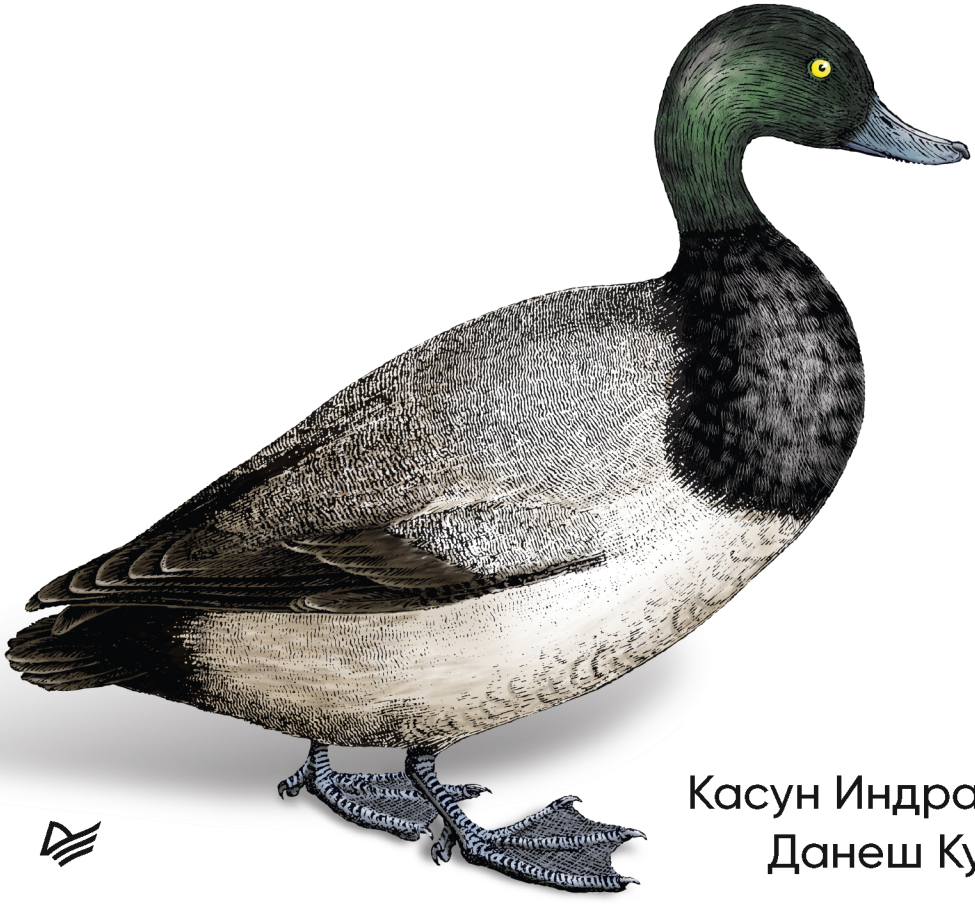
O'REILLY®

gRPC:

запуск и эксплуатация облачных приложений

Go и Java для Docker и Kubernetes



Касун Индрасири
Данеш Курупу

gRPC: Up and Running

Building Cloud Native Applications with Go and Java for Docker and Kubernetes

Kasun Indrasiri and Danesh Kuruppu

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

gRPC:
запуск и эксплуатация облачных приложений

Go и Java для Docker и Kubernetes

Касун Индрасири, Данеш Курупу

ПИТЕР®

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

2021

ББК 32.988.02-018
УДК 004.738.5
И60

Индрасири Касун, Курупу Данеш

И60 gRPC: запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes. — СПб.: Питер, 2021. — 224 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1737-6

Год от года обретая новых сторонников, облачно-ориентированные и микросервисные архитектуры стали основой современного IT. Такой переход значительно повлиял и на структуру коммуникаций. Теперь приложения часто подключаются друг к другу по сети, и это происходит с помощью технологий межпроцессной коммуникации. Одной из наиболее популярных и эффективных технологий такого рода является gRPC, но информации о ней не хватает. Так было, пока не вышла эта книга!

Наконец архитекторы и разработчики смогут подробно разобраться, как технология gRPC устроена «под капотом», и для этого не придется разгребать десятки устаревших ссылок в поисковике.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1492058335 англ.

Authorized Russian translation of the English edition of gRPC: Up and Running ISBN 9781492058335 © 2020 Kasun Indrasiri and Danesh Kuruppu
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.

978-5-4461-1737-6

Краткое содержание

Оглавление

Введение

В наши дни приложения часто «общаются» между собой по компьютерным сетям, используя технологии межпроцессного взаимодействия. gRPC — это разновидность межпроцессного взаимодействия, основанная на высокопроизводительных удаленных вызовах процедур (remote procedure calls, RPC) и предназначенная для создания распределенных приложений и микросервисов. Благодаря появлению микросервисов и облачно-ориентированных приложений популярность gRPC растет экспоненциально.

Зачем мы написали эту книгу

Наблюдая за распространением gRPC, мы чувствовали, что разработчикам необходимо полномасштабное руководство по данной технологии, которое можно использовать в качестве основного справочника на каждом этапе цикла разработки gRPC-приложений. На просторах Интернета gRPC посвящено множество материалов и примеров кода (документация, блоги, статьи, презентации и т. д.), но при этом отсутствует единый ресурс, пригодный для создания полноценных приложений. К тому же очень сложно найти информацию о внутреннем устройстве протокола gRPC и о том, как он работает.

Чтобы исправить ситуацию, мы написали эту книгу. Она поможет вам разобраться в основах gRPC, понять отличия данного протокола от традиционных средств межпроцессного взаимодействия, изучить практические методы коммуникации по gRPC и научиться создавать gRPC-приложения на Go и Java. Здесь вы узнаете, как устроена эта технология, как развертывать gRPC-приложения в промышленных условиях и как gRPC интегрируется в Kubernetes и облачную экосистему в целом.

Целевая аудитория

Эта книга ориентирована прежде всего на разработчиков, которые занимаются созданием распределенных приложений и микросервисов с помощью разных технологий межпроцессного взаимодействия. Такие разработчики должны разбираться в основах gRPC, понимать, когда и как следует использовать данный протокол, иметь представление о рекомендуемых методиках развертывания gRPC-сервисов в промышленных условиях и т. д. Кроме того, представленный материал будет весьма полезен архитекторам, которые внедряют микросервисную или облачно-ориентированную архитектуру и проектируют механизмы взаимодействия сервисов: gRPC сравнивается с аналогичными технологиями и даются рекомендации о том, когда его стоит использовать, а когда — нет.

Мы исходим из того, что разработчики и архитекторы, читающие данную книгу, имеют общее представление о распределенных вычислениях, включая методы межпроцессного взаимодействия, сервис-ориентированную архитектуру (service-oriented architecture, SOA) и микросервисы.

Структура издания

Эта книга написана так, что теоретический материал объясняется на примерах из реальной жизни. Мы активно используем демонстрационный код на Go и Java, чтобы читатель мог сразу получить практический опыт применения концепций, с которыми знакомится. Мы разделили эту книгу на восемь глав.

- В главе 1 вы получите общее представление об основах gRPC и сможете сравнить эту технологию с аналогичными разновидностями межпроцессного взаимодействия, такими как REST, GraphQL и прочими механизмами RPC.
- На страницах главы 2 вы получите свой первый практический опыт создания полноценного gRPC-приложения на Go или Java.
- В главе 3 вы исследуете методы работы с gRPC на реальных примерах.
- Если вы уже имеете опыт использования gRPC и хотите узнать, как устроена эта технология, то глава 4 для вас. Здесь мы пройдемся по каждому

этапу взаимодействия сервера и клиента и покажем, как gRPC работает на уровне сети.

- В главе 5 вы изучите некоторые из самых важных расширенных возможностей gRPC, такие как использование перехватчиков, работа с метаданными, мультиплексирование, балансировка нагрузки и т. д.
- В главе 6 вы досконально изучите методы защиты коммуникационных каналов и узнаете, как аутентифицировать пользователей и управлять их доступом к gRPC-приложениям.
- В главе 7 мы пройдемся по всему циклу разработки gRPC-приложений. Будут рассмотрены тестирование, интеграция с CI/CD, развертывание и запуск в Docker и Kubernetes, а также мониторинг.
- В главе 8 мы обсудим некоторые полезные вспомогательные компоненты, разработанные с расчетом на gRPC. Большинство из этих проектов подходит для создания реальных gRPC-приложений.

Использование примеров кода

Все примеры кода и дополнительные материалы доступны для скачивания по адресу grpc-up-and-running.github.io. Мы настоятельно рекомендуем использовать код, представленный в данном репозитории, по ходу чтения. Это позволит вам лучше понять концепции, с которыми вы знакомитесь.

Эти примеры кода поддерживаются и обновляются с учетом последних версий библиотек, зависимостей и инструментов разработки. Иногда содержимое репозитория может немного отличаться от кода, приведенного в книге. Если вы заметили некие расхождения (как отрицательные, так и положительные) в наших листингах, то мы очень просим вас оформить запрос на включение изменений (pull request, PR).

Вы можете использовать представленные здесь примеры кода в собственных программах и документации. Если вы не воспроизводите существенную часть кода, то связываться с нами не нужно. Это, например, касается ситуаций, когда вы включаете в свою программу несколько фрагментов кода, которые приводятся в данной книге. Однако на продажу или распространение примеров из книг издательства O'Reilly требуется отдельное разрешение. Цитировать книгу и приводить примеры кода при ответе на

вопрос вы можете свободно. Но если хотите включить существенную часть приведенного здесь кода в документацию своего продукта, то вам следует связаться с нами.

Мы приветствуем, но не требуем отсылки на оригинал. Отсылка обычно состоит из названия, имени автора, издательства, ISBN и копирайта. Например: «gRPC: запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes», Касун Индрасири и Данеш Куруппу (Питер). Copyright 2020 Kasun Indrasiri and Danesh Kuruppu, 978-5-4461-1737-6.

Если вам кажется, что ваше обращение с примерами кода выходит за рамки добросовестного использования или условий, перечисленных выше, то можете обратиться к нам по адресу permissions@oreilly.com.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсивный шрифт`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот рисунок указывает на совет или предложение.

Такой рисунок указывает на общее замечание.

Этот рисунок указывает на предупреждение.

Благодарности

Выражаем признательность научным редакторам этой книги, Жюльену Андрю, Тиму Реймонду и Райану Микеле. Кроме того, мы хотели бы поблагодарить старшего редактора Мелиссу Поттер за ее советы и поддержку, а также нашего редактора Райана Шоу за всю оказанную им помощь. И конечно же, мы говорим спасибо всему сообществу gRPC, которое создало этот замечательный проект с открытым исходным кодом.

От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

Введение в gRPC

Современные приложения редко работают в изоляции. Большинство из них общаются друг с другом по сети и координируют свои действия, обмениваясь сообщениями. Таким образом, современная программная система представляет собой набор распределенных приложений, которые работают на разных участках сети и взаимодействуют с помощью различных коммуникационных протоколов. Например, интернет-магазин может состоять из нескольких распределенных компонентов, таких как система управления заказами, каталог, база данных и т. д. Для реализации бизнес-возможностей подобного интернет-магазина необходимо наладить связь между этими компонентами.

Микросервисная архитектура

Микросервисная архитектура заключается в создании приложений на основе набора отдельных, автономных (с точки зрения разработки, развертывания и масштабирования), бизнес-ориентированных и слабо связанных между собой сервисов[1].

С появлением микросервисной (oreil.ly/q6N1P) и облачно-ориентированной (oreil.ly/8Ow2T) архитектуры традиционные приложения, обладающие разными бизнес-возможностями, подверглись разделению на более мелкие, автономные и бизнес-ориентированные сущности, известные как микросервисы. И эти микросервисы должны связываться друг с другом по сети, используя методики межпроцессного (межсервисного, межпрограммного) взаимодействия (inter-process communication, IPC). Если бы интернет-магазин из предыдущего примера был реализован с помощью микросервисной архитектуры, то состоял бы из нескольких взаимосвязанных микросервисов, которые бы отвечали за управление, поиск, оформление покупки, достав-

[1] *Indrasiri K., Siriwardena P.* Microservices for the Enterprise. — Apress, 2018.

ку и т. д. По сравнению с работой традиционных приложений данный подход требует большего количества сетевых соединений, поскольку задействовано множество мелких компонентов. Поэтому, какой бы архитектурный стиль вы ни выбрали (традиционный или на основе микросервисов), методы межпроцессного взаимодействия — один из важнейших аспектов современных распределенных приложений.

Межпроцессное взаимодействие обычно реализуется путем обмена сообщениями в асинхронном стиле (на основе запросов и ответов или событийной модели). Если взаимодействие происходит синхронно, то клиентский процесс шлет по сети сообщение серверному процессу и ждет ответа. При асинхронном событийном подходе процессы обмениваются асинхронными сообщениями, используя посредника, известного как *брокер событий*. Выбор метода взаимодействия зависит от бизнес-требований.

Что касается реализации синхронного взаимодействия в виде запросов и ответов, в современных облачно-ориентированных приложениях и микросервисах для этого чаще всего используют уже устоявшийся архитектурный стиль REST. Ваше приложение или сервис состоит из набора ресурсов, для чтения и модификации которых применяются сетевые вызовы, отправляемые по протоколу HTTP. Однако в большинстве случаев сервисы REST довольно громоздки, неэффективны и подвержены ошибкам в ходе построения межпроцессного взаимодействия. Часто возникает необходимость в хорошо масштабируемой технологии IPC со слабой связанностью, которая была бы более эффективной по сравнению с REST. Именно эту роль берет на себя gRPC — современный стиль межпроцессного взаимодействия для создания распределенных приложений и микросервисов (мы сравним gRPC и REST позже в данной главе). gRPC в основном использует синхронные запросы и ответы, но вместе с тем имеет полностью асинхронный или потоковый режим, доступный после установления соединения.

В настоящей главе мы расскажем, что такое gRPC, и объясним причины, приведшие к созданию данного протокола межпроцессного взаимодействия. Мы рассмотрим его ключевые составляющие, используя примеры из реальной жизни. Чтобы понимать основные проблемы, которые пытается решить gRPC, необходимо иметь четкое представление о методах IPC и о том, как они эволюционировали со временем. Поэтому мы разберем все эти методы и сравним их друг с другом. И начнем с разговора о том, что собой представляет gRPC.

Что такое gRPC

gRPC (значение буквы g меняется с каждой новой версией (oreil.ly/IKCi3)) — технология межпроцессного взаимодействия, позволяющая соединять, вызывать, администрировать и отлаживать распределенные гетерогенные приложения в стиле, который по своей простоте мало чем отличается от вызова локальных функций.

При разработке gRPC-приложения первым делом нужно определить интерфейс сервисов. Данное определение содержит информацию о том, как потребители могут обращаться к вашим сервисам, какие методы можно вызывать удаленно, какие параметры и форматы сообщений следует применять при вызове этих методов и т. д. Язык, который используется в спецификации сервиса, называется *языком описания интерфейсов* (interface definition language, IDL).

На основе спецификации сервиса можно сгенерировать серверный код (или *серверный каркас*), который упрощает создание логики на серверной стороне за счет предоставления абстракций для низкоуровневого взаимодействия. Вы также можете сгенерировать клиентский код (или *клиентскую заглушку*), инкапсулирующий низкоуровневые аспекты коммуникационного протокола в разных языках программирования. Методы, указанные в определении интерфейса сервиса, можно вызывать удаленно на клиентской стороне по примеру того, как вызываются локальные функции. Внутренний фреймворк gRPC возьмет на себя все сложные аспекты, присущие соблюдению строгих контрактов между сервисами, сериализации данных, сетевых коммуникаций, аутентификации, контролю доступа, наблюдаемости и т. д.

Понять основополагающие концепции gRPC попробуем на реальном примере использования микросервисов, основанных на этой технологии. Допустим, мы занимаемся разработкой приложения для розничной торговли, состоящего из нескольких микросервисов, и хотим, чтобы один из них возвращал описание товаров, доступных для продажи, как показано на рис. 1.1 (в главе 2 мы реализуем данный сценарий с нуля). Сервис `ProductInfo` спроектирован так, чтобы к нему был доступ в сети по протоколу gRPC.

Определение сервиса находится в файле `ProductInfo.proto`, который используется для генерации кода как на серверной, так и на клиентской стороне.

В этом примере мы исходим из того, что сервис написан на языке Go, а потребитель — на Java. Сетевое взаимодействие обоих объектов происходит по HTTP/2.

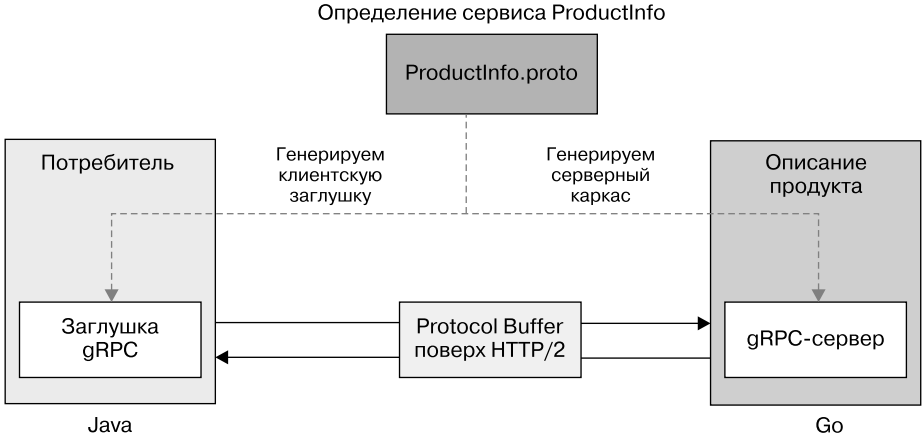


Рис. 1.1. Микросервис и потребитель на основе gRPC

Теперь рассмотрим подробности данного взаимодействия. Первый шаг при создании gRPC-сервиса — определение его интерфейса, содержащее описание методов, которые он предоставляет, а также их входящих параметров и типов возвращаемых значений. Обсудим этот процесс более подробно.

Определение сервиса

В качестве IDL для определения интерфейса сервисов gRPC использует Protocol Buffers (https://oreil.ly/iFi-b) — расширяемый механизм сериализации структурированных данных, который не зависит от языка и платформы (мы подробно рассмотрим его в главе 4, а пока вам достаточно знать, что это средство сериализации). Определение интерфейса сервиса хранится в proto-файле — обычном текстовом файле с расширением `.proto`. gRPC-сервисы описываются в стандартном формате Protocol Buffers; при этом параметры методов RPC и типы возвращаемых значений указываются в виде сообщений. Поскольку определение сервисов — расширение спецификации Protocol

Buffers, генерация кода из вашего proto-файла возможна при наличии специального дополнения gRPC.

В нашем демонстрационном примере интерфейс сервиса `ProductInfo` в формате Protocol Buffers выглядит так, как показано в листинге 1.1. Определение сервиса состоит из определения интерфейса, в котором указаны удаленные методы, их входящие и исходящие параметры, а также определения типов (или форматы сообщений) этих параметров.

Листинг 1.1. Определение gRPC-сервиса ProductInfo с помощью Protocol Buffers

```
// ProductInfo.proto
syntax = "proto3"; ❶
package ecommerce; ❷

service ProductInfo { ❸
    rpc addProduct(Product) returns (ProductID); ❹
    rpc getProduct(ProductID) returns (Product); ❺
}

message Product { ❻
    string id = 1; ❼
    string name = 2;
    string description = 3;
}

message ProductID { ❽
    string value = 1;
}
```

❶ Определение сервиса начинается с указания версии Protocol Buffers (proto3), которую мы используем.

❷ Имена пакетов позволяют предотвратить конфликты имен между типами сообщений и также применяются для генерации кода.

❸ Определение интерфейса gRPC-сервиса.

❹ Удаленный метод для добавления товара, который возвращает ID этого товара в качестве ответа.

❺ Удаленный метод для получения товара по его ID.

❻ Определение формата/типа сообщений `Product`.

❼ Поле (пара «имя — значение»), хранящее ID товара. Обладает уникальным номером, с помощью которого его можно идентифицировать в двоичном формате сообщений.

❽ Пользовательский тип для идентификационного номера товара.

Таким образом, сервис — это набор методов (например, `addProduct` и `getProduct`), которые можно вызывать удаленно. У каждого метода есть тип возвращаемых значений, либо определяющийся в рамках самого сервиса, либо импортирующийся в определение Protocol Buffers.

Входящие и возвращаемые параметры могут иметь пользовательские типы (например, `Product` и `ProductID`) или стандартные типы Protocol Buffers (https://oreil.ly/0Uc3A), описанные в определении сервиса. Тип состоит из сообщений, каждое из которых представляет собой небольшой логический блок информации с набором пар «имя — значение» (мы называем их полями). Каждое поле имеет уникальный номер (например, `string id = 1`), с помощью которого оно идентифицируется в двоичном формате сообщений.

На основе этого определения сервиса генерируется серверная и клиентская части вашего gRPC-приложения. В следующем подразделе мы подробно поговорим о реализации gRPC-сервера.

gRPC-сервер

Созданное вами определение сервиса можно использовать для генерации серверного или клиентского кода (а также стандартного кода Protocol Buffers). Для этого в Protocol Buffers предусмотрен компилятор *protoc*.

На серверной стороне для обработки клиентских вызовов используется gRPC-сервер. Таким образом, чтобы сервис `ProductInfo` мог выполнять свою работу, вы должны сделать следующее.

1. Реализовать логику сервиса на основе сгенерированного каркаса, переопределив его базовый класс.
2. Запустить gRPC-сервер, чтобы принимать запросы от клиентов и возвращать ответы сервиса.

При реализации логики сервиса первым делом нужно сгенерировать его каркас на основе его же определения. Например, в листинге 1.2 можно

видеть удаленные функции, сгенерированные для сервиса `ProductInfo` на языке Go. В теле каждой из этих удаленных функций можно реализовать ее логику.

Листинг 1.2. Серверная реализация сервиса ProductInfo на языке Go

```
import (
  ...
  "context"
  pb "github.com/grpc-up-and-running/samples/ch02/productinfo/go/proto"
  "google.golang.org/grpc"
  ...
)

// реализация ProductInfo на Go

// удаленный метод для добавления товара
func (s *server) AddProduct(ctx context.Context, in *pb.Product) (
      *pb.ProductID, error) {
  // бизнес-логика
}

// удаленный метод для получения товара
func (s *server) GetProduct(ctx context.Context, in *pb.ProductID) (
      *pb.Product, error) {
  // бизнес-логика
}
```

Закончив с реализацией сервиса, вы должны запустить gRPC-сервер, который будет прослушивать клиентские запросы, перенаправлять их сервису и возвращать его ответы клиенту. Фрагмент кода в листинге 1.3 демонстрирует реализацию gRPC-сервера на Go для работы с сервисом `ProductInfo`. Здесь мы открываем TCP-порт, запускаем gRPC-сервер и регистрируем на нем сервис `ProductInfo`.

Листинг 1.3. Запуск gRPC-сервера для сервиса ProductInfo, написанного на Go

```
func main() {
  lis, _ := net.Listen(„tcp", port)
  s := grpc.NewServer()
  pb.RegisterProductInfoServer(s, &server{})
  if err := s.Serve(lis); err != nil {
    log.Fatalf(„failed to serve: %v", err)
  }
}
```

Это все, что необходимо сделать на серверной стороне. Теперь перейдем к реализации клиента gRPC.

gRPC-клиент

С помощью определения сервиса можно сгенерировать не только серверный каркас, но и клиентскую заглушку. Она предоставляет те же методы, что и сервер, но при этом позволяет вызывать клиентский код; далее методы транслируются в удаленные (сетевые) вызовы функций, направленные к серверной стороне. Поскольку определения gRPC-сервиса не зависят от языка, вы можете сгенерировать клиентский и серверный код для любой поддерживаемой среды выполнения (с помощью сторонних реализаций (oreil.ly/psi72)) на ваш выбор. В нашем случае мы генерируем для сервиса `ProductInfo` заглушку на Java и серверный каркас на Go. Клиентский код представлен в листинге 1.4. Независимо от выбранного нами языка, реализация клиентской стороны состоит из установления соединения с удаленным сервером, подключения клиентской заглушки к этому соединению и вызова с ее помощью удаленных методов.

Листинг 1.4. gRPC-клиент для удаленного вызова методов сервиса

```
// создаем канал, используя адрес удаленного сервера
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8080)
   .usePlaintext(true)
   .build();

// инициализируем на основе этого канала блокирующую заглушку
ProductInfoGrpc.ProductInfoBlockingStub stub =
       ProductInfoGrpc.newBlockingStub(channel);

// вызываем удаленный метод с помощью блокирующей заглушки
StringValue productID = stub.addProduct(
       Product.newBuilder()
       .setName("Apple iPhone 11")
       .setDescription("Meet Apple iPhone 11." +
            "All-new dual-camera system with " +
            "Ultra Wide and Night mode.")
       .build());
```

Итак, вы уже хорошо понимаете ключевые концепции gRPC. Теперь попробуем детально разобрать процесс обмена сообщениями между клиентом и сервером.

Обмен сообщениями между клиентом и сервером

Когда gRPC-клиент обращается к gRPC-сервису, его gRPC-библиотека выполняет удаленный вызов процедуры по HTTP/2, используя Protocol Buffers и маршалинг. Серверная сторона распаковывает полученный запрос и вызывает соответствующую процедуру с помощью Protocol Buffers. Затем аналогичным образом сервер возвращает ответ клиенту. Для передачи данных gRPC задействует HTTP/2 — высокопроизводительный двоичный протокол обмена сообщениями с поддержкой двунаправленности. В главе 4 мы обсудим низкоуровневые аспекты передачи сообщений между клиентами и серверами, а также Protocol Buffers и то, как gRPC использует HTTP/2.

Маршалинг — процесс преобразования параметров и удаленной функции в пакет, который отправляется по сети. В результате данный пакет преобразуется в вызов метода.

Прежде чем углубляться в детали протокола gRPC, следует получить общее представление о разных технологиях межпроцессного взаимодействия и истории их развития.

Эволюция межпроцессного взаимодействия

С момента появления методы межпроцессного взаимодействия претерпели радикальные изменения. Мы можем наблюдать возникновение различных методик, направленных на удовлетворение современных нужд и предоставление лучших и более эффективных подходов к разработке. Поэтому важно, чтобы вы хорошо ориентировались в истории развития технологий IPC и понимали, как они в итоге эволюционировали в gRPC. Рассмотрим некоторые из наиболее популярных методик и сравним их с gRPC.

Традиционные подходы к RPC

Технология RPC была популярной разновидностью межпроцессного взаимодействия для создания клиент-серверных приложений. Благодаря ей удаленный вызов функций или методов ничем не отличался от локального. Среди ранних реализаций RPC особой популярностью пользовались CORBA

(Common Object Request Broker Architecture — общая архитектура брокера объектных запросов) и Java RMI (Remote Method Invocation — удаленный вызов методов); они использовались для создания и соединения сервисов и приложений. Однако большинство таких традиционных реализаций RPC страдали от излишней сложности, поскольку были построены поверх коммуникационных протоколов, таких как TCP, имеющих раздутые спецификации, которые затрудняли взаимодействие.

SOAP

Ввиду ограничений, присущих традиционным реализациям RPC наподобие CORBA, крупные корпорации, такие как Microsoft и IBM, разработали и начали активно продвигать SOAP (Simple Object Access Protocol — простой протокол доступа к объектам). Это стандартный подход к взаимодействию в сервис-ориентированной архитектуре (service-oriented architecture, SOA), который позволял сервисам (в контексте SOA их обычно называют веб-сервисами) обмениваться структурированными данными в формате XML; в качестве транспортного протокола чаще всего использовался HTTP.

Технология SOAP позволяла определить интерфейс сервиса и его операции, а также формат сообщений на основе XML, который будет применяться для вызова этих операций. Она была достаточно популярной, однако высокая сложность формата сообщений и спецификации, разработанной вокруг нее, препятствовала гибкому построению распределенных приложений. Поэтому с точки зрения современной разработки распределенных решений веб-сервисы SOAP считаются устаревшей технологией. В наши дни вместо нее обычно используют архитектурный стиль REST.

REST

REST (Representational State Transfer — передача состояния представления) — архитектурный стиль, который впервые предложил Рой Филдинг в своей докторской диссертации (oreil.ly/6tRrt). Филдинг был одним из главных авторов спецификации HTTP. REST — основа ресурсно-ориентированной архитектуры (resource-oriented architecture, ROA); согласно ей распределенные приложения моделируются в виде набора ресурсов, с которыми могут взаимодействовать (читать, создавать, обновлять и удалять) клиенты.

Фактическая реализация REST — протокол HTTP, позволяющий смоделировать приложение, которое состоит из набора REST-ресурсов, доступных по уникальному идентификатору (URL). Для выполнения операций, изменяющих состояние этих ресурсов, используются HTTP-методы (GET, POST, PUT, DELETE, PATCH и т. д.). Состояние ресурса представлено в текстовом формате, таком как JSON, XML, HTML, YAML и т. д.

Построение приложений с помощью архитектурного стиля REST вкупе с HTTP и JSON стало фактическим стандартом для создания микросервисов. Тем не менее, когда таких микросервисов, взаимодействующих по сети, становится слишком много, REST перестает удовлетворять современным требованиям. Он имеет несколько важных ограничений, которые не позволяют использовать его в качестве протокола обмена сообщениями в современных приложениях, основанных на микросервисах.

Неэффективные текстовые форматы и протоколы

В основе REST лежат текстовые транспортные протоколы, такие как HTTP 1.x, и текстовые форматы наподобие JSON, понятные человеку. Это делает взаимодействие сервисов довольно неэффективным, поскольку они обмениваются плохо оптимизированными сообщениями.

Клиентское приложение (отправитель) генерирует двоичные структурированные данные, затем переводит их в текстовый вид (поскольку в HTTP 1.x передаваемые сообщения должны быть текстовыми) и отправляет по сети (по HTTP) серверу (адресату), а тот, в свою очередь, анализирует их и превращает обратно в двоичную структуру. Вместо этого мы могли бы послать сообщение в двоичном формате, который можно привязать к бизнес-логике сервиса и потребителя. Один из популярных аргументов в пользу формата JSON — его проще применять, ведь он понятен человеку. Данная проблема относится скорее к инструментарию, чем к двоичным протоколам.

Нехватка сильно типизированных интерфейсов между приложениями

Современные приложения состоят из большого количества сервисов, взаимодействующих по сети и основанных на совершенно разных технологиях и языках. В таких условиях отсутствие четко заданных и сильно типизированных определений сервисов может стать существенным недостатком. Большинство существующих средств определения сервисов в REST, таких как OpenAPI/Swagger, были созданы задним числом, поэтому им не хва-

тает тесной интеграции с внутренними протоколами обмена сообщениями и архитектурным стилем.

Это приводит к множеству конфликтов, ошибок и проблем с взаимодействием при создании децентрализованных приложений. Например, в ходе разработки REST-сервисов вовсе не обязательно предоставлять их определения и описывать типы данных, которыми они обмениваются. Вместо этого разработчику приходится уже по факту сверяться с форматом полученного текстового сообщения или использовать сторонние технологии определения API, такие как OpenAPI. Поэтому наличие современных средств определения сервисов с поддержкой сильной типизации и фреймворков, которые генерируют основной серверный и клиентский код на разных языках, становится необходимостью.

Разработчикам сложно гарантировать соблюдение архитектурного стиля REST

Для создания REST-сервисов необходимо применять множество «общепринятых методик». Но эти методики не являются частью протоколов (таких как HTTP), поэтому обеспечить соблюдение архитектурного стиля на стадии реализации не слишком просто. Как следствие, большинство сервисов, которые якобы основаны на REST, не соблюдают данный стиль в полной мере. Многие из них на самом деле — всего лишь HTTP-сервисы, доступные по сети. Поэтому разработчикам приходится тратить много времени на то, чтобы сделать свои сервисы согласованными и совместимыми с REST.

Все эти ограничения, возникающие при создании современных облачно-ориентированных приложений на основе традиционных методов IPC, послужили толчком к изобретению более подходящего протокола для обмена сообщениями.

Появление gRPC

В прошлом для соединения тысяч микросервисов, расположенных в разных вычислительных центрах и созданных на основе различных технологий, компания Google использовала RPC-фреймворк общего назначения Stubby (oreil.ly/vat5r). Основной его слой RPC был рассчитан на обработку десятков миллиардов запросов в секунду в масштабах всего Интернета. Несмотря на множество отличных качеств, проект Stubby слишком сильно привязан к внутренней инфраструктуре Google, поэтому так и не стал общепринятым стандартом.

В 2015 году компания Google выпустила (oreil.ly/cUZSG) RPC-фреймворк с открытым исходным кодом под названием gRPC, который предоставляет стандартизированную кросс-платформенную RPC-инфраструктуру общего назначения. Этот проект был создан для того, чтобы сделать масштабируемость, производительность и функциональность, присущие Stubby, доступными широкому кругу пользователей.

На протяжении последних нескольких лет популярность gRPC росла взрывными темпами. Крупномасштабное внедрение этой технологии было проведено такими компаниями, как Netflix, Square, Lyft, Docker, Cisco и CoreOS. Позже она присоединилась (https://oreil.ly/GFffo) к CNCF (Cloud Native Computing Foundation) — одному из самых популярных фондов открытого программного обеспечения, который ставит перед собой задачу сделать облачно-ориентированные вычисления универсальными и стабильными; проект gRPC значительно укрепился благодаря экосистеме CNCF.

Теперь рассмотрим ряд ключевых аспектов gRPC, которые делают эту технологию более привлекательной в сравнении с традиционными протоколами межпроцессного взаимодействия.

Почему стоит выбрать gRPC

gRPC — технология межпроцессного взаимодействия, способная работать в масштабах Интернета и лишенная большинства недостатков, присущих традиционным средствам IPC. Благодаря своим сильным сторонам она все чаще применяется в современных приложениях и серверах. Но что делает ее особенной на фоне многих других коммуникационных протоколов? Обсудим некоторые ключевые преимущества gRPC более подробно.

Преимущества gRPC

Преимущества, предоставляемые технологией gRPC, — залог ее популярности. Некоторые из них перечислены ниже.

- *Высокая эффективность*. Вместо текстовых форматов, таких как JSON или XML, для взаимодействия с сервисами и клиентами gRPC использует двоичный протокол на основе Protocol Buffers. Кроме того, Protocol Buffers в gRPC работает поверх HTTP/2, что еще больше ускоряет межпроцессное взаимодействие. Благодаря этому gRPC — одна из самых эффективных технологий IPC.

- *Простые, четкие интерфейсы и форматы сообщений*. gRPC исповедует контрактный подход к разработке приложений. Сначала определяются интерфейсы сервисов, и только потом начинается работа над их реализацией. Поэтому, в отличие от OpenAPI/Swagger (для REST-сервисов) и WSDL (для веб-сервисов SOAP), gRPC делает процесс разработки приложений простым, но в то же время согласованным, надежным и масштабируемым.
- *Сильная типизация*. Поскольку для определения gRPC-сервисов используется Protocol Buffers, их контракты однозначно определяют типы данных, которыми будут обмениваться приложения. Это делает разработку распределенных приложений куда более стабильной, поскольку статическая типизация позволяет избежать большинства ошибок выполнения и совместимости, возникающих в ситуациях, когда облачно-ориентированные проекты создаются сразу несколькими командами и с помощью разных технологий.
- *Многоязычие*. Протокол gRPC рассчитан на поддержку многих языков программирования. Protocol Buffers позволяет определять gRPC-сервисы без привязки к конкретному языку. Поэтому вы можете взаимодействовать с любым существующим gRPC-сервисом или клиентом, используя тот язык, который вам нравится.
- *Двунаправленная потоковая передача*. gRPC имеет встроенную поддержку потоковой передачи на стороне клиента и сервера, интегрированную непосредственно в сервис. Это существенно упрощает разработку потоковых сервисов и клиентов. Возможность реализовывать как традиционный обмен сообщениями вида «запрос — ответ», так и потоковую передачу данных между клиентом и сервером — ключевое преимущество gRPC перед традиционным архитектурным стилем REST.
- *Встроенные практичные возможности*. gRPC имеет встроенную поддержку таких часто используемых возможностей, как аутентификация, шифрование, устойчивость (крайние сроки, время ожидания), обмен метаданными, сжатие, балансировка нагрузки, обнаружение сервисов и т. д. (мы исследуем их в главе 5).
- *Интеграция с облачно-ориентированными экосистемами*. gRPC входит в фонд CNCF и напрямую поддерживается в большинстве современных фреймворков и технологий. В частности, многие проекты, принадлежащие к CNCF (например, Envoy; oreil.ly/vGQsj), используют gRPC в качестве

коммуникационного протокола. Технология gRPC совместима со многими инструментами, предоставляющими такие стандартные возможности, как сбор метрик и мониторинг (например, для мониторинга gRPC-приложений можно задействовать Prometheus; oreil.ly/AU3-7).

- *Зрелость и широкая распространенность.* Зрелость проекта gRPC обусловлена его активным использованием в Google и внедрением другими крупными технологическими компаниями, такими как Square, Lyft, Netflix, Docker, Cisco и CoreOS.

Как и любая другая технология, gRPC имеет не только преимущества, но и недостатки. Понимание последних в процессе разработки приложений может оказаться весьма полезным. Поэтому рассмотрим некоторые ограничения, характерные для gRPC.

Недостатки gRPC

Ниже перечислен ряд недостатков gRPC, которые необходимо учитывать при выборе этой технологии для разработки своих приложений.

- *gRPC может не подойти для сервисов, доступных снаружи.* Если вы хотите, чтобы ваши приложения или сервисы были доступны снаружи по Интернету, то gRPC будет не самым удачным выбором, поскольку большинство потребителей совсем недавно начали обращать внимание на этот протокол, равно как и на REST/HTTP. Контрактно-ориентированная и сильно типизированная природа gRPC может сделать сервисы, доступ к которым вы открываете, менее гибкими и ограничить контроль со стороны потребителей (в отличие от таких протоколов, как GraphQL, о которых мы поговорим в следующем подразделе). Для преодоления этой проблемы был разработан шлюз gRPC, подробно описанный в главе 8.
- *Кардинальные изменения в определении сервисов требуют больших усилий.* Структурные изменения далеко не редкость при использовании современных средств межсервисного взаимодействия. Если определение gRPC-сервиса меняется слишком сильно, то разработчику обычно следует заново сгенерировать код как для клиента, так и для сервера. Эту процедуру необходимо внедрить в существующий процесс непрерывной интеграции, что может усложнить цикл разработки в целом. Тем не менее большинство изменений, вносимых в определение сервисов, можно применять, не нарушая их контракты. gRPC сможет без каких-либо проблем взаимодействовать с клиентами и серверами, основанными

на другой версии proto-файлов, — главное, чтобы сохранялась обратная совместимость. Таким образом, генерация кода в большинстве случаев не требуется.

- *Относительно небольшая экосистема*. Экосистема gRPC все еще уступает по своему охвату традиционному протоколу REST/HTTP. Поддержка gRPC в браузерах и мобильных приложениях по-прежнему находится в зачаточном состоянии.

Планируя разработку своих приложений, принимайте во внимание эти ограничения. Естественно, gRPC подходит не для всех разновидностей межпроцессного взаимодействия. Вы должны проанализировать свои бизнес-требования и выбрать подходящий протокол обмена сообщениями. Некоторые из рекомендаций о том, как сделать данный выбор, приводятся в главе 8.

Как уже обсуждалось в предыдущем разделе, в области межпроцессного взаимодействия существует множество известных и новых технологий. Вы должны четко представлять сильные и слабые стороны gRPC по сравнению с аналогичными инструментами, завоевавшими популярность в мире разработки современных приложений; это поможет вам выбрать для своих сервисов наиболее подходящий протокол.

Сравнение gRPC с другими протоколами: GraphQL и Thrift

Мы подробно обсудили некоторые ключевые ограничения REST, послужившие причиной создания gRPC. Но вместе с тем мы можем наблюдать появление большого количества новых технологий IPC, пытающихся удовлетворить те же потребности. Ниже мы рассмотрим часть из них и сравним их с gRPC.

Apache Thrift (thrift.apache.org) — RPC-фреймворк (изначально разработанный компанией Facebook и затем переданный фонду Apache), похожий на gRPC. Он использует собственный язык определения интерфейсов и поддерживает большое количество языков программирования. Thrift позволяет описывать типы данных и интерфейсы сервисов в файле определения. Компилятор Thrift принимает этот файл как входные данные и генерирует из него код для клиентской и серверной стороны. Транспортный слой Thrift предоставляет абстракции для сетевого ввода/вывода, что позволяет отделить ваши определения от остальной системы. Это значит, что он умеет работать поверх любого транспортного протокола, включая TCP, HTTP и т. д.

Если сравнить Thrift с gRPC, то оба проекта имеют практически одинаковые архитектурные принципы и сценарии использования. Однако между ними есть несколько различий.

