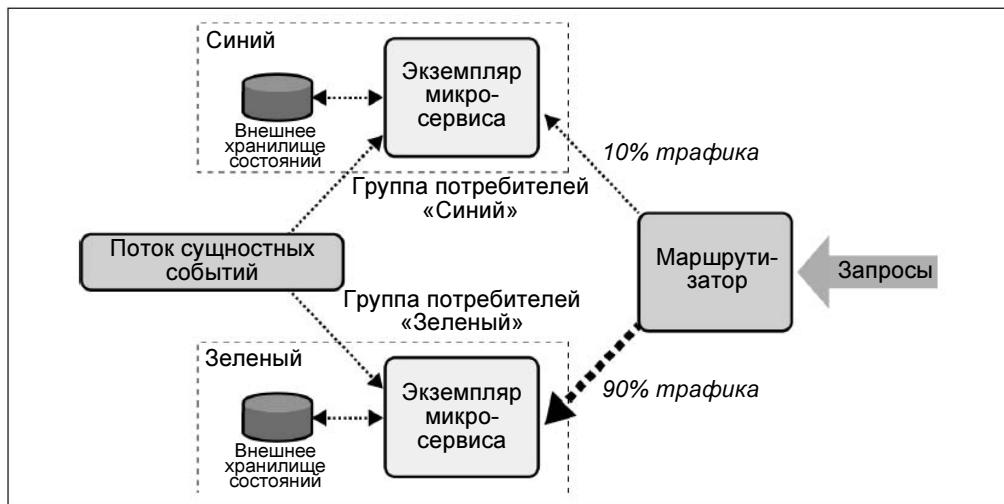


Рис. 16.3. Шаблон «сине-зеленого» развертывания

Как можно видеть, здесь полная копия нового микросервиса (**Синий**) создается параллельно со старым микросервисом (**Зеленый**). «Синий» сервис имеет полностью изолированный экземпляр внешнего хранилища данных, собственную группу потребителей потока событий и собственные IP-адреса для удаленного доступа. Он потребляет входные события до тех пор, пока мониторинг не покажет, что он в достаточной степени догнал «зеленый» сервис, после чего трафик из «зеленых» экземпляров может начать перенаправляться на него.

Переключение трафика выполняется маршрутизатором, расположенным перед сервисами. На новый — «синий» — экземпляр может сначала перенаправляться небольшой объем трафика, что позволит проверить развертывание в реальном времени. Если процесс развертывания не обнаружит сбоев или отклонений, все больше и больше трафика будет перенаправляться на «синий» экземпляр — до тех пор, пока «зеленая» сторона вообще не перестанет получать трафик.

На этом этапе, в зависимости от чувствительности вашего приложения и необходимости обеспечения быстрого отката, «зеленые» экземпляры могут быть немедленно отключены или оставлены в бездействии до тех пор, пока не пройдет достаточно время без инцидентов. В случае же возникновения во время этого переходного периода ошибки маршрутизатор сможет быстро перенаправить трафик обратно к «зеленым» экземплярам.



Мониторинг и оповещение, включая показатели использования ресурсов, задержку группы потребителей, триггеры автоматического масштабирования и системные оповещения, должны быть интегрированы в систему в качестве составной части процесса «переключения цвета».

«Сине-зеленые» развертывания вполне пригодны для микросервисов, которые потребляют данные из потоков событий. Они также могут хорошо работать тогда, когда события создаются *только* благодаря активности запросов-ответов, — например, когда запрос конвертируется непосредственно в событие (см. разд. «Обработка запросов внутри событийно-управляемого рабочего процесса» главы 13).



«Сине-зеленые» развертывания не работают, когда микросервис создает события для выходного потока в ответ на входной поток событий. Оба микросервиса этого варианта развертывания будут перезаписывать результаты друг друга в случае потоков сущностей или создавать дублированные события в случае потоков событий. Вместо «сине-зеленого» развертывания используйте тогда либо шаблон непрерывного обновления, либо базовый шаблон полного развертывания.

Резюме

Оптимизация развертывания микросервисов требует, чтобы ваша организация уплатила «налог на микросервисы» и вложила средства в необходимые системы развертывания. Из-за большого количества микросервисов, которыми может потребоваться управление, лучше всего делегировать ответственность за развертывание командам, владеющим микросервисами. Этим командам потребуются вспомогательные инструменты для контроля и управления их развертываниями.

Конвейеры непрерывной интеграции — важная часть процесса развертывания. Они обеспечивают основу для настройки и выполнения тестов, проверки сборок и обеспечения готовности контейнерных сервисов к развертыванию в производственной среде. Система управления контейнерами предоставляет средства для управления развертыванием контейнеров в вычислительных кластерах, распределения ресурсов и обеспечения масштабируемости.

Существует несколько способов развертывания сервисов, самый простой из которых — полностью остановить микросервис и повторно развернуть обновленный код. Однако это может привести к значительному простою и не подходит в зависимости от SLA. Есть несколько других шаблонов развертывания, каждый из которых имеет свои преимущества и недостатки. Набор шаблонов, рассмотренных в этой главе, ни в коем случае не является исчерпывающим, но они должны дать вам хорошую отправную точку для того, чтобы разобраться в способах организации развертывания ваших собственных сервисов.

Заключение

Архитектуры событийно-управляемых микросервисов обеспечивают мощный, гибкий и четко определенный подход к решению бизнес-задач. Вот краткий обзор того, что было рассмотрено в этой книге, а также несколько завершающих слов.

Уровни обмена информацией

Структура обмена данными обеспечивает универсальный доступ к важным бизнес-событиям во всей организации. Брокеры событий удовлетворяют эту потребность чрезвычайно хорошо, поскольку допускают наличие строгой организации данных, могут распространять обновления почти в реальном времени и способны работать в масштабе больших данных (*big data*). Обмен данными остается строго отвязанным от бизнес-логики, которая трансформирует и утилизирует эти данные, перекладывая ее требования на отдельные ограниченные контексты. Такое разделение задач позволяет брокеру событий оставаться в значительной степени независимым по отношению к требованиям бизнес-логики (помимо поддержки операций чтения и записи) и дает ему возможность сфокусироваться исключительно на хранении, поддержке и распределении событийных данных среди потребителей.

Зрелый уровень обмена данными отцепляет владение и производство данных от доступа к ним и их потребления. Приложения больше не должны выполнять двойные обязанности, обслуживая внутреннюю бизнес-логику и обеспечивая при этом также механизмы синхронизации и внешний прямой доступ для других сервисов.

Любой сервис может использовать долговечность и отказоустойчивость брокера событий для обеспечения высокой доступности своих данных, в том числе тех, которые задействуют этот брокер для хранения журналов изменений своего внутреннего состояния. Отказ экземпляра сервиса больше не приводит к тому, что данные станут недоступными, — просто новые данные будут отложены до тех пор, пока производитель не вернется в оперативный режим. В то же время потребители могут свободно потреблять данные из брокера событий во время любых отключений производителя, обрабатывая режимы отказов между сервисами.

Предметные области бизнеса и ограниченные контексты

Бизнес работает в определенном *домене* (предметной области), который можно разбить на поддомены. Решения бизнес-задач основываются на ограниченных кон-

текстах, сами границы которых определяют входы, выходы, события, требования, процессы и модели данных, относящиеся к поддомену.

Реализации микросервисов могут быть построены таким образом, чтобы обеспечивалось согласование с их ограниченными контекстами. При этом результирующие сервисы и рабочие процессы должны соответствовать задачам и бизнес-требованиям. Универсальный уровень передачи данных помогает облегчить это согласование, гарантируя, что реализации микросервисов будут достаточно гибки, чтобы адаптироваться к изменениям в бизнес-области и последующих ограниченных контекстах.

Совместные инструменты и инфраструктура

Событийно-управляемые микросервисы требуют инвестиций в системы и инструменты, которые обеспечивают их работу в большом масштабе. Такие инвестиции получили название «налог на микросервисы». Брокер событий находится в центре системы, обеспечивая фундаментальный обмен информацией между сервисами и освобождая каждый сервис от управления собственным решением по обмену данными.

Архитектура микросервисов охватывает все проблемы, связанные с созданием, управлением и развертыванием приложений, и выигрывает от стандартизации и оптимизации этих процессов. Такой охват приобретает все более критическую важность по мере роста числа микросервисов, и хотя каждый новый микросервис влечет за собой дополнительные накладные расходы, нестандартизированные микросервисы могут приводить к значительно большим издержкам, чем те, которые следуют протоколу.

Исключительно важные сервисы, требующие уплаты «налога на микросервисы», включают:

- ◆ брокер событий;
- ◆ реестр схем и сервис получения данных;
- ◆ систему управления контейнерами;
- ◆ непрерывную интеграцию, доставку и развертывание сервисов;
- ◆ сервис мониторинга и протоколирования.

Уплата «налога на микросервисы» вряд ли будет процессом «все или ничего». Организация, как правило, начинает либо с брокера событий, либо с системы управления контейнерами и добавляет другие элементы по мере необходимости. К счастью, ряд поставщиков вычислительных сервисов, такие как Google, Microsoft и Amazon, создали сервисы, которые значительно снижают накладные расходы. Вы должны взвесить свои возможности и выбрать между аутсорсингом этих операций поставщику сервисов либо созданием собственных систем своими силами.

Схематизированные события

Схемы играют ключевую роль в передаче смысла событий. События со строгой типизацией заставляют как производителей, так и потребителей учитывать реальность данных. Производители должны гарантировать, что они создают события в соответствии со схемой, в то время как потребители должны гарантировать, что они обрабатывают типы, диапазоны и определения потребляемых событий. Строго определенные схемы значительно снижают вероятность того, что потребители неверно истолковывают события, и обеспечивают контракт на будущие изменения.

Развитие схемы предоставляет механизм изменения событий и сущностей в ответ на новые бизнес-требования. Это позволяет производителю генерировать данные с новыми и измененными полями, а также дает возможность потребителям, которых не интересуют изменения, продолжать использовать старые схемы. Это значительно снижает частоту и риск несущественных изменений. В то же время оставшиеся потребители могут независимо обновлять свой код для потребления и обработки событий с использованием нового формата схемы, обеспечивающего им доступ к новым полям данных.