- *Транспортный протокол*. Разработчики gRPC пожертвовали универсальностью в пользу первоклассной поддержки HTTP/2. Применение расширенных возможностей этого протокола позволило повысить эффективность и реализовать потоковый обмен сообщениями.
- *Потоковая передача данных*. Определения gRPC-сервисов имеют встроенную поддержку двунаправленной потоковой передачи данных (между клиентом и сервером).
- *Распространенность и сообщество*. Что касается темпов внедрения, gRPC демонстрирует довольно приличные показатели и обладает хорошей экосистемой, построенной вокруг проектов фонда CNCF. К тому же у сообщества gRPC есть много хороших ресурсов, таких как документация, сторонние презентации и практические примеры приложений, что делает переход на эту технологию более гладким по сравнению с Thrift.
- *Производительность*. У нас нет официальных результатов сравнения gRPC и Thrift, но в Интернете можно найти несколько сайтов, которые оценивают скорость этих двух протоколов, и приводимые цифры говорят о преимуществе Thrift. Тем не менее производительность gRPC тщательно тестируется с выходом почти каждой новой версии (oreil.ly/Hy3mJ), так что данный фактор вряд ли будет решающим при выборе между этими технологиями. Существуют и другие RPC-фреймворки с аналогичными возможностями, но gRPC на сегодняшний день — наиболее стандартизированное, совместимое и широко распространенное решение в данной области.

GraphQL

GraphQL (graphql.org) — еще одна технология для межпроцессного взаимодействия (разработанная компанией Facebook и стандартизированная в качестве открытого протокола), набирающая довольно серьезную популярность[1]. Она представляет собой язык запросов для API и среду выполнения для возвращения в ответ на эти запросы имеющихся данных. GraphQL подходит к проблеме общения между клиентом и сервером совершенно по-другому, позволяя клиентам самим определять, какие данные им нужны и в каком

[1] См.: *Бэнкс А., Порселло Е.* GraphQL: язык запросов для современных веб-приложений. — СПб.: Питер, 2019.

формате их следует вернуть. В свою очередь, gRPC предлагает фиксированный контракт для удаленных методов, на основе которого происходит клиент-серверное взаимодействие.

GraphQL лучше подходит для сервисов и API, доступных непосредственно внешним потребителям, особенно если клиенты хотят лучше контролировать, какие именно данные должен возвращать сервер. Представьте, к примеру, что в нашем гипотетическом интернет-магазине потребителям сервиса `ProductInfo` требуется только часть информации о товарах, а не их полное описание, и при этом им нужен способ выбора подходящих атрибутов. GraphQL позволяет смоделировать сервис таким образом, чтобы потребители могли обращаться к нему за необходимой информацией, используя одноименный язык запросов.

С прагматической точки зрения GraphQL лучше всего применять для сервисов и API, доступных снаружи, тогда как внутренние сервисы, стоящие за этими интерфейсами, лучше реализовывать с помощью gRPC.

Теперь рассмотрим некоторые реальные примеры внедрения gRPC.

gRPC в реальных условиях

Успех любого протокола межпроцессного взаимодействия во многом зависит от его общеотраслевого внедрения, а также от его пользователей и разработчиков. Протокол gRPC широко применяется для создания микросервисов облачно-ориентированных приложений. Обсудим некоторые из ключевых историй успеха gRPC.

Netflix

Netflix (https://www.netflix.com/by/) — компания, предоставляющая потоковое видео по подписке. Она была одним из пионеров в практическом применении микросервисной архитектуры в крупных масштабах. Потребители могут просматривать видео через управляемые сервисы (или API), доступные снаружи, а их работу обеспечивают сотни внутренних сервисов. В связи с этим межпроцессное (или межсервисное) взаимодействие — один из самых важных аспектов данной системы. На начальном этапе реализации микросервисов компания Netflix создала собственный технологический стек для IPC на основе REST-сервисов, работающих поверх HTTP/1.1; это покрывало почти 98 % всех сценариев использования данного продукта.

Но со временем, когда этот подход начал применяться в масштабах Интернета, разработчики стали наблюдать некоторые его ограничения. Многие потребители REST-микросервисов приходилось писать с нуля, анализируя их ресурсы и форматы сообщений. Это занимало много времени, снижало продуктивность разработчиков и повышало риск допущения ошибок в коде. Реализация и потребление сервисов тоже затруднялись ввиду отсутствия технологий для исчерпывающего определения их интерфейсов. Вначале, чтобы обойти большинство из этих проблем, компания Netflix создала внутренний RPC-фреймворк, но после исследования уже готовых технологий выбор пал на gRPC. В ходе исследования обнаружилось, что протокол gRPC выгодно отличался с точки зрения простоты использования и предоставлял все необходимые возможности.

Внедрение gRPC в Netflix привело к огромному скачку продуктивности разработчиков. Например, сотни строчек самописного кода, которые раньше требовались для каждого клиента, были заменены двумя-тремя строчками в proto-файле. Благодаря gRPC на создание клиента уходит несколько минут, а не две или три недели, как раньше. Значительно улучшилась общая стабильность платформы, поскольку программистам больше не нужно вручную реализовывать большинство потребительских возможностей; к тому же теперь у них есть комплексный и безопасный способ определения интерфейсов сервисов. Благодаря повышению производительности снизилась общая латентность платформы Netflix. Похоже, что с момента внедрения gRPC в качестве основного средства межпроцессного взаимодействия некоторые внутренние проекты компании (такие как Ribbon; oreil.ly/qKgv4), реализовывавшие IPC поверх REST и HTTP, прекратили активное развитие и теперь просто поддерживаются в рабочем состоянии.

etcd

etcd (oreil.ly/wo4gM) — надежное распределенное хранилище типа «ключ — значение», предназначенное для хранения наиболее важных данных в распределенных системах. Это один из самых популярных открытых проектов фонда CNCF, который интегрирован во многие другие открытые инструменты, такие как Kubernetes. Один из ключевых факторов успеха протокола gRPC состоит в том, что у него есть простой, четкий и легкий в применении пользовательский API (oreil.ly/v-H-K). И благодаря этому интерфейсу etcd может использовать все преимущества gRPC.

Dropbox

Dropbox — сервис хранения файлов, который предоставляет облачное хранилище, синхронизацию данных, личное облако и клиентское ПО. Dropbox состоит из сотен микросервисов, написанных на разных языках и обменивающихся миллионами запросов каждую секунду. Изначально в нем использовалось несколько RPC-фреймворков, в том числе и самописный, с нестандартными протоколами для ручной сериализации и десериализации, а также Apache Thrift и устаревшая RPC-система на основе протокола HTTP/1.1 и сообщений в формате protobuf.

В итоге разработчики Dropbox полностью перешли на gRPC (что, помимо прочего, позволило им использовать уже готовые форматы сообщений, описанные с помощью Protocol Buffers). Так на свет появился RPC-фреймворк на основе gRPC под названием Courier (oreil.ly/msjcZ). Его целью было не создание нового RPC-протокола, а интеграция gRPC с уже существующей в Dropbox инфраструктурой. Разработчики Dropbox расширили gRPC с целью удовлетворить определенные требования, относящиеся к аутентификации, авторизации, обнаружению сервисов, сбору статистики, журналированию событий и средствам выполнения трассировки.

Эти истории успеха говорят о том, что gRPC — простой протокол межпроцессного взаимодействия, который повышает продуктивность и надежность, позволяя создавать системы, работающие в масштабах Интернета. Это лишь некоторые из наиболее известных первопроходцев в сфере внедрения gRPC. Общее же количество проектов, использующих данный протокол, постоянно растет.

Резюме

Современные распределенные приложения и сервисы редко находятся в изоляции, и методы межпроцессного взаимодействия, которые их объединяют, — один из самых важных их аспектов. gRPC — масштабируемое и типобезопасное решение со слабой связанностью, позволяющее организовывать IPC более эффективно, чем традиционные средства, основанные на REST/HTTP. Благодаря этому возможно соединять, вызывать, администрировать и отлаживать распределенные гетерогенные приложения в стиле, который по своей простоте мало чем отличается от вызова локальных функций по сетевым транспортным протоколам наподобие HTTP/2.

gRPC также можно считать следующим этапом эволюции традиционного подхода к RPC, лишенным прежних ограничений. Данный протокол широко применяется различными интернет-гигантами в качестве средства для межпроцессного взаимодействия и чаще всего используется в целях налаживания связи между внутренними сервисами.

Знания, полученные вами в этой главе, станут хорошим подспорьем для изучения дальнейшего материала, посвященного различным аспектам взаимодействия на основе gRPC. В следующей главе мы применим приобретенные навыки на практике и попытаемся создать с нуля настоящее gRPC-приложение.

ГЛАВА 2

Начало работы с gRPC

Хватит теории! Воспользуемся тем, что вы узнали в главе 1, для создания с нуля настоящего gRPC-приложения. С помощью языков Go и Java мы напишем простой gRPC-сервис и клиент, который будет к нему обращаться. В процессе разработки вы научитесь описывать gRPC-сервисы благодаря Protocol Buffers, генерировать серверный каркас и клиентскую заглушку, реализовывать бизнес-логику сервиса, запускать gRPC-сервер вместе с созданным вами сервисом и обращаться к последнему с помощью клиентского приложения.

Воспользуемся примером интернет-магазина из главы 1 и попробуем создать сервис, отвечающий за управление товарами. Он будет доступен удаленно, и его потребители смогут добавлять в систему новые товары и запрашивать информацию о них по их ID. Все описанное будет смоделировано с помощью gRPC. Для реализации данного примера можно выбрать любой язык, но в этой главе мы будем применять Go и Java.

В репозитории исходного кода для этой книги есть две реализации данного примера: на Go и на Java.

На рис. 2.1 мы показали, как происходит клиент-серверное взаимодействие при вызове каждого метода из сервиса `ProductInfo`. На сервере выполняется gRPC-сервис, который предоставляет два удаленных метода: `addProduct(product)` и `getProduct(productId)`. Любой из них может быть вызван клиентом.

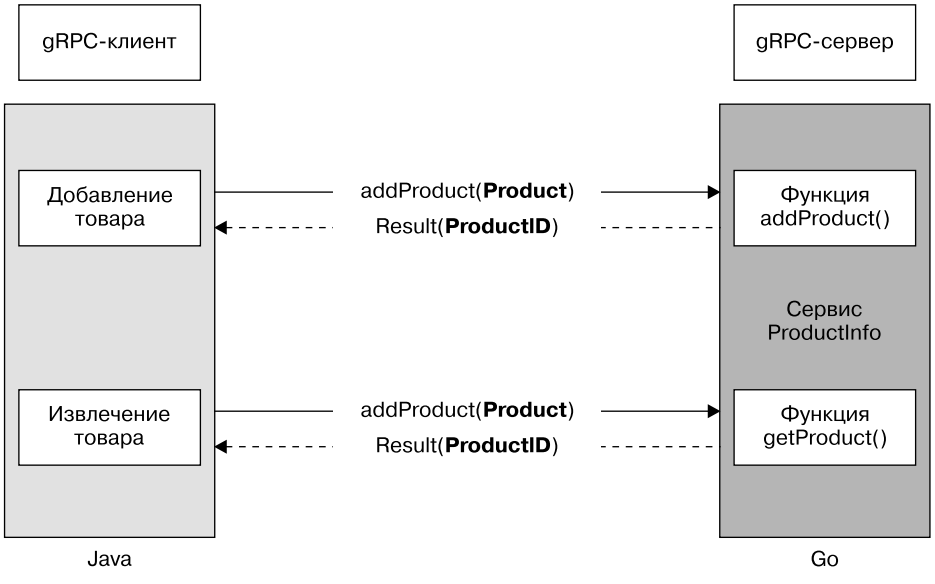


Рис. 2.1. Клиент-серверное взаимодействие сервиса ProductInfo

Начнем реализацию этого примера с определения gRPC-сервиса `ProductInfo`.

Определение сервиса

Как вы уже знаете из главы 1, первым делом при разработке gRPC-приложения нужно определить интерфейс сервиса — то есть описать методы, которые потребитель может вызывать удаленно, их параметры и форматы сообщений и т. д. Это делается с помощью языка определения интерфейсов (interface definition language, IDL) Protocol Buffers (oreil.ly/1X5Ws), используемого в gRPC.

В главе 3 мы более подробно обсудим методы определения сервисов, которые применяются в разных стилях обмена сообщениями. Все, что касается Protocol Buffers и реализации gRPC, будет рассмотрено в главе 4.

Определившись с бизнес-возможностями сервиса, вы можете описать интерфейс, который будет удовлетворять его требованиям. В нашем примере сервис `ProductInfo` будет иметь два удаленных метода, `addProduct(product)`

и `getProduct(productId)`, и два типа сообщений, `Product` и `ProductID`, которые эти методы принимают и возвращают.

Дальше все это нужно описать в формате Protocol Buffers, пригодном для определения как сервисов, так и типов сообщений. Сервис состоит из методов, каждый из которых имеет тип и входящие/исходящие параметры. Сообщение состоит из полей, а у каждого поля есть тип и уникальный индекс. Подробно поговорим о том, как определить структуру сообщения.

Определение сообщений

Сообщение — структура данных, которая передается между клиентом и сервисом. Как видно на рис. 2.1, в примере с `ProductInfo` сообщения бывают двух типов. Первый предназначен для описания товара (`Product`), используется при добавлении новых товаров в систему и возвращении уже существующих. Второй служит уникальным идентификатором (`ProductID`), применяется при извлечении определенного товара из системы и возвращается, когда этот товар добавляют впервые.

- `ProductID` — уникальный идентификатор товара, который может быть строковым значением. Мы можем либо определить собственный тип сообщений со строковым полем, либо использовать стандартный тип `google.protobuf.StringValue`, доступный в библиотеке Protocol Buffers. В этом примере мы выберем первый путь. Определение типа сообщений `ProductID` показано в листинге 2.1.

Листинг 2.1. Определение типа сообщений ProductID в формате Protocol Buffers

```
message ProductID {
   string value = 1;
}
```

- `Product` — наш собственный тип сообщений, который содержит описание товара в нашем интернет-магазине. Может иметь набор полей, связанных со свойствами товара. Мы будем исходить из того, что `Product` состоит из следующих полей:
  - `ID` — уникальный идентификатор товара;
  - `Name` — название товара;
  - `Description` — описание товара;
  - `Price` — цена товара.

Теперь мы можем определить наш тип сообщений с помощью Protocol Buffers, как показано в листинге 2.2.

Листинг 2.2. Определение типа сообщений Product в формате Protocol Buffers

```
message Product {
   string id = 1;
   string name = 2;
   string description = 3;
   float price = 4;
}
```

Номера, присвоенные полям сообщений, используются в целях их идентификации. Поэтому мы не можем указать один и тот же номер для двух полей в определении одного и того же сообщения. Более подробно о методиках определения сообщений в Protocol Buffers и о том, почему у каждого поля должен быть уникальный номер, мы поговорим в главе 4. А пока примите это за правило.

Библиотека protobuf предоставляет набор стандартных типов сообщений. Мы можем использовать их в определении нашего сервиса, вместо того чтобы создавать аналоги. Подробнее об этих типах можно почитать в документации Protocol Buffers (oreil.ly/D8Ysn).

Итак, мы закончили с определением типов сообщений для сервиса `ProductInfo`. Теперь можно перейти к определению его интерфейса.

Определение сервисов

Сервис — набор удаленных методов, доступных клиенту. В нашем примере у сервиса `ProductInfo` есть два таких метода: `addProduct(product)` и `getProduct(productId)`. Согласно правилам, которые действуют в Protocol Buffers, каждый удаленный метод может иметь только один входящий параметр и возвращать лишь одно значение. Если нам нужно передать методу несколько значений, то их необходимо сгруппировать в одном сообщении. Именно так мы поступили с методом `addProduct`, для которого был определен новый тип сообщений `Product`.

- `addProduct` создает в системе новый товар (`Product`). Принимает в качестве входных данных сведения о товаре, и если он был успешно добавлен, то возвращает его идентификационный номер. Определение метода `addProduct` показано в листинге 2.3.

Листинг 2.3. Определение метода addProduct в формате Protocol Buffers

```
rpc addProduct(Product) returns (google.protobuf.StringValue);
```

- `getProduct` извлекает информацию о товаре. Принимает в качестве входных данных `ProductID` и возвращает `Product`, если такой товар присутствует в системе. Определение метода `getProduct` показано в листинге 2.4.

Листинг 2.4. Определение метода getProduct в формате Protocol Buffers

```
rpc getProduct(google.protobuf.StringValue) returns (Product);
```

Если объединить все сообщения и сервисы, то получится полноценное определение `ProductInfo` в формате Protocol Buffers, как показано в листинге 2.5.

Листинг 2.5. Определение gRPC-сервиса ProductInfo в формате Protocol Buffers

```
syntax = "proto3"; ❶
package ecommerce; ❷

service ProductInfo { ❸
    rpc addProduct(Product) returns (ProductID); ❹
    rpc getProduct(ProductID) returns (Product); ❺
}

message Product { ❻
    string id = 1; ❼
    string name = 2;
    string description = 3;
}

message ProductID { ❽
    string value = 1;
}
```

❶ Определение сервиса начинается с указания версии Protocol Buffers (proto3), которую мы используем.

❷ Имена пакетов позволяют предотвратить конфликты имен между типами сообщений и также служат для генерации кода.

❸ Определение интерфейса gRPC-сервиса.

❹ Удаленный метод для добавления товара, который возвращает ID этого товара в качестве ответа.

❺ Удаленный метод для получения товара по его ID.

❻ Определение формата/типа сообщений `Product`.

❼ Поле (пара «имя — значение»), хранящее ID товара. Обладает уникальным номером, с помощью которого его можно идентифицировать в двоичном формате сообщений.

❽ Пользовательский тип для идентификационного номера товара.

В определении Protocol Buffers можно указать имя пакета (например, `ecommerce`), чтобы предотвратить конфликты имен в разных проектах. Если указать этот пакет при генерации кода для наших сервисов и клиентов, используя данное определение, то одноименные пакеты будут созданы для соответствующих языков программирования (конечно, только при наличии в самом языке такого понятия, как *пакет*). В имени пакета можно указать номер версии — например, `ecommerce.v1` или `ecommerce.v2`. Благодаря этому будущие мажорные изменения, вносимые в API, смогут сосуществовать в одной кодовой базе.

Распространенные IDE (integrated development environments — интегрированные среды разработки), такие как IntelliJ IDEA, Eclipse, VSCode и др., содержат дополнения для работы с Protocol Buffers. Установив одно из таких дополнений в своей IDE, вы сможете с легкостью создавать определения собственных сервисов в данном формате.

В этом контексте также следует упомянуть о процессе импорта из других proto-файлов. Если нам нужно использовать уже существующие типы сообщений, то мы можем их импортировать. Например, чтобы применить тип `StringValue` (`google.protobuf.StringValue`), определенный в файле `wrappers.proto`, мы можем импортировать файл `google/protobuf/wrappers.proto` в нашем определении:

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

package ecommerce;
...
```

Закончив с определением, вы можете перейти к реализации своего gRPC-сервиса и его клиента.

Реализация

Реализуем gRPC-сервис, состоящий из удаленных методов, которые мы указали в его определении. gRPC-клиенты смогут обращаться к серверу и вызывать эти методы.

Как показано на рис. 2.2, мы должны сначала скомпилировать определение сервиса `ProductInfo` и сгенерировать исходный код на нужном нам языке. gRPC имеет встроенную поддержку всех популярных языков, включая Java, Go, Python, Ruby, C, C++, Node и т. д. Язык можно выбрать при реализации сервиса или клиента. При этом серверная сторона вашего приложения может быть написана на одном языке, а клиентская — на другом. В данном примере мы реализуем клиент и сервер сразу на Go и на Java, чтобы вы могли выбрать ту реализацию, которая вам нравится больше.

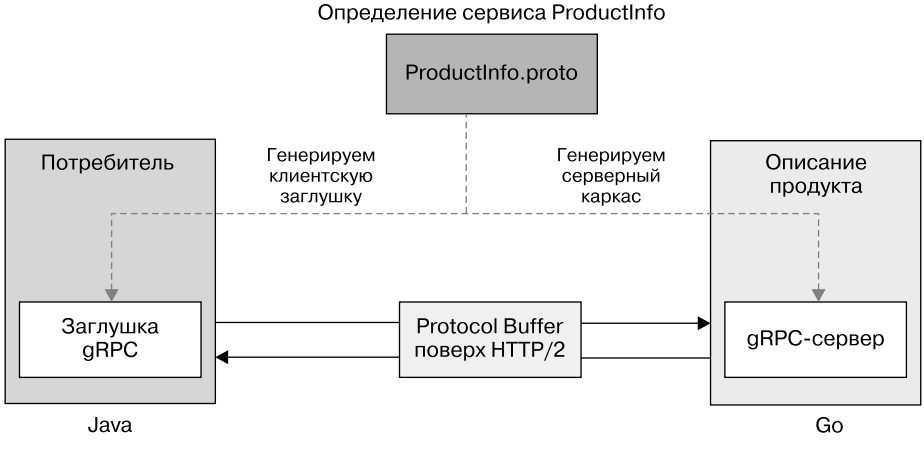


Рис. 2.2. Микросервис и потребитель на основе определения сервиса

Сгенерировать исходный код из определения сервиса можно либо вручную, скомпилировав proto-файл с помощью компилятора Protocol Buffers (oreil.ly/CYEbY), либо путем использования средств автоматизации, таких как Bazel, Maven или Gradle. Последние имеют набор готовых правил для генерации кода во время сборки проекта, и их зачастую легче интегрировать в этот процесс.

В этом примере мы будем использовать инструмент Gradle для сборки Java-приложения и дополнение к нему с целью сгенерировать из proto-файла код для сервиса и клиента. Что касается приложения, написанного на Go, мы задействуем компилятор Protocol Buffers.

Пошагово реализуем gRPC-сервер и клиент на Go и на Java. Но сначала убедитесь в том, что в вашей локальной системе установлены как минимум Java 7 и Go 1.11.

Разработка сервиса

Сгенерированный вами каркас сервиса содержит низкоуровневый код, необходимый для установления gRPC-соединения, а также нужные вам типы сообщений и интерфейсы. Создание сервиса заключается в реализации интерфейса, который был сгенерирован из его определения. Начнем с Go и затем посмотрим, как этот же сервис можно реализовать на языке Java.

Реализация gRPC-сервиса на Go

Реализация сервиса на Go состоит из трех шагов. Сначала нам нужно сгенерировать заглушки для определения сервиса, затем реализовать бизнес-логику всех его удаленных методов, создать сервер, прослушивающий заданный порт, и в конце зарегистрировать сервис для приема клиентских запросов. Начнем с создания нового модуля Go, `productinfo/service`, который будет хранить код сервиса. Внутри этого модуля будет находиться подкаталог ecommerce с автоматически сгенерированным файлом заглушки. Создайте внутри каталога productinfo подкаталог service. Войдите в него и выполните следующую команду, чтобы создать модуль `productinfo/service`:

```
go mod init productinfo/service
```

Создав модуль и подкаталог внутри него, вы получите следующую структуру модуля:

```
└─ productinfo
           └─ service
                 ├── go.mod
                 ├ . . .
                 └── ecommerce
                        └── . . .
```

Нам также нужно указать в файле `go.mod` зависимости с определенными версиями, как показано ниже:

```
module productinfo/service

require (
  github.com/gofrs/uuid v3.2.0
  github.com/golang/protobuf v1.3.2
  github.com/google/uuid v1.1.1
  google.golang.org/grpc v1.24.0
)
```

В Go 1.11 появилась новая концепция под названием «модули», которая позволяет разработчикам создавать и собирать проекты на Go за пределами GOPATH. Чтобы создать модуль Go, нужно войти в каталог, находящийся за пределами $GOPATH/src, и выполнить команду инициализации, указав имя модуля:

```
go mod init <имя_модуля>
```

После инициализации в корневом каталоге модуля появится файл go.mod. Затем внутри этого модуля можно создать и собрать исходный файл. Для поиска импортируемых файлов Go использует зависимости с определенными версиями, перечисленные в файле go.mod.

Генерация заглушек для клиента и сервера. Теперь мы вручную сгенерируем заглушки для клиента и сервера, используя компилятор Protocol Buffers. Для этого необходимо выполнить следующие действия.

- Загрузить и установить самый свежий компилятор Protocol Buffers версии 3 на странице выпусков GitHub (oreil.ly/Ez8qu).

Вы должны выбрать компилятор, который подходит для вашей платформы. Например, если вы используете компьютер с 64-битной версией Linux, а версия компилятора Protocol Buffers имеет вид x.x.x, то вам нужно загрузить файл protoc-x.x.x-linux-x86_64.zip.

- Установите gRPC-библиотеку с помощью следующей команды:

```
go get -u google.golang.org/grpc
```

- Установите дополнение protoc для Go, выполнив такую команду:

```
go get -u github.com/golang/protobuf/protoc-gen-go
```

Подготовив все необходимое, мы можем сгенерировать код для определения сервиса путем выполнения команды `protoc`, как показано ниже:

```
protoc -I ecommerce \ ❶
  ecommerce/product_info.proto \ ❷
  --go_out=plugins=grpc:<module_dir_path>/ecommerce ❸
```

❶ Каталог, в котором хранится исходный proto-файл и его зависимости (указывается с помощью параметра командной строки `--proto_path` или `-I`). Если ничего не указать, то в качестве исходного будет использоваться текущий каталог. Внутри него должны находиться proto-файлы с зависимостями, организованные в соответствии с именем пакета.

❷ Путь к proto-файлу, который вы хотите скомпилировать. Компилятор прочитает этот файл и сгенерирует код на Go.

❸ Каталог, в котором будет сохранен сгенерированный код.

В результате выполнения команды в заданном подкаталоге модуля (`ecommerce`) будет сгенерирован файл заглушки (`product_info.pb.go`). Теперь на основе этого сгенерированного кода необходимо реализовать бизнес-логику.

Реализация бизнес-логики. Для начала создадим новый файл `productinfo_service.go` внутри модуля Go (`productinfo/service`) и реализуем удаленные методы, как показано в листинге 2.6.

Листинг 2.6. Реализация gRPC-сервиса ProductInfo на Go

```
package main

import (
  "context"
  "errors"
  "log"
```

```
"github.com/gofrs/uuid"
pb "productinfo/service/ecommerce" ❶

)

// Сервер используется для реализации ecommerce/product_info
type server struct{ ❷
  productMap map[string]*pb.Product
}

// AddProduct реализует ecommerce.AddProduct
func (s *server) AddProduct(ctx context.Context,
                    in *pb.Product) (*pb.ProductID, error) { ❸❺❻
  out, err := uuid.NewV4()
  if err != nil {
    return nil, status.Errorf(codes.Internal,
        "Error while generating Product ID", err)
  }
  in.Id = out.String()
  if s.productMap == nil {
     s.productMap = make(map[string]*pb.Product)
  }
  s.productMap[in.Id] = in
  return &pb.ProductID{Value: in.Id}, status.New(codes.OK, "").Err()

}

// GetProduct реализует ecommerce.GetProduct
func (s *server) GetProduct(ctx context.Context, in *pb.ProductID)
                              (*pb.Product, error) { ❹❺❻
  value, exists := s.productMap[in.Value]
  if exists {
    return value, status.New(codes.OK, "").Err()
  }
  return nil, status.Errorf(codes.NotFound, "Product does not exist.", in.Value)

}
```

❶ Импортируем пакет со сгенерированным кодом, который мы только что создали с помощью компилятора protobuf.

❷ Структура `server` является абстракцией сервера и позволяет подключать к нему методы сервиса.

❸ Метод `AddProduct` принимает в качестве параметра `Product` и возвращает `ProductID`. Структуры `Product` и `ProductID` определены в файле

product_info.pb.go, который был автоматически сгенерирован из определения product_info.proto.

❹ Метод GetProduct принимает в качестве параметра ProductID и возвращает Product.

❺ У обоих методов также есть параметр Context. Объект Context существует на протяжении жизненного цикла запроса и содержит такие метаданные, как идентификатор конечного пользователя, авторизационные токены и крайний срок обработки запроса.

❻ Помимо итогового значения, оба метода могут возвращать ошибки (методы могут иметь несколько возвращаемых типов). Эти ошибки передаются потребителям и могут быть обработаны на клиентской стороне.

Этого достаточно для реализации бизнес-логики сервиса ProductInfo. Теперь мы можем создать простой сервер, который будет обслуживать наш сервис и принимать запросы от клиента.

Создание сервера на Go. Чтобы получить сервер на языке Go, создадим внутри того же пакета (productinfo/service) новый файл под названием main.go и реализуем в нем метод main, как показано в листинге 2.7.

Листинг 2.7. Реализация gRPC-сервера на Go для обслуживания сервиса ProductInfo

```
package main

import (
  "log"
  "net"

  pb "productinfo/service/ecommerce" ❶
  "google.golang.org/grpc"
)

const (
  port = ":50051"
)

func main() {
  lis, err := net.Listen("tcp", port) ❷
```

```
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
  s := grpc.NewServer() ❸
  pb.RegisterProductInfoServer(s, &server{}) ❹

  log.Printf("Starting gRPC listener on port " + port)
  if err := s.Serve(lis); err != nil { ❺
    log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Импортируем пакет со сгенерированным кодом, который мы только что создали с помощью компилятора protobuf.

❷ TCP-прослушиватель, к которому мы хотим привязать gRPC-сервер, создается на порте 50 051.

❸ Для создания нового экземпляра gRPC-сервера применяются API gRPC на Go.

❹ Используя сгенерированные API, регистрируем реализованный ранее сервис на только что созданном gRPC-сервере.

❺ Начинаем прослушивать входящие сообщения на порте 50 051.

Итак, мы завершили создание gRPC-сервиса на Go для нашего бизнес-сценария. Кроме того, мы создали простой сервер, который сделает методы этого сервиса доступными снаружи и позволит принимать сообщения от gRPC-клиентов.

То же самое можно сделать с помощью Java, если вы предпочитаете данный язык. Весь процесс будет довольно схож с описанным выше. Посмотрим, как это делается, на примере того же сервиса. Если вместо этого вы хотите реализовать клиентское приложение на Go, то можете сразу переходить к подразделу «Разработка gRPC-клиента» на с. 55.

Реализация gRPC-сервиса на Java

При создании gRPC-проекта на Java лучше всего использовать существующие средства сборки, такие как Gradle, Maven или Bazel, поскольку они берут на себя управление всеми зависимостями, генерацию кода и т. д.

Здесь мы будем применять Gradle, поэтому сначала посмотрим, как с помощью этого инструмента создать Java-проект, и затем уже перейдем к реализации бизнес-логики всех удаленных методов сервиса. В завершение мы создадим сервер и зарегистрируем на нем свой сервис, чтобы принимать клиентские запросы.

Gradle — инструмент, автоматизирующий процесс сборки. Он поддерживает множество языков и платформ, включая Java, Scala, Android, C/C++ и Groovy, и тесно интегрирован с такими средствами разработки, как Eclipse и IntelliJ IDEA. Инструкции по установке Gradle можно найти на официальной странице проекта (gradle.org/install).

Подготовка Java-проекта. Сначала создадим Java-проект с помощью Gradle (`product-info-service`). У вас должна получиться такая структура каталогов:

```
product-info-service

 ├── build.gradle
 ├ . . .
 └── src
     ├── main
     │   ├── java
     │   └── resources
     └── test
         ├── java
         └── resources
```

Создайте внутри src/main каталог proto и поместите туда файл с определением сервиса `ProductInfo` (файл `.proto`).

Теперь нужно обновить файл `build.gradle`. Добавьте в него зависимости и дополнение protobuf для Gradle. Итоговый результат показан в листинге 2.8.

Листинг 2.8. Конфигурация Gradle для gRPC-проекта на Java

```
apply plugin: 'java'
apply plugin: 'com.google.protobuf'

repositories {
   mavenCentral()
}
```

```
def grpcVersion = '1.24.1' ❶

dependencies { ❷
   compile "io.grpc:grpc-netty:${grpcVersion}"
   compile "io.grpc:grpc-protobuf:${grpcVersion}"
   compile "io.grpc:grpc-stub:${grpcVersion}"
   compile 'com.google.protobuf:protobuf-java:3.9.2'
}

buildscript {
   repositories {
      mavenCentral()
   }
   dependencies { ❸

       classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.10'
   }
}

protobuf { ❹
   protoc {
       artifact = 'com.google.protobuf:protoc:3.9.2'
   }
   plugins {
       grpc {
           artifact = "io.grpc:protoc-gen-grpc-java:${grpcVersion}"
       }
   }
   generateProtoTasks {
       all()*.plugins {
           grpc {}
       }
   }
}

sourceSets { ❺
   main {
      java {
          srcDirs 'build/generated/source/proto/main/grpc'
          srcDirs 'build/generated/source/proto/main/java'
      }
   }
}

jar { ❻
   manifest {
       attributes "Main-Class": "ecommerce.ProductInfoServer"
   }
```

```
    from {
        configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
    }
}

apply plugin: 'application'

startScripts.enabled = false
```

❶ gRPC-библиотека для Java, которая используется в проекте Gradle.

❷ Внешние зависимости, необходимые нам в этом проекте.

❸ Дополнение protobuf для Gradle, которое мы применяем в этом проекте. Укажите версию 0.7.5, если версия вашего пакета Gradle ниже 2.12.

❹ В настройках дополнения protobuf нужно указать версии компилятора и исполняемого файла для Java.

❺ Это нужно для того, чтобы IDE, такие как IntelliJ IDEA, Eclipse или NetBeans, знали о сгенерированном коде.

❻ Главный класс, который используется для запуска приложения.

Теперь выполним следующую команду, чтобы собрать библиотеку и сгенерировать код заглушки с помощью дополнения protobuf:

```
$ ./gradle build
```

Итак, мы подготовили проект на Java с автоматически сгенерированным кодом. Теперь реализуем интерфейс сервиса и добавим бизнес-логику в удаленные методы.

Реализация бизнес-логики. Для начала создадим в исходном каталоге src/main/java Java-пакет `ecommerce` и поместим внутрь этого пакета новый Java-класс `ProductInfoImpl.java`. Затем реализуем удаленные методы, как показано в листинге 2.9.

Листинг 2.9. Реализация gRPC-сервиса ProductInfo на Java

```
package ecommerce;

import io.grpc.Status;
import io.grpc.StatusException;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

public class ProductInfoImpl extends ProductInfoGrpc.ProductInfoImplBase { ❶

   private Map productMap = new HashMap<String, ProductInfoOuterClass.Product>();

   @Override
  public void addProduct(
       ProductInfoOuterClass.Product request,
      io.grpc.stub.StreamObserver
           <ProductInfoOuterClass.ProductID> responseObserver ) { ❷❹
      UUID uuid = UUID.randomUUID();
      String randomUUIDString = uuid.toString();
      productMap.put(randomUUIDString, request);
      ProductInfoOuterClass.ProductID id =
          ProductInfoOuterClass.ProductID.newBuilder()
             .setValue(randomUUIDString).build();
        responseObserver.onNext(id); ❺
        responseObserver.onCompleted();❻
   }

   @Override
  public void getProduct(
       ProductInfoOuterClass.ProductID request,
       io.grpc.stub.StreamObserver
            <ProductInfoOuterClass.Product> responseObserver ) { ❸❹
       String id = request.getValue();
       if (productMap.containsKey(id)) {
           responseObserver.onNext(
                (ProductInfoOuterClass.Product) productMap.get(id)); ❺
           responseObserver.onCompleted();❻
       } else {
           responseObserver.onError(new StatusException(Status.NOT_FOUND)); ❼
       }
   }
}
```

❶ Наследуем абстрактный класс `ProductInfoGrpc.ProductInfoImplBase`, сгенерированный дополнением. Это позволит нам добавить бизнес-логику в методы `AddProduct` и `GetProduct`, указанные в определении сервиса.

❷ Метод `AddProduct` принимает в качестве параметра `Product` (`ProductInfoOuterClass.Product`). Класс `Product` определен внутри класса `ProductInfoOuterClass`, сгенерированного из определения сервиса.

❸ Метод `GetProduct` принимает в качестве параметра `ProductID` (`ProductInfoOuterClass.ProductID`). Класс `ProductID` определен внутри класса `ProductInfoOuterClass`, сгенерированного из определения сервиса.

❹ Объект `responseObserver` используется для возвращения ответа клиенту и закрытия потока.

❺ Отправляем ответ клиенту.

❻ Закрываем поток, чтобы завершить клиентский вызов.

❼ Возвращаем клиенту ошибку.

Это все, что нужно для реализации на Java бизнес-логики сервиса `ProductInfo`. Теперь мы можем создать простой сервер, который будет обслуживать наш сервис и принимать запросы от клиента.

Создание сервера на Java. Чтобы сделать наш сервис `ProductInfo` доступным снаружи, нам нужно зарегистрировать его на gRPC-сервере. Этот сервер будет прослушивать указанный порт и передавать все запросы подходящему сервису. Создадим внутри пакета главный класс данного сервера, `ProductInfoServer.java`, как показано в листинге 2.10.

Листинг 2.10. Реализация gRPC-сервера на Java для обслуживания сервиса ProductInfo

```
package ecommerce;

import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.io.IOException;

public class ProductInfoServer {

   public static void main(String[] args)
           throws IOException, InterruptedException {
       int port = 50051;
       Server server = ServerBuilder.forPort(port) ❶
               .addService(new ProductInfoImpl())
               .build()
               .start();
```

```
        System.out.println("Server started, listening on " + port);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> { ❷
            System.err.println("Shutting down gRPC server since JVM is " +
                 "shutting down");
            if (server != null) {
                server.shutdown();
            }
            System.err.println("Server shut down");
         }));
         server.awaitTermination(); ❸
    }
}
```

❶ Создаем экземпляр сервера на порте 50 051. Это порт, к которому мы хотим привязать наш сервер и на котором он будет принимать входящие сообщения. Затем на сервер добавляется наша реализация сервиса `ProductInfo`.

❷ Добавляем хук, чтобы gRPC-сервер останавливался вместе с JVM.

❸ Поток сервера продолжает выполняться, пока сервер не завершит работу.

Итак, мы реализовали gRPC-сервис на обоих языках. Теперь можно переходить к реализации gRPC-клиента.

Разработка gRPC-клиента

Теперь, когда у нас есть реализация gRPC-сервиса, можно поговорить о создании приложения, которое будет взаимодействовать с ним. Для начала сгенерируем клиентские заглушки из определения сервиса. На их основе мы создадим простой gRPC-клиент, который станет подключаться к gRPC-серверу и вызывать предоставляемые им удаленные методы.

В данном примере мы напишем клиентские приложения на двух языках: Go и Java. Стоит отметить: для написания клиента и сервера не обязательно использовать один и тот же язык или платформу. gRPC работает в разных языках и системах, так что вы можете выбрать поддерживаемые инструменты. Сначала обсудим реализацию на Go. Если вас интересует, как это делается в языке Java, то можете пропустить следующий пункт и перейти сразу к Java-клиенту.

Реализация gRPC-клиента на Go

Сначала создадим новый модуль Go, `productinfo/client`, и подкаталог ecommerce внутри него. Чтобы реализовать клиентское приложение, нам также нужно сгенерировать заглушку, как мы делали при создании сервиса. Для этого следует создать тот же файл, `product_info.pb.go`, и выполнить те же шаги, поэтому здесь мы пропустим данный этап. Инструкции по генерации файлов с заглушками можно найти в подпункте «Генерация заглушек для клиента и сервера» на с. 45.

Теперь создадим внутри модуля `productinfo/client` файл под названием `productinfo_client.go` и реализуем в нем главный метод, который будет обращаться к удаленным методам сервиса, как показано в листинге 2.11.

Листинг 2.11. Клиентское gRPC-приложение на Go

```
package main

import (
  "context"
  "log"
  "time"

  pb "productinfo/client/ecommerce" ❶
  "google.golang.org/grpc"

)

const (
  address = "localhost:50051"
)

func main() {

  conn, err := grpc.Dial(address, grpc.WithInsecure()) ❷
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close() ❼
  c := pb.NewProductInfoClient(conn) ❸

  name := "Apple iPhone 11"
  description := `Meet Apple iPhone 11. All-new dual-camera system with
              Ultra Wide and Night mode.`
  price := float32(1000.0)
  ctx, cancel := context.WithTimeout(context.Background(), time.Second) ❹
  defer cancel()
```

```
  r, err := c.AddProduct(ctx,
        &pb.Product{Name: name, Description: description, Price: price}) ❺
  if err != nil {
     log.Fatalf("Could not add product: %v", err)
  }
  log.Printf("Product ID: %s added successfully", r.Value)

  product, err := c.GetProduct(ctx, &pb.ProductID{Value: r.Value}) ❻
  if err != nil {
    log.Fatalf("Could not get product: %v", err)
  }
  log.Printf("Product: ", product.String())

}
```

❶ Импортируем пакет со сгенерированным кодом, который мы создали только что с помощью компилятора protobuf.