Наконец, схемы позволяют использовать такие полезные функциональности, как генерация кода и обнаружение данных. Генераторы кода способны создавать классы/структуры, относящиеся к созданию и потреблению приложений. Такие строго определенные классы/структуры могут помочь обнаруживать ошибки форматирования в производстве и потреблении либо во время компиляции, либо во время локального тестирования, обеспечивая беспрепятственное создание и использование событий. Это позволяет разработчикам сосредоточиться строго на бизнес-логике обработки и преобразования этих событий и гораздо меньше на составляющих их компонентах. Между тем реестр схемы предоставляет средства поиска, дающие возможность определить, какие данные к какому потоку событий относятся, что упрощает обнаружение содержимого потока.

Освобождение данных и единственный источник истины

Как только уровень передачи данных станет доступен, самое время добавить в него данные, критически важные для бизнеса. Процесс освобождения данных из различных сервисов и хранилищ данных в вашей организации может быть длительным, и потребуется некоторое время, чтобы заполучить все необходимые данные в брокер событий. Это важный шаг в отделении систем друг от друга и для перехода к событийно-управляемой архитектуре. Освобождение данных отвязывает производство данных и владение ими от доступа к ним нижестоящих потребителей.

Начните с освобождения данных, которые наиболее часто используются и наиболее важны для достижения главных целей вашей организации. Существуют различные пути извлечения информации из разных сервисов и хранилищ данных, и каждый имеет свои преимущества и недостатки. Важно взвесить последствия для существующих

вующего сервиса рисков устаревания данных, отсутствия схем и открытия внешнему миру внутренних моделей данных в освобожденных потоках событий.



Наличие легкодоступных бизнес-данных в виде потоков событий позволяет создавать сервисы на основе их композиции. Новый сервис должен только подписаться на интересующие его потоки событий через брокер событий, а не напрямую подключаться к каждому сервису, который в таком случае предоставлял бы ему требуемые данные.

Микросервисы

Будучи реализацией ограниченного контекста, микросервисы ориентированы на решение *бизнес-задач* этого ограниченного контекста и соответствующим образом согласованы. Основным драйвером обновлений микросервиса являются изменения бизнес-требований, при этом все другие несвязанные микросервисы остаются неизменными.

Избегайте внедрения микросервисов, основанных на технических границах. Обычно они создаются в качестве краткосрочной оптимизации для обслуживания нескольких бизнес-процессов, но при этом сцепляются с каждым рабочим процессом. Технический микросервис становится чувствительным к изменениям в бизнесе и объединяет вместе не связанные друг с другом рабочие процессы. Сбой или не преднамеренное изменение технического микросервиса может нарушить работу нескольких бизнес-процессов. По возможности избегайте технических взаимосвязей и вместо этого сосредоточьтесь на выполнении ограниченного контекста бизнеса.

Наконец, не все микросервисы должны быть «микро». Организациям разумно поначалу использовать несколько более крупных сервисов — особенно если они еще не уплатили «налог на микросервисы» частично или полностью. Это нормальная эволюция архитектуры организации. Если ваш бизнес использует несколько крупных сервисов, то постепенная ориентация на следующие принципы поможет вам создать новые детализированные сервисы, отцепленные от существующих крупных:

- ◆ помещайте важные бизнес-сущности и события в брокер событий;
- ◆ используйте брокер событий в качестве единственного источника истины;
- ◆ избегайте использования прямых вызовов между сервисами.

Варианты реализации микросервисов

Существует широкий спектр вариантов создания событийно-управляемых микросервисов — все со своими плюсами и минусами. В настоящее время «легковесные» фреймворки, как правило, имеют наибольшую функциональность прямо «из коробки». Потоки могут быть материализованы в таблицы и поддерживаться бесконечно. Соединения, в том числе по внешним ключам, могут выполняться для множества

потоков и таблиц. «Горячие реплики», долговременное хранилище и журналы изменений обеспечивают отказоустойчивость и масштабируемость.

«Тяжеловесные» фреймворки обеспечивают функциональность, аналогичную «легковесным» фреймворкам, но не обладают достаточной независимостью обработки данных, поскольку требуют отдельного выделенного кластера ресурсов. При этом все более популярными становятся новые варианты управления кластерами — такие как прямая интеграция с Kubernetes, одной из ведущих систем управления контейнерами. «Тяжеловесные» фреймворки часто используются сейчас в средних и крупных организациях, как правило, специалистами, выполняющими анализ больших данных.

Решения на основе базового шаблона производителя и потребителя (BPC) и технологии «Функция как сервис» (FaaS) предоставляют гибкие возможности для многих языков и сред выполнения. Но оба эти варианта ограничены базовой реализацией взаимосвязи потребителя и производителя. Обеспечить с их помощью детерминированное поведение сложно, поскольку ни один из вариантов не имеет возможностей встроенного планирования событий, поэтому сложные операции требуют либо значительных инвестиций в собственные пользовательские библиотеки для обеспечения этой функциональности, либо ограничены применением простых шаблонов на основе варианта использования.

Тестирование

Событийно-управляемые микросервисы очень хорошо поддаются полной интеграции и модульному тестированию. Как преобладающая форма входных данных в событийно-управляемом микросервисе, события могут легко комбинироваться, охватывая любые возможные случаи. Схемы событий ограничивают диапазон значений, которые должны тестироваться, и обеспечивают необходимую структуру для композиции входных тестовых потоков.

Локальное тестирование может включать как модульное, так и интеграционное тестирование, причем последнее опирается на динамическое создание брокера событий, реестра схем и любых других зависимостей, требуемых тестируемым сервисом. Например, брокер событий может быть создан для запуска в том же исполняемом файле, что и тест, как в случае с многочисленными решениями на основе JVM, или для запуска в собственном контейнере вместе с тестируемым приложением. Имея полный контроль над брокером событий, вы можете моделировать нагрузку, временные условия, сбои, отказы или любые другие взаимодействия брокера и приложения.

Вы можете провести интеграционное тестирование производственной среды, динамически создав временный кластер брокера событий, заполнив его копиями производственных потоков событий (за исключением вопросов информационной безопасности) и выполнив приложение для него. Это может обеспечить проверку «на дымность» перед развертыванием в производственной среде, которая поможет вам убедиться, что ничего не было упущено. Вы также можете выполнять тесты

производительности в этой среде, проверяя как производительность отдельного экземпляра, так и способность всего приложения к горизонтальному масштабированию. Среда тестирования после его завершения может быть легко отброшена.

Развертывание

Для масштабного развертывания микросервисов требуется, чтобы владельцы микросервисов могли сами быстро и легко разворачивать и откатывать свои сервисы. Такая самостоятельность позволяет командам двигаться вперед и действовать независимо, устранивая «узкие места», которые в противном случае возникали бы в инфраструктурной команде, ответственной исключительно за развертывание. Конвейеры непрерывной интеграции, доставки и развертывания имеют важное значение для обеспечения этой функциональности, поскольку они обеспечивают оптимизированный и все же индивидуально настраиваемый процесс развертывания, который сокращает шаги и вмешательства в ручном режиме и может масштабироваться на другие микросервисы. В зависимости от вашего выбора система управления контейнерами может предоставлять дополнительную функциональность для помощи в развертывании и откате, что еще больше упрощает этот процесс.

Процесс развертывания должен учитывать соглашения об уровне обслуживания (SLA), восстановление состояния и повторное потребление входных потоков событий. Соглашения об уровне обслуживания — это не просто вопросы простого. Вы также должны учитывать влияние развертывания на всех нижестоящих потребителей и работоспособность сервиса брокера событий. Микросервис, который должен полностью перестроить свое состояние и распространять новые выходные события, может создать значительную нагрузку на брокер событий, а также вызывать необходимость в масштабировании нижестоящих потребителей вверх — до многочисленных дополнительных экземпляров обработки. Нередко сервис восстановления обрабатывает миллионы или миллиарды событий в короткие сроки. Квоты могут смягчать это воздействие, но, в зависимости от требований к нижестоящим сервисам, сервис восстановления может находиться в несогласованном состоянии в течение неприемлемого периода времени.

Всегда существуют компромиссы между SLA, последствиями для нижестоящих потребителей, для брокера событий и для системы мониторинга и оповещения. Например, «сине-зеленое» развертывание требует двух групп потребителей, которые должны учитываться в мониторинге и оповещении, включая автоматическое масштабирование на основе задержки. Хотя вы, безусловно, можете выполнить работу, необходимую для адаптации этого шаблона развертывания, еще один вариант состоит в том, чтобы просто изменить дизайн приложения. Альтернативой «сине-зеленым» развертываниям является использование тонкого, постоянно работающего уровня сервисов для обслуживания синхронных запросов, в то время как бэкендовый обработчик событий может быть заменен и повторно обработан в свое время. Хотя ваш уровень сервисов будет в течение некоторого периода времени обслуживать устаревшие данные, он не требует каких-либо дополнений к инструмента-

ментам или более сложных операций замещения и, возможно, все еще может соответствовать SLA зависимых сервисов.

Завершающие слова

Событийно-управляемые микросервисы требуют от вас переосмыслиния того, что такие данные на самом деле, как сервисы получают к ним доступ и как их используют. Объем данных, относящихся к какой-либо конкретной предметной области, растет с каждым годом не по дням, а по часам, и этот рост не показывает никаких признаков замедления. Данные становятся все больше и более вездесущими, и прошли те времена, когда их можно было просто засунуть в одно большое хранилище данных и использовать для всех целей. Надежный и четко определенный уровень обмена данными освобождает сервисы от выполнения двойных обязанностей и позволяет им вместо этого сфокусироваться на обслуживании только своих бизнес-функций, а не на данных и запросах других ограниченных контекстов.

Событийно-управляемые микросервисы — это естественная эволюция вычислений для обработки больших и разнообразных наборов данных. Композиционный характер потоков событий обеспечивает беспрецедентную гибкость и позволяет отдельным бизнес-командам сосредоточиться на использовании любых данных, необходимых для достижения их бизнес-целей. Организации, использующие несколько сервисов, могут извлечь большую выгоду из обмена данными, предоставляемого брокером событий, который открывает путь для создания новых сервисов, полностью отвязанных от старых бизнес-требований и реализаций.