❷ Устанавливаем соединение с сервером с использованием предоставленного адреса (`"localhost:50051"`). В данном случае соединение между сервером и клиентом будет незащищенным.

❸ Передаем соединение и создаем экземпляр заглушки, который содержит все удаленные методы, доступные на сервере.

❹ Создаем объект `Context`, который будет передаваться вместе с удаленными вызовами. Он содержит метаданные, такие как идентификатор конечного пользователя, авторизационные токены и крайний срок обработки запроса. Этот объект будет существовать до тех пор, пока запрос не будет обработан.

❺ Вызываем метод `addProduct`, передавая ему описание товара. В ответ получим ID новой записи, если все пройдет успешно. В противном случае будет возвращена ошибка.

❻ Вызываем метод `getProduct`, передавая ему ID товара. В ответ получим описание товара, если все пройдет успешно. В противном случае будет возвращена ошибка.

❼ Закончив работу, закрываем соединение.

Итак, мы завершили разработку gRPC-клиента на языке Go. Теперь создадим аналогичный клиент на Java. Данный этап не является обязательным. Если хотите узнать, как в Java создаются gRPC-клиенты, то можете продолжать; в противном случае можете сразу переходить к разделу «Сборка и запуск» на с. 59.

Реализация клиента на Java

Для создания клиентского приложения на Java нужно сначала подготовить проект Gradle (`product-info-client`) и сгенерировать классы с помощью специального дополнения, как мы делали при реализации сервиса. Пожалуйста, следуйте инструкциям, приведенным в подпункте «Подготовка Java-проекта» на с. 50.

Сгенерировав код клиентской заглушки для нашего проекта с помощью средства сборки Gradle, мы можем создать внутри пакета `ecommerce` новый класс под названием `ProductInfoClient` и добавить в него бизнес-логику, как показано в листинге 2.12.

Листинг 2.12. Клиентское gRPC-приложение на Java

```
package ecommerce;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;

import java.util.logging.Logger;

/**
* Пример gRPC-клиента для сервиса productInfo
*/
public class ProductInfoClient {

   public static void main(String[] args) throws InterruptedException {
       ManagedChannel channel = ManagedChannelBuilder
           .forAddress("localhost", 50051) ❶
           .usePlaintext()
           .build();

       ProductInfoGrpc.ProductInfoBlockingStub stub =
              ProductInfoGrpc.newBlockingStub(channel); ❷

       ProductInfoOuterClass.ProductID productID = stub.addProduct( ❸
               ProductInfoOuterClass.Product.newBuilder()
                       .setName("Apple iPhone 11")
                       .setDescription("Meet Apple iPhone 11. " +
                            All-new dual-camera system with " +
                            "Ultra Wide and Night mode.");
                       .setPrice(1000.0f)
                       .build());
      System.out.println(productID.getValue());
```

```
        ProductInfoOuterClass.Product product = stub.getProduct(productID); ❹
        System.out.println(product.toString());
        channel.shutdown(); ❺
    }
}
```

❶ Создаем gRPC-канал, указывая адрес и порт сервера, к которому мы хотим подключиться. Наш сервер находится на том же компьютере и прослушивает порт 50 051. Мы также включили передачу данных в открытом виде; это значит, что соединение между клиентом и сервером будет незащищенным.

❷ Создаем клиентскую заглушку на основе только что созданного канала. Заглушки бывают двух типов: `BlockingStub` и `NonBlockingStub`. Первые ждут получения ответа от сервера, а вторые регистрируют для получения ответов «объект — наблюдатель». Чтобы не усложнять наш пример, мы используем `BlockingStub`.

❸ Вызываем метод `addProduct`, передавая ему описание товара. В ответ получим ID новой записи, если все пройдет успешно.

❹ Вызываем метод `getProduct`, передавая ему ID товара. В ответ получим описание товара, если все пройдет успешно.

❺ Закончив работу, закрываем соединение, чтобы безопасно освободить сетевые ресурсы, которые мы использовали в нашем приложении.

На этом разработка gRPC-клиента закончена. Теперь сделаем так, чтобы он взаимодействовал с сервером.

Сборка и запуск

Пришло время собрать и запустить созданные нами клиентское и серверное gRPC-приложения. Для развертывания и запуска можно использовать локальную систему, виртуальную машину, Docker или Kubernetes. В этом разделе мы обсудим процесс сборки и запуска серверного и клиентского gRPC-приложений на локальном компьютере.

Развертывание и запуск gRPC-приложений в средах Docker и Kubernetes рассматриваются в главе 7.

Запустим в локальной системе серверное и клиентское gRPC-приложения, которые вы только что разработали. Поскольку они написаны на двух языках, серверный код будет собираться отдельно.

Сборка сервера, написанного на Go

В результате реализации сервиса на языке Go итоговая структура каталогов будет выглядеть так:

```
└─ productinfo
           └─ service
                 ├─ go.mod
                 ├─ main.go
                 ├─ productinfo_service.go
                 └─ ecommerce
                    └── product_info.pb.go
```

Теперь мы можем собрать наш сервис и сгенерировать двоичный файл (`bin/server`). Для этого следует зайти в корневой каталог модуля Go (productinfo/service) и выполнить в терминале такую команду:

```
$ go build -i -v -o bin/server
```

В случае успешного завершения сборки исполняемый файл (`bin/server`) можно будет найти в каталоге bin.

Теперь займемся клиентом.

Сборка клиента, написанного на Go

Итоговая структура каталогов клиента, который мы реализовали на языке Go, будет выглядеть так:

```
└─ productinfo
           └─ client
                ├─ go.mod
                ├──main.go
                └─ ecommerce
                       └── product_info.pb.go
```

Клиентский код можно собрать так же, как мы сделали с кодом сервиса, — используя следующую команду:

```
$ go build -i -v -o bin/client
```

В случае успешного завершения сборки исполняемый файл (`bin/client`) можно будет найти в каталоге bin. Теперь запустим собранные нами файлы!

Запуск сервера и клиента, написанных на Go

Запустим клиент и сервер, которые мы собрали, в отдельных терминалах и сделаем так, чтобы они общались друг с другом:

```
// запуск сервера
$ bin/server
2019/08/08 10:17:58 Starting gRPC listener on port :50051

// запуск клиента
$ bin/client
2019/08/08 11:20:01 Product ID: 5d0e7cdc-b9a0-11e9-93a4-6c96cfe0687d
added successfully
2019/08/08 11:20:01 Product: id:"5d0e7cdc-b9a0-11e9-93a4-6c96cfe0687d"
        name:"Apple iPhone 11"
        description:"Meet Apple iPhone 11. All-new dual-camera system with
        Ultra Wide and Night mode."
        price:1000
```

Теперь соберем сервер на языке Java.

Сборка сервера, написанного на Java

Поскольку наш Java-сервис реализован в виде проекта Gradle, мы можем легко его собрать с помощью следующей команды:

```
$ gradle build
```

В результате успешной сборки в каталоге build/libs появится исполняемый JAR-файл (`server.jar`).

Сборка клиента, написанного на Java

Как и в случае с сервисом, для сборки проекта достаточно одной команды:

```
$ gradle build
```

Если сборка пройдет успешно, то в каталоге build/libs можно будет найти исполняемый JAR-файл (`client.jar`).

Запуск сервера и клиента, написанных на Java

Итак, мы собрали клиентский и серверный код, написанный на языке Java. Теперь запустим его:

```
$ java -jar build/libs/server.jar
INFO: Server started, listening on 50051

$ java -jar build/libs/client.jar
INFO: Product ID: a143af20-12e6-483e-a28f-15a38b757ea8 added successfully.
INFO: Product: name: "Apple iPhone 11"
description: "Meet Apple iPhone 11. All-new dual-camera system with
Ultra Wide and Night mode."
price: 1000.0
```

Наш пример успешно запущен на локальном компьютере. Сначала клиентское приложение вызывает метод `addProduct`, передавая ему описание товара, и получает в ответ идентификатор новой записи. Затем, с целью получить описание только что созданного товара, он передает его идентификатор методу `getProduct`. Как упоминалось ранее в данной главе, клиент и сервер могут быть написаны на разных языках. Например, если мы запустим Java-сервер и Go-клиент, то у нас не должно возникнуть никаких проблем.

На этом данную главу можно считать завершенной.

Резюме

При разработке gRPC-приложения нужно сначала определить интерфейс сервиса с помощью Protocol Buffers — расширяемого средства сериализации структурированных данных, не зависящего от языка и платформы. Затем следует сгенерировать серверный и клиентский код для выбранного языка программирования. В результате низкоуровневое взаимодействие будет инкапсулировано, что упростит разработку серверной и клиентской логики. На стороне сервера нужно реализовать методы, с вашего разрешения доступные для удаленного вызова, и запустить gRPC-сервер, который станет обслуживать ваш сервис. На клиентской стороне следует подключиться к удаленному gRPC-серверу и вызвать удаленные методы, используя сгенерированный клиентский код.

Эта глава в основном посвящена практическим аспектам разработки и запуска серверных и клиентских gRPC-приложений. Опыт, который вы приобрели в ходе выполнения приведенных здесь упражнений, пригодится вам при создании реальных gRPC-проектов, поскольку, независимо от выбранного языка, вам придется совершать аналогичные действия. В следующей главе мы продолжим обсуждать способы практического применения изученных вами концепций и технологий.

ГЛАВА 3

Методы взаимодействия на основе gRPC

В первых двух главах вы познакомились с основами методов межпроцессного взаимодействия, которые предлагает gRPC, и получили практический опыт создания простого gRPC-приложения. Вы уже знаете, как определить и реализовать интерфейс сервиса, запустить gRPC-сервер и удаленно вызвать метод сервиса из gRPC-клиента. Взаимодействие клиента и сервера проходит в простом стиле: на каждый запрос клиента приходит ровно один ответ сервера. Тем не менее gRPC поддерживает и другие, более сложные разновидности (стили) IPC.

В этой главе мы исследуем четыре фундаментальных метода взаимодействия, которые применяются в gRPC-приложениях: простой (унарный) RPC, потоковый режим на стороне сервера, потоковый режим на стороне клиента и двунаправленный потоковый режим. Мы продемонстрируем каждый из этих подходов на реальных примерах, определим сервис с помощью языка IDL, используемого в gRPC, и реализуем серверный и клиентский код на Go.

Примеры кода на Go и Java

Чтобы сохранить однородность наших примеров, все фрагменты кода, представленные в текущей главе, написаны на Go. Но если вы программируете на Java, то полноценные примеры на этом языке для тех же сценариев использования можно найти в репозитории кода для данной книги.

Простой (унарный) RPC

Начнем обсуждение методов взаимодействия на основе gRPC с простейшего варианта — *простого RPC* (известного также как *унарный RPC*). При использовании данного подхода клиент, вызывающий удаленную функцию,

отправляет серверу один запрос и получает один ответ, содержащий информацию о состоянии и заключительные метаданные. На самом деле описанное ничем не отличается от метода взаимодействия, с которым вы познакомились в главах 1 и 2. Попробуем еще тщательнее разобраться в этом подходе, прибегнув к реальному сценарию использования.

Представьте, что нам нужно создать для нашего интернет-магазина, основанного на gRPC, сервис `OrderManagement`. В ходе его реализации нам предстоит создать метод `getOrder`, который позволяет клиенту извлекать существующие заказы по их идентификаторам. Как видно на рис. 3.1, клиент отправляет один запрос с ID заказа, а сервер возвращает ему один ответ с информацией о данном заказе. Это взаимодействие происходит по принципу простого RPC.

Рис. 3.1. Простой/унарный RPC

Теперь перейдем к реализации данного подхода. Первым делом нужно создать определение сервиса `OrderManagement` с методом `getOrder`. Как показано в листинге 3.1, сервис можно определить с помощью Protocol Buffers; удаленный метод `getOrder` принимает один запрос с ID заказа и возвращает один ответ, содержащий сообщение `Order`. Последнее имеет структуру, необходимую для представления заказа в этом конкретном случае.

Листинг 3.1. Определение сервиса OrderManagement с методом getOrder по принципу простого RPC

```
syntax = "proto3";

import "google/protobuf/wrappers.proto"; ❶

package ecommerce;
```

```
service OrderManagement {
    rpc getOrder(google.protobuf.StringValue) returns (Order); ❷
}

message Order { ❸
    string id = 1;
    repeated string items = 2;  ❹
    string description = 3;
    float price = 4;
    string destination = 5;
}
```

❶ В этом пакете находятся стандартные типы, такие как `StringValue`.

❷ Удаленный метод для извлечения заказа.

❸ Определяем тип `Order`.

❹ Модификатор `repeated` используется для представления полей, которые могут повторяться в сообщении сколько угодно (или вообще отсутствовать). В нашем случае сообщение `Order` может содержать любое количество заказанных единиц товара.

Затем, применив proto-файл с определением gRPC-сервиса, можно сгенерировать каркас серверного кода и реализовать логику метода `getOrder`. Во фрагменте кода в листинге 3.2 показана реализация сервиса `OrderManagement`. На вход в качестве запроса метод `getOrder` принимает один идентификатор заказа (типа `String`); он ищет этот заказ на стороне сервера и возвращает его в виде структуры типа `Order`. Сообщение `Order` может прийти вместе с `nil` в качестве ошибки; так протокол gRPC будет знать, что мы закончили работу с RPC и заказ можно вернуть клиенту.

Листинг 3.2. Реализация сервиса OrderManagement с методом getOrder на Go

```
// server/main.go
func (s *server) GetOrder(ctx context.Context,
        orderId *wrapper.StringValue) (*pb.Order, error) {
     // Реализация сервиса
        ord := orderMap[orderId.Value]
        return &ord, nil
}
```

Теперь реализуем клиентскую логику, чтобы удаленно вызывать метод `getOrder`. Как и в случае с серверной стороной, вы можете сгенерировать код для нужного вам языка и затем использовать полученную заглушку для обращения к сервису. В листинге 3.3 наш gRPC-клиент на языке Go об-

ращается к сервису `OrderManagement`. В первую очередь, конечно же, следует установить соединение с сервером и вызвать удаленный метод, задействовав метод клиентской заглушки `getOrder`. В качестве ответа вы получите сообщение `Order` с информацией о заказе, которое описано в определении нашего сервиса с помощью Protocol Buffers.

Низкоуровневые аспекты обмена сообщениями по gRPC между сервером и клиентом объясняются в главе 4. Как можно видеть в представленном выше определении, помимо наших параметров метод getOrder принимает объект Context, который передавался в предыдущей реализации сервиса OrderManagement. Данный объект содержит элементы, применяемые для управления поведением gRPC, — например, крайние сроки и механизм отмены запроса. Подробнее об этом читайте в главе 5.

Листинг 3.3. Реализация клиента на Go, которая вызывает удаленный метод getOrder

```
// установление соединения с сервером
...
orderMgtClient := pb.NewOrderManagementClient(conn)
...

// получаем заказ
retrievedOrder , err := orderMgtClient.GetOrder(ctx,
       &wrapper.StringValue{Value: "106"})
log.Print("GetOrder Response -> : ", retrievedOrder)
```

Метод простого RPC довольно прост в реализации и хорошо подходит для большинства сценариев использования межпроцессного взаимодействия. Его реализация мало чем отличается в разных языках программирования. Исходный код для Go и Java можно найти в репозитории к этой книге.

Теперь перейдем к *потоковому RPC на стороне сервера*.

Потоковый RPC на стороне сервера

При взаимодействии сервера и клиента по принципу простого RPC каждый запрос инициирует отдельный ответ. Что касается потокового RPC на стороне сервера, один клиентский запрос генерирует цепочку из нескольких ответов. Эту цепочку называют потоком. После возвращения своего последнего ответа сервер передает клиенту заключительные метаданные с информацией о своем состоянии.

Чтобы лучше понять, как это работает, рассмотрим реальный пример. Допустим, наш сервис `OrderManagement` должен предоставлять возможность поиска по заказам: мы отправляем поисковый запрос и получаем найденные результаты (рис. 3.2). Сервис `OrderManagement` может возвращать подходящие заказы не все сразу, а по мере их обнаружения. Это значит, что в ответ на один запрос клиент получит несколько сообщений.

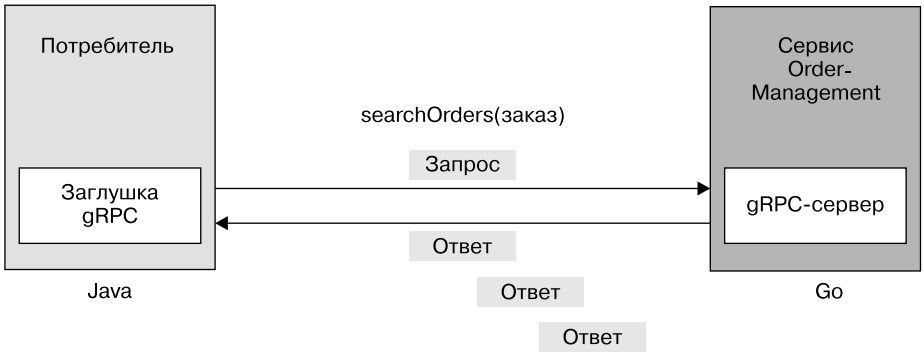


Рис. 3.2. Потоковый RPC на стороне сервера

Теперь добавим в определение нашего gRPC-сервиса `OrderManagement` метод `searchOrder`. Как показано в листинге 3.4, определение этого метода довольно похоже на то, которое применялось в случае с простым RPC, но теперь мы возвращаем заказы в виде *потока*, используя в proto-файле выражение `returns (stream Order)`.

Листинг 3.4. Определение сервиса с потоковым RPC на стороне сервера

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

package ecommerce;

service OrderManagement {
    ...
    rpc searchOrders(google.protobuf.StringValue) returns (stream Order); ❶
    ...
}

message Order {
    string id = 1;
```

```
    repeated string items = 2;
    string description = 3;
    float price = 4;
    string destination = 5;
}
```

❶ Включаем потоковую передачу, возвращая поток сообщений типа `Order`.

Из определения gRPC-сервиса `OrderManagement` можно сгенерировать серверный код и реализовать полученные интерфейсы, чтобы наполнить логикой метод `searchOrder`. В реализации на языке Go, представленной в листинге 3.5, метод `SearchOrders` принимает два параметра: строковое значение `searchQuery` и специальный объект `OrderManagement_SearchOrdersServer`, в который мы будем записывать наши ответы. Этот объект представляет собой ссылку на поток, через который будут возвращаться результаты. Задача бизнес-логики — найти подходящие заказы и отправить их один за другим в виде потока. Каждый обнаруженный заказ записывается в поток с помощью метода `Send(…)`, принадлежащего упомянутому выше объекту. После записи последнего ответа вы можете сообщить и о завершении потока, вернув клиенту `nil`, состояние сервера и заключительные метаданные.

Листинг 3.5. Реализация сервиса OrderManagement с методом searchOrders на Go

```
func (s *server) SearchOrders(searchQuery *wrappers.StringValue,
        stream pb.OrderManagement_SearchOrdersServer) error {

        for key, order := range orderMap {
            log.Print(key, order)
            for _, itemStr := range order.Items {
                log.Print(itemStr)
                if strings.Contains(
                    itemStr, searchQuery.Value) { ❶
                    // Отправляем подходящие заказы в поток
                    err := stream.Send(&order) ❷
                    if err != nil {
                        return fmt.Errorf(
                            "error sending message to stream : %v", err) ❸
                    }
                    log.Print("Matching Order Found : " + key)
                    break
                }
            }
        }
        return nil
}
```

❶ Находим подходящие заказы.

❷ Отправляем найденный заказ в поток.

❸ Проверяем, не возникли ли какие-либо ошибки при потоковой передаче сообщений клиенту.

Удаленный вызов методов на клиентской стороне сильно напоминает простой RPC. Но в данном случае необходимо обрабатывать множественные ответы, поскольку сервер записывает в поток цепочку ответов. Таким образом, в реализации gRPC-клиента на Go (листинг 3.6) мы последовательно извлекаем сообщения из клиентского потока, используя метод `Recv()`, и делаем это до тех пор, пока данный поток не закончится.

Листинг 3.6. Реализация клиента OrderManagement с методом searchOrders на Go

```
// устанавливаем соединение с сервером
...
       c := pb.NewOrderManagementClient(conn)
...
    searchStream, _ := c.SearchOrders(ctx,
       &wrapper.StringValue{Value: "Google"}) ❶

       for {
               searchOrder, err := searchStream.Recv() ❷
               if err == io.EOF { ❸
                       break
               }
           // обрабатываем другие потенциальные ошибки
               log.Print("Search Result : ", searchOrder)
       }
```

❶ Функция `SearchOrders` возвращает клиентский поток `OrderManagement_SearchOrdersClient`, у которого есть метод `Recv`.

❷ Вызываем метод `Recv()` из клиентского потока для последовательного получения ответов типа `Order`.

❸ При обнаружении конца потока `Recv` возвращает `io.EOF`.

Теперь рассмотрим потоковый RPC на стороне клиента, который фактически является противоположностью подхода, описанного в текущем разделе.

Потоковый RPC на стороне клиента

При использовании этого метода клиент отправляет серверу не одно, а несколько сообщений, а тот возвращает один ответ. Но, чтобы ответить, серверу вовсе не обязательно ждать получения всех клиентских запросов. Ответ можно отправить после извлечения из потока одного, нескольких или всех сообщений.

Чтобы понять, как это работает, расширим наш сервис `OrderManagement`. Допустим, вам нужно добавить новый метод для обновления нескольких заказов, `updateOrders` (рис. 3.3). Мы будем передавать список заказов в виде потока сообщений, а сервер станет их обрабатывать и возвращать ответ с состоянием заказов, которые были обновлены.

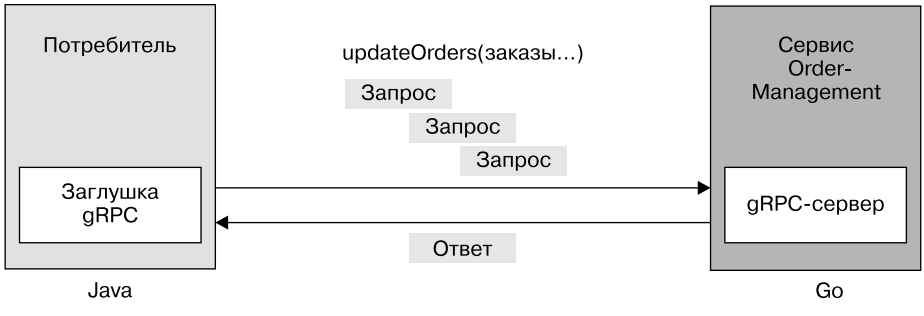


Рис. 3.3. Потоковый RPC на стороне клиента

Добавим метод `updateOrders` в определение сервиса `OrderManagement`, как показано в листинге 3.7. В качестве параметра можно использовать `stream order`; благодаря этому метод `updateOrders` будет знать, что клиент передает ему множественные сообщения. Поскольку сервер отправляет обратно только один ответ, возвращаемое значение будет иметь обычный тип `StringValue`.

Листинг 3.7. Определение сервиса с потоковым RPC на стороне клиента

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

package ecommerce;
```

```
service OrderManagement {
...
    rpc updateOrders(stream Order) returns (google.protobuf.StringValue);
...
}

message Order {
    string id = 1;
    repeated string items = 2;
    string description = 3;
    float price = 4;
    string destination = 5;
}
```

Обновив определение сервиса `OrderManagement`, мы можем сгенерировать серверный и клиентский код. На серверной стороне нужно реализовать интерфейс сгенерированного метода `UpdateOrders`. В реализации на языке Go, представленной в листинге 3.8, `UpdateOrders` принимает объект `OrderManagement_UpdateOrdersServer`, который ссылается на поток сообщений, отправляемых клиентом. Таким образом, сообщения можно читать с помощью метода `Recv()`.

Количество сообщений, которые вам нужно прочитать, зависит от бизнес-логики. Чтобы вернуть ответ, сервису достаточно вызвать из объекта `OrderManagement_UpdateOrdersServer` метод `SendAndClose`; это будет сигнализировать об окончании потока серверных сообщений. Если сервер решит прекратить чтение из клиентского потока, не доходя до его конца, то должен отменить данный поток; таким образом, клиент будет знать, что ему больше не нужно отправлять свои сообщения.

Листинг 3.8. Реализация сервиса OrderManagement с методом updateOrders на Go

```
func (s *server) UpdateOrders(stream pb.OrderManagement_UpdateOrdersServer)
error {

        ordersStr := "Updated Order IDs : "
        for {
            order, err := stream.Recv() ❶
            if err == io.EOF { ❷
                // завершаем чтение потока заказов
                return stream.SendAndClose(
                    &wrapper.StringValue{Value: "Orders processed "
                    + ordersStr})
            }
```

```
            // обновляем заказ
            orderMap[order.Id] = *order

            log.Printf("Order ID ", order.Id, ": Updated")
            ordersStr += order.Id + ", "
        }
}
```

❶ Читаем сообщение из клиентского потока.

❷ Проверяем, не закончился ли поток.

Теперь взглянем на клиентскую реализацию этого подхода. Как видно в листинге 3.9, клиент может отправлять множественные сообщения, используя ссылку на клиентский поток и метод `updateStream.Send`. Закончив передачу сообщений, клиент может просигнализировать о завершении потока и получить ответ от сервиса. Для этого у ссылки на поток предусмотрен метод `CloseAndRecv`.

Листинг 3.9. Реализация клиента OrderManagement с методом updateOrders на Go

```
// устанавливаем соединение с сервером
...
        c := pb.NewOrderManagementClient(conn)
...
    updateStream, err := client.UpdateOrders(ctx) ❶

       if err != nil { ❷
               log.Fatalf("%v.UpdateOrders(_) = _, %v", client, err)
       }

       // обновляем заказ 1
       if err := updateStream.Send(&updOrder1); err != nil { ❸
               log.Fatalf("%v.Send(%v) = %v",
                       updateStream, updOrder1, err) ❹
       }

       // обновляем заказ 2
       if err := updateStream.Send(&updOrder2); err != nil {
               log.Fatalf("%v.Send(%v) = %v",
                       updateStream, updOrder2, err)
       }

       // обновляем заказ 3
       if err := updateStream.Send(&updOrder3); err != nil {
               log.Fatalf("%v.Send(%v) = %v",
                       updateStream, updOrder3, err)
       }
```

```
updateRes, err := updateStream.CloseAndRecv() ❺
if err != nil {
        log.Fatalf("%v.CloseAndRecv() got error %v, want %v",
                updateStream, err, nil)
}
log.Printf("Update Orders Res : %s", updateRes)
```

❶ Вызов удаленного метода `UpdateOrders`.

❷ Обработка ошибок, связанных с `UpdateOrders`.

❸ Отправка обновленных заказов в клиентский поток.

❹ Обработка ошибок, связанных с отправкой сообщений в поток.

❺ Закрытие потока и получение ответа.

В результате вызова этой функции вы получите от сервиса ответное сообщение. Вы уже должны хорошо ориентироваться в потоковом RPC как на стороне сервера, так и на стороне клиента, поэтому перейдем к двунаправленному потоковому режиму, который в некотором роде сочетает оба описанных способа.

Двунаправленный потоковый RPC

В двунаправленном потоковом режиме запрос клиента и ответ сервера представлены в виде потоков сообщений. Вызов должен быть инициирован на клиентской стороне, но дальнейшее взаимодействие зависит лишь от логики клиентского и серверного кода. Чтобы получше разобраться, как это работает, рассмотрим пример. Допустим, нам нужно добавить в наш сервис `OrderManagement` возможность обрабатывать заказы, как показано на рис. 3.4: клиент отправляет непрерывный поток заказов, а сервер объединяет их в партии, в зависимости от адреса доставки.

Этот бизнес-сценарий можно разделить на следующие ключевые этапы.

- Процесс инициируется клиентским приложением, которое устанавливает соединение с сервером и передает ему метаданные (заголовки) вызова.
- Успешно установив соединение, клиентское приложение начинает передачу непрерывного потока идентификаторов заказов, которые должен обработать сервис `OrderManagement`.

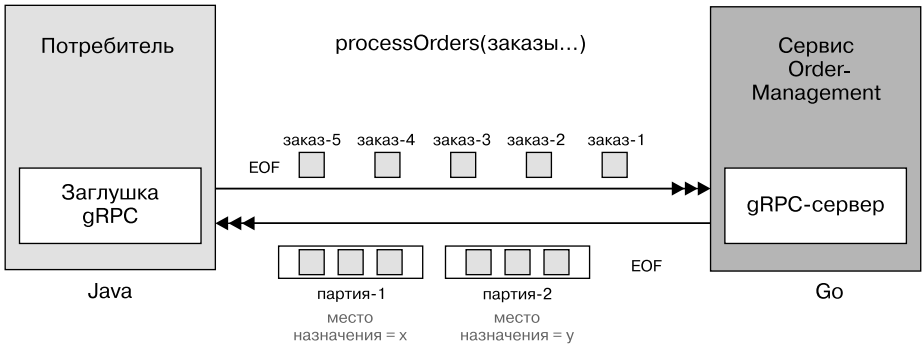


Рис. 3.4. Двунаправленный потоковый RPC

- Каждый ID передается серверу в виде отдельного gRPC-сообщения.
- Сервер обрабатывает все заказы, соответствующие полученным идентификаторам, и объединяет их в партии на основе адреса доставки.
- Партия может содержать несколько заказов, которые должны быть доставлены в одно место назначения.
- Заказы обрабатываются партиями. Когда партия достигает предельного размера, все заказы в текущей партии возвращаются клиенту.
- Например, упорядоченный поток из четырех заказов, два из которых отправляются по адресу X и еще два — по адресу Y, можно обозначить как X, Y, X, Y. Если размер партии равен 3, то заказы следует сгруппировать так: [X, X], [Y] и [Y]. После группирования они возвращаются клиенту в виде потока.

Ключевая идея данного бизнес-сценария состоит в том, что после вызова RPC-метода клиент и сервис могут свободно обмениваться сообщениями (при этом каждая из сторон может завершить поток в любой момент).

Теперь посмотрим, как в этом случае будет выглядеть определение сервиса. В листинге 3.10 показан метод `processOrders`, который принимает в качестве параметра поток строк (представляющий идентификаторы заказов) и возвращает поток объектов `CombinedShipment`. При объявлении входящих и исходящих параметров метода используется модификатор `stream`, что позволяет нам определить двунаправленное потоковое взаимодействие. В определении сервиса также объявляется сообщение со списком сгруппированных заказов.

Листинг 3.10. Определение сервиса для двунаправленного потокового RPC

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

package ecommerce;

service OrderManagement {
    ...
    rpc processOrders(stream google.protobuf.StringValue)
        returns (stream CombinedShipment); ❶
}

message Order { ❷
    string id = 1;
    repeated string items = 2;
    string description = 3;
    float price = 4;
    string destination = 5;
}

message CombinedShipment { ❸
    string id = 1;
    string status = 2;
    repeated Order ordersList = 3;
}
```

❶ В двунаправленном RPC входящие и исходящие параметры объявляются в качестве потоков.

❷ Структура сообщения `Order`.

❸ Структура сообщения `CombinedShipment`.

Теперь из этого обновленного определения сервиса можно сгенерировать серверный код. Сервис `OrderManagement` должен реализовывать метод `processOrders`. В реализации на языке Go, представленной в листинге 3.11, этот метод принимает объект `OrderManagement_ProcessOrdersServer`, который ссылается на поток сообщений между клиентом и сервисом. С помощью этого объекта сервис может читать из потока клиентские запросы и записывать в него свои ответы. В методе `processOrders` сервис одновременно читает из потока входящие сообщения с помощью метода `Recv()` и выполняет запись в тот же поток, используя метод `Send()`.

В целях упрощения в листинге 3.10 опущена часть кода. Полный исходный текст данного примера находится в репозитории кода этой книги.

Листинг 3.11. Реализация сервиса OrderManagement с методом processOrders на Go

```
func (s *server) ProcessOrders(
        stream pb.OrderManagement_ProcessOrdersServer) error {
        ...
        for {
                orderId, err := stream.Recv() ❶
                if err == io.EOF {            ❷
                        ...
                        for _, comb := range combinedShipmentMap {
                                stream.Send(&comb) ❸
                        }
                        return nil                ❹
                }
                if err != nil {
                        return err
                }

                // логика для объединения заказов в партии
                // на основе адреса доставки
                ...
                //

                if batchMarker == orderBatchSize { ❺
                    // передаем клиенту поток заказов, объединенных в партии
                    for _, comb := range combinedShipmentMap {
                        // передаем клиенту партию объединенных заказов
                        stream.Send(&comb)         ❻
                    }
                    batchMarker = 0
                    combinedShipmentMap = make(
                        map[string]pb.CombinedShipment)
                } else {
                        batchMarker++
                }
        }
}
```

❶ Читаем ID заказов из входящего потока.

❷ Продолжаем читать, пока не обнаружим конец потока.

❸ При обнаружении конца потока отправляем клиенту все сгруппированные заказы, которые еще остались.

❹ Сервер завершает поток, возвращая `nil`.

❺ Заказы обрабатываются группами. Когда достигается предельный размер партии, все объединенные заказы возвращаются клиенту в виде потока.

❻ Запись объединенных заказов в поток.

Мы обрабатываем входящие заказы на основе их идентификаторов, и в момент создания новой партии сервис записывает ее в тот же поток (в отличие от потокового RPC на стороне клиента, когда поток закрывается после записи с помощью метода `SendAndClose`). При обнаружении конца клиентского потока сервер закрывает этот поток со своей стороны, возвращая `nil`.

Клиентская реализация (листинг 3.12) тоже имеет много общего с предыдущими примерами. При вызове метода `processOrders` из объекта `OrderManagement` клиент получает ссылку на поток (`streamProcOrder`), с помощью которой он отправляет свои сообщения и читает ответы, возвращаемые сервером.

Листинг 3.12. Реализация клиента OrderManagement с методом processOrders на Go

```
// обработка заказов
streamProcOrder, _ := c.ProcessOrders(ctx) ❶
        if err := streamProcOrder.Send(
                &wrapper.StringValue{Value:"102"}); err != nil { ❷
                log.Fatalf("%v.Send(%v) = %v", client, "102", err)
        }

        if err := streamProcOrder.Send(
                &wrapper.StringValue{Value:"103"}); err != nil {
                log.Fatalf("%v.Send(%v) = %v", client, "103", err)
        }

        if err := streamProcOrder.Send(
                &wrapper.StringValue{Value:"104"}); err != nil {
                log.Fatalf("%v.Send(%v) = %v", client, "104", err)
        }

        channel := make(chan struct{}) ❸
    go asncClientBidirectionalRPC(streamProcOrder, channel) ❹
    time.Sleep(time.Millisecond * 1000) ❺
```

```
        if err := streamProcOrder.Send(
                &wrapper.StringValue{Value:"101"}); err != nil {
                log.Fatalf("%v.Send(%v) = %v", client, "101", err)
        }

        if err := streamProcOrder.CloseSend(); err != nil { ❻
                log.Fatal(err)
}

<- channel

func asncClientBidirectionalRPC (
            streamProcOrder pb.OrderManagement_ProcessOrdersClient,
            c chan struct{}) {
        for {
                combinedShipment, errProcOrder := streamProcOrder.Recv() ❼
                if errProcOrder == io.EOF { ❽
                        break
                }
                log.Printf("Combined shipment : ", combinedShipment.OrdersList)
        }
        <-c
}
```

❶ Вызываем удаленный метод и получаем ссылку на поток для записи и чтения на клиентской стороне.

❷ Отправляем сообщение сервису.

❸ Создаем канал для горутин (goroutines).

❹ Вызываем функцию с помощью горутин, чтобы распараллелить чтение сообщений, возвращаемых сервисом.

❺ Имитируем задержку при отправке сервису некоторых сообщений.

❻ Сигнализируем о завершении клиентского потока (с ID заказов).

❼ Читаем сообщения сервиса на клиентской стороне.

❽ Условие для обнаружения конца потока.

Клиент может отправить свои сообщения сервису и в любой момент закрыть поток. То же самое касается и чтения. В данном примере мы выполняем код для записи и чтения сообщений в двух параллельных потоках, используя механизм *горутин* (goroutines), доступный в языке Go.

Горутины

Горутины в языке Go представляют собой функции или методы, которые выполняются параллельно с другими функциями или методами. Их можно считать легковесными потоками.

Таким образом, клиент может одновременно читать и записывать в один и тот же поток, а входящий и исходящий потоки работают независимо друг от друга. Чтобы продемонстрировать всю мощь двунаправленного RPC, мы использовали довольно сложный пример. Следует понимать, что клиент и сервер могут читать и записывать сообщения в любом порядке, поскольку их потоки никак не связаны. Следовательно, после установления соединения клиент и сервис сами решают, как будет проходить взаимодействие.

На этом мы завершили обсуждение всех существующих методов организации взаимодействия в gRPC-приложениях. Выбор подходящего метода не обусловлен какими-либо жесткими правилами, но вам точно не помешает проанализировать бизнес-сценарии, которые необходимо реализовать.

В заключение подробно поговорим об использовании gRPC для организации взаимодействия микросервисов.

Взаимодействие микросервисов на основе gRPC

Один из основных способов применения gRPC заключается в создании микросервисов и механизмов их взаимодействия. В данном контексте gRPC используется в сочетании с другими коммуникационными протоколами, а сервисы, которые общаются между собой, обычно написаны на разных языках программирования. Чтобы лучше понять, как все работает, рассмотрим расширенную версию интернет-магазина, который мы обсуждали ранее (рис. 3.5).

В этом примере у нас будет ряд микросервисов с определенными бизнес-возможностями. Некоторые из них, такие как `Product`, реализованы с помощью gRPC, а сервис `Catalog`, к примеру, для выполнения своей работы должен обращаться к другим сервисам. Как уже отмечалось в главе 1, gRPC можно использовать в большинстве сценариев с синхронным обменом сообщений. Но при асинхронном взаимодействии, требующем наличия постоянного

хранилища, можно воспользоваться брокерами событий или сообщений, такими как Kafka[1] (kafka.apache.org), Active MQ (activemq.apache.org), RabbitMQ (www.rabbitmq.com) или NATS (http://nats.io). Если вам нужно сделать определенные бизнес-функции доступными снаружи, то можете использовать традиционные сервисы на основе REST/OpenAPI или технологию GraphQL. Итак, `Catalog` и `Checkout` взаимодействуют с внутренними gRPC-сервисами, но в то же время предоставляют внешние интерфейсы, основанные на REST или GraphQL.

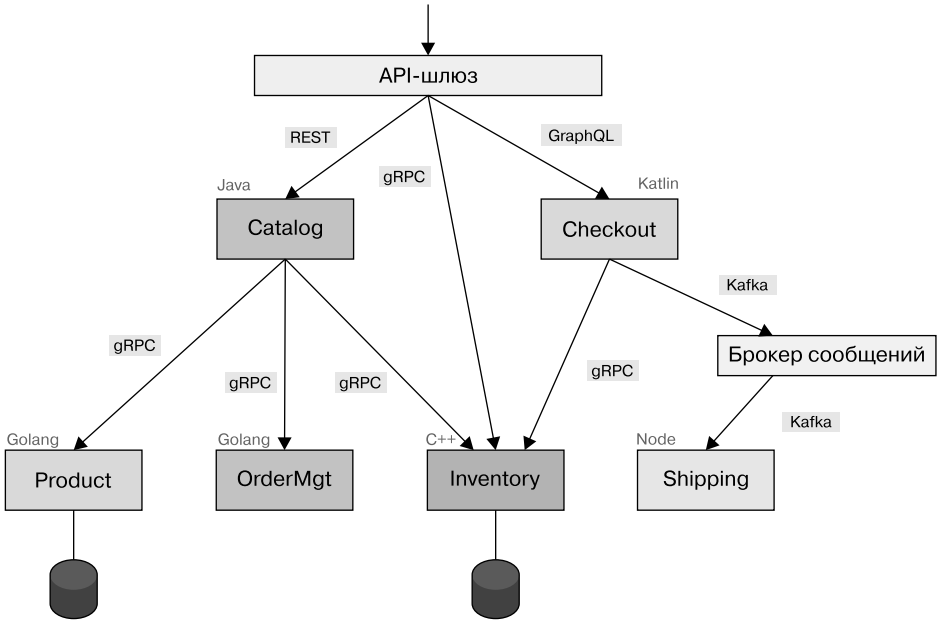


Рис. 3.5. Распространенный метод развертывания микросервисов с использованием gRPC и других протоколов

В большинстве реальных ситуаций наружный доступ к сервисам осуществляется через API-шлюз. Это место, где используются различные нефункциональные механизмы, такие как обеспечение безопасности, ограничение трафика, ведение версий и т. д. Большинство подобных API основаны

[1] См.: *Нархид Н., Шапира Г., Палино Т.* Apache Kafka. Потоковая обработка и анализ данных. — СПб.: Питер, 2020.

на REST или GraphQL. Но можно использовать и gRPC, если ваш шлюз поддерживает данный протокол (хотя так мало кто делает). API-шлюз реализует общесистемные функции, такие как аутентификация, журналирование, ведение версий, ограничение трафика и балансировка нагрузки. Это позволяет вынести все указанные функции за пределы ваших основных gRPC-сервисов. Один из важных аспектов этой архитектуры — возможность использовать разные языки программирования в контексте одних и тех же контрактов между сервисами (то есть генерировать многоязычный код из определения одного gRPC-сервиса). Благодаря этому мы можем выбрать подходящие технологии с учетом бизнес-возможностей наших сервисов.

Резюме

gRPC поддерживает широкий набор стилей организации межпроцессного взаимодействия gRPC-приложений. В данной главе мы исследовали четыре основных подхода. Самый тривиальный из них — простой RPC; это фактически обычный удаленный вызов процедур в стиле «запрос — ответ». Потоковый RPC на стороне сервера позволяет сервису вернуть несколько сообщений в ответ на вызов удаленного метода; потоковый RPC на стороне клиента дает возможность передавать множественные сообщения от клиента к сервису. Мы подробно обсудили реализацию каждого из этих стилей, используя реальные примеры.

Материал, представленный в текущей главе, будет полезен в любых сценариях использования gRPC и позволит вам выбрать метод взаимодействия, наиболее подходящий для ваших задач. Тем не менее мы обошли вниманием низкоуровневые аспекты gRPC, которые видны пользователю. Следующая глава призвана исправить это упущение.

ГЛАВА 4

Внутреннее устройство gRPC

Из предыдущей главы вы уже знаете, что gRPC-приложения общаются по сети с помощью RPC. Разработчику не нужно беспокоиться о подробностях реализации этого взаимодействия, о том, какие методы кодирования сообщений используются внутри и как RPC работает по сети. Имея определение сервиса, вы можете сгенерировать как серверный, так и клиентский код для нужного вам языка. Данный код будет инкапсулировать все низкоуровневые механизмы, предоставляя вам удобные абстракции. Но, если вы разрабатываете сложные системы на основе gRPC и запускаете их в промышленных условиях, то понимать внутреннее устройство этого протокола совершенно необходимо.

В данной главе мы поговорим о том, как в gRPC реализован поток взаимодействия, какие методы кодирования использует эта технология, как она обращается с внутренними сетевыми коммуникациями и т. д. Мы пошагово разберем процесс передачи сообщений при вызове конкретной удаленной процедуры, покажем, как они упаковываются в вызов gRPC, который отправляется по сети, какую роль играет транспортный протокол, как происходит распаковка сообщений на сервере, каким образом выбираются подходящий сервис и удаленная функция и т. д.

Мы также поговорим об использовании Protocol Buffers в качестве средства кодирования и HTTP/2 в качестве транспортного протокола для gRPC. В конце будет рассмотрена архитектура gRPC и основанный на ней стек поддержки языков. В большинстве приложений понимать низкоуровневые аспекты gRPC, с которыми вы здесь познакомитесь, не обязательно, но эти знания могут пригодиться вам при разработке сложных проектов и отладке существующих приложений.

Процесс передачи сообщений в RPC

В RPC-системе сервер реализует набор функций, доступных для удаленного вызова. Клиентское приложение может сгенерировать заглушку с абстракциями, которые можно использовать напрямую для взаимодействия с этими функциями сервера.

Чтобы понять, как происходит удаленный вызов процедур по сети, вернемся к нашему сервису `ProductInfo` из главы 2. Одной из функций, которые мы реализовали в этом сервисе, была `getProduct`; с ее помощью клиент мог получить описание товара, предоставив его ID. Действия, выполняемые в ходе вызова удаленной функции, показаны на рис. 4.1.

Как видно на рис. 4.1, вызов клиентом функции `getProduct` из сгенерированной заглушки можно разделить на следующие ключевые этапы.

1. Клиентский процесс вызывает функцию `getProduct` из сгенерированной заглушки.
2. Клиентская заглушка создает HTTP-запрос типа POST с закодированным сообщением. В gRPC все запросы передаются методом POST и с заголовком `content-type` вида `application/grpc`. Удаленная функция, которая при этом вызывается (`/ProductInfo/getProduct`), передается в отдельном HTTP-заголовке.
3. HTTP-запрос с сообщением отправляется на серверный компьютер по сети.
4. Получив сообщение, сервер анализирует его заголовки, чтобы узнать, какую функцию нужно вызвать, и передает его заглушке сервиса.
5. Заглушка сервиса преобразует байты сообщения в структуры данных конкретного языка.
6. Затем сервис, используя преобразованное сообщение, вызывает локальную функцию `getProduct`.
7. Ответ функции сервиса кодируется и возвращается клиенту. Ответ с сообщением проходит тот же процесс, который мы наблюдали на клиентской стороне (ответ → кодирование → HTTP-ответ, готовый к передаче по сети). Сообщение распаковывается, и его значение возвращается ожидающему клиентскому процессу.

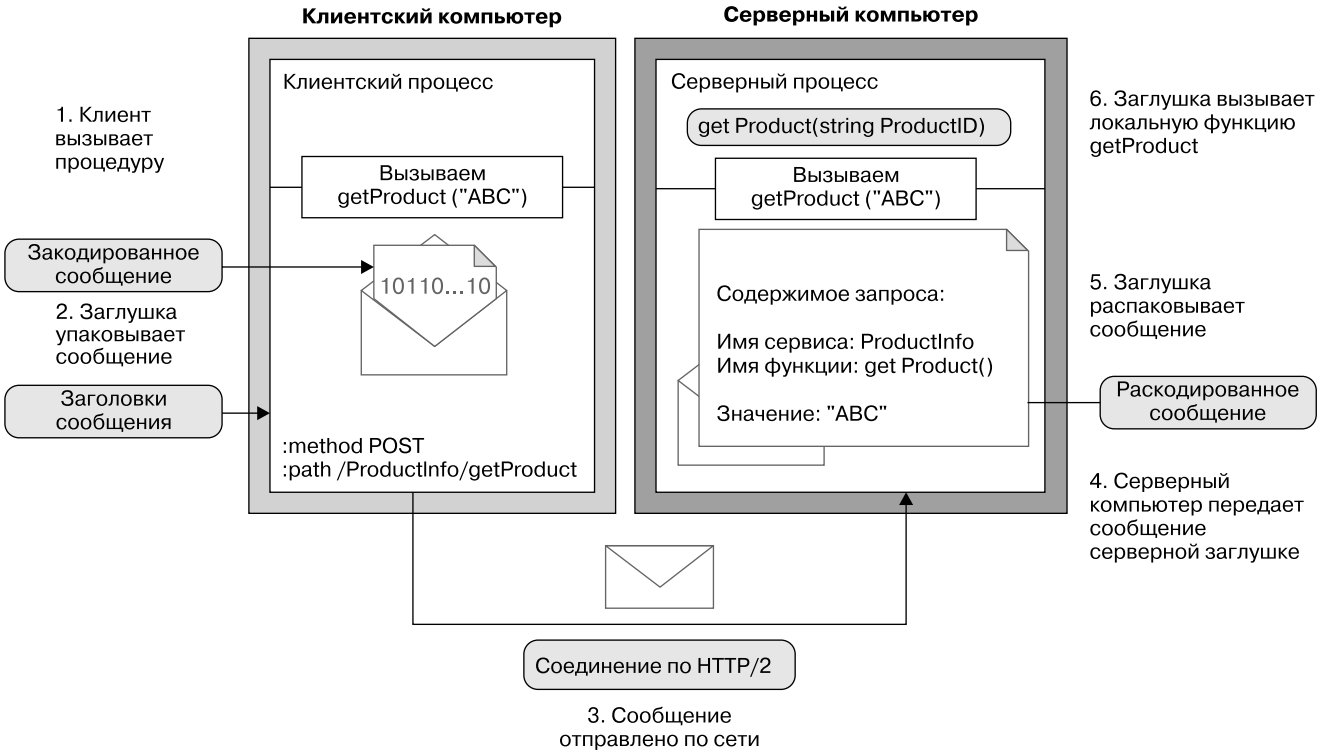


Рис. 4.1. Как выполняется удаленный вызов процедуры по сети

Указанные этапы характерны для большинства систем RPC, таких как CORBA, Java RMI и т. д. Главная особенность протокола gRPC в этом контексте — то, как он кодирует сообщения (см. рис. 4.1). Для кодирования сообщений в gRPC применяется Protocol Buffers (oreil.ly/u9YJI) — расширяемый механизм сериализации структурированных данных, который не зависит от языка и платформы. Один раз определив то, как должны быть структурированы ваши данные, вы затем можете записывать их в разного рода потоки и читать их оттуда, используя специально сгенерированный исходный код.

Поговорим о том, как gRPC применяет Protocol Buffers для кодирования сообщений.

Кодирование сообщений с помощью Protocol Buffers

Как уже обсуждалось в предыдущих главах, для определения сервисов в gRPC используется Protocol Buffers. Чтобы определить сервис, нужно перечислить его удаленные методы и сообщения, которые мы хотим передавать по сети. Возьмем, к примеру, метод `getProduct` из сервиса `ProductInfo`. В качестве входящего параметра он принимает сообщение `ProductID`, а на выходе возвращает сообщение `Product`. Эти входящие и исходящие структуры можно описать с помощью Protocol Buffers, как показано в листинге 4.1.

Листинг 4.1. Определение сервиса ProductInfo с функцией getProduct

```
syntax = "proto3";

package ecommerce;

service ProductInfo {
   rpc getProduct(ProductID) returns (Product);
}

message Product {
    string id = 1;
    string name = 2;
    string description = 3;
    float price = 4;
}

message ProductID {
    string value = 1;
}
```

Как видите, сообщение `ProductID` содержит всего одно поле строкового типа — уникальный ID товара. Сообщение имеет структуру, которая описывает товар. Важно, чтобы оно было правильно определено, поскольку от этого зависит, как оно будет кодироваться. Подробнее об этом мы поговорим позже в данном разделе.

Итак, мы определили сообщение. Теперь посмотрим, как его закодировать и превратить в равнозначный набор байтов. Обычно за это отвечает исходный код, сгенерированный на основе определения сообщения. Для генерации исходного кода используются компиляторы поддерживаемых языков. Передав компилятору подходящее определение, разработчик получает код, с помощью которого можно читать и записывать сообщения.

Представьте, что нам нужно получить информацию о товаре с ID `15`; мы создаем объект сообщения со значением `15` и передаем его функции `getProduct`. В следующем фрагменте кода показано, как создать сообщение `ProductID` со значением, равным `15`, и передать его функции `getProduct`, чтобы получить сведения о товаре:

```
product, err := c.GetProduct(ctx, &pb.ProductID{Value: "15"})
```

Этот фрагмент написан на Go. Определение сообщения `ProductID` находится в сгенерированном исходном коде. Мы создаем экземпляр `ProductID` и присваиваем ему в качестве значения `15`. Точно так же в языке Java мы используем сгенерированные методы, чтобы создать экземпляр `ProductID`:

```
ProductInfoOuterClass.Product product = stub.getProduct(
       ProductInfoOuterClass.ProductID.newBuilder()
               .setValue("15").build());
```

Структура сообщения `ProductID`, показанная ниже, состоит из одного поля `value` с индексом `1`. Когда мы создаем экземпляр сообщения со значением `15`, соответствующее байтовое представление выглядит как идентификатор поля `value`, за которым идет его закодированное значение. Этот идентификатор еще называют *тегом*:

```
message ProductID {
    string value = 1;
}
```

Байтовое представление данной структуры показано на рис. 4.2: каждое поле состоит из идентификатора и закодированного значения.

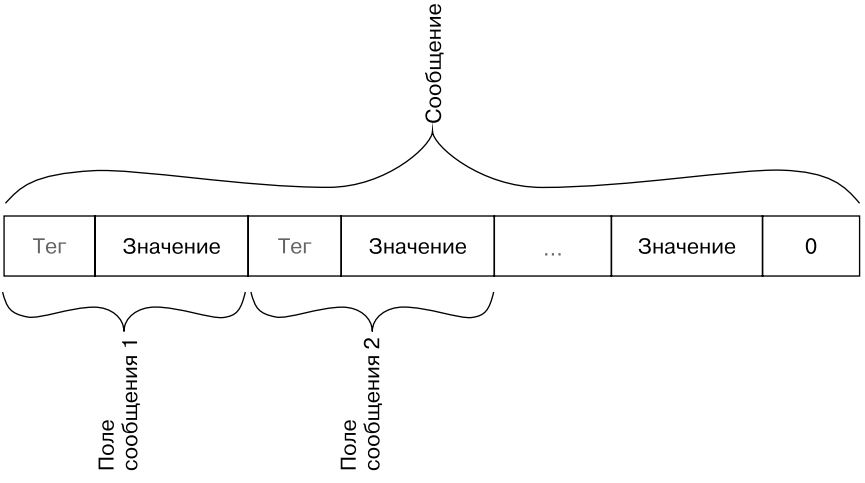


Рис. 4.2. Байтовый поток, закодированный с помощью Protocol Buffers

Данный тег состоит из двух значений: индекса поля и транспортного типа. Индекс — уникальное число, которое присваивается каждому полю в определении сообщения внутри proto-файла. Транспортный тип основан на типе поля и определяет то, какого рода данные могут находиться в этом поле; предоставляемая им информация позволяет узнать длину значения. Транспортные типы и соответствующие типы полей перечислены в табл. 4.1. Это соответствие определено в спецификации Protocol Buffers, и более подробно о нем можно почитать в официальной документации (oreil.ly/xeLBr).

Таблица 4.1. Доступные транспортные типы и соответствующие им типы полей

Транспортный тип	Категория	Типы полей
0	Переменной длины	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-битные	fixed64, sfixed64, double
2	Ограниченной длины	string, bytes, встроенные сообщения, упакованные повторяющиеся поля
3	Начальная группа	Группы (устаревший)
4	Конечная группа	Группы (устаревший)
5	32-битные	fixed32, sfixed32, float

Зная индекс и транспортный тип поля, мы можем определить значение его тега, используя следующую формулу. Здесь мы сдвигаем двоичное представ-

ление индекса на три позиции влево и выполняем его битовое объединение с двоичным представлением транспортного типа:

```
Значение тега = (индекс_поля << 3) | транспортный_тип
```

На рис. 4.3 показано, как индекс и транспортный тип поля размещаются в значении тега.

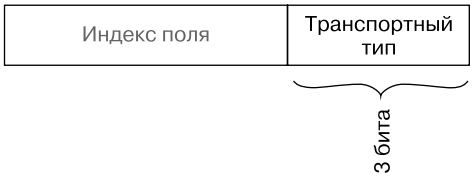


Рис. 4.3. Структура значения тега

Попробуем разобраться в этих терминах с помощью уже знакомого нам примера. Сообщение `ProductID` содержит одно строковое поле с индексом `1` и транспортным типом `2`. В двоичном представлении индекс поля и транспортный тип будут выглядеть как `00000001` и `00000010` соответственно. Если подставить эти значения в формулу, приведенную выше, то значение тега будет равно 10:

```
Значение тега = (00000001 << 3) | 00000010
              = 000 1010
```

Дальше нужно закодировать значение поля. Protocol Buffers использует разные методы кодирования для различных типов данных. Строковое значение кодируется с помощью UTF-8, а целочисленное (int32) — с использованием кодирования переменной длины. Более подробно о разных способах кодирования и о том, в каких ситуациях они применяются, мы поговорим в следующем разделе. А пока вернемся к нашему примеру и закодируем строковое значение.

В формате Protocol Buffers для строковых значений предусмотрена кодировка UTF-8 (формат преобразования Unicode, в котором каждый символ занимает 8 бит). Это метод кодирования символов переменной длины, который преобладает в веб-страницах и электронных письмах.

В нашем примере значение поля `value` в сообщении `ProductID` равно `15`, что в кодировке UTF-8 выглядит как `\x31 \x35`. В UTF-8 длина закодированных значений не является постоянной. Иными словами, для представления

значения может применяться разное количество восьмибитных блоков. В нашем примере используются два блока. Поэтому перед самим закодированным значением нужно указать его длину (количество блоков, которое занимает это значение). Шестнадцатеричное представление 15 в кодировке UTF-8 выглядит следующим образом:

```
A 02 31 35
```

Два крайних справа байта — это строковое значение 15, закодированное в UTF-8. 0x02 представляет длину данного значения в восьмибитных блоках.

Теги и значения закодированного сообщения объединяются в байтовый поток. На рис. 4.2 показано, как значения нескольких полей размещаются в байтовом потоке. Чтобы обозначить его конец, отправляется тег со значением 0.

Теперь вы знаете, как закодировать простое сообщение со строковым полем с помощью Protocol Buffers. Для некоторых типов полей, которые поддерживает Protocol Buffers, применяются разные методики кодирования. Коротко рассмотрим их.

Методики кодирования

В Protocol Buffers поддерживается много разных способов кодирования, предназначенных для различных типов данных. Например, для строковых значений используется кодировка UTF-8, а для чисел типа int32 — кодирование переменной длины. Понимание того, как кодируется то или иное поле, необходимо при определении сообщения; это позволяет выбрать для каждого поля сообщения наиболее подходящий тип и тем самым повысить эффективность кодирования на этапе выполнения.

Типы полей, которые поддерживает Protocol Buffers, делятся на разные группы в зависимости от используемой методики кодирования значений. Ниже перечислены распространенные кодировки, которые применяются в Protocol Buffers.

Кодирование переменной длины

Это метод сериализации целых чисел с помощью одного или нескольких байтов. Основная идея состоит в том, что большинство чисел распределены неравномерно. Поэтому для каждого значения выделяется разное количество

байтов. Как было показано в табл. 4.1, данный метод применяется к группе таких типов полей, как int32, int64, uint32, uint64, sint32, sint64, bool и enum. В табл. 4.2 приводится описание и назначение каждого из этих типов.

Таблица 4.2. Описание типов полей

Тип поля	Описание
int32	Представляет целые числа со знаком в диапазоне от –2 147 483 648 до 2 147 483 647. Обратите внимание: данный тип неэффективен для кодирования отрицательных чисел
int64	Представляет целые числа со знаком в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Обратите внимание: данный тип неэффективен для кодирования отрицательных чисел
uint32	Представляет беззнаковые целые числа со значениями в диапазоне от 0 до 4 294 967 295
uint64	Представляет беззнаковые целые числа со значениями в диапазоне от 0 до 18 446 744 073 709 551 615
sint32	Представляет целые числа со знаком в диапазоне от –2 147 483 648 до 2 147 483 647. Этот тип кодирует отрицательные значения более эффективно, чем int32
sint64	Представляет целые числа со знаком в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Этот тип кодирует отрицательные значения более эффективно, чем int64
bool	Представляет два возможных значения, которые обычно обозначаются как true и false
enum	Представляет набор именованных значений

При кодировании переменной длины старший бит (most significant bit, MSB) каждого байта (кроме последнего) указывает на то, что за ним идут другие байты. Младшие семь бит каждого байта используются для хранения числа, представленного в дополнительном коде (https://ru.wikipedia.org/wiki/Дополнительный_код). Кроме того, младшая группа идет в самом начале, поэтому к ней следует прибавить бит, обозначающий продолжение значения.

Целые числа со знаком

Целые числа со знаком представляют как положительные, так и отрицательные значения. К ним относятся такие типы полей, как sint32 и sint64. Сначала эти значения преобразуются в беззнаковые с помощью кодировки zigzag, а затем к ним применяется метод кодирования переменной длины, рассмотренный выше.

Zigzag привязывает целое число со знаком к беззнаковому числу, чередуя положительные и отрицательные значения (зигзагом). В табл. 4.3 показано, как это работает.

Таблица 4.3. Кодировка zigzag, которая используется для целых чисел со знаком

Исходное значение	Итоговое значение
0	0
–1	1
1	2
–2	3
2	4

Как видите, в оригинале и результате ноль соответствует нулю, а остальные значения привязываются зигзагом к положительным числам. Отрицательные значения становятся нечетными, а положительные — четными. В результате кодирования всегда получается положительное число, независимо от знака исходного значения. Дальше используется кодирование переменной длины.

Для отрицательных целых чисел рекомендуется применять целочисленные типы со знаком, такие как sint32 и sint64, поскольку обычные типы наподобие int32 или int64 преобразуют отрицательные значения в двоичный вид, используя метод кодирования переменной длины. Данный метод требует больше байтов для представления отрицательных чисел по сравнению с положительными. Как следствие, для эффективности отрицательное значение сначала преобразуется в положительное, а затем уже кодируется. Именно этот способ применяется в таких типах, как sint32.

Числа фиксированной длины

Это противоположность типов переменной длины. Любому значению, независимо от его размера, выделяется одно и то же количество байтов. Для представления чисел фиксированной длины Protocol Buffers использует два транспортных типа: один для 64-битных значений, таких как fixed64, sfixed64 и double, а другой — для 32-битных, таких как fixed32, sfixed32 и float.

Строковый тип

В Protocol Buffers строки принадлежат к транспортному типу ограниченной длины. Это значит, что за значением, закодированным методом переменной длины, идет заданное количество байтов с данными. Для строковых значений используется кодировка UTF-8.

Итак, мы провели краткий обзор методик, которые используются для кодирования распространенных типов данных. Подробнее об этом можно почитать на официальной странице Protocol Buffers (oreil.ly/hH_gL).

Но прежде, чем отправлять наше закодированное сообщение по сети, его необходимо обрамить.

Обрамление сообщений с префиксом длины

Если говорить в общих чертах, то обрамление сообщений — процесс организации данных таким образом, чтобы тот, кому они предназначены, мог легко их извлечь. Данный подход используется и в gRPC. Перед отправкой другой стороне закодированную информацию необходимо как следует упаковать. Для этого gRPC применяет обрамление сообщений с префиксом длины.

При использовании этого подхода перед записью каждого сообщения указывается его размер. На рис. 4.4 можно видеть, что перед закодированным двоичным сообщением находится четырехбайтный блок. Поскольку сообщение имеет конечную длину, а этот блок занимает четыре байта, мы знаем, что gRPC поддерживает сообщения длиной до 4 Гбайт.

Как показано на рис. 4.4, Protocol Buffers кодирует сообщение в двоичный формат и указывает в начале его размер с порядком следования байтов от старшего к младшему.

Порядок следования байтов от старшего к младшему (big-endian) — это способ организации данных в системах и сообщениях. Старшее значение в последовательности (с наибольшей степенью двойки) хранится в ячейке памяти с наименьшим адресом.

Помимо размера сообщения, при обрамлении также используется однобайтное беззнаковое целое число, которое говорит о том, являются ли данные сжатыми. Это так называемый флаг сжатия. Если он равен 1, то двоичные данные были сжаты методом, указанным в HTTP-заголовке Message-Encoding; значение 0 говорит о том, что байты сообщения не сжимались. В следующем разделе мы подробно обсудим HTTP-заголовки, которые поддерживает gRPC.

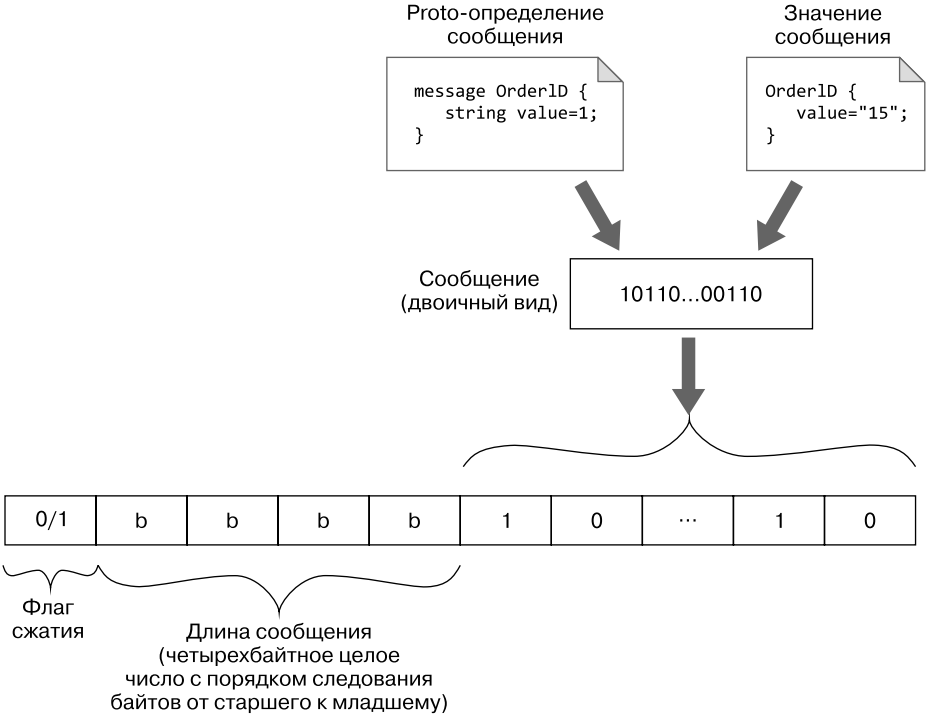


Рис. 4.4. Обрамление с префиксом длины в gRPC-сообщении

После обрамления сообщение готово к отправке по сети. Если это клиентский запрос, то получателем будет сервер. Если ответ, то получателем будет клиент. Получив сообщение, мы должны прочитать его первый байт, чтобы понять, является ли оно сжатым. Следующие четыре байта позволят узнать размер закодированного двоичного сообщения — то есть сколько байтов нам нужно прочитать из потока. В унарном/простом RPC нам предстоит обработать только одно сообщение, а при потоковой передаче — несколько.

Теперь вы должны хорошо ориентироваться в том, как сообщения подготавливаются для передачи по сети. В следующем разделе мы обсудим, как, собственно, передаются эти сообщения с префиксом длины. На сегодня ядро gRPC поддерживает три транспортных механизма: HTTP/2, Cronet (https://oreil.ly/D0laq) и in-process (https://oreil.ly/lRgXF). Среди них для отправки сообщений чаще всего используется HTTP/2. Посмотрим, как gRPC применяет этот протокол для эффективного обмена данными.

gRPC поверх HTTP/2

HTTP/2 — вторая мажорная версия интернет-протокола HTTP. Она призвана исправить некоторые проблемы с безопасностью и производительностью, присущие предыдущей версии (HTTP/1.1). HTTP/2 поддерживает все основные возможности HTTP/1.1, но делает это более эффективно. В результате приложения, написанные с применением данного протокола, получаются более быстрыми, простыми и надежными.

gRPC использует HTTP/2 в качестве своего транспортного протокола для передачи сообщений по сети. Это одна из причин, почему gRPC является высокопроизводительным RPC-фреймворком. Поговорим о том, как соотносятся обе технологии.

В HTTP/2 все взаимодействие клиента и сервера выполняется в рамках одного TCP-соединения, по которому может передаваться любое количество двунаправленных байтовых потоков. Чтобы понять, как это работает, нужно сначала разобраться в терминологии данного протокола:

- поток — двунаправленная передача данных в рамках установленного соединения; может передавать любое количество сообщений;
- фрейм — наименьшая единица взаимодействия в HTTP/2. Принадлежность каждого фрейма к тому или иному потоку (и, возможно, другая информация) определяется в его заголовке;
- сообщение — полная последовательность фреймов, которая относится к одному логическому HTTP-сообщению. Благодаря этому клиент и сервер могут мультиплексировать сообщения: разбивать их на отдельные фреймы, перемешивать и затем собирать на другом конце соединения.

Как видно на рис. 4.5, канал gRPC представляет соединение с конечной точкой (то есть соединение по HTTP/2). Когда клиентское приложение создает такой канал, gRPC внутри устанавливает HTTP/2-соединение с сервером. Один и тот же канал можно использовать для отправки серверу любого количества удаленных вызовов. Они привязываются к потокам HTTP/2. Сообщения передаются в виде HTTP/2-фреймов. Фрейм может вмещать

одно сообщение с префиксом длины; если же сообщение слишком большое, то может занять несколько фреймов.

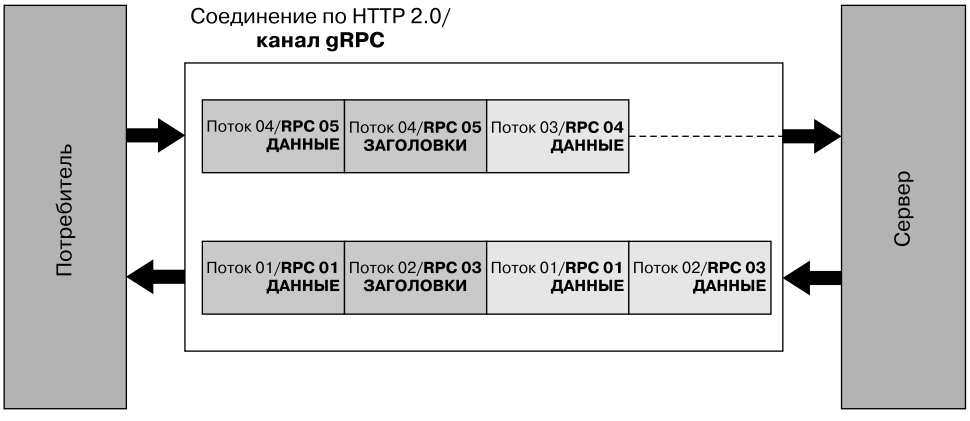


Рис. 4.5. Семантическое соответствие между gRPC и HTTP/2

В предыдущем разделе мы обсуждали обрамление сообщений с префиксом длины. Когда такие сообщения передаются по сети в качестве запросов и ответов, к ним необходимо прилагать дополнительные заголовки. Поговорим о том, как структурировать запросы/ответы и какие заголовки при этом нужно передавать.

Запрос

Запрос — это сообщение, которое инициирует удаленный вызов. В gRPC запрос всегда отправляется клиентским приложением и содержит три основных компонента: заголовок, сообщение с префиксом длины и флаг, обозначающий конец потока (рис. 4.6). Удаленный вызов инициируется, когда клиент отправляет заголовки запроса. Затем передаются сами сообщения. В конце отправляется флаг EOS (end of stream — «конец потока»), который оповещает получателя о том, что передача запроса завершена.

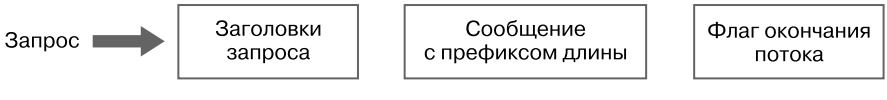


Рис. 4.6. Порядок отправки элементов запроса

Воспользуемся уже знакомой нам функцией `getProduct` из сервиса `ProductInfo` и покажем отправку запроса по протоколу HTTP/2. Чтобы инициировать вызов функции, клиент отправляет следующие заголовки:

```
HEADERS (flags = END_HEADERS)
:method = POST ❶
:scheme = http ❷
:path = /ProductInfo/getProduct ❸
:authority = abc.com ❹
te = trailers ❺
grpc-timeout = 1S ❻
content-type = application/grpc ❼
grpc-encoding = gzip ❽
authorization = Bearer xxxxxx ❾
```

❶ Определяет HTTP-метод. В случае с gRPC заголовок `:method` всегда равен `POST`.

❷ Определяет режим взаимодействия. Если включен протокол защиты транспортного уровня (Transport Level Security, TLS), то значение равно `https`; если нет — то `http`.

❸ Определяет путь конечной точки. В случае с gRPC это значение имеет вид `/{имя сервиса}/{имя метода}`.

❹ Определяет доменное имя в URI-адресе.

❺ Определяет способ обнаружения несовместимых прокси-серверов. В случае с gRPC это значение должно равняться `trailers`.

❻ Определяет время ожидания вызова. Если его не указать, то сервер должен считать, что оно не ограничено.

❼ Определяет тип содержимого. В случае с gRPC это значение должно быть равно `application/grpc`, иначе gRPC-сервер вернет HTTP-код 415 (неподдерживаемый тип данных).

❽ Определяет способ сжатия сообщения. Допустимые значения: `identity`, `gzip`, `deflate`, `snappy` и `{custom}`.

❾ Это необязательные метаданные. Заголовок `authorization` используется для доступа к защищенной конечной точке.

Несколько замечаний по данному примеру:

- заголовки, названия которых начинаются с :, являются зарезервированными и должны идти перед обычными;
- заголовки, участвующие во gRPC-взаимодействии, делятся на две категории: определение вызова и пользовательские метаданные;
- заголовки, из которых состоит определение вызова, входят в стандарт HTTP/2 и должны отправляться перед пользовательскими метаданными;
- пользовательские метаданные — произвольные пары вида «ключ — значение», которые определяются на уровне приложения. Названия заголовков, в которых передаются эти метаданные, зарезервированы в ядре gRPC и должны начинаться с grpc-.

Инициировав вызов, клиент отправляет серверу сообщения с префиксом длины в виде HTTP/2-фреймов. Если сообщение слишком большое, то может занимать больше одного фрейма. Конец запроса обозначается флагом `END_STREAM` в последнем фрейме `DATA`. Если данных больше не осталось, но нам нужно закрыть поток запроса, то флаг `END_STREAM` можно послать в пустом фрейме:

```
DATA (flags = END_STREAM)
<сообщение с префиксом длины>
```

Это всего лишь краткий обзор структуры запросов в gRPC. Подробную информацию можно найти в официальном репозитории gRPC на сайте GitHub (https://oreil.ly/VIhYs).

Сообщение, содержащее ответ, тоже имеет свою структуру. Посмотрим, как оно устроено и какие заголовки передает.

Ответ

Ответ — это сообщение, возвращаемое сервером после получения клиентского запроса. Как и в предыдущем случае, большинство ответов состоит из трех основных компонентов: заголовков, сообщений с префиксом длины и заключительных блоков. Но, как видно на рис. 4.7, некоторые ответы, возвращаемые клиенту, могут не содержать сообщений с префиксом длины.

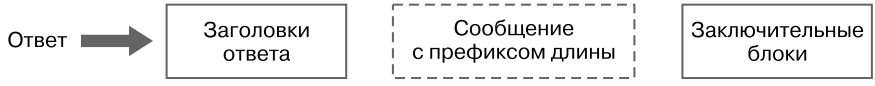


Рис. 4.7. Порядок отправки элементов ответа

Чтобы разобраться в последовательности обрамления ответа в HTTP/2, рассмотрим тот же пример. В первую очередь сервер возвращает клиенту заголовки ответа, как показано ниже:

```
HEADERS (flags = END_HEADERS)
:status = 200 ❶
grpc-encoding = gzip ❷
content-type = application/grpc ❸
```

❶ Обозначает код состояния HTTP-ответа.

❷ Определяет способ сжатия сообщения. Допустимые значения: `identity`, `gzip`, `deflate`, `snappy` и `{custom}`.

❸ Определяет тип содержимого. В случае с gRPC это значение должно быть равно `application/grpc`.

Ответ, как и запрос, может содержать заголовки с пользовательскими метаданными в виде пар «ключ — значение», определяемых на уровне приложения.

Вслед за заголовками сервер отправляет сообщения с префиксом длины, представленные в виде HTTP/2-фреймов `DATA`. Как и в случае с запросом, если сообщение слишком большое, то может занимать больше одного фрейма. Ниже вы можете видеть, что флаг `END_STREAM` возвращается не во фрейме `DATA`, а в отдельном заголовке — так называемом заключительном блоке:

```
DATA
<сообщение с префиксом длины>
```

В конце клиент получает заключительные блоки, сигнализирующие о том, что сервер закончил передачу. Помимо прочего, эти блоки содержат код и сообщение о состоянии ответа:

```
HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK ❶
grpc-message = xxxxxx ❷
```

❶ Определяет код состояния. В gRPC использует набор стандартных кодов, описание которых можно найти в официальной документации (https://oreil.ly/3MH72).

❷ Определяет описание ошибки. Это необязательный заголовок. Он устанавливается только в тех случаях, когда во время обработки запроса возникла ошибка.

Заключительные блоки тоже возвращаются в виде заголовочных фреймов протокола HTTP/2, но в конце ответа. Конец ответного потока обозначается флагом END_STREAM, установленным в заключительном блоке. Вместе с ним могут также передаваться заголовки grpc-status и grpc-message.

Иногда неудачное завершение запроса происходит в самом начале. В подобных случаях серверу нужно вернуть ответ без фреймов `DATA`. В результате клиент получает только заключительные блоки, которые тоже имеют вид заголовочных фреймов протокола HTTP/2 и содержат флаг `END_STREAM`. В заключительном блоке могут также находиться следующие заголовки:

- HTTP-Status → `:status`;
- Content-Type → `content-type`;
- Status → `grpc-status`;
- Status-Message → `grpc-message`.

Теперь вы знаете, как gRPC-сообщение передается по HTTP/2-соединению. Рассмотрим процесс передачи сообщений в gRPC при использовании разных методов взаимодействия.

Передача сообщений с помощью разных методов взаимодействия на основе gRPC

В предыдущей главе мы обсудили четыре метода взаимодействия, которые поддерживает gRPC: простой RPC, потоковый RPC на стороне сервера, потоковый RPC на стороне клиента и двунаправленный потоковый RPC.

Мы объяснили, как они работают, на примере реальных сценариев применения. В этом подразделе мы еще раз рассмотрим эти четыре метода, но уже под другим углом. Поговорим об их поведении на транспортном уровне, используя знания, приобретенные в данной главе.

Простой RPC

В простом RPC серверное и клиентское gRPC-приложения всегда обмениваются единичными запросами и ответами. Как видно на рис. 4.8, запрос содержит заголовки и сообщение с префиксом длины, которое может занимать один или несколько фреймов. Чтобы закрыть поток на своей стороне и сообщить об окончании запроса, клиент отправляет флаг EOS (end of stream — «конец потока»). Когда соединение становится «полузакрытым», клиент больше не может передавать сообщения, но у него все еще остается возможность принимать ответы от сервера. Тот генерирует свой ответ только после получения всего запроса. Ответ содержит заголовочный фрейм, за которым следует сообщение с префиксом длины. Взаимодействие завершается, когда сервер возвращает заключительный заголовок с подробностями о состоянии.

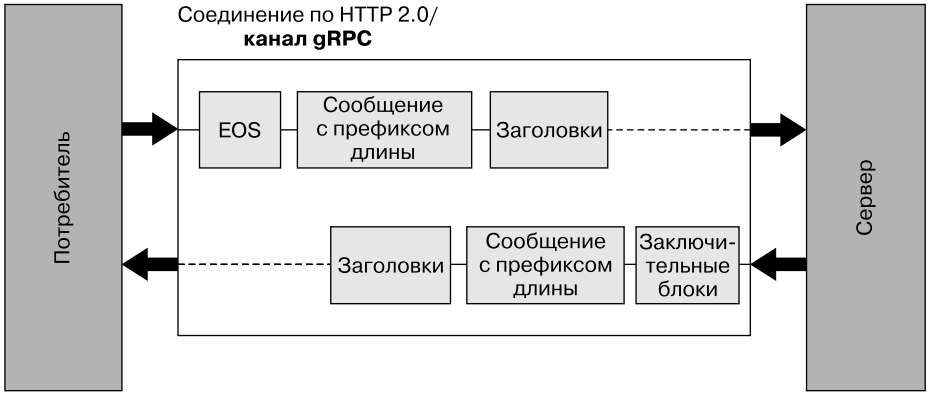


Рис. 4.8. Простой RPC: процесс обмена сообщениями

Это простейший метод взаимодействия. Перейдем к более сложному сценарию: потоковому RPC на стороне сервера.

Потоковый RPC на стороне сервера

С точки зрения клиента простой RPC и потоковый RPC на стороне сервера используют один и тот же способ передачи сообщений. В обоих случаях отправляется только один запрос. Хотя в ответ сервер возвращает не одно сообщение, а несколько. Дождавшись прихода всего запроса, он отправляет заголовки ответа и множественные сообщения с префиксом длины, как показано на рис. 4.9. Взаимодействие завершается, когда сервер возвращает заключительный заголовок с подробностями о состоянии.

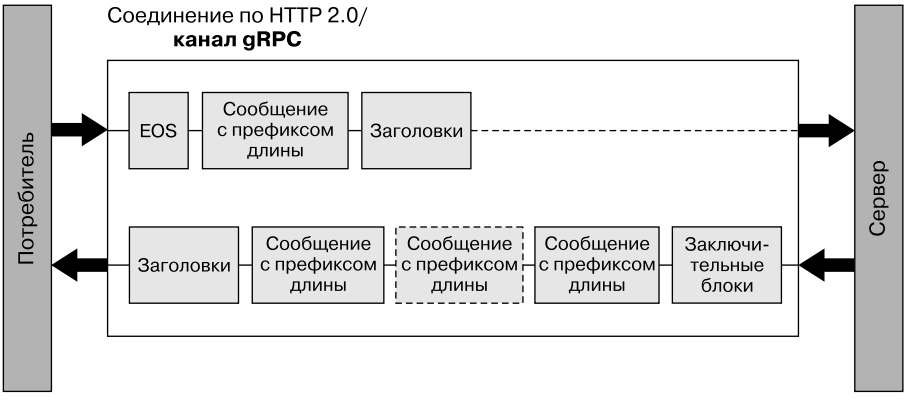


Рис. 4.9. Потоковый RPC на стороне сервера: процесс обмена сообщениями

Теперь взглянем на противоположный подход: потоковый RPC на стороне клиента.

Потоковый RPC на стороне клиента

В данном подходе клиент передает серверу множественные сообщения, а тот возвращает только один ответ. Вначале клиент устанавливает соединение с сервером, отправляя ему заголовочные фреймы. Затем передает несколько сообщений с префиксом длины в виде фреймов `DATA`, как показано на рис. 4.10. В конце клиент закрывает соединение на своей стороне, отправляя в последнем фрейме `DATA` флаг EOS. Тем временем сервер читает полученные сообщения. Получив весь запрос, сервер возвращает ответное сообщение вместе с заключительным заголовком и закрывает соединение.

Итак, мы подошли к последнему методу взаимодействия, двунаправленному RPC, в котором клиент и сервер обмениваются множественными сообщениями, пока не закроют свое соединение.

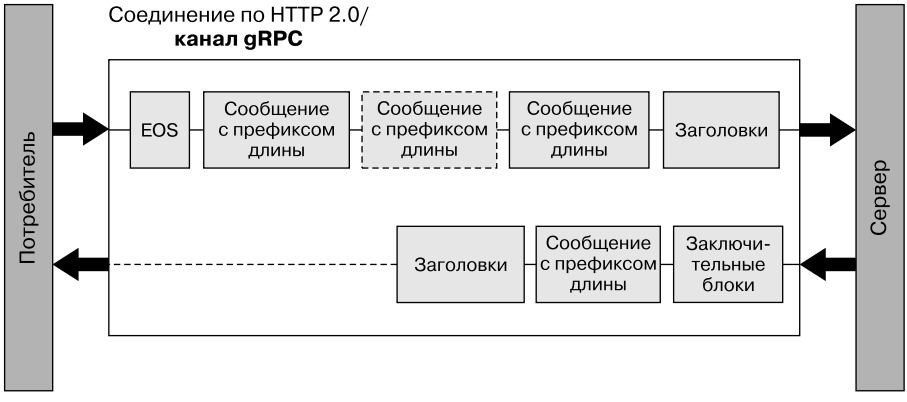


Рис. 4.10. Потоковый RPC на стороне клиента: процесс обмена сообщениями

Двунаправленный потоковый RPC

В данном подходе клиент устанавливает соединение, отправляя заголовочные фреймы. Затем клиентское и серверное приложения обмениваются сообщениями с префиксом длины, не дожидаясь завершения взаимодействия с противоположной стороны. Как видно на рис. 4.11, клиент и сервер отправляют сообщения одновременно. Любой из них может закрыть соединение на своей стороне, теряя тем самым возможность отправлять дальнейшие сообщения.

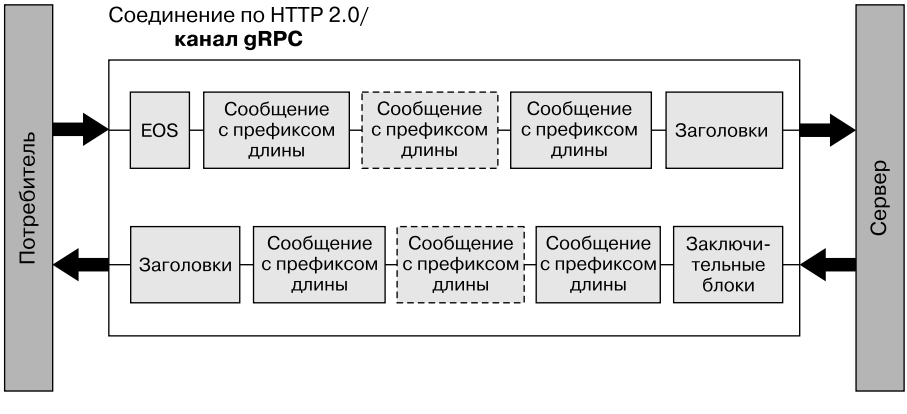


Рис. 4.11. Потоковый двунаправленный RPC: процесс обмена сообщениями

На этом мы завершаем наш подробный обзор методов взаимодействия на основе gRPC. За сетевые и транспортные операции обычно отвечает сам

протокол, и разработчику gRPC-приложений не обязательно знать их в подробностях.

Прежде чем переходить к следующей главе, рассмотрим то, как архитектура gRPC реализована на практике и какие языки поддерживает.

Практическая реализация архитектуры gRPC

Как показано на рис. 4.12, реализацию протокола gRPC можно разделить на несколько слоев. Основной, довольно тонкий, абстрагирует все сетевые операции от слоев более высокого уровня, чтобы разработчики приложений могли с легкостью выполнять RPC-вызовы по сети. Здесь же находятся расширения, дополняющие основную функциональность; некоторые из них предоставляют фильтры для аутентификации, обеспечивающие безопасность вызовов, и для установки крайних сроков.

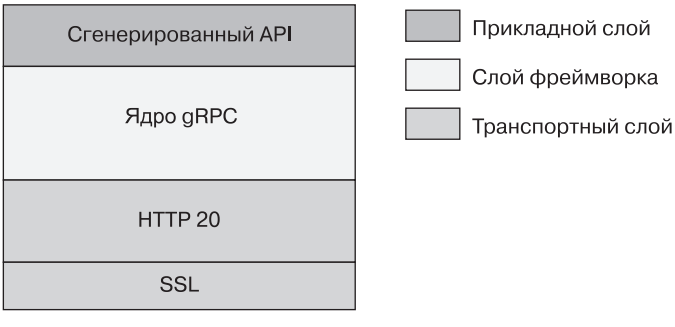


Рис. 4.12. Стандартная реализация архитектуры gRPC

Протокол gRPC имеет встроенную поддержку языков C/C++, Go и Java. Он также предоставляет привязки для многих других популярных языков, таких как Python, Ruby, PHP и т. д. Эти привязки представляют собой обертки вокруг низкоуровневого API на C.

Вслед за привязками для языков программирования идет код приложения. Этот слой отвечает за прикладную логику и кодирование данных. Обычно код для кодирования данных генерируется с помощью компиляторов, доступных для тех или иных языков. Например, можно воспользоваться компилятором Protocol Buffers. Таким образом, разработчики могут сосредоточиться на логике своего приложения и вызывать методы из уже сгенерированного кода.

Итак, мы обсудили большинство низкоуровневых аспектов реализации и работы приложений, основанных на gRPC. Разбирающийся в них разработчик лучше понимает те методики, которые применяет в своем коде. Это, в свою очередь, позволяет ему проектировать надежные приложения и с легкостью справляться с возникающими проблемами.

Резюме

Технология gRPC основана на двух быстрых и эффективных протоколах: Protocol Buffers и HTTP/2. Protocol Buffers — расширяемый механизм сериализации структурированных данных, не зависящий от языка и платформы. Двоичные данные, которые он генерирует, более компактные по сравнению с обычными JSON-сообщениями и обладают сильной типизацией. Для их передачи по сети используется транспортный протокол под названием HTTP/2.

HTTP/2 — новая мажорная версия протокола HTTP с полноценной поддержкой мультиплексирования. То есть приложение, написанное с помощью HTTP/2, может отправлять множество параллельных запросов по одному TCP-соединению; это делает его более быстрым, простым и надежным.

Своей высокой производительностью PRC-фреймворк gRPC обязан всем этим факторам.

В этой главе мы обсудили низкоуровневые аспекты взаимодействия на основе gRPC. С точки зрения разработки приложений они могут быть не такими уж существенными, поскольку их реализация встроена в библиотеку. Однако понимание того, как на самом деле происходит обмен сообщениями, совершенно незаменимо при диагностике проблем, возникающих в промышленных gRPC-приложениях. В следующей главе будут рассмотрены некоторые из расширенных возможностей gRPC, нацеленные на решение реальных задач.

ГЛАВА 5

gRPC: расширенные возможности

Иногда при разработке реальных gRPC-приложений приходится реализовывать дополнительные возможности, такие как перехват входящих и исходящих RPC-сообщений, эластичная обработка сетевых задержек, обработка ошибок, обмен метаданными между сервисами и потребителями и т. д.

Чтобы сохранить однородность наших примеров, все фрагменты кода, представленные в этой главе, написаны на Go. Но если вы лучше разбираетесь в Java, то полноценные примеры на данном языке для тех же сценариев использования можно найти в репозитории кода книги.

В этой главе вы познакомитесь с некоторыми ключевыми расширенными возможностями gRPC: с применением перехватчиков для перехвата RPC-сообщений на серверной и клиентской стороне, использованием крайних сроков для определения времени ожидания вызовов, обработкой ошибок на стороне сервера и клиента с учетом общепринятых рекомендаций. Кроме того, мы рассмотрим запуск нескольких сервисов на одном сервере с помощью мультиплексирования, обмен пользовательскими метаданными между приложениями, балансировку нагрузки и сопоставление имен при вызове других сервисов, а также сжатие RPC-вызовов для эффективного использования пропускной способности сети.

Начнем с обсуждения перехватчиков.

Перехватчики

Иногда перед вызовом удаленной функции на клиентской или серверной стороне или после него нужно выполнить некие рутинные операции. На этот случай gRPC позволяет перехватывать вызов для выполнения таких задач,

как ведение журнала, аутентификация, сбор метрик и пр., используя механизм расширения под названием *«перехватчик»* (interceptor). Установка перехватчиков в клиентском или серверном приложении возможна с помощью предоставленного gRPC простого API. Это один из ключевых способов расширения gRPC, который довольно хорошо подходит для реализации журналирования, аутентификации, авторизации, сбора метрик, трассировки и других потребительских требований.

Перехватчики доступны не для всех языков, которые поддерживаются в gRPC, и их реализация может варьироваться. В этой главе мы уделяем внимание только Go и Java.

Перехватчики можно разделить на две категории в зависимости от перехватываемых типов вызовов. В унарном RPC можно использовать *унарные перехватчики*, а в потоковом — *потоковые*. Они могут применяться как на серверной, так и на клиентской стороне. Сначала посмотрим, как они работают на стороне сервера.

Серверные перехватчики

В момент вызова клиентом удаленного метода gRPC-сервиса можно выполнить определенную логику, используя серверный перехватчик. Это помогает в ситуациях, когда перед удаленным вызовом, к примеру, нужно провести аутентификацию. Как видно на рис. 5.1, к любому gRPC-серверу, который вы разрабатываете, можно подключить один или несколько перехватчиков. Например, чтобы подключить новый серверный перехватчик к сервису `OrderManagement`, вы можете реализовать перехватчик и зарегистрировать его при создании gRPC-сервера.

Унарные и потоковые перехватчики предназначены для соответствующих видов RPC. Сначала рассмотрим серверные унарные перехватчики.

Унарный перехватчик

Если вы хотите перехватить унарный удаленный вызов на стороне сервера, то вам необходимо реализовать серверный унарный перехватчик. В листинге 5.1

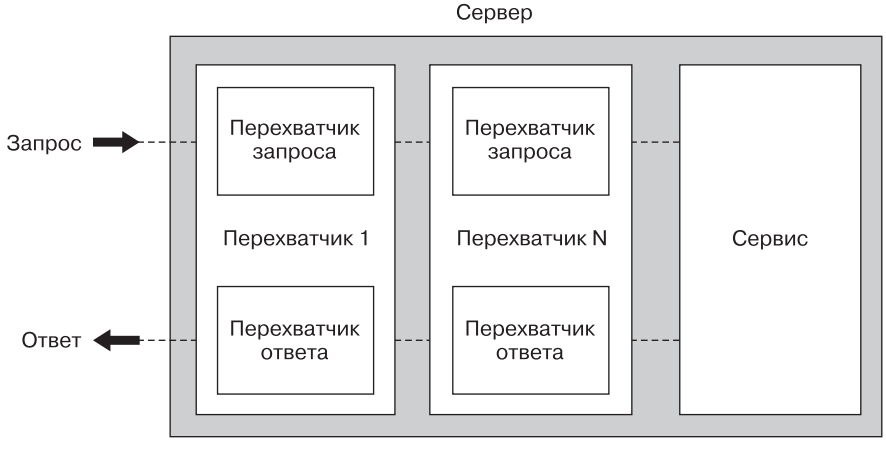


Рис. 5.1. Серверные перехватчики

показан фрагмент кода на языке Go, в котором при создании gRPC-сервера реализуется и регистрируется функция типа `UnaryServerInterceptor`. Это тип серверных унарных перехватчиков со следующей сигнатурой:

```
func(ctx context.Context, req interface{}, info *UnaryServerInfo,
                handler UnaryHandler) (resp interface{}, err error)
```

Внутри этой функции вы имеете полный контроль над унарными удаленными вызовами, которые получает ваш gRPC-сервер.

Листинг 5.1. Серверный унарный перехватчик в gRPC

```
// унарный перехватчик на стороне сервера
func orderUnaryServerInterceptor(ctx context.Context, req interface{},
                info *grpc.UnaryServerInfo, handler grpc.UnaryHandler)
                     (interface{}, error) {

        // логика перед вызовом
        // получает информацию о текущем RPC-вызове путем
        // анализа переданных аргументов
        log.Println("======= [Server Interceptor] ", info.FullMethod) ❶

        // вызываем обработчик, чтобы завершить
        // нормальное выполнение унарного RPC-вызова
        m, err := handler(ctx, req) ❷

        // логика после вызова
        log.Printf(" Post Proc Message : %s", m) ❸
        return m, err ❹
}
```

```
// ...

func main() {

...
    // регистрируем перехватчик на стороне сервера
        s := grpc.NewServer(
                grpc.UnaryInterceptor(orderUnaryServerInterceptor)) ❺
...
```

❶ Перед вызовом: здесь вы можете перехватить сообщение до того, как будет вызван соответствующий удаленный метод.

❷ Вызов RPC-метода с помощью `UnaryHandler`.

❸ После вызова: здесь вы можете обработать ответ, сгенерированный в результате удаленного вызова.

❹ Возвращение RPC-ответа.

❺ Регистрация унарного перехватчика на gRPC-сервере.

Реализацию серверного унарного перехватчика обычно можно разделить на три этапа: предобработки, вызова RPC-метода, постобработки. Как понятно из названия, этап предобработки выполняется до вызова удаленного метода. На этапе постобработки пользователь может получить информацию о текущем RPC-вызове, проанализировав переданные аргументы, такие как контекст (`ctx`), запрос (`req`) и состояние сервера. Возможно даже модифицировать RPC-вызов.

Дальше используется обработчик `UnaryHandler` для вызова удаленного метода. Затем идет этап постобработки, на котором при необходимости можно обработать возвращаемый ответ и сообщение об ошибке; по его окончании вы должны вернуть ответ и сообщение об ошибке в виде параметров функции своего перехватчика. Если постобработка не требуется, то вы можете просто вернуть вызов обработчика (`handler(ctx, req)`).

Теперь обсудим потоковые перехватчики.

Потоковый перехватчик

Серверные потоковые перехватчики перехватывают любые вызовы потокового RPC, которые приходят на gRPC-сервер. Они состоят из двух этапов: предобработки и перехвата потоковой операции.

Во фрагменте кода на языке Go, показанном в листинге 5.2, перехватываются вызовы потокового RPC, направленные к сервису `OrderManagement`. Тип `StreamServerInterceptor` — это тип серверного потокового перехватчика, а `orderServerStreamInterceptor` — функция данного типа, имеющая следующую сигнатуру:

```
func(srv interface{}, ss ServerStream, info *StreamServerInfo,
                                    handler StreamHandler) error
```

Как и в унарном перехватчике, этап предобработки позволяет перехватить вызов потокового RPC, пока он не дошел до реализации сервиса. Далее, чтобы завершить выполнение удаленного метода, можно вызвать `StreamHandler`. После этапа предобработки вы можете перехватить сообщение потокового RPC, воспользовавшись оберткой, реализующей интерфейс `grpc.ServerStream`. Эту обертку можно передать при вызове `grpc.StreamHandler`, используя выражение `handler(srv, newWrappedStream(ss))`. Интерфейс `grpc.ServerStream` перехватывает потоковые сообщения, отправляемые или принимаемые gRPC-сервисом. Он реализует функции `SendMsg` и `RecvMsg`, которые вызываются в момент получения или отправления сервисом сообщения потокового RPC.

Листинг 5.2. Серверный потоковый перехватчик в gRPC

```
// Потоковый перехватчик на стороне сервера
// wrappedStream – обертка вокруг встроенного интерфейса
// grpc.ServerStream, которая перехватывает вызовы методов
// RecvMsg и SendMsg

type wrappedStream struct { ❶
        grpc.ServerStream
}

❷
func (w *wrappedStream) RecvMsg(m interface{}) error {
        log.Printf("====== [Server Stream Interceptor Wrapper] " +
                "Receive a message (Type: %T) at %s",
                m, time.Now().Format(time.RFC3339))
        return w.ServerStream.RecvMsg(m)
}

❸
func (w *wrappedStream) SendMsg(m interface{}) error {
        log.Printf("====== [Server Stream Interceptor Wrapper] " +
                "Send a message (Type: %T) at %v",
                m, time.Now().Format(time.RFC3339))
```

```
        return w.ServerStream.SendMsg(m)
}

❹
func newWrappedStream(s grpc.ServerStream) grpc.ServerStream {
        return &wrappedStream{s}
}

❺
func orderServerStreamInterceptor(srv interface{},
        ss grpc.ServerStream, info *grpc.StreamServerInfo,
        handler grpc.StreamHandler) error {
        log.Println("====== [Server Stream Interceptor] ",
                info.FullMethod) ❻
        err := handler(srv, newWrappedStream(ss)) ❼
        if err != nil {
                log.Printf("RPC failed with error %v", err)
        }
        return err
}

...
// регистрация перехватчика
s := grpc.NewServer(
                grpc.StreamInterceptor(orderServerStreamInterceptor)) ❽
...
```

❶ Обертка для потока `grpc.ServerStream`.

❷ Реализация функции `RecvMsg`, принадлежащей обертке; обрабатывает сообщения, принимаемые с помощью потокового RPC.

❸ Реализация функции `SendMsg`, принадлежащей обертке; обрабатывает сообщения, отправляемые с помощью потокового RPC.

❹ Создание экземпляра обертки.

❺ Реализация потокового перехватчика.

❻ Этап предобработки.

❼ Вызов метода потокового RPC с помощью обертки.

❽ Регистрация перехватчика.

Понять, как потоковый перехватчик ведет себя на серверной стороне, помогут следующие журнальные записи gRPC-сервера. Учитывая порядок, в котором

они выводятся, можно проследить действия, выполняемые потоковым перехватчиком. Здесь вызывается удаленный потоковый метод `SearchOrders`:

```
[Server Stream Interceptor] /ecommerce.OrderManagement/searchOrders
[Server Stream Interceptor Wrapper] Receive a message

Matching Order Found : 102 -> Writing Order to the stream ...
[Server Stream Interceptor Wrapper] Send a message...
Matching Order Found : 104 -> Writing Order to the stream ...
[Server Stream Interceptor Wrapper] Send a message...
```

В клиентских перехватчиках используется очень похожая терминология, с некоторыми незначительными отличиями, касающимися интерфейсов и сигнатур функций. Поговорим об этом подробно.

Клиентские перехватчики

RPC-вызов, который клиент делает с целью выполнить удаленный метод gRPC-сервиса, можно перехватить на клиентской стороне. На рис. 5.2 показано, что вы можете перехватывать вызовы как унарного, так и потокового RPC.

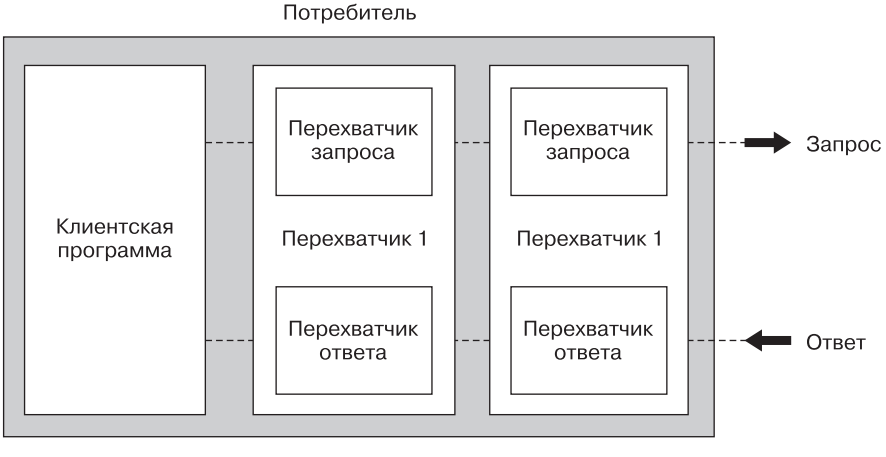


Рис. 5.2. Клиентские перехватчики

Это особенно полезно в ситуациях, когда необходимо реализовать операции с возможностью повторного применения, такие как безопасный вызов gRPC-сервиса за пределами клиентского кода.

Унарный перехватчик

Клиентский унарный перехватчик используется для перехвата вызовов унарного RPC на стороне клиента. Он представляет собой функцию типа `UnaryClientInterceptor` со следующей сигнатурой:

```
func(ctx context.Context, method string, req, reply interface{},
        cc *ClientConn, invoker UnaryInvoker, opts ...CallOption) error
```

Клиентский унарный перехватчик, как и серверный, состоит из разных этапов. Его элементарная реализация на языке Go представлена в листинге 5.3. На этапе предобработки можно перехватить RPC-вызов, прежде чем вызывать удаленный метод. Здесь возможен доступ к информации о текущем RPC-вызове, если проанализировать передаваемые аргументы, такие как контекст (`ctx`), строка `method`, отправляемый запрос (`req`) и параметры `CallOption`. Можно даже модифицировать RPC-вызов, прежде чем отправлять его серверному приложению. Затем использование аргумента `UnaryInvoker` позволяет вызвать фактический унарный RPC. На этапе постобработки доступны ответ или ошибка, возникшая в результате удаленного вызова.

Листинг 5.3. Клиентский унарный перехватчик в gRPC

```
func orderUnaryClientInterceptor(
        ctx context.Context, method string, req, reply interface{},
        cc *grpc.ClientConn,
        invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
        // этап предобработки
        log.Println("Method : " + method) ❶

        // вызов удаленного метода
        err := invoker(ctx, method, req, reply, cc, opts...) ❷

        // этап постобработки
        log.Println(reply) ❸

        return err ❹
}
...

func main() {
        // установление соединения с сервером
        conn, err := grpc.Dial(address, grpc.WithInsecure(),
                grpc.WithUnaryInterceptor(orderUnaryClientInterceptor)) ❺
...
```

❶ На этапе предобработки есть доступ к RPC-запросу перед его отправкой на сервер.

❷ Вызов RPC-метода с помощью `UnaryInvoker`.

❸ Этап постобработки, где можно обработать ответ или возникшую ошибку.

❹ Возвращение ошибки клиентскому gRPC-приложению вместе с ответом, который передается в виде аргумента.

❺ Установление соединения с сервером путем передачи унарного перехватчика в качестве параметра метода `Dial`.

Регистрация функции перехватчика выполняется внутри операции `grpc.Dial` с помощью `grpc.WithUnaryInterceptor`.

Потоковый перехватчик

Клиентский потоковый перехватчик перехватывает любые вызовы потокового RPC, с которыми имеет дело клиент. Его реализация похожа на ту, что была показана выше на серверной стороне. Он представляет собой функцию типа `StreamClientInterceptor` со следующей сигнатурой:

```
func(ctx context.Context, desc *StreamDesc, cc *ClientConn,
                          method string, streamer Streamer,
                          opts ...CallOption) (ClientStream, error)
```

Как показано в листинге 5.4, реализация клиентского потокового перехватчика включает в себя этапы предобработки и перехвата потоковой операции.

Листинг 5.4. Клиентский потоковый перехватчик в gRPC

```
func clientStreamInterceptor(
        ctx context.Context, desc *grpc.StreamDesc,
        cc *grpc.ClientConn, method string,
        streamer grpc.Streamer, opts ...grpc.CallOption)
        (grpc.ClientStream, error) {
        log.Println("======= [Client Interceptor] ", method) ❶
        s, err := streamer(ctx, desc, cc, method, opts...) ❷
        if err != nil {
                return nil, err
        }
        return newWrappedStream(s), nil ❸
}
```

```
type wrappedStream struct { ❹
        grpc.ClientStream
}

func (w *wrappedStream) RecvMsg(m interface{}) error { ❺
        log.Printf("====== [Client Stream Interceptor] " +
                "Receive a message (Type: %T) at %v",
                m, time.Now().Format(time.RFC3339))
        return w.ClientStream.RecvMsg(m)
}

func (w *wrappedStream) SendMsg(m interface{}) error { ❻
        log.Printf("====== [Client Stream Interceptor] " +
                "Send a message (Type: %T) at %v",
                m, time.Now().Format(time.RFC3339))
        return w.ClientStream.SendMsg(m)
}

func newWrappedStream(s grpc.ClientStream) grpc.ClientStream {
        return &wrappedStream{s}
}

...

func main() {
        // установление соединения с сервером
        conn, err := grpc.Dial(address, grpc.WithInsecure(),
                grpc.WithStreamInterceptor(clientStreamInterceptor)) ❼
...
```

❶ На этапе предобработки есть доступ к RPC-запросу перед его отправкой на сервер.

❷ Вызов переданной функции `streamer` для получения `ClientStream`.

❸ Создание обертки вокруг интерфейса `ClientStream`, перегрузка его методов с использованием логики перехвата и возвращение его клиентскому приложению.

❹ Обертка для интерфейса `grpc.ClientStream`.

❺ Функция для перехвата сообщений, принимаемых в рамках потокового RPC.

❻ Функция для перехвата сообщений, отправляемых в рамках потокового RPC.

❼ Регистрация потокового перехватчика.

Перехват потоковых операций производится с помощью новой структуры-обертки вокруг потока `grpc.ClientStream`. Внутри этой обертки нужно реализовать два метода, `RecvMsg` и `SendMsg`, с помощью которых можно будет перехватывать потоковые сообщения, принимаемые или отправляемые клиентской стороной. Как и в случае с унарным перехватчиком, для регистрации используется операция `grpc.Dial`.

Рассмотрим крайние сроки — еще одну возможность, которая часто применяется при вызове gRPC-сервисов из клиентских приложений.

Крайние сроки

Крайние сроки и время ожидания — две методики, которые часто применяются в распределенных вычислениях. *Время ожидания* позволяет указать, как долго клиентское приложение должно ждать завершения удаленного вызова, прежде чем прервать его с ошибкой. Эта величина обычно представлена отрезком времени, который действует локально в рамках отдельно взятого клиента. Например, один запрос может состоять из цепочки удаленных вызовов, связывающих разные сервисы. Как следствие, можно задать время ожидания для каждого вызова при обращении к каждому следующему сервису. В связи с этим время ожидания нельзя применить ко всему жизненному циклу запроса целиком. Для таких случаев предусмотрены крайние сроки.

Крайний срок — абсолютная величина времени, которое проходит с момента отправки запроса (хотя в API он представлен в виде смещения продолжительности) и охватывает вызов нескольких сервисов. Приложение, инициирующее запрос, устанавливает крайний срок, до завершения которого должна быть выполнена вся цепочка вызова. Данная концепция поддерживается в API gRPC, и ее использование обусловлено множеством причин. Взаимодействие в gRPC происходит по сети, поэтому между вызовами и ответами могут возникать задержки. Кроме того, в некоторых случаях самому gRPC-сервису требуется некоторое время на то, чтобы ответить, — это зависит от его бизнес-логики. Если при разработке клиентского приложения не используются крайние сроки, то ожидание ответов на его RPC-запросы может занимать неограниченное количество времени, и пока приложение ждет, его ресурсы остаются недоступными для других активных вызовов. В результате

сервер и клиент рискуют исчерпать доступные им ресурсы и тем самым повысить латентность ответов или даже вывести из строя весь gRPC-сервис.

На рис. 5.3 показано клиентское gRPC-приложение, вызывающее сервис `ProductMgt`, который затем обращается к сервису `Inventory`.

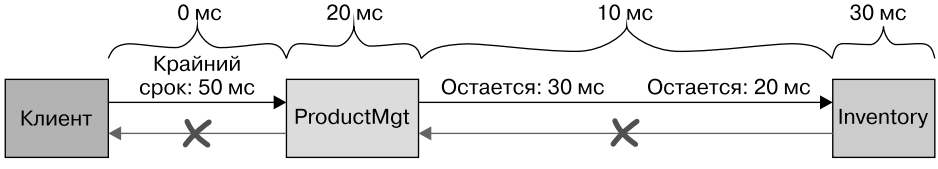


Рис. 5.3. Использование крайних сроков при вызове сервисов

Клиент указывает смещение продолжительности (то есть крайний срок = = текущее время + смещение) в размере 50 мс. Латентность между сервисом `ProductMgt` и клиентом равна 0 мс, а на обработку запроса уходит 20 мс. Смещение, которое должен указать сервис `ProductMgt`, должно быть равно 30 мс. Но, поскольку на возврат ответа сервису `Inventory` нужно 30 мс, крайний срок истечет на обеих клиентских сторонах (`ProductMgt` вызывает сервис `Inventory` и клиентское приложение).

Латентность, которую вносит бизнес-логика сервиса `ProductMgt`, равна 20 мс. В результате истечения крайнего срока логика вызова `ProductMgt` генерирует соответствующее событие и передает его обратно клиентскому приложению. Таким образом, если вы используете крайние сроки, то убедитесь, что они действуют во всех сервисах.

Клиентское приложение может установить крайний срок во время соединения с gRPC-сервисом. Сделав вызов, оно ждет на протяжении заданного периода; если ответ не получен до его истечения, то удаленный вызов прерывается с ошибкой `DEADLINE_EXCEEDED`.

Рассмотрим реальный пример использования крайних сроков при обращении к gRPC-сервисам. Возьмем тот же сервис `OrderManagement` и предположим, что выполнение его метода `AddOrder` занимает существенное время (мы будем имитировать эту задержку). Но клиентское приложение ждет только до тех пор, пока ответ не станет бесполезным. Например, метод `AddOrder` может выполняться 2 с, что составляет максимальное время ожидания

клиента. Чтобы реализовать это (как показано во фрагменте кода на Go в листинге 5.5), клиентское приложение может установить двухсекундное время ожидания с помощью операции `context.WithDeadline`. Для обработки кода ошибки используется пакет `status`, который мы подробно рассмотрим в одном из следующих разделов.

Листинг 5.5. Крайние сроки в клиентском gRPC-приложении

```
conn, err := grpc.Dial(address, grpc.WithInsecure())
if err != nil {
    log.Fatalf("did not connect: %v", err)
}
defer conn.Close()
client := pb.NewOrderManagementClient(conn)

clientDeadline := time.Now().Add(
    time.Duration(2 * time.Second))
ctx, cancel := context.WithDeadline(
    context.Background(), clientDeadline) ❶

defer cancel()

// добавляем заказ
order1 := pb.Order{Id: "101",
    Items:[]string{"iPhone XS", "Mac Book Pro"},
    Destination:"San Jose, CA",
    Price:2300.00}
res, addErr := client.AddOrder(ctx, &order1) ❷

if addErr != nil {
    got := status.Code(addErr) ❸
    log.Printf("Error Occured -> addOrder : , %v:", got) ❹
} else {
    log.Print("AddOrder Response -> ", res.Value)
}
```

❶ Устанавливаем двухсекундный крайний срок в текущем контексте.

❷ Вызываем удаленный метод `AddOrder` и перехватываем любые потенциальные ошибки внутри `addErr`.

❸ Определяем код ошибки с помощью пакета `status`.

❹ Если вызов превысит крайний срок, то возвращается ошибка типа `DEADLINE_EXCEEDED`.

Как же определить оптимальное значение для крайнего срока? Однозначного ответа на этот вопрос не существует, но ваш выбор может быть обусловлен несколькими факторами. В частности, следует учитывать сквозную латентность каждого вызываемого вами сервиса; вы должны знать, какие вызовы выполняются последовательно, а какие — параллельно, какая латентность характерна вашей сети и какие крайние сроки установлены для остальных сервисов в цепочке вызовов. Выбрав какое-то конкретное значение, откорректируйте его с учетом поведения ваших gRPC-приложений.

Установка крайнего срока в gRPC на языке Go выполняется с помощью пакета context (oreil.ly/OTrmY), который содержит функцию WithDeadline. В Go контекст часто используется для передачи данных по цепочке вызовов, чтобы они были доступны в последующих операциях. При вызове WithDeadline на клиентской стороне библиотека gRPC создает соответствующий заголовок, который представляет собой крайний срок для выполнения удаленного метода. В Java это делается немного иначе, так как реализация концепции крайних сроков берется непосредственно из заглушки пакета io.grpc.stub.*, а сам крайний срок устанавливается с помощью метода blockingStub.withDeadlineAfter(long, java.util.concurrent.TimeUnit). Подробности реализации данной возможности на языке Java можно найти в репозитории кода.

Клиент и сервер могут судить об успешности удаленного вызова независимо друг от друга; это значит, что сделанные ими выводы могут не совпадать. Например, если в приведенном выше коде клиент удовлетворяет условию `DEADLINE_EXCEEDED`, то сервис все равно может попытаться ответить. Поэтому сервису необходимо определить, является ли текущий удаленный вызов действительным. На стороне сервера тоже можно узнать, закончился ли крайний срок, заданный клиентом. Для этого внутри операции `AddOrder` можно проверить условие `ctx.Err() == context.DeadlineExceeded` и в случае его выполнения прервать работу удаленного метода и вернуть ошибку (в Go это часто реализуется с помощью неблокирующей конструкции `select`).

Помимо истечения крайних сроков, могут возникать ситуации, когда клиентскому или серверному приложению нужно прекратить активное взаимодействие. На этот случай в gRPC предусмотрен механизм отмены.

Механизм отмены

Клиент и сервер, общающиеся по gRPC, определяют успешность вызовов локально и независимо друг от друга. Например, один и тот же удаленный вызов может успешно отработать на сервере, но завершиться ошибкой на клиентской стороне. Точно так же клиент и сервер могут иметь разные мнения о результатах вызова. Если одна из сторон хочет прервать взаимодействие, то это можно сделать, *отменив* удаленный вызов, после чего обмен сообщениями становится невозможным, а информация о том, что вызов был отменен, передается противоположной стороне.

В языке Go механизм отмены, как и крайние сроки, реализован в пакете context (oreil.ly/OTrmY) в виде функции WithCancel. Когда приложение вызывает эту функцию, библиотека gRPC создает на клиентской стороне необходимый заголовок, который сигнализирует о прекращении взаимодействия клиента и сервера.

Рассмотрим пример двунаправленного потокового взаимодействия клиентского и серверного приложения. В коде на языке Go, представленном в листинге 5.6, функция `cancel` берется из вызова `context.WithTimeout`. Получив ссылку на эту функцию, вы можете вызвать ее в любом месте, где необходимо прервать взаимодействие.

Листинг 5.6. Отмена gRPC

```
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second) ❶

streamProcOrder, _ := client.ProcessOrders(ctx) ❷
_ = streamProcOrder.Send(&wrapper.StringValue{Value:"102"}) ❸
_ = streamProcOrder.Send(&wrapper.StringValue{Value:"103"})
_ = streamProcOrder.Send(&wrapper.StringValue{Value:"104"})

channel := make(chan bool, 1)

go asncClientBidirectionalRPC(streamProcOrder, channel)
time.Sleep(time.Millisecond * 1000)

// отмена gRPC
cancel() ❹
log.Printf("RPC Status : %s", ctx.Err()) ❺
```

```
_ = streamProcOrder.Send(&wrapper.StringValue{Value:"101"})
_ = streamProcOrder.CloseSend()

<- channel

func asncClientBidirectionalRPC (
    streamProcOrder pb.OrderManagement_ProcessOrdersClient, c chan bool) {
...

            combinedShipment, errProcOrder := streamProcOrder.Recv()
            if errProcOrder != nil {
                log.Printf("Error Receiving messages %v", errProcOrder) ❻
...
}
```

❶ Получаем ссылку на `cancel`.

❷ Инициируем потоковый RPC.

❸ Отправляем сервису поток сообщений.

❹ Отменяем (прерываем) удаленный вызов на клиентской стороне.

❺ Состояние текущего контекста.

❻ Возвращаем ошибку при попытке получить сообщения из отмененного контекста.

Чтобы узнать об отмене удаленного вызова противоположной стороной, можно проверить контекст. В этом примере серверное приложение проверяет, отменен ли текущий контекст, используя выражение `stream.Context().Err() == context.Canceled`.

Как показывают примеры приложений с крайними сроками и механизмом отмены, обработка ошибок, возникающих в RPC, — очень распространенная операция. В следующем разделе мы подробно поговорим о методиках обработки ошибок в gRPC.

Обработка ошибок

При вызове в gRPC клиент получает ответ, состояние которого говорит либо об успешном выполнении, либо об ошибке. Клиентское приложение должно быть написано с учетом всех потенциальных ошибок и условий, которые

к ним приводят. На стороне сервера нужно не только обрабатывать ошибки, но и генерировать для них подходящие коды состояния.

При возникновении ошибки gRPC возвращает один из кодов состояния и (при необходимости) сообщение с подробностями о том, что произошло. Объект состояния содержит целочисленный код и строковое сообщение, которые являются типичными для всех реализаций gRPC на разных языках.

В gRPC используется стандартный для этого протокола набор кодов состояния, включая следующие:

- `OK` — успешный вызов, не ошибка;
- `CANCELLED` — операция была отменена (обычно вызывающей стороной);
- `DEADLINE_EXCEEDED` — истечение крайнего срока до завершения операции;
- `INVALID_ARGUMENT` — клиент указал недопустимый аргумент.

Коды ошибок, доступные в gRPC, а также их описания перечислены в табл. 5.1. Их полный список можно найти в официальной документации gRPC (oreil.ly/LiNLn) или в документации для Go (oreil.ly/E61Q0) или Java (oreil.ly/Ugtg0).

Таблица 5.1. Коды ошибок в gRPC

Код	Номер	Описание
OK	0	Успех
CANCELLED	1	Операция была отменена (вызывающей стороной)
UNKNOWN	2	Неизвестная ошибка
INVALID_ARGUMENT	3	Клиент указал недопустимый аргумент
DEADLINE_EXCEEDED	4	Крайний срок истек до завершения операции
NOT_FOUND	5	Не удалось найти запрошенный элемент
ALREADY_EXISTS	6	Элемент, который клиент пытался создать, уже существует
PERMISSION_DENIED	7	У вызывающей стороны нет права выполнять указанную операцию

Код	Номер	Описание
RESOURCE_EXHAUSTED	8	Исчерпан какой-то ресурс
FAILED_PRECONDITION	9	Операция была отклонена, поскольку система не находится в состоянии, необходимом для ее выполнения
ABORTED	10	Операция была прервана
OUT_OF_RANGE	11	Попытка выполнения операции за пределами допустимого диапазона
UNIMPLEMENTED	12	Операция не реализована, не поддерживается или не включена в этом сервисе
INTERNAL	13	Внутренние ошибки
UNAVAILABLE	14	В данный момент сервис недоступен
DATA_LOSS	15	Потеря или повреждение данных без возможности восстановления
UNAUTHENTICATED	16	Запрос не содержит аутентификационных данных, необходимых для выполнения операции

Модель ошибок, встроенная в протокол gRPC, довольно ограничена и лишена всякой связи с его внутренним форматом данных (в качестве которого чаще всего выступает Protocol Buffers). Если вы используете данные в формате Protocol Buffers, то вам доступна более развитая модель ошибок, реализованная в пакете Google API `google.rpc`. Но она поддерживается только в библиотеках для C++, Go, Java, Python и Ruby; вы должны помнить об этом, если планируете применять другие языки.

Посмотрим, как эти концепции можно использовать в реальных сценариях обработки ошибок в gRPC. Еще раз вернемся к нашему примеру с управлением заказами и представим, что иногда удаленному методу `AddOrder` могут передаваться недействительные ID. В листинге 5.7 показана ситуация, когда в ответ на получение идентификатора `-1` сервис генерирует ошибку и возвращает ее потребителю.

Листинг 5.7. Создание и передача ошибки на стороне сервера

```
if orderReq.Id == "-1" { ❶
    log.Printf("Order ID is invalid! -> Received Order ID %s",
        orderReq.Id)
```