Независимо от будущего самих событийно-управляемых микросервисов, совершенно очевидно, что уровень обмена данными расширяет возможности имеющихся в организации данных на любого сотрудника или команду, которым они требуются, устраняет границы доступа и снижает ненужную сложность, связанную с производством и распространением важной бизнес-информации.

Предметный указатель

А

- Apache Gobblin 76
- Apache Zookeeper 197
- API «запрос-ответ» 230, 236–238, 248, 254

* * *

А

- Автомасштабирование 209
- Автономия: роль микросервиса в обеспечении автономии дизайна 25
- Авторитетный источник истины 47
- Алгоритм
 - ◊ назначения разделов 105
 - ◊ распределения событий 101
- Аналитические события 228, 229
- Архитектура
 - ◊ в стиле микросервисов 22
 - ◊ компьютерных систем 21
 - ◊ микросервисов 303
 - ◊ микросервисов прямого вызова 156
 - ◊ развертывания микросервисов 292
 - ◊ событийно-управляемых микросервисов 71, 155, 302
- Архитектуры
 - ◊ микросервисов прямого вызова 155
 - ◊ событийно-управляемых микросервисов 155
- Асинхронные
 - ◊ прямые вызовы 180, 181
 - ◊ событийно-управляемые архитектуры 70
- Асинхронный
 - ◊ запуск 175
 - ◊ пользовательский интерфейс 240
- Атомарная транзакция 145

Б

- Базовые клиенты потребителя 187
- Базовый шаблон производителя и потребителя (Basic Producer and Consumer, BPC) 187, 194, 306
- Бизнес-логика приложения 27
- Бизнес-топология 42
- Бизнес-требования 25, 27
 - ◊ к продукту 25
- Блокирующий вызов 230
- Брокер
 - ◊ событий 41, 46, 47, 51, 53, 56, 71, 84, 97, 110, 118, 132, 134, 145, 171, 177, 180, 184, 188, 195, 196, 216, 233
 - ◊ сообщений 51
- Бэкендовый подход с использованием микросервисов 248

В

- Валидация 88
 - ◊ схемы 90, 91
- Варианты хранения состояния и доступа к нему 130
- Ведущий узел (master node) 197
- Вертикальное масштабирование 55
- Ветвление потоков событий 101
- Виртуальная машина 54
- Виртуальный тупик (deadlock) перебалансирования 185
- Владелец данных 47
- Внешне сгенерированное событие 228
- Внешнее хранилище состояний 130, 139, 140, 142, 153, 236, 237
- Внешние хранилища состояний 187
- Внешний ввод/вывод в хранилища состояний 184
- Внешний сервис перераспределения (External Shuffle Service, ESS) 207

Внутреннее хранилище состояний 130, 132–136, 140, 142, 153
 Внутренний поток событий 118, 216
 Внутренняя модель данных 84, 85
 Водяные знаки (watermarks) 108, 114, 115, 127, 129, 212
 Возможности DevOps 255, 264
 Восстановление после отказов 107
 Временные метки 108, 109, 111
 Временные среды для интеграционного тестирования 283
 Временный кластер брокера событий 306
 Время
 ◊ жизни функции (lifespan) 172
 ◊ потока (stream time) 117, 122, 127, 129, 212
 ◊ потоковой передачи 108
 ◊ события 212
 Встраивание библиотек тестирования 278
 Выделение достаточных ресурсов 184
 Выделенный кластер тестирования 289
 Высокая тестопригодность 36

Г

Генератор кода 60, 61
 Гибкость бизнес-требований 36
 Гибридные архитектуры 40
 Главная цель детерминированной обработки 109
 Главные обрабатывающие состояния событийно-управляемого микросервиса 109
 Горизонтальное масштабирование 55
 «Горячая реплика» 136, 138, 220
 «Горячие реплики» хранилищ состояний 233
 Гранулярность 35
 Графический пользовательский интерфейс (GUI) 241
 Группа потребителей 52

Д

Денормализация данных 84
 Детерминизм максимальных усилий 109
 Динамическое масштабирование приложений по мере их исполнения 218
 Диск
 ◊ подключенный к сети 141
 ◊ подключенный через сеть 134, 135

Диспетчер задач (task manager) 198
 Домен (предметная область) 23, 302
 Допустимая задержка 124
 Достоверность информации 34
 Драйвер 203
 Дуальность табличного потока 45
 Дублирующиеся события 149, 150

Е

Единая общая среда тестирования 289
 Единственный источник истины 53, 79
 Единый источник истины 72, 75, 97

Ж

Журнал
 ◊ изменений 131, 132, 138, 142, 147, 153, 216, 218, 233
 ◊ хранилища данных 80
 Журналы регистрации изменений в данных 80

З

Задание (job) 198
 Задача (task) 198
 Задержка потребителей (consumer lag) 262
 Закон Конвея 29
 Запаздывание потребителя 49
 Запоздалое событие (late) 120, 122, 129
 Захват изменений в данных (Change-Data Capture, CDC) 76