```
errorStatus := status.New(codes.InvalidArgument,
    "Invalid information received") ❷
ds, err := errorStatus.WithDetails( ❸
    &epb.BadRequest_FieldViolation{
        Field:"ID",
        Description: fmt.Sprintf(
            "Order ID received is not valid %s : %s",
            orderReq.Id, orderReq.Description),
    },
)
if err != nil {
    return nil, errorStatus.Err()
}

return nil, ds.Err() ❹
}
...
```

❶ Некорректный запрос. Нужно сгенерировать ошибку и отправить ее клиенту.

❷ Создаем новое состояние с кодом ошибки `InvalidArgument`.

❸ Описываем произошедшее с помощью типа ошибки `BadRequest_FieldViolation` из `google.golang.org/genproto/googleapis/rpc/errdetails`.

❹ Возвращаем сгенерированную ошибку.

Состояние с подходящим кодом ошибки и дополнительной информацией можно создать из пакетов `grpc.status`. В нашем примере мы использовали выражение `status.New(codes.Invalid Argument, "Invalid information received")`. Чтобы отправить эту ошибку клиенту, достаточно выполнить `return nil, errorStatus.Err()`. Но вы также можете воспользоваться более развитой моделью ошибок Google API из пакета `google.rpc`. Здесь мы указали подробности о случившемся с помощью специального типа `google.golang.org/genproto/googleapis/rpc/errdetails`.

На клиентской стороне мы просто обрабатываем возвращенную ошибку в рамках удаленного вызова. То, как это делается в нашем примере с управлением заказами, показано во фрагменте кода на Go в листинге 5.8. Мы вызываем метод `AddOrder` и присваиваем возвращенную ошибку переменной `addOrderError`. Далее мы проверяем результат, сохраненный в `addOrderError`, и корректно обрабатываем ошибку. Для этого используем код и тип ошибки, установленные на стороне сервера.

Листинг 5.8. Обработка ошибок на стороне клиента

```
order1 := pb.Order{Id: "-1",
        Items:[]string{"iPhone XS", "Mac Book Pro"},
        Destination:"San Jose, CA", Price:2300.00} ❶
res, addOrderError := client.AddOrder(ctx, &order1) ❷

if addOrderError != nil {
        errorCode := status.Code(addOrderError) ❸
        if errorCode == codes.InvalidArgument { ❹
                log.Printf("Invalid Argument Error : %s", errorCode)
                errorStatus := status.Convert(addOrderError) ❺
                for _, d := range errorStatus.Details() {
                        switch info := d.(type) {
                        case *epb.BadRequest_FieldViolation: ❻
                              log.Printf("Request Field Invalid: %s", info)
                        default:
                              log.Printf("Unexpected error type: %s", info)
                        }
                }
        } else {
                log.Printf("Unhandled error : %s ", errorCode)
        }
} else {
        log.Print("AddOrder Response -> ", res.Value)
}
```

❶ Этот заказ недействителен.

❷ Вызываем удаленный метод `AddOrder` и присваиваем ошибку переменной `addOrderError`.

❸ Получаем код ошибки из пакета `grpc/status`.

❹ Сравниваем код ошибки с `InvalidArgument`.

❺ Получаем состояние.

❻ Проверяем, имеет ли ошибка тип `BadRequest_FieldViolation`.

Коды ошибок gRPC и расширенную модель ошибок рекомендуется использовать везде, где это возможно. Состояние с ошибкой и подробностями передается с помощью заключительных заголовков на уровне транспортного протокола.

Теперь поговорим о мультиплексировании — механизме размещения нескольких сервисов на одном gRPC-сервере.

Мультиплексирование

До сих пор во всех наших примерах на одном gRPC-сервере всегда размещался лишь один сервис, а клиентское соединение всегда использовалось только одной клиентской заглушкой. Однако на самом деле gRPC позволяет выполнять на одном сервере сразу несколько сервисов (рис. 5.4) и задействовать одно и то же клиентское соединение из разных клиентских заглушек. Эта возможность называется *мультиплексированием*.

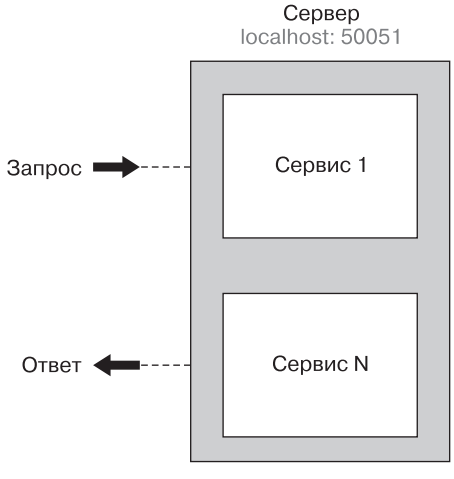


Рис. 5.4. Мультиплексирование нескольких gRPC-сервисов в одном серверном приложении

Представьте, к примеру: для управления заказами вместе с `OrderManagement` на том же gRPC-сервере нужно запустить еще один сервис, чтобы клиентское приложение могло обращаться к ним с помощью одного и того же соединения. Для регистрации обоих сервисов на gRPC-сервере можно использовать их соответствующие функции (то есть `ordermgt_pb.RegisterOrderManagementServer` и `hello_pb.RegisterGreeterServer`). Таким образом, вы можете зарегистрировать любое количество сервисов на одном и том же gRPC-сервере (как показано в листинге 5.9).

Листинг 5.9. Два gRPC-сервиса внутри одного экземпляра grpc.Server

```
func main() {
        initSampleData()
        lis, err := net.Listen("tcp", port)
        if err != nil {
```

```
            log.Fatalf("failed to listen: %v", err)
        }
        grpcServer := grpc.NewServer() ❶

        // регистрируем на сервере orderMgtServer сервис OrderManagement
        ordermgt_pb.RegisterOrderManagementServer(grpcServer, &orderMgtServer{}) ❷

        // регистрируем на сервере orderMgtServer сервис Hello
        hello_pb.RegisterGreeterServer(grpcServer, &helloServer{}) ❸

        ...
}
```

❶ Создаем gRPC-сервер.

❷ Регистрируем сервис `OrderManagement` на gRPC-сервере.

❸ Регистрируем сервис `Hello` на gRPC-сервере.

На стороне клиента аналогичным образом можно разделять одно и то же gRPC-соединение между двумя клиентскими заглушками.

Поскольку оба сервиса работают на одном gRPC-сервере, можно создать gRPC-соединение и использовать его при создании экземпляров клиента для разных сервисов. Это показано в листинге 5.10.

Листинг 5.10. Две клиентские заглушки, разделяющие одно и то же соединение grpc.ClientConn

```
// устанавливаем соединение с сервером
conn, err := grpc.Dial(address, grpc.WithInsecure()) ❶
...

orderManagementClient := pb.NewOrderManagementClient(conn) ❷

...

// удаленный вызов AddOrder
        ...
res, addErr := orderManagementClient.AddOrder(ctx, &order1)

...

helloClient := hwpb.NewGreeterClient(conn) ❸

        ...
        // удаленный вызов SayHello
helloResponse, err := helloClient.SayHello(hwcCtx,
        &hwpb.HelloRequest{Name: "gRPC Up and Running!"})
...
```

❶ Создаем gRPC-соединение.

❷ Используем это соединение, чтобы создать клиент сервиса `OrderManagement`.

❸ Задействуем то же соединение с целью создать клиент сервиса `Hello`.

Запуск нескольких сервисов или использование одного соединения разными заглушками — архитектурное решение, не имеющее отношения к концепциям gRPC. В большинстве повседневных ситуаций, таких как создание микросервисов, один экземпляр gRPC-сервера сознательно не разделяют между несколькими сервисами.

Один из самых полезных сценариев применения мультиплексирования в контексте микросервисов — обслуживание нескольких мажорных версий одного и того же сервиса в рамках одного серверного процесса. Это позволяет продолжить поддержку устаревших клиентов после нарушения обратной совместимости в API. Как только старая версия контракта сервиса выйдет из употребления, ее можно убрать с сервера.

В следующем разделе мы обсудим обмен данными, не являющимися частью запросов и ответов, которыми обмениваются клиентские и серверные приложения.

Метаданные

В gRPC-приложениях для обмена информацией между сервисами и потребителями обычно используются вызовы. Если информация имеет непосредственное отношение к бизнес-логике серверной и клиентской сторон, то большинстве случаев передается в виде аргументов при вызове удаленных методов. Но в ряде ситуаций может возникнуть необходимость в передаче сведений об RPC-вызовах, которые не имеют отношения к бизнес-контексту (и следовательно, не должны быть частью аргументов). На этот случай в gRPC предусмотрены *метаданные*, которые могут отправлять и принимать как сервис, так и клиент. Как видно на рис. 5.5, для обмена метаданными, созданными на той или иной стороне и представленными в виде списка пар «ключ — значение», можно применять заголовки gRPC.

Один из наиболее распространенных способов применения метаданных заключается в обмене заголовками безопасности между gRPC-приложениями, но вы можете обмениваться любой информацией. API для работы с метаданными часто используют в перехватчиках. В следующем подразделе мы исследуем передачу в gRPC метаданных между клиентом и сервером.

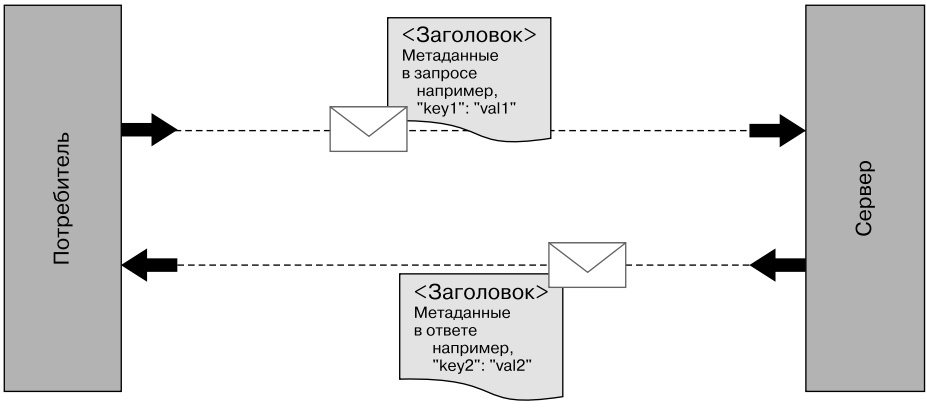


Рис. 5.5. Обмен метаданными между клиентским и серверным gRPC-приложениями

Создание и извлечение метаданных

Процесс создания метаданных в gRPC-приложении довольно прост и понятен. В следующем фрагменте кода на языке Go показано сразу два способа. В Go метаданные имеют вид обычной хеш-таблицы и создаются в таком формате: `metadata.New(map[string]string{"key1": "val1", "key2": "val2"})`. Вы также можете создавать метаданные в виде пар, используя `metadata.Pairs`; в этом случае значения с одним и тем же ключом объединяются в список:

```
// создание метаданных: первый вариант
md := metadata.New(map[string]string{"key1": "val1", "key2": "val2"})

// создание метаданных: второй вариант
md := metadata.Pairs(
    "key1", "val1",
    "key1", "val1-2", // у "key1" будет значение []string{"val1", "val1-2"}
    "key2", "val2",
)
```

В качестве метаданных можно передавать и двоичные значения; перед отправкой они будут переводиться в кодировку base64 и декодироваться после получения.

Чтение метаданных на стороне клиента и сервера можно выполнять с помощью входящего контекста, используя вызов `metadata.FromIncomingContext(ctx)`, который в языке Go возвращает хеш-таблицу с метаданными:

```
func (s *server) AddOrder(ctx context.Context, orderReq *pb.Order)
    (*wrappers.StringValue, error) {

md, metadataAvailable := metadata.FromIncomingContext(ctx)
// читаем нужные нам метаданные из хеш-таблицы 'md'
```

Теперь подробно обсудим процесс отправки и получения метаданных на стороне клиента и сервера с помощью унарного и потокового стилей RPC.

Отправка и получение метаданных на стороне клиента

Чтобы отправить метаданные gRPC-сервису, их нужно создать и указать в контексте удаленного вызова. В языке Go это можно сделать двумя способами. Как показано в листинге 5.11, вы можете создать метаданные вместе с новым контекстом, используя функцию `NewOutgoingContext`, или просто добавить их в существующий контекст, вызвав `AppendToOutgoingContext`. Обратите внимание: в первом варианте любые имеющиеся в контексте метаданные будут заменены. Созданный вами контекст можно передать с помощью унарного или потокового RPC. Как вы уже знаете из главы 4, метаданные, указанные в контексте, превращаются на транспортном уровне в заголовки gRPC (в HTTP/2) или заключительные блоки. Поэтому принимающая сторона получает их в виде заголовков.

Листинг 5.11. Отправка метаданных по gRPC на стороне клиента

```
md := metadata.Pairs(
        "timestamp", time.Now().Format(time.StampNano),
        "kn", "vn",
) ❶
mdCtx := metadata.NewOutgoingContext(context.Background(), md) ❷

ctxA := metadata.AppendToOutgoingContext(mdCtx,
      "k1", "v1", "k1", "v2", "k2", "v3") ❸
```

```
// делаем унарный удаленный вызов
response, err := client.SomeRPC(ctxA, someRequest) ❹

// или делаем потоковый удаленный вызов
stream, err := client.SomeStreamingRPC(ctxA) ❺
```

❶ Создаем метаданные.

❷ Создаем новый контекст с новыми метаданными.

❸ Добавляем дополнительные метаданные в существующий контекст.

❹ Унарный RPC на основе нового контекста с метаданными.

❺ Тот же контекст можно использовать и для потокового RPC.

Таким образом, метаданные, которые принимает клиент, нужно воспринимать либо как заголовки, либо как заключительные блоки. В листинге 5.12 показан код на Go с примерами получения метаданных в двух стилях RPC: унарном и потоковом.

Листинг 5.12. Чтение метаданных gRPC на стороне клиента

```
var header, trailer metadata.MD ❶

// ***** унарный RPC *****

r, err := client.SomeRPC( ❷
    ctx,
    someRequest,
    grpc.Header(&header),
    grpc.Trailer(&trailer),
)

// обрабатываем здесь хеш-таблицу с заголовками и заключительными блоками

// ***** потоковый RPC *****

stream, err := client.SomeStreamingRPC(ctx)

// извлекаем заголовок
header, err := stream.Header() ❸

// извлекаем заключительный блок
trailer := stream.Trailer() ❹

// обрабатываем хеш-таблицу с заголовками и заключительными блоками
```

❶ Переменная для хранения заголовка и заключительного блока, возвращенных удаленным вызовом.

❷ Передаем ссылку на заголовок и заключительный блок, чтобы сохранить значения для унарного RPC.

❸ Получаем заголовки из потока.

❹ Получаем из потока заключительные блоки, которые используются для отправки кодов и сообщений о состоянии.

Получив значения из соответствующих операций RPC, вы можете обработать их в виде обычной хеш-таблицы и извлечь нужные вам метаданные.

Теперь перейдем к обработке метаданных на стороне сервера.

Отправка и получение метаданных на стороне сервера

Получение метаданных на стороне сервера — довольно простой и понятный процесс. В языке Go для этого достаточно выполнить `metadata.FromIncomingContext(ctx)` внутри реализации вашего удаленного метода (листинг 5.13).

Листинг 5.13. Чтение метаданных на стороне gRPC-сервера

```
func (s *server) SomeRPC(ctx context.Context,
    in *pb.someRequest) (*pb.someResponse, error) { ❶
    md, ok := metadata.FromIncomingContext(ctx) ❷
    // некие действия с метаданными
}

func (s *server) SomeStreamingRPC(
    stream pb.Service_SomeStreamingRPCServer) error { ❸
    md, ok := metadata.FromIncomingContext(stream.Context()) ❹
    // некие действия с метаданными
}
```

❶ Унарный RPC.

❷ Считываем хеш-таблицу с метаданными из входящего контекста удаленного метода.

❸ Потоковый RPC.

❹ Получаем контекст из потока и считываем из него метаданные.

Чтобы отправить метаданные на серверной стороне, задействуйте заголовок или заключительный блок. Здесь используется метод создания метаданных, обсуждаемый в предыдущем подразделе. В листинге 5.14 показан пример кода на Go, в котором метаданные отправляются из унарного и потокового удаленных методов на серверной стороне.

Листинг 5.14. Отправка метаданных на стороне gRPC-сервера

```
func (s *server) SomeRPC(ctx context.Context,
    in *pb.someRequest) (*pb.someResponse, error) {
    // создаем и отправляем заголовок
    header := metadata.Pairs("header-key", "val")
    grpc.SendHeader(ctx, header) ❶
    // создаем и отправляем заключительный блок
    trailer := metadata.Pairs("trailer-key", "val")
    grpc.SetTrailer(ctx, trailer) ❷
}

func (s *server) SomeStreamingRPC(stream pb.Service_SomeStreamingRPCServer) error {
    // создаем и отправляем заголовок
    header := metadata.Pairs("header-key", "val")
    stream.SendHeader(header) ❸
    // создаем и отправляем заключительный блок
    trailer := metadata.Pairs("trailer-key", "val") stream.SetTrailer(trailer) ❹
}
```

❶ Отправляем метаданные в виде заголовка.

❷ Отправляем метаданные вместе с заключительным блоком.

❸ Отправляем метаданные в виде заголовка в потоке.

❹ Отправляем метаданные вместе с заключительным блоком в потоке.

И в унарном, и в потоковом стиле метаданные можно отправлять с помощью метода `grpc.SendHeader`. Вы также можете сделать их частью заключительного блока в текущем контексте, используя метод `grpc.SetTrailer` или `SetTrailer` соответствующего потока.

Теперь обсудим еще один распространенный прием, который применяется при обращении к gRPC-приложениям: сопоставление имен.

Сопоставление имен

DNS-сопоставитель принимает имя сервиса и возвращает список IP-адресов, принадлежащих его серверам. Сопоставитель, показанный в листинге 5.15, сопоставляет `lb.example.grpc.io` с `localhost:50051` и `localhost:50052`.

Листинг 5.15. Реализация DNS-сопоставителя для gRPC на языке Go

```
type exampleResolverBuilder struct{} ❶

func (*exampleResolverBuilder) Build(target resolver.Target,
        cc resolver.ClientConn,
        opts resolver.BuildOption) (resolver.Resolver, error) {

        r := &exampleResolver{ ❷
                target: target,
                cc:     cc,
                addrsStore: map[string][]string{
           exampleServiceName: addrs, ❸
                },
        }
        r.start()
        return r, nil
}
func (*exampleResolverBuilder) Scheme() string { return exampleScheme } ❹

type exampleResolver struct { ❺
        target     resolver.Target
        cc         resolver.ClientConn
        addrsStore map[string][]string
}

func (r *exampleResolver) start() {
        addrStrs := r.addrsStore[r.target.Endpoint]
        addrs := make([]resolver.Address, len(addrStrs))
        for i, s := range addrStrs {
                addrs[i] = resolver.Address{Addr: s}
        }
        r.cc.UpdateState(resolver.State{Addresses: addrs})
}
func (*exampleResolver) ResolveNow(o resolver.ResolveNowOption) {}
func (*exampleResolver) Close() {}

func init() {
        resolver.Register(&exampleResolverBuilder{})
}
```

❶ Построитель (билдер, builder), который отвечает за создание сопоставителя.

❷ Создание демонстрационного сопоставителя, который находит адреса для `lb.example.grpc.io`.

❸ `lb.example.grpc.io` сопоставляется с `localhost:50051` и `localhost:50052`.

❹ Этот сопоставитель создан для схемы `example`.

❺ Структура DNS-сопоставителя.

Таким образом, вы можете реализовать сопоставители для любого реестра сервисов на ваш выбор, включая Consul (www.consul.io), etcd (etcd.io) и Zookeeper (zookeeper.apache.org). Требования к балансировке нагрузки в gRPC могут сильно зависеть от выбранных вами методов развертывания или от сценариев использования. С ростом популярности платформ оркестрации контейнеров, таких как Kubernetes, и более высокоуровневых абстракций наподобие механизмов межсервисного взаимодействия логику балансировки нагрузки все реже реализуют на стороне клиента. В главе 7 мы исследуем некоторые практические рекомендации по локальному развертыванию gRPC-приложений в контейнерах и Kubernetes.

Теперь поговорим об одном из самых распространенных элементов gRPC-приложений — о балансировке нагрузки — и посмотрим, как в этом контексте могут пригодиться DNS-сопоставители.

Балансировка нагрузки

При подготовке gRPC-приложения к использованию в промышленных условиях необходимо убедиться в том, что оно удовлетворяет требованиям высокой производительности и масштабируемости. Поэтому в реальных проектах gRPC-сервер всегда запускается в нескольких экземплярах, и для распределения между ними удаленных вызовов нужен некий механизм. Данная роль отводится балансировщику нагрузки. В gRPC балансировщики обычно применяются как *на стороне клиента*, так и *в виде прокси-сервера*. Сначала рассмотрим второй вариант.

Прокси-сервер для балансировки нагрузки

При использовании этого подхода клиент направляет свои удаленные вызовы к прокси-серверу (рис. 5.6), а тот направляет их к доступным внутренним gRPC-серверам, которые реализуют логику для обслуживания этих вызовов. Прокси-сервер следит за тем, насколько загружены внутренние сервисы, и задействует разные алгоритмы для распределения трафика между ними.

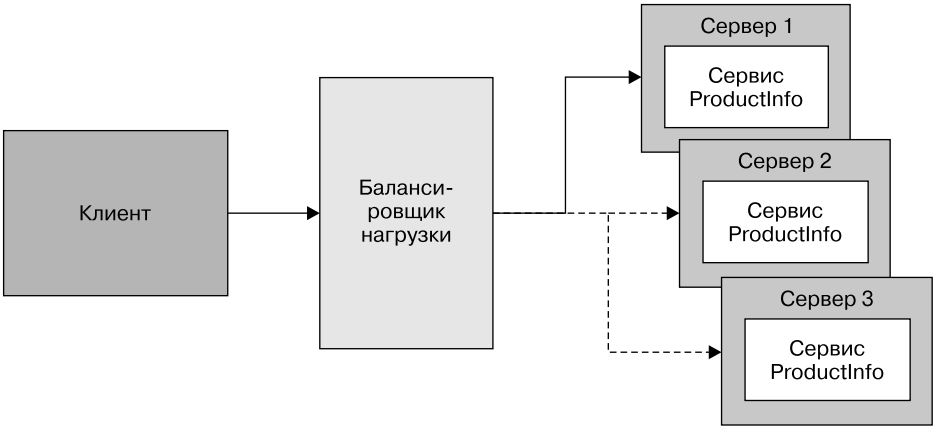


Рис. 5.6. Клиентское приложение обращается к балансировщику нагрузки, за которым находится несколько gRPC-сервисов

gRPC-клиенты ничего не знают о топологии внутренних сервисов; им известно только о конечной точке балансировщика нагрузки. Поэтому для распределения трафика клиентская сторона не требует никаких изменений; она просто должна задействовать конечную точку балансировщика в качестве места назначения всех своих gRPC-соединений. Внутренние сервисы могут сообщать прокси-серверу о своей загруженности, а тот, в свою очередь, использует данную информацию при распределении нагрузки.

Для gRPC-приложения теоретически подойдет любой балансировщик нагрузки с поддержкой HTTP/2. Но эта поддержка должна быть полноценной. Как следствие, лучше выбирать балансировщики, специально предназначенные для gRPC. Например, вы можете использовать такие решения, как Nginx (oreil.ly/QH_1c), Envoy proxy (www.envoyproxy.io) и т. д.

Если вы не используете балансировщик нагрузки, то распределение трафика можно реализовать в рамках клиентского приложения. Рассмотрим этот подход подробно.

Балансировка нагрузки на стороне клиента

Балансировку нагрузки можно реализовать не только в виде промежуточного прокси-сервера, но и на уровне gRPC-клиента. При использовании этого подхода клиент знает о существовании разных gRPC-серверов и сам выбирает один из них при каждом удаленном запросе. На рис. 5.7 показано, что балансировка нагрузки может быть как частью клиентского приложения (это называется *толстым клиентом*), так и прерогативой отдельного сервера, известного как ассоциативный балансировщик. Последний выдает клиенту адрес наиболее подходящего gRPC-сервера, к которому тот может подключиться напрямую.

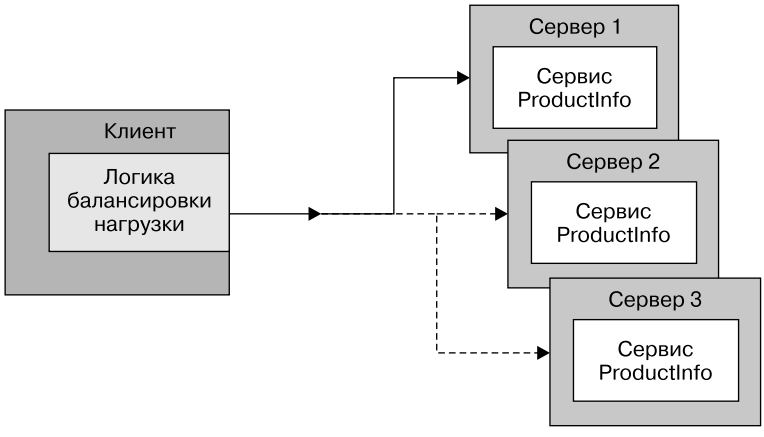


Рис. 5.7. Балансировка нагрузки на стороне клиента

Чтобы вам было легче понять, как выполняется балансировка нагрузки на клиентской стороне, рассмотрим пример толстого клиента, написанного на Go. Допустим, у нас есть два внутренних gRPC-сервиса на портах :50051 и :50052, которые вместе с ответами возвращают адрес своего сервера. Их можно считать участниками одного кластера. Теперь предположим, что нам нужно разработать два клиентских gRPC-приложения: первое будет выбирать подходящий gRPC-сервер с помощью алгоритма циклического перебора

(который чередует серверы), а второе всегда обращается к первой конечной точке в списке. Реализация этих толстых клиентов с поддержкой балансировки нагрузки представлена в листинге 5.16. Как видите, клиент направляет свои вызовы по адресу `example:///lb.example.grpc.io`. Здесь используется схема `example` и имя сервера `lb.example.grpc.io`. Все будет выглядеть так, будто DNS-сопоставитель обнаруживает абсолютное значение адреса внутреннего сервиса. Используя возвращенный им список значений, gRPC-приложение выполняет разные алгоритмы для выбора подходящего сервера. Алгоритм указывается с помощью выражения `grpc.WithBalancerName("round_robin")`.

Листинг 5.16. Балансировка нагрузки на стороне толстого клиента

```
pickfirstConn, err := grpc.Dial(
                fmt.Sprintf("%s:///%s",
        // exampleScheme      = "example"
        // exampleServiceName = "lb.example.grpc.io"
        exampleScheme, exampleServiceName), ❶
        // по умолчанию используется параметр "pick_first" ❷
                grpc.WithBalancerName("pick_first"),

                grpc.WithInsecure(),)
if err != nil {
    log.Fatalf("did not connect: %v", err)
}
defer pickfirstConn.Close()

log.Println("==== Calling helloworld.Greeter/SayHello " +
        "with pick_first ====")
makeRPCs(pickfirstConn, 10)

// устанавливаем еще одно соединение ClientConn с помощью алгоритма round_robin
roundrobinConn, err := grpc.Dial(
    fmt.Sprintf("%s:///%s", exampleScheme, exampleServiceName),
    // "example:///lb.example.grpc.io"
    grpc.WithBalancerName("round_robin"), ❸
    grpc.WithInsecure(),
)
if err != nil {
    log.Fatalf("did not connect: %v", err)
}
defer roundrobinConn.Close()

log.Println("==== Calling helloworld.Greeter/SayHello " +
        "with round_robin ====")
makeRPCs(roundrobinConn, 10)
```

❶ Устанавливаем gRPC-соединение с помощью схемы `example` и имени сервиса. Протокол возвращается сопоставителем в рамках клиентского приложения.

❷ Указываем алгоритм балансировки нагрузки, который выбирает первый сервер в списке конечных точек.

❸ Балансируем нагрузки с помощью алгоритма циклического перебора.

Поддержка этих двух алгоритмов балансировки нагрузки, `pick_first` и `round_robin`, встроена в gRPC. Алгоритм `pick_first` пытается соединиться с первым адресом и в случае успеха использует его для всех дальнейших удаленных вызовов; если попытка не удалась, берется следующий адрес. Алгоритм `round_robin` соединяется со всеми доступными ему адресами и поочередно применяет их для выполнения удаленных вызовов.

В примере балансировки нагрузки на стороне клиента, показанном в листинге 5.16, DNS-сопоставитель возвращает схему `example` с логикой обнаружения значений конечных точек URL. Теперь поговорим еще об одном механизме gRPC, который используется для отправки больших объемов данных, — о сжатии.

Сжатие

Эффективное использование пропускной способности сети при выполнении удаленных вызовов между клиентом и сервисами возможно в случае сжатия. На клиентской стороне это можно реализовать за счет выбора компрессора. Например, в языке Go для этого достаточно выполнить команду `client.AddOrder(ctx, &order1, grpc.UseCompressor(gzip.Name))`; пакет `gzip` имеет путь `"google.golang.org/grpc/encoding/gzip"`.

Серверная сторона автоматически выбирает компрессоры для декодирования запросов и кодирования ответов. В Go для регистрации компрессора достаточно импортировать пакет `"google.golang.org/grpc/encoding/gzip"` в серверном gRPC-приложении. Сервер всегда отвечает с помощью того метода сжатия, который указал клиент. Если соответствующий компрессор не был зарегистрирован, то клиенту возвращается состояние `Unimplemented`.

Резюме

Разработка реальных gRPC-приложений, готовых к промышленному использованию, не ограничивается определением интерфейса сервиса, генерацией серверного и клиентского кода и реализацией бизнес-логики. Как было показано в этой главе, протокол gRPC предлагает целый ряд возможностей, которые могут понадобиться в настоящих проектах, включая использование перехватчиков, крайние сроки, механизм отмены и обработку ошибок.

В следующей главе мы подробно обсудим тему безопасности gRPC-приложений и то, как они используются.

ГЛАВА 6

Безопасность в gRPC

gRPC-приложения взаимодействуют удаленно, по сети. Для этого каждое из них должно предоставлять точку входа, к которой можно обращаться. Такое решение сомнительно с точки зрения безопасности. Чем больше точек входа, тем шире поверхность атаки и выше риск взлома. Поэтому в любом реальном сценарии использования взаимодействие и точки входа необходимо защищать. Каждое gRPC-приложение должно поддерживать зашифрованные запросы и ответы, шифровать все вызовы между своими узлами, аутентифицировать и подписывать все сообщения и т. д.

В этой главе мы рассмотрим основы безопасности и ряд методик, позволяющих реализовать защиту на прикладном уровне. Говоря простым языком, вы узнаете, как защитить каналы взаимодействия микросервисов и организовать аутентификацию пользователей с контролем доступа.

Начнем с защиты каналов взаимодействия.

Аутентификация gRPC-канала с помощью TLS

Протокол защиты транспортного уровня (Transport Level Security, TLS) предназначен для шифрования и обеспечения целостности данных, передающихся между приложениями. В gRPC он используется для установления безопасного соединения между клиентом и сервером. Согласно спецификации TLS (oreil.ly/n4iIE) безопасное соединение должно обладать как минимум одним из следующих свойств.

- *Секретность*. Для шифрования и расшифровки данных применяется один и тот же (секретный) ключ (так называемая симметричная криптография). Ключ уникален в рамках каждого соединения и генерируется на основе общего секретного ключа, выбранного обеими сторонами в начале сессии.

- *Надежность*. Каждое сообщение проходит проверку целостности, чтобы потеря или модификация передаваемых данных не могли остаться незамеченными.

Таким образом, данные должны передаваться по безопасному соединению. Защита gRPC-соединений с помощью TLS не требует больших усилий, поскольку данный механизм встроен в библиотеку gRPC. Это способствует использованию TLS в целях аутентификации и шифрования взаимодействия.

Как же включить безопасность транспортного уровня в gRPC-соединениях? Защитить передачу данных между клиентом и сервером можно с помощью однонаправленного подхода, а также двунаправленного (известного как взаимный TLS, или mTLS). В следующих подразделах мы обсудим оба варианта.

Однонаправленное защищенное соединение

В однонаправленном соединении проверкой подлинности занимается только клиент. В начале сессии сервер передает клиенту свой открытый сертификат, чтобы тот его проверил. Для этого используется удостоверяющий центр (certificate authority, CA). Убедившись в подлинности сертификата, клиент отправляет данные, зашифрованные с помощью закрытого ключа.

Роль CA играет доверенная организация, которая администрирует и выдает сертификаты и открытые ключи, применяемые для безопасного взаимодействия в публичной сети.

Включить TLS можно, сначала создав следующие сертификаты и ключи:

- `server.key` — закрытый RSA-ключ, с помощью которого подписывается и аутентифицируется открытый ключ;
- `server.pem/server.crt` — самоподписанные открытые X.509-ключи для публичного распространения.

Чтобы сгенерировать ключи, можно использовать открытый набор инструментов под названием OpenSSL, предназначенный для работы с протоколами TLS и SSL. Он поддерживает генерацию закрытых ключей, имеющих разные размеры и пароли, публичные сертификаты и т. д. Для удобного создания ключей и сертификатов можно применять и другие инструменты, такие как mkcert (mkcert.dev) и certstrap (oreil.ly/Mu4Q6).

Алгоритм RSA назван в честь трех его создателей (на основе первых букв их фамилий): Rivest, Shamir и Adleman. Это одна из самых популярных криптографических систем с открытым ключом, которая широко применяется для безопасной передачи данных. В RSA для шифрования используется открытый (публично доступный) ключ, а для расшифровки — закрытый. Идея заключается в том, что для расшифровки сообщений, зашифрованных с помощью открытого ключа, необходим закрытый ключ (по крайней мере, если результат нужен в разумные сроки).

Мы не станем описывать процесс генерации ключей, которые являются самоподписанными сертификатами; соответствующие пошаговые инструкции можно найти в файле README в репозитории с исходным кодом.

Представьте, что у нас уже есть закрытый ключ и публичный сертификат. Воспользуемся ими для организации безопасного клиент-серверного взаимодействия в нашем интернет-магазине, который мы обсуждали в главах 1 и 2.

Включение однонаправленного безопасного соединения на стороне сервера

Это самый простой способ зашифровать взаимодействие клиента и сервера. Сервер должен быть инициализирован с помощью открытого и закрытого ключей. Мы покажем, как это делается, на примере gRPC-сервера, написанного на Go.

Чтобы наш сервер поддерживал безопасные соединения, нужно изменить реализацию его главной функции, как показано в листинге 6.1.

Листинг 6.1. Безопасная реализация gRPC-сервера для обслуживания сервиса ProductInfo

```
package main

import (
  "crypto/tls"
  "errors"
  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/credentials"
  "log"
  "net"
)
```

```
var (
  port = ":50051"
  crtFile = "server.crt"
  keyFile = "server.key"
)

func main() {
  cert, err := tls.LoadX509KeyPair(crtFile,keyFile) ❶
  if err != nil {
     log.Fatalf("failed to load key pair: %s", err)
  }
  opts := []grpc.ServerOption{
     grpc.Creds(credentials.NewServerTLSFromCert(&cert)) ❷
  }

  s := grpc.NewServer(opts...) ❸

  pb.RegisterProductInfoServer(s, &server{}) ❹

  lis, err := net.Listen("tcp", port) ❺
  if err != nil {
     log.Fatalf("failed to listen: %v", err)
  }

  if err := s.Serve(lis); err != nil { ❻
     log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Считываем и анализируем открытый/закрытый ключи и создаем сертификат, чтобы включить TLS.

❷ Включаем TLS для всех входящих соединений, используя сертификаты для аутентификации.

❸ Создаем новый экземпляр gRPC-сервера, передавая ему аутентификационные данные.

❹ Регистрируем реализованный сервис на только что созданном gRPC-сервере с помощью сгенерированных API.

❺ Начинаем прослушивать TCP-соединение на порте 50 051.

❻ Привязываем gRPC-сервер к прослушивателю и ждем появления сообщений на порте 50 051.

Теперь клиент, отправляющий запросы серверу, сможет проверить подлинность его сертификата. Отредактируем клиентский код, чтобы он мог общаться с этим сервером.

Включение однонаправленного безопасного соединения на стороне клиента

Чтобы подключиться к серверу, клиент должен получить его самоподписанный публичный сертификат. Отредактируем наш клиентский код на языке Go, как показано в листинге 6.2.

Листинг 6.2. Безопасное клиентское gRPC-приложение

```
package main

import (
  "log"

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc/credentials"
  "google.golang.org/grpc"
)

var (
  address = "localhost:50051"
  hostname = "localhost
  crtFile = "server.crt"
)

func main() {
  creds, err := credentials.NewClientTLSFromFile(crtFile, hostname) ❶
  if err != nil {
     log.Fatalf("failed to load credentials: %v", err)
  }
  opts := []grpc.DialOption{
     grpc.WithTransportCredentials(creds), ❷
  }

  conn, err := grpc.Dial(address, opts...) ❸
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close() ❺
  c := pb.NewProductInfoClient(conn) ❹

  .... // пропускаем вызов удаленного метода
}
```

❶ Считываем и анализируем публичный сертификат, чтобы включить TLS.

❷ Указываем аутентификационные данные для транспортного протокола с помощью `DialOption`.

❸ Устанавливаем безопасное соединение с сервером, передавая параметры аутентификации.

❹ Передаем соединение и создаем заглушку. Ее экземпляр содержит все удаленные методы, которые можно вызвать на сервере.

❺ Закрываем соединение после окончания работы.

Это довольно простой процесс. Нам нужно было добавить всего три новые строчки и отредактировать одну существующую. Сначала мы создали объект `creds` из файла с открытым серверным ключом, затем передали его функции `grpc.Dial`. Благодаря этому при создании каждого соединения клиент и сервер будут обмениваться аутентификационными данными TLS (данный процесс называется рукопожатием).

В однонаправленном соединении аутентификацию проходит только сервер. В следующем подразделе аутентифицируем обе стороны (клиент и сервер).

Включение безопасного соединения mTLS

Основное назначение mTLS состоит в том, чтобы сервер мог контролировать клиентские приложения, которые подключаются к нему. В отличие от однонаправленного TLS-соединения сервер принимает запросы от ограниченной группы проверенных клиентов. Обе стороны обмениваются друг с другом публичными сертификатами и проверяют их подлинность. Соединение устанавливается по следующему принципу.

1. Клиент отправляет серверу запрос, чтобы получить доступ к защищенной информации.
2. Сервер возвращает клиенту сертификат X.509.
3. Клиент проверяет полученный сертификат в соответствующем удостоверяющем центре.
4. В случае успешной проверки клиент передает серверу свой сертификат.

5. Теперь уже сервер проверяет сертификат клиента в удостоверяющем центре.
6. В случае успеха сервер открывает доступ к защищенным данным.

Чтобы включить mTLS в нашем примере, сначала нужно понять, как поступать с клиентским и серверным сертификатами. Следует создать удостоверяющий центр с самоподписанными сертификатами и выполнить запросы на подпись сертификатов для клиента и сервера. Как и в случае с однонаправленным безопасным соединением, генерация ключей и сертификатов возможна с помощью утилиты OpenSSL.

Представим, что у нас уже есть все сертификаты, необходимые для включения mTLS в целях клиент-серверного взаимодействия. В результате корректной процедуры генерации у вас должны получиться следующие ключи и сертификаты:

- `server.key` — закрытый RSA-ключ сервера;
- `server.crt` — публичный сертификат сервера;
- `client.key` — закрытый RSA-ключ клиента;
- `client.crt` — публичный сертификат клиента;
- `ca.crt` — публичный сертификат удостоверяющего центра, с помощью которого подписываются все публичные сертификаты.

Сначала отредактируем серверную часть нашего примера, чтобы напрямую создать пары ключей X.509 и сгенерировать пул сертификатов на основе открытого ключа удостоверяющего центра.

Включение mTLS на gRPC-сервере

Чтобы включить mTLS на сервере, написанном на Go, нужно обновить реализацию его главной функции, как показано в листинге 6.3.

Листинг 6.3. Безопасная реализация gRPC-сервера для обслуживания сервиса ProductInfo

```
package main

import (
  "crypto/tls"
```

```
  "crypto/x509"
  "errors"
  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/credentials"
  "io/ioutil"
  "log"
  "net"
)

var (
  port = ":50051"
  crtFile = "server.crt"
  keyFile = "server.key"
  caFile = "ca.crt"
)

func main() {
  certificate, err := tls.LoadX509KeyPair(crtFile, keyFile) ❶
  if err != nil {
     log.Fatalf("failed to load key pair: %s", err)
  }

  certPool := x509.NewCertPool() ❷
  ca, err := ioutil.ReadFile(caFile)
  if err != nil {
     log.Fatalf("could not read ca certificate: %s", err)
  }

  if ok := certPool.AppendCertsFromPEM(ca); !ok { ❸
     log.Fatalf("failed to append ca certificate")
  }

  opts := []grpc.ServerOption{
    // включаем TLS для всех входящих соединений
    grpc.Creds( ❹
       credentials.NewTLS(&tls.Config {
          ClientAuth:   tls.RequireAndVerifyClientCert,
          Certificates: []tls.Certificate{certificate},
          ClientCAs:    certPool,
          },
       )),
  }

  s := grpc.NewServer(opts...) ❺
  pb.RegisterProductInfoServer(s, &server{}) ❻
  lis, err := net.Listen("tcp", port) ❼
  if err != nil {
```

```
    log.Fatalf("failed to listen: %v", err)
  }

  if err := s.Serve(lis); err != nil { ❽
    log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Создаем пары ключей X.509 непосредственно из ключа и сертификата сервера.

❷ Генерируем пул сертификатов в удостоверяющем центре.

❸ Добавляем клиентские сертификаты из удостоверяющего центра в сгенерированный пул.

❹ Включаем TLS для всех входящих соединений путем создания аутентификационных данных.

❺ Создаем новый экземпляр gRPC-сервера, передавая ему аутентификационные данные.

❻ Регистрируем сервис на только что созданном gRPC-сервере, используя сгенерированные API.

❼ Начинаем прослушивать TCP-соединение на порте 50 051.

❽ Привязываем gRPC-сервер к прослушивателю и ждем появления сообщений на порте 50 051.

Теперь сервер будет принимать запросы только от проверенных клиентов. Отредактируем наш клиентский код, чтобы он мог общаться с этим сервером.

Включение mTLS для gRPC-клиента

Чтобы установить соединение, клиент должен выполнить аналогичные шаги. Отредактированный клиентский код на Go показан в листинге 6.4.

Листинг 6.4. Безопасное клиентское gRPC-приложение на Go

```
package main

import (
  "crypto/tls"
  "crypto/x509"
```

```
  "io/ioutil"
  "log"

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/credentials"
)

var (
  address = "localhost:50051"
  hostname = "localhost"
  crtFile = "client.crt"
  keyFile = "client.key"
  caFile = "ca.crt"
)

func main() {
  certificate, err := tls.LoadX509KeyPair(crtFile, keyFile) ❶
  if err != nil {
     log.Fatalf("could not load client key pair: %s", err)
  }

  certPool := x509.NewCertPool() ❷
  ca, err := ioutil.ReadFile(caFile)
  if err != nil {
     log.Fatalf("could not read ca certificate: %s", err)
  }

  if ok := certPool.AppendCertsFromPEM(ca); !ok { ❸
     log.Fatalf("failed to append ca certs")
  }

  opts := []grpc.DialOption{
     grpc.WithTransportCredentials( credentials.NewTLS(&tls.Config{ ❹
        ServerName:   hostname, // Примечание: это обязательно!
        Certificates: []tls.Certificate{certificate},
        RootCAs:      certPool,
     })),
  }

  conn, err := grpc.Dial(address, opts...) ❺
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close() ❼
  c := pb.NewProductInfoClient(conn) ❻

  .... // пропускаем вызов удаленного метода
}
```

❶ Создаем пары ключей X.509 непосредственно из ключа и сертификата сервера.

❷ Генерируем пул сертификатов в удостоверяющем центре.

❸ Добавляем клиентские сертификаты из удостоверяющего центра в сгенерированный пул.

❹ Указываем транспортные аутентификационные данные в виде параметров соединения. Поле `ServerName` должно быть равно значению `Common Name`, указанному в сертификате.

❺ Устанавливаем безопасное соединение с сервером и передаем параметры.

❻ Передаем соединение и создаем экземпляр заглушки, который содержит все удаленные методы, доступные на сервере.

❼ Закончив работу, закрываем соединение.

Мы создали безопасный канал взаимодействия клиентской и серверной сторон gRPC-приложения, используя как обычный однонаправленный протокол TLS, так и mTLS. Следующим шагом будет включение аутентификации на уровне вызова; это значит, что к каждому вызову будут прикрепляться аутентификационные данные. Решение о дальнейшем выполнении или отклонении вызова принимает сервер, который его получил.

Аутентификация вызовов в gRPC

Протокол gRPC был создан с расчетом на использование серьезных механизмов аутентификации. В предыдущем разделе мы обсуждали шифрование обмена данными между клиентом и сервером с применением TLS. Теперь же пришло время поговорить о том, как проверить подлинность вызывающей стороны и обеспечить контроль доступа с помощью разных методов аутентификации — например, на основе токенов.

Чтобы сделать возможной проверку подлинности вызывающей стороны, протокол gRPC позволяет клиенту внедрять свои аутентификационные данные (такие как имя пользователя и пароль) в каждый вызов. gRPC-сервер может перехватывать запросы клиента и проверять данную информацию.

Чтобы показать, как это работает, рассмотрим для начала простой сценарий аутентификации каждого отдельного вызова.

Использование базовой аутентификации

Самый простой механизм проверки подлинности — *базовая аутентификация*. Клиент отправляет запрос с заголовком `Authorization`, значение которого начинается со слова `Basic`; дальше идет пробел и строка `имя_пользователя:пароль` в кодировке base64. Например, для пользователя `admin` с паролем `admin` заголовок будет выглядеть так:

```
Authorization: Basic YWRtaW46YWRtaW4=
```

В целом протокол gRPC не поощряет применение имени пользователя и пароля для аутентификации вызовов, поскольку эти данные, в отличие от других токенов (JSON Web Tokens [JWTs], токенов доступа OAuth2), не ограничены по времени. То есть при генерации токена мы можем указать срок его действия. У базовой аутентификации такого ограничения нет: пароль остается действительным, пока его не изменят. Если вам нужно включить эту возможность в своем приложении, то обмен учетными данными между клиентом и сервером рекомендуется проводить по безопасному соединению. Мы выбрали данный метод, поскольку он позволяет проще всего объяснить принцип работы аутентификации в gRPC.

Сначала посмотрим, как учетные данные пользователя внедряются в вызов. Поскольку в gRPC нет встроенной поддержки базовой аутентификации, данные необходимо добавлять в контекст клиента вручную. На языке Go это можно легко сделать путем создания отдельной структуры и реализации интерфейса `PerRPCCredentials` (листинг 6.5).

Листинг 6.5. Реализация интерфейса PerRPCCredentials для передачи пользовательских аутентификационных данных

```
type basicAuth struct { ❶
  username string
  password string
}

func (b basicAuth) GetRequestMetadata(ctx context.Context,
  in ...string) (map[string]string, error) { ❷
  auth := b.username + ":" + b.password
```

```
  enc := base64.StdEncoding.EncodeToString([]byte(auth))
  return map[string]string{
     "authorization": "Basic " + enc,
  }, nil
}

func (b basicAuth) RequireTransportSecurity() bool { ❸
  return true
}
```

❶ Определяем структуру для хранения полей, которые нужно внедрять в удаленные вызовы (в нашем случае это такие учетные данные, как имя пользователя и пароль).

❷ Реализуем метод `GetRequestMetadata` и преобразуем данные аутентификации в метаданные запроса. В нашем случае ключ равен `Authorization`, а значение состоит из слова `Basic`, за которым следует строка `<имя_пользователя>:<пароль>` в кодировке base64.

❸ Как уже упоминалось, для передачи этих данных рекомендуется использовать безопасный канал, что мы и делаем.

Реализованный нами объект для хранения аутентификационных данных нужно инициализировать с помощью корректных значений и указать при установлении соединения, как показано в листинге 6.6.

Листинг 6.6. Безопасное клиентское gRPC-приложение с базовой аутентификацией

```
package main

import (
  "log"
  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc/credentials"
  "google.golang.org/grpc"
)

var (
  address = "localhost:50051"
  hostname = "localhost"
  crtFile = "server.crt"
)

func main() {
  creds, err := credentials.NewClientTLSFromFile(crtFile, hostname)
```

```
  if err != nil {
     log.Fatalf("failed to load credentials: %v", err)
  }

  auth := basicAuth{ ❶
    username: "admin",
    password: "admin",
  }

  opts := []grpc.DialOption{
    grpc.WithPerRPCCredentials(auth), ❷
    grpc.WithTransportCredentials(creds),
  }

  conn, err := grpc.Dial(address, opts...)
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close()
  c := pb.NewProductInfoClient(conn)

  .... // Пропускаем вызов удаленного метода
}
```

❶ Инициализируем переменную `auth` с помощью корректных учетных данных (имени пользователя и пароля). Мы будем использовать хранящиеся в ней значения.

❷ Переменная `auth` соответствует интерфейсу, который функция `grpc.WithPerRPCCredentials` принимает в качестве параметра. Поэтому мы можем передать `auth` этой функции.

Теперь во время выполнения вызовов клиент передает серверу дополнительные метаданные, но сервер о них ничего не знает. Научим наш сервер читать и проверять эту информацию, как показано в листинге 6.7.

Листинг 6.7. Безопасный gRPC-сервер с базовой проверкой подлинности пользователя

```
package main

import (
  "context"
  "crypto/tls"
  "encoding/base64"
  "errors"
```

```
  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/codes"
  "google.golang.org/grpc/credentials"
  "google.golang.org/grpc/metadata"
  "google.golang.org/grpc/status"
  "log"
  "net"
  "path/filepath"
  "strings"
)

var (
  port = ":50051"
  crtFile = "server.crt"
  keyFile = "server.key"
  errMissingMetadata = status.Errorf(codes.InvalidArgument, "missing metadata")
  errInvalidToken    = status.Errorf(codes.Unauthenticated, "invalid credentials")
)

type server struct {
  productMap map[string]*pb.Product
}

func main() {
  cert, err := tls.LoadX509KeyPair(crtFile, keyFile)
  if err != nil {
     log.Fatalf("failed to load key pair: %s", err)
  }
  opts := []grpc.ServerOption{
    // включаем TLS для всех входящих соединений
    grpc.Creds(credentials.NewServerTLSFromCert(&cert)),

    grpc.UnaryInterceptor(ensureValidBasicCredentials), ❶
  }

  s := grpc.NewServer(opts...)
  pb.RegisterProductInfoServer(s, &server{})

  lis, err := net.Listen("tcp", port)
  if err != nil {
     log.Fatalf("failed to listen: %v", err)
  }

  if err := s.Serve(lis); err != nil {
     log.Fatalf("failed to serve: %v", err)
  }
}
```

```
func valid(authorization []string) bool {
  if len(authorization) < 1 {
     return false
  }
  token := strings.TrimPrefix(authorization[0], "Basic ")
  return token == base64.StdEncoding.EncodeToString([]byte("admin:admin"))
}

func ensureValidBasicCredentials(ctx context.Context, req interface{}, info
*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) { ❷
  md, ok := metadata.FromIncomingContext(ctx) ❸
  if !ok {
     return nil, errMissingMetadata
  }
  if !valid(md["authorization"]) {
    return nil, errInvalidToken
  }
  // если токен действителен, то продолжаем выполнение обработчика
  return handler(ctx, req)
}
```

❶ Добавляем новый серверный параметр (`grpc.ServerOption`) вместе с TLS-сертификатом сервера. Добавляем с помощью вызова `grpc.UnaryInterceptor` перехватчик, который будет направлять все клиентские запросы к функции `ensureValidBasicCredentials`.

❷ Определяем функцию `ensureValidBasicCredentials` для проверки подлинности вызывающей стороны. Объект `context.Context` содержит нужные нам метаданные, которые будут существовать на протяжении обработки запроса.

❸ Извлекаем метаданные из контекста, получаем значение поля `authentication` и проверяем аутентификационные данные. Ключи внутри `metadata.MD` переводятся в нижний регистр, поэтому при проверке их значений нужно использовать прописные буквы.

Теперь сервер проверяет подлинность клиента при каждом вызове. Это очень простой пример. Логика аутентификации внутри перехватчика может быть более сложной.

Итак, вы получили общее представление о том, как аутентифицируются клиентские запросы. Теперь поговорим о широко распространенном и рекомендованном методе аутентификации на основе токенов (OAuth 2.0).

Использование OAuth 2.0

OAuth 2.0 (oauth.net/2) — фреймворк для делегирования доступа. С его помощью сервисы могут обращаться к определенным ресурсам от имени пользователей, не обладая полными правами доступа (как в случае с базовой аутентификацией). Мы не станем обсуждать OAuth 2.0 во всех подробностях, но вам будет полезно иметь общее представление о том, как работает эта технология, чтобы использовать ее в своих приложениях.

В процессе аутентификации OAuth 2.0 участвуют четыре стороны: клиент, сервер авторизации, сервер ресурсов и владелец ресурсов. Клиент хочет получить доступ к ресурсу, размещенному на соответствующем сервере. Для этого ему следует получить токен (представляющий собой произвольную строку) у сервера авторизации. Данный токен должен быть непредсказуемым и иметь подходящую длину. Клиент, получивший его, может отправить токен серверу ресурсов, который затем свяжется с сервером авторизации. Если владелец ресурса подтвердит подлинность токена, то клиент получит доступ к ресурсу.

Протокол gRPC имеет встроенную поддержку OAuth 2.0, и вы можете включить ее в своих приложениях. Но сначала поговорим о том, как внедрить токен в вызов. В нашем примере нет сервера авторизации, поэтому в качестве значения токена будет использоваться произвольная строка, прописанная в коде. Добавление токенов OAuth в клиентский запрос проиллюстрировано в листинге 6.8.

Листинг 6.8. Безопасное клиентское gRPC-приложение на Go с токеном OAuth

```
package main

import (
  "google.golang.org/grpc/credentials"
  "google.golang.org/grpc/credentials/oauth"
  "log"

  pb "productinfo/server/ecommerce"
  "golang.org/x/oauth2"
  "google.golang.org/grpc"
)
```

```
var (
  address = "localhost:50051"
  hostname = "localhost"
  crtFile = "server.crt"
)

func main() {
  auth := oauth.NewOauthAccess(fetchToken()) ❶

  creds, err := credentials.NewClientTLSFromFile(crtFile, hostname)
  if err != nil {
     log.Fatalf("failed to load credentials: %v", err)
  }

  opts := []grpc.DialOption{
    grpc.WithPerRPCCredentials(auth), ❷
    grpc.WithTransportCredentials(creds),
  }

  conn, err := grpc.Dial(address, opts...)
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close()
  c := pb.NewProductInfoClient(conn)

  .... // пропускаем вызов удаленного метода
}

func fetchToken() *oauth2.Token {
  return &oauth2.Token{
     AccessToken: "some-secret-token",
  }
}
```

❶ Подготавливаем учетные данные для соединения. Но прежде необходимо предоставить значение токена OAuth2. Здесь мы используем строку, прописанную в коде.

❷ Указываем один и тот же токен OAuth в параметрах всех вызовов в рамках одного соединения. Если вы хотите указывать токен для каждого вызова отдельно, то используйте `CallOption`.

Обратите внимание: мы включили безопасный канал, поскольку OAuth требует, чтобы транспортный протокол был защищен. Внутри gRPC мы

предоставили токен, указав его тип в качестве префикса, и прикрепили метаданные с ключом `authorization`.

Как показано в листинге 6.9, в серверном коде нужно добавить аналогичный перехватчик, который будет проверять подлинность клиентского токена, приходящего вместе с запросом.

Листинг 6.9. Безопасный gRPC-сервер с проверкой подлинности пользователя на основе OAuth

```
package main

import (
  "context"
  "crypto/tls"
  "errors"
  "log"
  "net"
  "strings"

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/codes"
  "google.golang.org/grpc/credentials"
  "google.golang.org/grpc/metadata"
  "google.golang.org/grpc/status"
)

// структура для реализации ecommerce/product_info
type server struct {
  productMap map[string]*pb.Product
}

var (
  port = ":50051"
  crtFile = "server.crt"
  keyFile = "server.key"
  errMissingMetadata = status.Errorf(codes.InvalidArgument, "missing metadata")
  errInvalidToken    = status.Errorf(codes.Unauthenticated, "invalid token")
)

func main() {
  cert, err := tls.LoadX509KeyPair(crtFile, keyFile)
  if err != nil {
     log.Fatalf("failed to load key pair: %s", err)
  }
```

```
  opts := []grpc.ServerOption{
     grpc.Creds(credentials.NewServerTLSFromCert(&cert)),
     grpc.UnaryInterceptor(ensureValidToken), ❶
  }

  s := grpc.NewServer(opts...)
  pb.RegisterProductInfoServer(s, &server{})

  lis, err := net.Listen("tcp", port)
  if err != nil {
     log.Fatalf("failed to listen: %v", err)
  }

  if err := s.Serve(lis); err != nil {
     log.Fatalf("failed to serve: %v", err)
  }
}

func valid(authorization []string) bool {
  if len(authorization) < 1 {
     return false
  }
  token := strings.TrimPrefix(authorization[0], "Bearer ")
  return token == "some-secret-token"
}

func ensureValidToken(ctx context.Context, req interface{},
     info *grpc.UnaryServerInfo,
     handler grpc.UnaryHandler) (interface{}, error) { ❷
  md, ok := metadata.FromIncomingContext(ctx)
  if !ok {
     return nil, errMissingMetadata
  }
  if !valid(md["authorization"]) {
     return nil, errInvalidToken
  }
  return handler(ctx, req)
}
```

❶ Добавляем новый серверный параметр (`grpc.ServerOption`) вместе с TLS-сертификатом сервера. Добавляем с помощью вызова `grpc.UnaryInterceptor` перехватчик, который будет направлять все клиентские запросы.

❷ Определяем функцию `ensureValidToken` для проверки подлинности токена. Если тот отсутствует или недействителен, тогда перехватчик бло-

кирует выполнение и возвращает ошибку. В противном случае вызывается следующий обработчик, которому передается контекст и интерфейс.

Перехватчик позволяет настроить проверку токена для всех удаленных вызовов. Для этого можно использовать либо `grpc.UnaryInterceptor`, либо `grpc.StreamInterceptor`, в зависимости от типа сервиса.

Помимо OAuth 2.0, протокол gRPC поддерживает аутентификацию на основе веб-токенов в формате JSON (JSON Web Token, JWT). В следующем подразделе мы покажем, как ее включить.

Использование JWT

JWT — это контейнер для обмена аутентификационной информацией между клиентом и сервером. Если его подписать, то он может выступать в качестве автономного токена доступа, для проверки которого серверу ресурсов не нужно обращаться к серверу аутентификации. Чтобы убедиться в его подлинности, достаточно проанализировать его подпись. Клиент запрашивает доступ у сервера аутентификации; тот проверяет его учетные данные, создает токен JWT и возвращает его клиенту. Вместе с JWT клиент получает доступ к ресурсам.

gRPC имеет встроенную поддержку JWT. JWT-файл, полученный от сервера аутентификации, нужно использовать для создания учетных данных. Фрагмент кода в листинге 6.10 показывает создание учетных данных из JWT-файла (`token.json`) и их передачу клиентскому приложению на Go в виде параметров `DialOption`.

Листинг 6.10. Установление соединения в клиентском приложении на Go с использованием JWT

```
jwtCreds, err := oauth.NewJWTAccessFromFile("token.json") ❶
if err != nil {
  log.Fatalf("Failed to create JWT credentials: %v", err)
}

creds, err := credentials.NewClientTLSFromFile("server.crt",
     "localhost")
if err != nil {
    log.Fatalf("failed to load credentials: %v", err)
}
```

```
opts := []grpc.DialOption{
  grpc.WithPerRPCCredentials(jwtCreds),
  // данные аутентификации для транспортного протокола
  grpc.WithTransportCredentials(creds), ❷
}

// устанавливаем соединение с сервером
conn, err := grpc.Dial(address, opts...)
if err != nil {
  log.Fatalf("did not connect: %v", err)
}
  .... // пропускаем генерацию заглушки и вызов удаленного метода
```

❶ Вызываем `oauth.NewJWTAccessFromFile` с целью инициализации `credentials.PerRPCCredentials`. Для создания учетных данных нужно предоставить файл с действительным токеном.

❷ Настраиваем параметры `DialOption WithPerRPCCredentials`, чтобы токен JWT применялся ко всем удаленным вызовам в рамках одного соединения.

Помимо этих методов аутентификации, можно использовать любой другой механизм — достаточно расширить учетные данные на клиентской стороне и добавить новый перехватчик на стороне сервера. В протокол gRPC также встроена поддержка работы с gRPC-сервисами, развернутыми в Google Cloud. Более подробно о том, как вызывать эти сервисы, мы поговорим в следующем подразделе.

Аутентификация в Google Cloud с использованием токенов

Идентификацией пользователей и контролем их доступа к сервисам, развернутым на платформе Google Cloud, занимается система Extensible Service Proxy (ESP). Она поддерживает различные методы аутентификации, включая токены Firebase, Auth0 и Google ID. В каждом из этих случаев клиент должен указывать в своих запросах действительный токен JWT. Такие токены генерируются отдельно для каждого развернутого сервиса с помощью отдельной служебной учетной записи.

Получив токен, мы можем передать его методу соответствующего сервиса вместе с запросом. Создадим канал, в котором будут передаваться учетные данные, как показано в листинге 6.11.

Листинг 6.11. Установление соединения с конечной точкой Google Cloud в клиентском приложении на Go

```
perRPC, err := oauth.NewServiceAccountFromFile("service-account.json", scope) ❶
if err != nil {
  log.Fatalf("Failed to create JWT credentials: %v", err)
}

pool, _ := x509.SystemCertPool()
creds := credentials.NewClientTLSFromCert(pool, "")

opts := []grpc.DialOption{
  grpc.WithPerRPCCredentials(perRPC),
  grpc.WithTransportCredentials(creds), ❷
}

conn, err := grpc.Dial(address, opts...)
if err != nil {
  log.Fatalf("did not connect: %v", err)
}
.... // пропускаем генерацию заглушки и вызов удаленного метода
```

❶ Вызываем `oauth.NewServiceAccountFromFile` в целях инициализации `credentials.PerRPCCredentials`. Для создания учетных данных нужно предоставить файл с действительным токеном.

❷ Настраиваем параметры `DialOption` `WithPerRPCCredentials`, чтобы аутентификационный токен применялся ко всем удаленным вызовам в рамках одного соединения, — по аналогии с тем, как мы это делали в уже рассмотренных механизмах аутентификации.

Резюме

gRPC-приложение, рассчитанное на использование в промышленных условиях, должно удовлетворять минимальным требованиям безопасности, необходимым для защиты взаимодействия клиента и сервера. Библиотека gRPC поддерживает разные механизмы аутентификации и позволяет реализовывать дополнительные методы проверки подлинности, что делает ее удобным и безопасным средством для общения с другими системами.

Учетные данные в gRPC могут действовать и на уровне канала (такого как TLS), и для каждого отдельного вызова (как в случае с токенами OAuth 2.0,

базовой аутентификацией и т. д.). Эти два подхода можно применять в одном и том же gRPC-приложении. Например, мы можем включить TLS-соединение между клиентом и сервером и притом прикреплять учетные данные к каждому удаленному вызову, который выполняется в рамках этого соединения.

Вы узнали, как использовать оба типа учетных данных в своих gRPC-приложениях. В следующей главе мы применим уже известные концепции и технологии, чтобы создать реальный проект, готовый к промышленному использованию. Мы также покажем, как писать тесты для серверного и клиентского кода, развертывать приложения в Docker и Kubernetes и организовывать мониторинг системы, которая работает в промышленной среде.

ГЛАВА 7

Использование gRPC в промышленных условиях

Предыдущие главы были посвящены различным аспектам проектирования и разработки приложений, основанных на gRPC. Теперь пришло время поговорить о том, как нужно обращаться с этими приложениями в промышленных условиях. Мы обсудим модульное и интеграционное тестирование серверного и клиентского кода и покажем, как можно внедрить его в средства непрерывной интеграции. Затем речь пойдет о непрерывном развертывании gRPC-приложений в таких средах, как виртуальные машины (ВМ), Docker и Kubernetes. В финале мы уделим внимание администрированию приложений в промышленной среде и выбору надежной платформы для наблюдения за сервисами. Мы исследуем разные средства наблюдаемости для gRPC-приложений, а также обсудим методы их диагностирования и отладки. Начнем с тестирования.

Тестирование gRPC-приложений

В процессе разработки любых приложений (включая те, что основаны на gRPC) необходимо применять модульное тестирование. А поскольку gRPC-приложения работают с сетью, вы должны тестировать соответствующие функции в их клиентском и серверном коде. Начнем с тестирования gRPC-сервера.

Тестирование gRPC-сервера

Для тестирования gRPC-сервисов часто задействуют клиентские приложения. Тестирование на серверной стороне предусматривает запуск gRPC-сервера с нужным вам сервисом и последующее его соединение с клиентом, который реализует тестовые сценарии. Рассмотрим пример тестового сценария, написанного для сервиса `ProductInfo`. Как показано в листинге 7.1, в языке Go для этого используется универсальный пакет `testing`.

Листинг 7.1. Тестирование gRPC-сервера на языке Go

```
func TestServer_AddProduct(t *testing.T) { ❶
        grpcServer := initGRPCServerHTTP2() ❷
        conn, err := grpc.Dial(address, grpc.WithInsecure()) ❸
        if err != nil {

           grpcServer.Stop()
           t.Fatalf("did not connect: %v", err)
        }
        defer conn.Close()
        c := pb.NewProductInfoClient(conn)

        name := "Sumsung S10"
        description := "Samsung Galaxy S10 is the latest smart phone, launched in
        February 2019"
        price := float32(700.0)
        ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        r, err := c.AddProduct(ctx, &pb.Product{Name: name,
                                Description: description, Price: price}) ❹
        if err != nil { ❺
                t.Fatalf("Could not add product: %v", err)
        }

        if r.Value == "" {
                t.Errorf("Invalid Product ID %s", r.Value)
        }
        log.Printf("Res %s", r.Value)
      grpcServer.Stop()
}
```

❶ Традиционный тест, который запускает сервер и клиент для проверки удаленного метода сервиса.

❷ Запускаем стандартный gRPC-сервер поверх HTTP/2.

❸ Подключаемся к серверному приложению.

❹ Вызываем удаленный метод `AddProduct`.

❺ Проверяем ответ.

Здесь используются стандартные для языка Go тестовые сценарии, поэтому тестирование выполняется так же, как и с любыми другими приложениями. У тестирования серверного gRPC-кода есть лишь одна особенность:

вы должны открыть порт, к которому может подключиться клиент. Если вы не хотите этого делать (или ваша среда не позволяет), то можете использовать библиотеку, которая позволит избежать запуска сервиса на реальном порте. В Go для этого предусмотрен пакет `bufconn` (oreil.ly/gOq46). Он предоставляет реализацию `net.Conn` на основе буфера и сопутствующие возможности, связанные с отправкой и приемом сообщений. Полный пример кода находится в репозитории к данной главе. Если вы предпочитаете Java, то та же процедура написания серверного тестового сценария выполнима с помощью такого фреймворка, как *JUnit*. Кроме того, в Java gRPC-сервер можно запускать в том же процессе, что и сам тест. Полный пример кода на Java доступен в репозитории к этой книге.

Модульное тестирование бизнес-логики удаленных функций, которые вы разрабатываете, можно выполнить, не прибегая к использованию сети. Функции доступны для вызова напрямую, без применения клиента.

Теперь вы знаете, как писать тесты для gRPC-сервисов. Поговорим о тестировании клиентских gRPC-приложений.

Тестирование gRPC-клиента

Один из возможных подходов к тестированию gRPC-клиента заключается в запуске gRPC-сервера с предварительно реализованным макетом сервиса. Но это не самая простая задача, поскольку вам придется открывать порт и подключаться к серверу. Во избежание данного действия можно воспользоваться фреймворком для создания макетов. Это даст разработчикам возможность писать легковесные модульные тесты, позволяющие проверять функциональность клиентского кода без обращения к удаленному серверу.

Если вы пишете свой gRPC-клиент на Go, то для создания макета клиентского интерфейса (с помощью сгенерированного кода) и программного определения значений, которые принимают и возвращают его методы, можно использовать Gomock (oreil.ly/8GAWB). Данный фреймворк позволяет сгенерировать макеты интерфейсов клиентского gRPC-приложения, применив следующую команду:

```
mockgen github.com/grpc-up-and-running/samples/ch07/grpc-docker/go/proto-gen \
ProductInfoClient > mock_prodinfo/prodinfo_mock.go
```

В качестве интерфейса, для которого нужно создать макет, мы указали `ProductInfoClient`. В своем тестовом коде вы можете импортировать пакет, сгенерированный утилитой `mockgen`, а также пакет `gomock`, который поможет в написании модульных тестов для клиентской логики. Как показано в листинге 7.2, вы можете создать макет и указать результат, который должен возвращаться при вызове его метода.

Листинг 7.2. Тест для gRPC-клиента с использованием Gomock

```
func TestAddProduct(t *testing.T) {
        ctrl := gomock.NewController(t)
        defer ctrl.Finish()
        mocklProdInfoClient := NewMockProductInfoClient(ctrl) ❶
     ...
        req := &pb.Product{Name: name, Description: description, Price: price}
        mocklProdInfoClient. ❷
         EXPECT().AddProduct(gomock.Any(), &rpcMsg{msg: req},). ❸
         Return(&wrapper.StringValue{Value: "ABC123" + name}, nil) ❹

        testAddProduct(t, mocklProdInfoClient) ❺
}

func testAddProduct(t *testing.T, client pb.ProductInfoClient) {
        ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        ...

        r, err := client.AddProduct(ctx, &pb.Product{Name: name,
        Description: description, Price: price})

        // проверяем ответ
}
```

❶ Создаем макет, который ожидает вызова своих удаленных методов.

❷ Программируем макет.

❸ Ожидаем вызова метода `AddProduct`.

❹ Возвращаем фиктивное значение для ID заказа.

❺ Вызываем метод теста, который обращается к удаленному методу клиентской заглушки.

В Java для тестирования клиентского приложения можно использовать Mockito (site.mockito.org) и соответствующую межпроцессную серверную

реализацию gRPC. Более подробно эти примеры освещены в репозитории исходного кода. Подготовив тесты для серверной и клиентской сторон, вы можете внедрить их в свое средство непрерывной интеграции.

Необходимо понимать, что макет gRPC-сервера ведет себя не совсем так, как оригинал. Если вы не реализуете всю логику обработки ошибок, присутствующую в gRPC-сервере, то некоторые возможности могут оказаться недоступными для тестирования. На практике вы можете проверить одну часть функциональности с помощью макета, а другую — с использованием настоящей реализации сервера. Теперь посмотрим, как проводить нагрузочное тестирование и оценивать производительность gRPC-приложений.

Нагрузочное тестирование

Традиционные средства плохо подходят для нагрузочного тестирования и оценки производительности gRPC-приложений, так как последние в той или иной степени привязаны к конкретному протоколу (например, HTTP). Поэтому в случае с gRPC необходимо использовать специальные инструменты, способные генерировать виртуальную нагрузку на gRPC-сервер в виде удаленных вызовов.

Один из таких инструментов — ghz (ghz.sh); это утилита командной строки, написанная на Go. Ее можно использовать как для локального тестирования и отладки сервисов, так и для регрессивного тестирования в автоматизированных средах непрерывной интеграции. Например, вы можете выполнить нагрузочный тест с помощью следующего кода:

```
ghz --insecure \
  --proto ./greeter.proto \
  --call helloworld.Greeter.SayHello \
  -d '{"name":"Joe"}'\
  -n 2000 \
  -c 20 \

  0.0.0.0:50051
```

Здесь мы вызываем из сервиса `Greeter` удаленный метод `SayHello`, не используя механизмы безопасности. Мы можем указать общее количество запросов (`-n 2000`) и степень параллелизма (20 потоков). Результаты можно генерировать в разных форматах.

Разработанные вами серверные и клиентские тесты впоследствии можно внедрить в систему непрерывной интеграции.

Непрерывная интеграция

На случай, если вы незнакомы с понятием *непрерывной интеграции* (continuous integration, CI), отметим, что это методика, согласно которой разработчики должны постоянно интегрировать свой код в общий репозиторий. Во время каждой фиксации код проходит процесс автоматической сборки, что позволяет выявлять проблемы на ранних этапах. В gRPC-приложениях клиентская и серверная стороны часто независимы друг от друга и могут быть реализованы с помощью разных технологий. Поэтому в рамках процесса CI вам нужно будет проверить работу клиентского и серверного кода, задействуя методики модульного и интеграционного тестирования, с которыми мы познакомились выше. Затем, в зависимости от используемого вами языка, вы можете внедрить средства тестирования (такие как пакет `testing` в Go или JUnit в Java) своих приложений в свою систему CI. Например, если ваши тесты написаны на Go, то легко интегрируются с такими инструментами, как Jenkins (jenkins.io), TravisCI (travis-ci.com), Spinnaker (www.spinnaker.io) и т. д.

Наладив тестирование и непрерывную интеграцию своего gRPC-приложения, вы должны позаботиться о его развертывании.

Развертывание

Рассмотрим разные методы развертывания разрабатываемых gRPC-приложений. Если вы хотите запускать gRPC-сервер и клиент локально или в ВМ, то процесс развертывания будет сводиться к генерации двоичных файлов для тех языков программирования, которые вы используете в своем коде. В таких случаях для обеспечения масштабируемости и высокой доступности серверных gRPC-приложений обычно применяют стандартные средства наподобие балансировщиков нагрузки с поддержкой протокола gRPC.

Тем не менее большинство современных приложений развертываются в виде контейнеров, так что вам будет полезно узнать, как это делается в контексте gRPC. Начнем с Docker — стандартной платформы для контейнерного развертывания приложений.

Развертывание в Docker

Docker (www.docker.com) — открытая платформа для разработки, доставки и выполнения приложений. С ее помощью вы можете отделить свой прикладной код от инфраструктуры. Docker позволяет упаковывать и запускать приложения в изолированной среде, *контейнере*; вы можете запустить несколько контейнеров на одном и том же сервере. Контейнеры гораздо более легковесны, чем ВМ, и могут выполняться непосредственно в ядре операционной системы.

Рассмотрим некоторые примеры развертывания gRPC-приложений в виде контейнера Docker.

Основы работы с Docker выходят за рамки нашей книги. Поэтому, если вы незнакомы с данной системой, то можете обратиться к официальной документации (docs.docker.com) и другим ресурсам.

Разработав серверное gRPC-приложение, вы можете создать для него контейнер Docker. В листинге 7.3 показан файл Dockerfile gRPC-сервера, написанного на Go. Здесь вы можете отметить много концепций, характерных для gRPC. Мы выполняем многоэтапную сборку: на первом этапе собираем приложение, а на втором запускаем его, используя намного более легковесную среду. Перед сборкой приложения в контейнер добавляется сгенерированный серверный код.

Листинг 7.3. Dockerfile для gRPC-сервера на Go

```
# Многоэтапная сборка

# этап I: ❶
FROM golang AS build
ENV location /go/src/github.com/grpc-up-and-running/samples/ch07/grpc-docker/go
WORKDIR ${location}/server

ADD ./server ${location}/server
ADD ./proto-gen ${location}/proto-gen

RUN go get -d ./... ❷
RUN go install ./... ❸

RUN CGO_ENABLED=0 go build -o /bin/grpc-productinfo-server ❹
```

```
# этап II: ❺
FROM scratch
COPY --from=build /bin/grpc-productinfo-server /bin/grpc-productinfo-server ❻
ENTRYPOINT ["/bin/grpc-productinfo-server"]
EXPOSE 50051
```

❶ Для сборки этой программы требуется только язык Go и Alpine Linux.

❷ Загружаем все зависимости.

❸ Устанавливаем все пакеты.

❹ Собираем серверное приложение.

❺ Собранные программы на Go — самодостаточные исполняемые файлы.

❻ Копируем двоичный файл, собранный на предыдущем этапе, в новое место.

После создания Dockerfile можно собрать образ Docker, используя следующую команду:

```
docker image build -t grpc-productinfo-server -f server/Dockerfile
```

Клиентское gRPC-приложение можно создать аналогичным образом. Изменятся только сетевое имя и порт, к которому оно будет подключаться, поскольку серверный код теперь выполняется в Docker.

Клиентское и серверное gRPC-приложения, запущенные в Docker, должны как-то общаться между собой и с внешним миром (через основную систему). Поэтому нам нужен сетевой слой. Docker поддерживает разные типы сетей для разных сценариев использования. При запуске контейнеров можно указать общую сеть, чтобы клиент мог обнаружить местоположение сервера по его сетевому имени. То есть нужно поправить клиентский код, чтобы он подключался к сетевому имени сервера. Например, в нашем gRPC-приложении на языке Go вместо `localhost` нужно указать следующее:

```
conn, err := grpc.Dial("productinfo:50051", grpc.WithInsecure())
```

Сетевое имя можно не прописывать в коде, а считывать из переменной окружения. После внесения изменений в клиентское приложение вы должны заново собрать образ Docker и запустить его вместе с образом сервера так, как показано ниже:

```
docker run -it --network=my-net --name=productinfo \
    --hostname=productinfo
    -p 50051:50051 grpc-productinfo-server ❶
```

```
docker run -it --network=my-net \
    --hostname=client grpc-productinfo-client ❷
```

❶ Запускаем gRPC-сервер в сети Docket *my-net*, используя сетевое имя `productinfo` и порт 50 051.

❷ Запускаем gRPC-клиент в сети Docker *my-net*.

При запуске контейнеров можно указать сеть Docker, в которой они будут работать. Если они находятся в одной сети, клиентское приложение может обнаружить адрес сервиса по его сетевому имени, указанному в команде `docker run`.

Если вы запускаете небольшое количество контейнеров, которые взаимодействуют относительно простым образом, то вся ваша система может быть целиком построена на основе Docker. Но в большинстве реальных ситуаций контейнерами и их взаимодействием необходимо как-то управлять. Создавать такие системы исключительно с помощью Docker довольно утомительно. Здесь вам может помочь платформа оркестрации контейнеров.

Развертывание в Kubernetes

Kubernetes — открытая платформа для автоматизации развертывания, масштабирования и администрирования контейнеризованных приложений. Само по себе контейнеризованное gRPC-приложение на основе Docker не масштабируется и не предоставляет гарантии высокой доступности. Эти возможности нужно реализовывать отдельно. Kubernetes обладает широким набором подобных характеристик и может взять на себя администрирование и оркестрацию контейнеров.

Kubernetes (kubernetes.io) — надежная и масштабируемая платформа для выполнения контейнеризованных рабочих заданий. Она берет на себя такие функции, как масштабирование, обработка отказов, обнаружение сервисов, управление конфигурацией, обеспечение безопасности, поддержка разных методов развертывания и др.

Основы работы с Kubernetes выходят за рамки этой книги. За дополнительной информацией советуем обратиться к официальной документации (oreil.ly/csW_8) и другим ресурсам[1].

[1] См.: *Бернс Б., Вильяльба Э., Штребель Д., Эвенсон Л.* Kubernetes: Лучшие практики. — СПб.: Питер, 2021.

Посмотрим, как развернуть в Kubernetes ваше серверное gRPC-приложение.

Ресурс развертывания Kubernetes для gRPC-сервера

Чтобы выполнить развертывание в Kubernetes, нужно сначала создать для вашего gRPC-сервера контейнер Docker. В предыдущем подразделе мы уже показали, как это сделать, так что здесь вы можете воспользоваться тем же контейнером. Образ контейнера можно загрузить в реестр — например, в Docker Hub.

Мы загрузили образ нашего gRPC-сервера в Docker Hub, используя тег `kasunindrasiri/grpc-productinfo-server`. Платформа Kubernetes не управляет контейнерами напрямую, применяя вместо этого абстракцию под названием *«pod-оболочка»*. Pod-оболочка — логический элемент, который может состоять из одного или нескольких контейнеров; это единица репликации в Kubernetes. Например, чтобы запустить несколько экземпляров gRPC-сервера, вам нужно создать дополнительные pod-оболочки. Контейнеры, размещенные внутри одной pod-оболочки, находятся в одной локальной сети и разделяют общие ресурсы. Но в данном примере нам достаточно одного контейнера. Kubernetes не управляет pod-оболочками напрямую; вместо этого используется еще одна абстракция, *ресурс развертывания*. В нем определяется количество pod-оболочек, которые должны работать одновременно. При создании нового развертывания Kubernetes запускает столько pod-оболочек, сколько указано в его ресурсе.

Чтобы развернуть наше gRPC-приложение в Kubernetes, нужно создать ресурс развертывания в формате YAML, как показано в листинге 7.4.

Листинг 7.4. Ресурс, описывающий развертывание gRPC-сервера в Kubernetes

```
apiVersion: apps/v1
kind: Deployment ❶
metadata:
  name: grpc-productinfo-server ❷
spec:
  replicas: 1 ❸
  selector:
    matchLabels:
      app: grpc-productinfo-server
  template:
    metadata:
```