И

Имитация (mocking) конечной точки 274
 Индекс 49
 Индивидуально настраиваемое назначение 107
 Инструменты
 ◊ непрерывной доставки (Continuous Delivery, CD) 265
 ◊ непрерывной интеграции (Continuous Integration, CI) 265
 Интеграционное тестирование 306
 Интеллектуальный балансировщик нагрузки 236
 Исполнитель (executor) 197
 Исходящая таблица (outbox) 82

К

- Клиент-потребитель 187
 Клиент-производитель 187
 Комментарии к схемам 58
 Коммуникационная структура 26
 Компенсационные действия 168
 Компенсационный рабочий процесс 168
 Композиционный фронтенд 250
 Компоненты функционально-ориентированных решений 172
 Конвейер непрерывной интеграции (CI) 293, 295
 Конвейеры
 ◊ на основе потоковой передачи 197
 ◊ непрерывной интеграции, доставки и развертывания 307
 ◊ развертывания 41, 56
 Контейнеризация 54
 Контейнеры 54
 Контракт на передачу данных 57, 62, 66, 72, 75, 84, 93
 Контроллер разделов 105, 107
 Контроль доступа к данным 260
 Контрольная точка 204, 205, 207, 209, 210, 214, 215
 Контрольные точки (checkpoints) 204, 219
 Концепции предметной области 29
 Концепция модулей Kubernetes 55
 «Крупнозернистый» режим (coarse-grained mode) 207

Л

- «Легковесный» фреймворк 195, 216, 220, 217
 Лимит на объем ресурсов 257
 Локализация данных 102, 103
 Локальное
 ◊ интеграционное тестирование 275, 277, 280, 282
 ◊ тестирование 306

М

- Максимальное время исполнения 184
 Масштабируемость 36
 ◊ подходов 33
 Материализация потока событий 45
 Материализованное состояние 130
 Метрика задержки группы потребителей (consumer group's lag metric) 176

- Механизм получения уведомлений о событии 177
 Миграция данных 93
 Мигрирование 144
 Микросервис прямого вызова 156
 Микросервисы 22
 Микрофронтенды 247, 249, 254
 Модель
 ◊ контрольных точек 218
 ◊ предметной области 24, 70
 Модульное тестирование 273, 306
 ◊ без поддержки состояния 273
 ◊ с поддержкой состояния 273
 Моментальный снимок 143
 Мультитенантность (multitenancy) 210

Н

- Назначение разделов по круговой схеме 105
 Налог на микросервисы 56, 303, 305
 Неблокирующий запрос 231
 Недостатки синхронных микросервисов «запрос-ответ» 38
 Независимый кластер обрабатывающих ресурсов 195
 Неизменяемый журнал 45, 52, 53
 Непрерывная
 ◊ доставка (Continuous delivery) 293
 ◊ интеграция (Continuous Integration, CI) 293
 Непрерывное развертывание (Continuous deployment) 293
 Неупорядоченное событие (out-of-order) 119, 120, 129
 Нефункциональное тестирование 272

О

- Обрабатывающая топология микросервиса 100
 Обработка неупорядоченных событий 126
 Обработчик
 ◊ с поддержкой состояния 102
 ◊ не поддерживающий сохранение состояния 102
 Общий брокер событий тестирования 288
 Ограниченный контекст 24, 42
 Окно пакета 175
 Оконная обработка 122
 Оконные функции 124
 Операторное состояние 204, 219

Операция материализации 41
 Определение данных 57–59, 93
 Определения исходных данных 75
 Оркестрованные транзакции 165, 167, 168
 Оркестровка
 ◊ процессов с прямым вызовом 160
 ◊ событийно-управляемых процессов 160
 Освобождение данных 71–77, 79, 80, 82, 92, 93, 95, 97
 Освобожденные данные 79
 Основные типы окон событий 122
 Откат транзакции 167
 Отметка об удалении 46
 Отозванный раздел 132
 Очередь 48, 53
 ◊ с возможностью индивидуальной фиксации 183

П

Первичные выгоды от использования событийно-управляемых микросервисов 35
 Переворачивающееся (tumbling) окно 122
 Переподразделение (repartitioning) 102
 ◊ событий 100, 102
 Переработка (reprocessing) 126
 Перераспределение (shuffle) 115
 ◊ событий 219
 Перестраивание хранилищ состояний 143
 Планирование событий 112, 126
 Повторная обработка 126, 129
 Подготовленные для тестирования наборы данных 285
 Поддержание состояния 46
 Поддержка
 ◊ непрерывной поставки 36
 ◊ триггеров 89
 Поддомен 23
 Подразделение доменов на поддомены 24
 Подход с фиксацией смещений 170
 Политика «без локального состояния» («no local state» policy) 178
 Пользовательский интерфейс (UI) 240
 Понятия, связанные с временными метками 110
 Популярность фреймворков 212
 Поток освобожденных событий 73
 Потоки событий 35
 Потоковые фреймворки 195
 Потребляющие микросервисы 41

Правила развития схемы 59, 75
 Право владения потоком 255, 256
 Практически однократная обработка 145, 147, 148, 151
 Предметно-ориентированное проектирование 23
 Преимущества синхронных микросервисов 39
 Преобразование 100
 Привязка микросервисов к техническим требованиям 25
 Примеры полезных метаданных 256
 Принцип
 ◊ единственного автора 164
 ◊ единственного источника событий 255, 261
 ◊ единственной ответственности (single responsibility) 36
 Принципы
 ◊ определения контракта на передачу данных 66
 ◊ проектирования FaaS 169
 Проблемы
 ◊ использования синхронных микросервисов 38
 ◊ модификации унаследованных систем 74
 Программное создание имитационных событий 285
 Продолженная миграция 299
 Производящие микросервисы 41
 Пространства имен 211
 Пространство решений 24
 Протоколы статического назначения 106
 Процесс создания микросервиса 263
 Процессы прямого вызова 161

Р

Рабочий процесс 154, 156, 158, 162, 164, 165, 167
 ◊ хореографической транзакции 164
 Рабочий узел (worker node) 197
 Разворачивание кластера на CMS 201
 Развитие схемы 70
 Разделение команды 30
 ◊ на две меньшие 32
 Размер пакета 175, 184
 Разрушительное изменение схемы 61, 297
 Распределенная файловая система Hadoop (Hadoop Distributed File System, HDFS) 204

Распределенные транзакции 162, 163
 ◇ с хореографическими рабочими процессами 163

Реактивные события 228

Реализации брокера событий 145

Реестр схем 258

Решения на основе FaaS с поддержкой состояния 178

Риск запуска и выполнения новых сервисов 30

Риски

- ◇ добавления нового бизнес-требования в существующий сервис 31
- ◇ создания нового сервиса 30

C

Саги (распределенные транзакции) 163

Сброс внутреннего состояния приложения 261

Сеансовое (session) окно 124

Серверы сетевого протокола времени (Network Time Protocol, NTP) 111

Сервисно-ориентированные архитектуры (Service-Oriented Architectures, SOA) 22

Сервисы, требующие уплаты «налога на микросервисы» 303

Сериализация схемы 86

Сильная зацепленность (highly cohesive) 24

«Сине-зеленое» развертывание 300, 301, 307

Синхронизированные миграции 300

Синхронные

- ◇ вызовы функций 182
- ◇ потоки «запрос-ответ» 161
- ◇ триггеры 175

Синхронный ввод/вывод в формате «запрос-ответ» 187

Система управления контейнерами (Container Management System, CMS) 171, 195, 201, 216, 294

Системы

- ◇ контейнеризации и управления контейнерами 56
- ◇ управления контейнерами 41, 55, 264

Скользжение окна 123

Скользящее (sliding) окно 123

Слабая

- ◇ связанность (loosely coupled) 24
- ◇ сцепленность 36

Слияние потоков событий 101

Случайный отказ в обслуживании 257

Смещение 52

Снимок существующих данных 80

Собственный планировщик событий 113

Событие 43

Событизация (eventification) 85

- ◇ пользователя 85

Событийно-управляемая

- ◇ архитектура 71, 96, 98
- ◇ структура обмена информацией 34
- ◇ топология 41

Событийно-управляемые архитектуры 76, 155

Событийно-управляемые

- ◇ микросервисы (Event-Driven Microservice, EDM) 22, 23, 25, 26, 32, 35, 37, 47, 64, 71, 73, 86, 99, 107, 108, 126, 130, 134, 187, 215, 230, 232, 259
- ◇ микросервисы без поддержки состояния 107
- ◇ топологии 273

Событийно-управляемый

- ◇ микросервис 41, 45
- ◇ подход 33

Событийные данные 47

Совместные базы данных 29

Соглашение об уровне обслуживания (SLA) 307

Соединения

- ◇ по внешнему ключу (foreign-key joins) 221
- ◇ по первичному ключу (primary-key joins) 220

Создание

- ◇ единой общей среды тестирования 286
- ◇ определений событий 64

Соподразделение (copartitioning) 103

Соподразделенные потоки 107

Состояние по ключу 205, 219

Списки управления доступом (Access Control List, ACL) 259

Способы

- ◇ обработки запоздалого события 125
- ◇ поддержки ограниченных контекстов с функциями 170

Средства обработки пакетов данных big data 196

Средство извлечения метки времени 114

Среды для тестирования микросервиса 289

Стратегии масштабирования приложений с поддержкой состояния 206

Стратегия

- ◊ миграции 72
- ◊ обработки неупорядоченных и запоздалых событий 124
- Строго одноразовая обработка 145
- Структура обмена
 - ◊ бизнес-информацией 27
 - ◊ данными 28, 40
 - ◊ информацией 26, 28, 40
 - ◊ технической информацией 27
 - ◊ формализованного обмена данными 35
- Сценарий
 - ◊ единоличное владение 25
 - ◊ трансграничное владение 25

Т

- Тест топологии 274
- Тестиирование
- ◊ с использованием реальных производственных данных 284
 - ◊ топологии 275
- Технологическая гибкость 36
- Технология
- ◊ Debezium 81
 - ◊ Maxwell 81
 - ◊ «Функция как сервис» (Functions-as-a-Service, FaaS) 169, 306
 - «Теплая» функция 173
- Типы
- ◊ запросов шаблона освобождения данных по запросу 77
 - ◊ событий 43
 - ◊ совместимости 59
- Топология 41, 42
- ◊ микросервисов 41, 42, 100
- Транзакции 145, 147–149, 152, 153
- Триггер 90, 174
- ◊ на основе расписания 177
 - ◊ по задержке группы потребителей 177
 - ◊ слушателя потока 175, 176
 - ◊ слушателя потока событий (event-stream listener trigger) 174
- Триггерная логика 57, 58, 68, 70
- Триггерные системы со слушателем событий 185
- Триггеры 89, 91
- «Тяжеловесные»
- ◊ потоковые фреймворки 209
 - ◊ фреймворки 195, 197, 200, 204, 209, 306

- ◊ фреймворки пакетной обработки 196
- ◊ фреймворки потоковой обработки 196, 215
- «Тяжеловесный»
- ◊ кластер потоковой обработки 197
- ◊ фреймворк 196, 217

У

- Удаленное интеграционное тестирование 275
- Узловое время события 116
- Унаследованные системы 188
- Уплотнение 46
- Управление
- ◊ данными предприятия (EDM, Enterprise Data Management) 154, 232
 - ◊ окнами (windowing) 122
- Уровень параллелизации 223

Ф

- Физически подключенный локальный диск 134
- Фиксация
- ◊ смещений 170
 - ◊ смещений до завершения обработки 171
- Формат
- ◊ события 63
 - ◊ схемы 59
- Формы событий 52
- Фреймворк FaaS 172, 173, 177, 180, 186
- Фреймворки тестирования топологии 274
- Функциональное тестирование 272
- Функция 169
- ◊ соединения (join) 225

Х

- «Холодный старт» (cold start) 173
- Хореографическая архитектура 155
- Хореографический
- ◊ рабочий процесс 155, 157, 163
 - ◊ шаблон 156, 163, 168
 - саги 163
- Хранилище
- ◊ глобального состояния 133
 - ◊ состояний 130–132, 136, 138, 140, 142–144, 151, 153

Ц

- Целевые уровни обслуживания
(Service-Level Objectives, SLO) 210
Централизованный фреймворк 94, 95

Ш

- Шаблон
- ◊ «запрос-ответ» 230
 - ◊ «Коляска» (sidecar) 188, 189
 - ◊ «Сара» 163
 - ◊ архитектурного дизайна, основанный на хореографии 180
 - ◊ гибридного приложения (hybrid application pattern) 192–194
 - ◊ нагрузки (load pattern) 191
 - ◊ оркестратора 186
 - ◊ оркестровки 158, 163, 179, 180, 182

- ◊ прямого вызова (direct-call pattern) 180
- ◊ слушателя потока событий (listener pattern) 173
- ◊ фильтрации 189, 190, 200
- ◊ хореографии 155, 179, 180
- Шаблоны
- ◊ использования «тяжеловесных» фреймворков 199
- ◊ освобождения данных 75

Э

- Эмерджентное поведение 155

Я

- Явная схема для каждого события 58
Язык определения данных (Data Definition Language, DDL) 93

Об авторе

Адам Беллемар — штатный инженер платформы данных в Shopify. Он занимает эту должность с 2020 года. Ранее, с 2014 по 2020 год, он работал в качестве штатного инженера во Flipp. А до этого занимал должность разработчика программного обеспечения в BlackBerry, где впервые начал работать с событийно-управляемыми системами.

Его опыт включает разработку и сопровождение (DevOps) кластеров Kafka, Spark, Mesos, Kubernetes, Solr, Elasticsearch, HBase и Zookeeper — создание программ, масштабирование, мониторинг; техническую поддержку — помочь предприятиям в организации их уровня обмена данными, в интеграции с существующими системами, в разработке новых систем с акцентом на поставку продуктов; разработку программного обеспечения — построение событийно-управляемых микросервисов на Java и Scala с помощью потоковых библиотек Beam, Flink, Spark и Kafka, а также инженерию данных — изменение способов сбора поведенческих данных с пользовательских устройств и обмена ими внутри организации.

Об обложке

На обложке книги изображена желтощекая синица (*Machlolophus spilonotus*). Эту птичку можно встретить в широколиственных и смешанных холмистых лесах, а также в созданных людьми парках и садах Юго-Восточной Азии.

Ярко-желтая область вокруг клюва и задняя часть головы желтощекой синицы, контрастирующие с ее черным гребнем, зобом и грудью, делают ее легко узнаваемой. У самца, чье изображение приведено на обложке, серое тело и черные крылья, испещренные белыми пятнами и полосами, тело самки же оливковое, а крылья у нее бледно-желтые и тоже полосатые.

Желтощекие синицы питаются мелкими беспозвоночными, пауками и некоторыми фруктами и ягодами, добывая пищу в нижних и средних уровнях леса. Как и другие птицы из семейства синиц, желтощекая синица порхает над лесом короткими волнообразными виражами, быстро-быстро взмахивая крыльями.

К счастью, выживание желтощекой синицы пока не вызывает беспокойства, однако многие представители животного мира, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Все они чрезвычайно важны для нашей планеты.

Иллюстрация на обложке создана Карен Монтгомери и основана на черно-белой гравюре из книги «Иллюстрированный музей живой природы» («Pictorial Museum of Animated Nature»).