```
    labels:
      app: grpc-productinfo-server
  spec:
    containers:
    - name: grpc-productinfo-server ❹
      image: kasunindrasiri/grpc-productinfo-server ❺
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
      ports:
      - containerPort: 50051
        name: grpc
```

❶ Объявляем объект `Deployment`.

❷ Название развертывания.

❸ Количество pod-оболочек с gRPC-сервером, которые должны выполняться одновременно.

❹ Название соответствующего контейнера с gRPC-сервером.

❺ Имя и тег образа контейнера с gRPC-сервером.

В случае применения этого ресурса с помощью команды `kubectl apply -f server/grpc-prodinfo-server.yaml` Kubernetes развернет в кластере pod-оболочку с одним gRPC-сервером. Если клиентское gRPC-приложение работает в том же кластере, то для отправки удаленного вызова оно должно знать IP-адрес и порт данной pod-оболочки. Но при перезапуске pod-оболочки ее IP-адрес может измениться, и если вы используете несколько реплик, то вам придется иметь дело с разными IP-адресами. Для обхода этого ограничения Kubernetes предоставляет абстракцию под названием *«сервис»*.

Ресурс Kubernetes-сервиса для gRPC-сервера

Вы можете создать Kubernetes-сервис и привязать его к подходящим pod-оболочкам (в данном случае к pod-оболочкам с gRPC-сервером); для автоматического перенаправления трафика используется доменное имя. Поэтому сервис можно считать прокси-сервером или балансировщиком нагрузки, направляющим запросы к pod-оболочкам, которые к нему привязаны. В листинге 7.5 показан ресурс Kubernetes-сервиса для серверного gRPC-приложения.

Листинг 7.5. Ресурс, описывающий Kubernetes-сервис для gRPC-сервера

```
apiVersion: v1
kind: Service ❶
metadata:
  name: productinfo ❷
spec:
  selector:
    app: grpc-productinfo-server ❸
  ports:
  - port: 50051 ❹
    targetPort: 50051
    name: grpc
type: NodePort
```

❶ Указываем `Service` в качестве типа ресурса.

❷ Имя, с помощью которого клиентское приложение будет подключаться к сервису.

❸ Это говорит сервису о том, что запросы должны направляться к pod-оболочкам с меткой `grpcproductinfo-server`.

❹ Сервис работает на порте 50 051 и перенаправляет запросы на порт 50 051.

Создав ресурсы `Deployment` и `Service`, вы можете развернуть это приложение в Kubernetes с помощью команды `kubectl apply -f server/grpc-prodinfo-server.yaml` (оба ресурса могут находиться в одном и том же YAML-файле). В результате успешного развертывания этих ресурсов вы должны получить pod-оболочку с запущенным gRPC-сервером, привязанный к ней Kubernetes-сервис и объект `Deployment`.

Теперь развернем в кластере Kubernetes gRPC-клиент.

Kubernetes-задание для выполнения gRPC-клиента

Серверное и клиентское gRPC-приложения можно запустить в одном кластере Kubernetes. Для доступа к серверу клиент может использовать gRPC-сервис `productinfo`, который мы создали выше. Поэтому для удаленных вызовов в клиентском коде следует использовать сетевое имя и порт сервиса. Например, в нашей реализации клиента на Go это будет выглядеть как `grpc.Dial("productinfo:50051", grpc.WithInsecure())`. Если наш клиент просто вызывает gRPC-сервис, записывает ответ и завершает работу и его нужно запустить определенное количество раз, то вместо ресурса `Deployment`

лучше использовать `Job` (задание). Задания в Kubernetes предназначены для многократного запуска pod-оболочек.

Контейнер клиентского приложения можно создать так же, как мы делали это в случае с gRPC-сервером. Загрузив его в реестр Docker, вы можете описать ресурс `Job` следующим образом (листинг 7.6).

Листинг 7.6. Клиентское gRPC-приложение, которое запускается в виде Kubernetes-задания

```
apiVersion: batch/v1
kind: Job ❶
metadata:
  name: grpc-productinfo-client ❷
spec:
  completions: 1 ❸
  parallelism: 1 ❹
  template:
    spec:
      containers:
      - name: grpc-productinfo-client ❺
        image: kasunindrasiri/grpc-productinfo-client ❻
      restartPolicy: Never
  backoffLimit: 4
```

❶ Указываем `Job` в качестве типа ресурса.

❷ Имя задания.

❸ Количество раз, которые pod-оболочка должна отработать успешно, прежде чем задание можно считать завершенным.

❹ Сколько pod-оболочек должно работать одновременно.

❺ Название соответствующего контейнера с gRPC-клиентом.

❻ Образ контейнера, с которым связано это задание.

Теперь вы можете развернуть задание с клиентским gRPC-приложением, используя команду `kubectl apply -f client/grpc-prodinfo-client-job.yaml`, и проверить состояние pod-оболочки.

В результате успешного выполнения этого задания gRPC-сервису `ProductInfo` будет отправлен удаленный вызов для добавления нового товара. Чтобы

убедиться в получении нужной информации, можно проверить журнальные записи клиентской и серверной pod-оболочек.

Теперь можно обсудить предоставление доступа к gRPC-сервисам из-за пределов кластера Kubernetes с помощью ресурсов Ingress.

Открытие доступа к gRPC-сервису снаружи с помощью ресурсов Ingress

Итак, мы развернули gRPC-сервер в Kubernetes и сделали его доступным для другой pod-оболочки, которая выполняет задание в том же кластере. Но что, если к сервису нужно обращаться из-за пределов кластера? Как вы уже знаете, с помощью ресурса `Service` трафик перенаправляется от одной pod-оболочки к другой, вследствие чего внешние приложения не могут получить доступ к данному ресурсу. Для этого в Kubernetes предусмотрена другая абстракция — *Ingress*.

Ресурс `Ingress` можно считать балансировщиком нагрузки, который находится между Kubernetes-сервисом и внешними приложениями. Он направляет внешний трафик к сервису, а тот распределяет уже внутренний трафик между подходящими pod-оболочками. Ресурс `Ingress` в кластере Kubernetes управляется одноименным контроллером. Тип и поведение последнего могут зависеть от того, какой кластер вы используете. Один из важнейших аспектов при открытии доступа к gRPC-сервису для внешних приложений — поддержка маршрутизации протокола gRPC на уровне Ingress. Следовательно, мы должны выбрать контроллер Ingress с поддержкой gRPC.

В этом примере мы будем использовать контроллер Nginx (oreil.ly/0UC0a), основанный на одноименном балансировщике нагрузки (вы можете выбрать другой контроллер Ingress с поддержкой gRPC, который лучше подходит для вашего кластера). Nginx Ingress (oreil.ly/wZo5w) умеет направлять внешний трафик к внутренним сервисам по протоколу gRPC.

Итак, чтобы сделать наше серверное gRPC-приложение `ProductInfo` доступным внешнему миру (то есть за пределами кластера Kubernetes), мы можем создать ресурс `Ingress`, как показано в листинге 7.7.

Листинг 7.7. Ресурс Ingress для gRPC-сервера на Go

```
apiVersion: extensions/v1beta1
kind: Ingress ❶
metadata:
```

```
  annotations: ❷
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/backend-protocol: "GRPC"
  name: grpc-prodinfo-ingress ❸
spec:
  rules:
  - host: productinfo ❹
    http:
      paths:
      - backend:
        serviceName: productinfo ❺
        servicePort: grpc ❻
```

❶ Указываем `Ingress` как тип ресурса.

❷ Аннотация, относящаяся к контроллеру Nginx Ingress; указывает gRPC в качестве внутреннего протокола.

❸ Название ресурса `Ingress`.

❹ Это сетевое имя, доступное снаружи.

❺ Имя соответствующего Kubernetes-сервиса.

❻ Название порта, указанного для Kubernetes-сервиса.

Прежде чем развертывать представленный здесь ресурс Ingress, необходимо установить контроллер Nginx Ingress. Информацию о его установке и использовании вместе с gRPC можно найти в репозитории Ingress-Nginx (oreil.ly/l-vFp) проекта Kubernetes. После развертывания этого ресурса к вашему gRPC-серверу сможет обратиться любое внешнее приложение, используя сетевое имя (`productinfo`) и порт по умолчанию (80).

Итак, вы изучили основы развертывания в Kubernetes gRPC-приложений, готовых к промышленному использованию. Теперь вы знаете, что благодаря возможностям Kubernetes и Docker не нужно беспокоиться о большинстве инфраструктурных требований, таких как масштабируемость, высокая доступность, балансировка нагрузки, обработка отказов и т. д. Все это уже поддерживается в Kubernetes. Таким образом, если вы запускаете свои gRPC-приложения с помощью этой платформы, то вам не нужно реализовывать в своем коде некоторые концепции, представленные в главе 6 (включая балансировку нагрузки, сопоставление имен и т. д.).

Запустив свое gRPC-приложение, вы должны убедиться в том, что оно корректно работает в промышленных условиях. Для этого следует постоянно отслеживать его выполнение и при необходимости принимать соответствующие меры. Подробно рассмотрим понятие *наблюдаемости* в контексте gRPC-приложений.

Наблюдаемость

В предыдущем разделе было сказано, что gRPC-приложения обычно развертываются и выполняются в виде нескольких контейнеров, которые взаимодействуют по сети. В связи с этим возникает проблема: каким образом отслеживать все контейнеры и быть уверенными в том, что они работают на самом деле? Для этого была создана такая концепция, как *наблюдаемость*.

Согласно «Википедии», «наблюдаемость — свойство системы, показывающее, можно ли по выходу полностью восстановить информацию о состояниях системы». Иными словами, наблюдаемость позволяет узнать о каких-либо проблемах в работе системы. При обнаружении проблем мы должны иметь возможность ответить на ряд вопросов наподобие «Что пошло не так?» и «Что происходит?». Если мы можем ответить на все эти вопросы в любой момент времени и в контексте любого участка системы, то такую систему можно считать *наблюдаемой*.

Кроме того, необходимо отметить, что наблюдаемость — такое же важное свойство системы, как эффективность, удобство использования и надежность. Поэтому о нем следует позаботиться на самых ранних этапах разработки gRPC-приложения.

Наблюдаемость опирается на три основные методики: метрики, ведение журнала и трассировка. Обсудим каждую из них по отдельности.

Метрики

Метрики — числовое представление данных, которые собираются на определенных отрезках времени. Эти данные бывают двух видов: общесистемные (использование процессора, памяти и т. д.) и прикладные (частота входящих запросов, частота возникновения ошибок и т. д.).

Общесистемные метрики обычно собираются во время работы приложений. В наши дни для их сбора существует множество инструментов, которые, как правило, используются командой DevOps. Но прикладные метрики зависят от конкретного приложения, поэтому выбор того, какие из них необходимо собирать, ложится на разработчиков. В данном подразделе основное внимание будет уделяться тому, как реализовать поддержку прикладных метрик в наших приложениях.

OpenCensus в сочетании с gRPC

Стандартные метрики в gRPC-приложениях могут предоставляться библиотекой OpenCensus (oreil.ly/EMfF-). Чтобы их включить, достаточно добавить обработчики в клиентский и серверный код. Мы также можем реализовать собственный сборщик метрик (листинг 7.8).

OpenCensus (opencensus.io) — набор открытых библиотек для сбора прикладных метрик и распределенной трассировки с поддержкой разных языков. Собранные метрики приложения передаются внутреннему сервису (который вы сами выбираете) в режиме реального времени. На сегодняшний день в число поддерживаемых сервисов входят Azure Monitor, Datadog, Instana, Jaeger, SignalFX, Stackdriver и Zipkin. Мы также можем написать собственное средство экспорта для других сервисов.

Листинг 7.8. Включение мониторинга OpenCensus для gRPC-сервера на Go

```
package main

import (
  "errors"
  "log"
  "net"
  "net/http"

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "go.opencensus.io/plugin/ocgrpc" ❶
  "go.opencensus.io/stats/view"
  "go.opencensus.io/zpages"
  "go.opencensus.io/examples/exporter"
)
```

```
const (
  port = ":50051"
)

// сервер, который реализует ecommerce/product_info
type server struct {
  productMap map[string]*pb.Product
}

func main() {

  go func() { ❼
    mux := http.NewServeMux()
    zpages.Handle(mux, "/debug")
    log.Fatal(http.ListenAndServe("127.0.0.1:8081", mux))
  }()

   view.RegisterExporter(&exporter.PrintExporter{}) ❷

  if err := view.Register(ocgrpc.DefaultServerViews...); err != nil { ❸
     log.Fatal(err)
  }

  grpcServer := grpc.NewServer(grpc.StatsHandler(&ocgrpc.ServerHandler{})) ❹
  pb.RegisterProductInfoServer(grpcServer, &server{}) ❺

  lis, err := net.Listen("tcp", port)
  if err != nil {
     log.Fatalf("Failed to listen: %v", err)
  }

  if err := grpcServer.Serve(lis); err != nil { ❻
     log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Указываем внешние библиотеки, необходимые для включения мониторинга. Пакет gRPC OpenCensus предоставляет готовые обработчики для поддержки OpenCensus, и здесь мы будем использовать их.

❷ Регистрируем средства экспорта собираемых данных. Добавленный нами объект `PrintExporter` экспортирует данные в консоль. Это сделано в целях демонстрации; обычно мы не рекомендуем вести журнал для всех промышленных заданий.

❸ Регистрируем представления для сбора количества запросов к серверу. Нам доступно несколько готовых представлений, которые собирают байты,

полученные/отправленные в ходе каждого удаленного вызова, латентность этих вызовов и информацию о том, сколько из них завершилось. Мы также можем создавать собственные представления для сбора данных.

❹ Создаем gRPC-сервер с обработчиком статистики.

❺ Регистрируем на gRPC-сервере сервис `ProductInfo`.

❻ Начинаем прослушивать входящие сообщения на порте 50 051.

❼ Запускаем сервер z-Pages. Для визуализации метрик служит HTTP-путь на порте 8081, начинающийся с `/debug`.

Мониторинг OpenCensus можно включить не только для gRPC-сервера, но и для клиентов (используя клиентские обработчики). В листинге 7.9 показан фрагмент кода, который добавляет обработчик метрик в gRPC-клиент, написанный на Go.

Листинг 7.9. Включение мониторинга OpenCensus для gRPC-клиента

```
package main

import (
  "context"
  "log"
  "time"

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "go.opencensus.io/plugin/ocgrpc" ❶
  "go.opencensus.io/stats/view"
  "go.opencensus.io/examples/exporter"
)

const (
  address = "localhost:50051"
)

func main() {
  view.RegisterExporter(&exporter.PrintExporter{}) ❷

   if err := view.Register(ocgrpc.DefaultClientViews...); err != nil { ❸
       log.Fatal(err)
   }
```

```
  conn, err := grpc.Dial(address, ❹
        grpc.WithStatsHandler(&ocgrpc.ClientHandler{}),
          grpc.WithInsecure(),
          )
  if err != nil {
    log.Fatalf("Can't connect: %v", err)
  }
  defer conn.Close() ❻

  c := pb.NewProductInfoClient(conn) ❺

  ... // пропускаем вызов удаленного метода
}
```

❶ Указываем внешние библиотеки, необходимые для включения мониторинга.

❷ Регистрируем средства экспорта собранных данных и трассировок. Здесь мы добавляем объект `PrintExporter`, который записывает экспортированные данные в консоль. Это сделано в целях демонстрации; обычно мы не рекомендуем вести журнал для всех промышленных заданий.

❸ Регистрируем представления для сбора количества запросов к серверу. Нам доступно несколько готовых представлений, которые собирают байты, полученные/отправленные в ходе каждого удаленного вызова, латентность этих вызовов и информацию о том, сколько из них завершилось. Мы также можем создавать собственные представления для сбора данных.

❹ Устанавливаем соединение с сервером, используя клиентские обработчики статистики.

❺ Создаем на основе установленного соединения клиентскую заглушку.

❻ Закончив работу, закрываем соединение.

После запуска сервера и клиента мы можем просматривать их метрики с помощью созданного нами HTTP-пути (например, метрики удаленных вызовов и трассировки доступны по адресам localhost:8081/debug/rpcz и localhost:8081/debug/tracez соответственно).

Как уже упоминалось ранее, нам доступны готовые средства экспорта для публикации данных в поддерживаемые внутренние сервисы, но мы можем написать собственные, чтобы передавать метрики и результаты трассировки любым системам, способным их потреблять.

Далее мы обсудим еще одну популярную технологию, Prometheus (prometheus.io), которая часто используется для сбора метрик в gRPC-приложениях.

Prometheus в сочетании с gRPC

Prometheus — набор открытых инструментов для мониторинга и создания уведомлений. Вы можете использовать его в целях сбора метрик в своих gRPC-приложениях с помощью библиотеки gRPC Prometheus (oreil.ly/nm84_). Для ее подключения достаточно добавить в клиентский и серверный код по одному перехватчику, но можно также создать собственный сборщик метрик.

Для сбора метрик система Prometheus обращается по HTTP-пути, который начинается с /metrics. Она хранит все собранные данные и применяет к ним специальные правила, предназначенные либо для агрегации и записи на основе метрик новых временных рядов, либо для генерации уведомлений. Агрегированные результаты можно визуализировать с помощью таких инструментов, как Grafana (grafana.com).

В листинге 7.10 показано, как добавить перехватчик метрик и создать собственный сборщик в нашем сервере для управления товарами, написанном на Go.

Листинг 7.10. Включение мониторинга Prometheus для gRPC-сервера на Go

```
package main

import (
  ...
  "github.com/grpc-ecosystem/go-grpc-prometheus" ❶
  "github.com/prometheus/client_golang/prometheus"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  reg = prometheus.NewRegistry() ❷

  grpcMetrics = grpc_prometheus.NewServerMetrics() ❸

   customMetricCounter = prometheus.NewCounterVec(prometheus.CounterOpts{
       Name: "product_mgt_server_handle_count",
       Help: "Total number of RPCs handled on the server.",
```

```
   }, []string{"name"}) ❹
)

func init() {
   reg.MustRegister(grpcMetrics, customMetricCounter) ❺
}

func main() {
  lis, err := net.Listen("tcp", port)
  if err != nil {
     log.Fatalf("failed to listen: %v", err)
  }

  httpServer := &http.Server{
      Handler: promhttp.HandlerFor(reg, promhttp.HandlerOpts{}),
        Addr: fmt.Sprintf("0.0.0.0:%d", 9092)} ❻

  grpcServer := grpc.NewServer(
    grpc.UnaryInterceptor(grpcMetrics.UnaryServerInterceptor()), ❼
  )

  pb.RegisterProductInfoServer(grpcServer, &server{})
  grpcMetrics.InitializeMetrics(grpcServer) ❽

  // запускаем HTTP-сервер для Prometheus
  go func() {
     if err := httpServer.ListenAndServe(); err != nil {
        log.Fatal("Unable to start a http server.")
     }
  }()

  if err := grpcServer.Serve(lis); err != nil {
     log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Указываем внешние библиотеки, необходимые для включения мониторинга. В экосистеме gRPC уже есть ряд готовых перехватчиков для поддержки Prometheus, и здесь мы будем использовать их.

❷ Создаем реестр метрик. Он станет хранить все данные, которые сборщики регистрируют в системе. Если нужно добавить новый сборщик, то его следует указать в реестре.

❸ Создаем стандартные клиентские метрики, которые определены в библиотеке.

❹ Создаем собственный счетчик метрик с именем `product_mgt_server_handle_count`.

❺ Регистрируем стандартные серверные метрики и наш сборщик в реестре, созданном в пункте ❷.

❻ Создаем HTTP-сервер для Prometheus. HTTP-путь для сбора метрик начинается с `/metrics` и находится на порте 9092.

❼ Создаем gRPC-сервер с перехватчиком метрик. Поскольку наш сервис унарный, мы используем `grpcMetrics.UnaryServerInterceptor`. Для потоковых сервисов предусмотрен другой перехватчик, `grpcMetrics.StreamServerInterceptor()`.

❽ Инициализируем все стандартные метрики.

Наш счетчик, который мы создали в пункте 4, позволяет выполнять мониторинг дополнительных метрик. Представьте, что мы хотим подсчитывать, сколько товаров с одинаковыми названиями добавляется в нашу систему. Как показано в листинге 7.11, с помощью счетчика `customMetricCounter` в методе `addProduct` можно добавить новую метрику.

Листинг 7.11. Добавление новой метрики с помощью созданного нами счетчика

```
// AddProduct реализует ecommerce.AddProduct
func (s *server) AddProduct(ctx context.Context,
     in *pb.Product) (*wrapper.StringValue, error) {
     customMetricCounter.WithLabelValues(in.Name).Inc()
  ...
}
```

Мониторинг Prometheus можно включить не только на gRPC-сервере, но и на его клиентах, с помощью клиентских перехватчиков. В листинге 7.12 показан пример кода для добавления перехватчика метрик в gRPC-клиент, написанный на Go.

Листинг 7.12. Включение мониторинга Prometheus для gRPC-клиента на Go

```
package main

import (
  ...
  "github.com/grpc-ecosystem/go-grpc-prometheus" ❶
```

```
  "github.com/prometheus/client_golang/prometheus"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

const (
  address = "localhost:50051"
)

func main() {
  reg := prometheus.NewRegistry() ❷
  grpcMetrics := grpc_prometheus.NewClientMetrics() ❸
  reg.MustRegister(grpcMetrics) ❹

  conn, err := grpc.Dial(address,
        grpc.WithUnaryInterceptor(grpcMetrics.UnaryClientInterceptor()), ❺
          grpc.WithInsecure(),
          )

  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }
  defer conn.Close()

   // создаем HTTP-сервер для Prometheus
   httpServer := &http.Server{
        Handler: promhttp.HandlerFor(reg, promhttp.HandlerOpts{}),
          Addr: fmt.Sprintf("0.0.0.0:%d", 9094)} ❻

   // запускаем HTTP-сервер для Prometheus
   go func() {
       if err := httpServer.ListenAndServe(); err != nil {
           log.Fatal("Unable to start a http server.")
       }
   }()

  c := pb.NewProductInfoClient(conn)
  ...
}
```

❶ Указываем внешние библиотеки, необходимые для включения мониторинга.

❷ Создаем реестр метрик. Как и в случае с серверным кодом, реестр будет хранить все данные, которые сборщики регистрируют в системе. Если нужно добавить новый сборщик, то его следует указать в реестре.

❸ Создаем стандартные клиентские метрики, которые определены в библиотеке.

❹ Регистрируем стандартные серверные метрики и наш сборщик в реестре, созданном в пункте ❷.

❺ Устанавливаем соединение с сервером, в котором реализован перехватчик метрик. Это унарный клиент, вследствие чего мы используем `grpcMetrics.UnaryClientInterceptor`. Для потоковых клиентов предусмотрен другой перехватчик, `grpcMetrics.StreamClientInterceptor()`.

❻ Создаем HTTP-сервер для Prometheus. HTTP-путь для сбора метрик начинается с `/metrics` и находится на порте 9094.

После запуска сервера и клиента мы можем просматривать их метрики с помощью созданного нами HTTP-пути (например, серверные метрики доступны по адресу http://localhost:9092/metrics, а клиентские — по адресу http://localhost:9094/metrics).

Как уже упоминалось, система Prometheus умеет собирать метрики с представленных выше URL. Она хранит все собранные данные и применяет к ним специальные правила, предназначенные для агрегации и создания новых записей. Используя Prometheus в качестве источника данных, вы можете визуализировать метрики на информационной панели с помощью таких инструментов, как Grafana.

Grafana — открытый инструмент, позволяющий создавать информационные панели с метриками и диаграммами для Graphite, Elasticsearch и Prometheus. Он позволяет запрашивать, визуализировать и анализировать собранные метрики.

Одно из преимуществ мониторинга на основе метрик — затраты на обработку данных не зависят от нагрузки на систему. Например, увеличение объема трафика, поступающего в приложение, не повлияет на загруженность диска, сложность обработки, скорость визуализации, операционные затраты и т. д. Накладные расходы остаются неизменными. Кроме того, с собранными метриками можно производить различные математические и статистические расчеты, делая ценные выводы о состоянии системы.

Еще один ключевой элемент наблюдаемости — журнальные записи. О них мы поговорим в следующем подразделе.

Журнальные записи

Журнальные записи — неизменяемые фрагменты данных с временными метками, относящиеся к отдельным событиям, произошедшим в определенный момент. Разработчики приложений обычно сохраняют в журнал информацию о текущем внутреннем состоянии системы. Преимущество журнальных записей по сравнению с метриками в том, что их легче генерировать и они могут содержать больше дополнительных данных. Мы можем прикреплять к ним определенные действия или такие контекстные сведения, как уникальные ID, описание того, что мы собираемся сделать, трассировку стека и т. д. Недостатком журнальных записей является их высокая стоимость, поскольку их нужно хранить и индексировать так, чтобы можно было легко находить и использовать.

В gRPC-приложениях журналирование можно реализовать с помощью перехватчиков. Как уже обсуждалось в главе 5, мы можем создать по одному новому перехватчику на клиентской и серверной стороне и записывать в журнал запросы и ответы для каждого удаленного вызова.

Экосистема gRPC предлагает ряд готовых перехватчиков для реализации журналирования в приложениях на языке Go. Среди них можно выделить библиотеку grpc_ctxtags, добавляющую в контекст хеш-таблицу Tag с содержимым тела запроса; grpc_zap, проект по интеграции библиотеки журналирования zap (oreil.ly/XMlIg) в обработчики gRPC; проект grpc_logrus, который интегрирует в обработчики gRPC библиотеку журналирования logrus (oreil.ly/oKJX5). Подробности об этих перехватчиках ищите в репозитории go-grpc-middleware (oreil.ly/8lNaH).

После добавления поддержки журналирования в gRPC-приложение журнальные записи будут выводиться либо в консоль, либо в журнальный файл в зависимости от вашей конфигурации. Каждый фреймворк для ведения журнала настраивается по-разному.

Итак, мы обсудили два ключевых элемента наблюдаемости: метрики и журнальные записи. Этого уже достаточно для анализа производительности и поведения отдельных систем. Но если вы хотите иметь

представление о жизненном цикле запросов, которые проходят через разные компоненты, то вам нужно нечто большее. Распределенная трассировка — методика, позволяющая отслеживать прохождение запросов по различным системам.

Трассировка

Трассировка — представление ряда связанных между собой событий, которые возникают при прохождении запроса по распределенной системе. Как уже обсуждалось в разделе «Взаимодействие микросервисов на основе gRPC» на с. 80, реальные проекты состоят из множества микросервисов, каждый из которых выполняет свою бизнес-задачу. Следовательно, прежде, чем клиент получит ответ, его запрос должен пройти через цепочку сервисов и разных систем. Все эти промежуточные события — часть его жизненного цикла. С помощью трассировки мы можем отследить как путь, пройденный запросом, так и его структуру.

Базовые элементы распределенной трассировки — *интервалы* (spans), которые формируют дерево. Интервал содержит метаданные и прочие атрибуты задания, такие как время, потраченное на его выполнение (латентность). У трассировки есть свой идентификатор, TraceID, представляющий собой уникальную последовательность байтов; он позволяет группировать и отличать друг от друга разные интервалы. Попробуем включить трассировку в нашем gRPC-приложении.

Помимо метрик, библиотека OpenCensus предоставляет поддержку трассировки для gRPC-приложений. С ее помощью мы реализуем трассировку в нашем сервисе для управления товарами. Как уже отмечалось ранее, в целях передачи результатов трассировки любым внутренним системам можно подключить любые поддерживаемые средства экспорта. В данном примере мы будем использовать Jaeger.

В библиотеке gRPC для языка Go трассировка включена по умолчанию. Поэтому, чтобы начать сбор результатов, достаточно лишь зарегистрировать средство экспорта. Инициируем Jaeger в клиентском и серверном приложениях. Как это делается, показано в листинге 7.13.

Листинг 7.13. Инициализация средства экспорта OpenCensus Jaeger

```
package tracer

import (
  "log"

  "go.opencensus.io/trace" ❶
  "contrib.go.opencensus.io/exporter/jaeger"

)

func initTracing() {

   trace.ApplyConfig(trace.Config{DefaultSampler: trace.AlwaysSample()})
   agentEndpointURI := "localhost:6831"
   collectorEndpointURI := "http://localhost:14268/api/traces" ❷
    exporter, err := jaeger.NewExporter(jaeger.Options{
            CollectorEndpoint: collectorEndpointURI,
            AgentEndpoint: agentEndpointURI,
            ServiceName:    "product_info",
    })
    if err != nil {
       log.Fatal(err)
    }
    trace.RegisterExporter(exporter) ❸

}
```

❶ Импортируем библиотеки OpenCensus и Jaeger.

❷ Создаем средство экспорта Jaeger, указывая путь для сбора данных, имя сервиса и конечную точку агента.

❸ Регистрируем средство экспорта в трассировщике OpenCensus.

Зарегистрировав средство экспорта, мы можем выполнить трассировку метода сервиса на стороне сервера. Это показано в листинге 7.14.

Листинг 7.14. Трассировка метода gRPC-сервиса

```
// GetProduct реализует ecommerce.GetProduct
func (s *server) GetProduct(ctx context.Context, in *wrapper.StringValue) (
         *pb.Product, error) {
  ctx, span := trace.StartSpan(ctx, "ecommerce.GetProduct") ❶
  defer span.End() ❷
  value, exists := s.productMap[in.Value]
```

```
  if exists {
     return value, status.New(codes.OK, "").Err()
  }
  return nil, status.Errorf(codes.NotFound, "Product does not exist.", in.Value)
}
```

❶ Начинаем новый интервал, указывая его имя и контекст.

❷ Закончив работу, завершаем интервал.

Как показано в листинге 7.15, аналогичным образом трассировку можно реализовать и на стороне клиента.

Листинг 7.15. Реализация трассировки в gRPC-клиенте

```
package main

import (
  "context"
  "log"
  "time"

  pb "productinfo/client/ecommerce"
  "productinfo/client/tracer"
  "google.golang.org/grpc"
  "go.opencensus.io/plugin/ocgrpc" ❶
  "go.opencensus.io/trace"
  "contrib.go.opencensus.io/exporter/jaeger"
)

const (
  address = "localhost:50051"
)

func main() {
  tracer.initTracing() ❷

  conn, err := grpc.Dial(address, grpc.WithInsecure())
  if err != nil {
     log.Fatalf("did not connect: %v", err)
  }

  defer conn.Close()
  c := pb.NewProductInfoClient(conn)

  ctx, span := trace.StartSpan(context.Background(),
          "ecommerce.ProductInfoClient") ❸
```

```
  name := "Apple iphone 11"
  description := "Apple iphone 11 is the latest smartphone,
            launched in September 2019"
  price := float32(700.0)
  r, err := c.AddProduct(ctx, &pb.Product{Name: name,
      Description: description, Price: price}) ❺
  if err != nil {
     log.Fatalf("Could not add product: %v", err)
  }
  log.Printf("Product ID: %s added successfully", r.Value)

  product, err := c.GetProduct(ctx, &pb.ProductID{Value: r.Value}) ❻
  if err != nil {
    log.Fatalf("Could not get product: %v", err)
  }
  log.Printf("Product: ", product.String())
  span.End() ❹
}
```

❶ Импортируем библиотеки OpenCensus и Jaeger.

❷ Вызываем функцию `initTracing`, инициализируем средство экспорта Jaeger и регистрируем его в трассировщике.

❸ Начинаем новый интервал, указывая его имя и контекст.

❹ Закончив работу, завершаем интервал.

❺ Вызываем удаленный метод `addProduct`, передавая ему описание нового товара.

❻ Вызываем удаленный метод `getProduct`, передавая ему `productID`.

После запуска сервера и клиента интервалы трассировки будут переданы агенту Jaeger; последний представляет собой фоновый процесс, который играет роль буфера и скрывает пакетную обработку и маршрутизацию от клиента. Получив клиентские результаты трассировки, агент Jaeger направляет их сборщику. Тот их обрабатывает и сохраняет. Визуализировать трассировку можно с помощью сервера Jaeger.

На этом мы завершаем обсуждение наблюдаемости. Журнальные записи, метрики и трассировка служат каждые своей цели, и, чтобы получить как можно более подробную картину внутреннего состояния системы, лучше использовать все три ключевых инструмента.

Если для вашего gRPC-приложения, запущенного в промышленной среде, можно настроить наблюдаемость, то вы можете отслеживать его состояние и с легкостью обнаруживать любые проблемы и перебои в работе. Процесс разработки, тестирования и развертывания исправлений должен быть максимально быстрым. Для этого следует предусмотреть хорошие механизмы отладки и устранения неполадок. Подробно поговорим об этих механизмах в контексте gRPC-приложений.

Отладка и устранение неполадок

Отладка и устранение неполадок — процесс поиска и устранения изначальной причины возникновения проблемы. Для этого необходимо сначала воспроизвести данную проблему в среде более низкого уровня (отладочной или тестовой). Как следствие, нужен набор инструментов для генерации запросов, аналогичных тем, которые наблюдаются в промышленных условиях.

В gRPC этот процесс проходит сложнее, чем в HTTP, поскольку наши инструменты должны поддерживать кодирование и декодирование сообщений с учетом определения сервиса и уметь работать с HTTP/2. Распространенные средства, такие как curl или Postman, которые используются для тестирования HTTP-сервисов, не годятся для gRPC.

Но отладка и тестирование gRPC-сервисов возможны с помощью множества других интересных инструментов. Их список можно найти в репозитории awesome-grpc (oreil.ly/Ki2aZ), который содержит отличный набор ресурсов для gRPC. Один из наиболее распространенных способов отладки gRPC-приложений — использование дополнительных журнальных записей.

Включение дополнительных журнальных записей. Мы можем включить дополнительные журнальные записи и трассировку в целях диагностики наших gRPC-приложений. В коде на языке Go для этого можно установить следующие переменные окружения:

```
GRPC_GO_LOG_VERBOSITY_LEVEL=99 ❶
GRPC_GO_LOG_SEVERITY_LEVEL=info ❷
```

❶ *Уровень детализации* (verbosity) определяет количество информационных сообщений, которые выводятся каждые пять минут. По умолчанию это значение равно 0.

❷ Это *степень серьезности* `info`, согласно которому должны выводиться все информационные сообщения.

В gRPC-приложениях, написанных на Java, уровнем журналирования нельзя управлять с помощью переменных окружения. Вместо этого для включения дополнительных журнальных записей можно предоставить файл `logging.properties`. Представим, к примеру, что нам нужно проанализировать фреймы транспортного протокола. Мы можем создать в нашем проекте файл `logging.properties` и указать пониженный уровень журналирования с помощью Java-пакета (а именно, транспортного пакета netty):

```
handlers=java.util.logging.ConsoleHandler
io.grpc.netty.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

После этого мы можем запустить скомпилированное приложение со следующим флагом JVM:

```
-Djava.util.logging.config.file=logging.properties
```

В результате все журнальные записи, чей уровень равен тому, который мы сконфигурировали, или выше его, будут выводиться в консоль или журнальный файл. Просматривая их, мы сможем получить ценную информацию о состоянии системы.

На этом мы закончили рассмотрение большей части того, что необходимо знать о работе gRPC-приложений в промышленных условиях.

Резюме

При подготовке gRPC-приложения к промышленному использованию мы сначала проектируем контракт сервиса и генерируем серверный и клиентский код, после чего реализуем бизнес-логику. Разработав gRPC-приложение, мы должны убедиться в том, что оно готово для работы в промышленных условиях. Неотъемлемая часть этого процесса — *тестирование* серверной и клиентской сторон.

В *развертывании* gRPC-приложений применяются стандартные методологии. При развертывании в локальной системе или ВМ мы просто используем сгенерированные двоичные файлы для сервера и клиента. gRPC-приложения

можно запускать в виде контейнеров Docker; вы можете найти примеры стандартных файлов Dockerfile для кода на языках Go и Java. Способ развертывания gRPC-сервисов в Kubernetes похож на стандартный. gRPC-приложение задействует внутренние возможности Kubernetes, такие как балансировка нагрузки, высокая доступность, контроллеры Ingress и т. д. Крайне важно в промышленных условиях обеспечить наблюдаемость; в реальных gRPC-приложениях часто используются метрики прикладного уровня.

В одной из наиболее популярных реализаций сбора метрик для gRPC, библиотеке gRPC-Prometheus, применяются перехватчики на стороне сервера и клиента. Аналогичным образом в gRPC реализуется и поддержка журналирования. Для отладки и устранения неполадок в промышленных условиях иногда необходимо включать дополнительное журналирование. В следующей главе мы исследуем некоторые компоненты экосистемы gRPC, которые могут пригодиться в разработке gRPC-приложений.

ГЛАВА 8

Экосистема gRPC

В этой главе мы исследуем технологии, которые не входят в основную реализацию gRPC, но могут оказаться довольно полезными в реальных сценариях разработки и выполнения gRPC-приложений. Они являются частью родительного проекта gRPC Ecosystem, и ни одна из этих технологий не требуется для работы кода, в котором используется gRPC. Если какая-то из них предлагает возможности, в той или иной степени соответствующие вашим требованиям, то вам стоит познакомиться с ней поближе.

Начнем наше обсуждение с gRPC-шлюза.

gRPC-шлюз

gRPC-шлюз — дополнение для компилятора Protocol Buffers, которое позволяет ему считывать определения gRPC-сервисов и генерировать обратные прокси-серверы для перевода API на основе REST и JSON в gRPC. Данный инструмент написан специально для Go; благодаря ему клиентские приложения могут обращаться к gRPC-сервису как по gRPC, так и по HTTP. Это показано на рис. 8.1.

В данном примере у нас есть контракт сервиса `ProductInfo`, с помощью которого мы создаем gRPC-сервис под названием `ProductInfoService`. Ранее мы уже показывали, как создать клиент для `ProductInfo`, но здесь вместо этого используется обратный прокси-сервер, который предоставляет доступ к каждому удаленному методу gRPC-сервиса в виде API REST. Таким образом, REST-клиенты могут отправлять свои запросы по HTTP. Получив HTTP-запрос, обратный прокси-сервер превращает его в gRPC-сообщение и вызывает удаленный метод внутреннего сервиса. Сообщение, возвращаемое внутренним сервисом, переводится обратно в формат HTTP-ответа и передается клиенту.

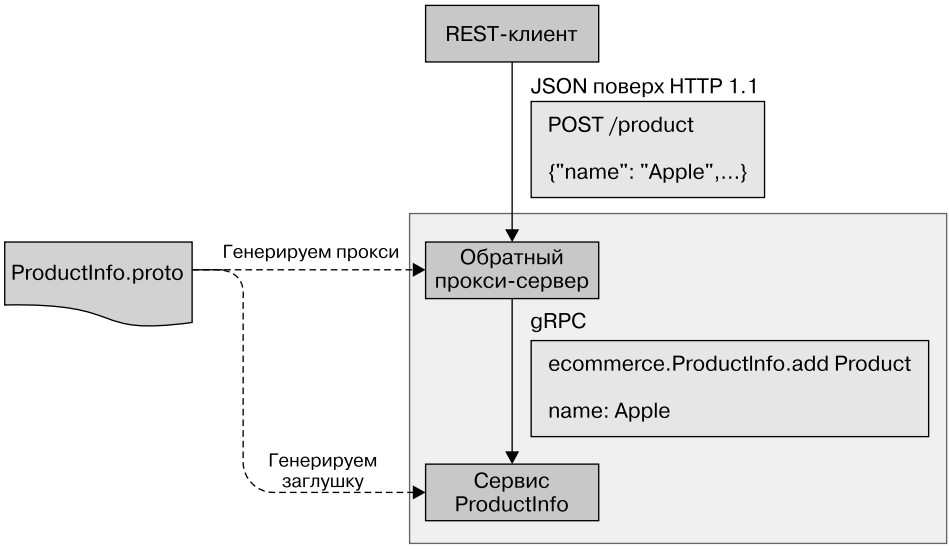


Рис. 8.1. gRPC-шлюз

Чтобы сгенерировать обратный прокси-сервер из определения сервиса, нужно сначала привязать gRPC-методы к HTTP-ресурсам. Для этого определение сервиса `ProductInfo`, которое мы создали, нужно обновить. Добавим в него связующий код, как показано в листинге 8.1.

Листинг 8.1. Обновленное определение сервиса ProductInfo в формате Protocol Buffers

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";
import "google/api/annotations.proto"; ❶

package ecommerce;

service ProductInfo {
   rpc addProduct(Product) returns (google.protobuf.StringValue) {
       option (google.api.http) = { ❷
           post: "/v1/product"
           body: "*"
       };
   }
   rpc getProduct(google.protobuf.StringValue) returns (Product) {
        option (google.api.http) = { ❸
```

```
            get:"/v1/product/{value}"
        };
  }
}

message Product {
   string id = 1;
   string name = 2;
   string description = 3;
   float price = 4;
}
```

❶ Импортируем proto-файл `google/api/annotations.proto`, чтобы добавить в Protocol Buffers поддержку аннотаций.

❷ Добавляем в метод `addProduct` привязку gRPC/HTTP. Указываем шаблон URL-пути (`v1/product`), HTTP-метод (POST) и описание тела сообщения. Символ * говорит о том, что любое поле, не соответствующее шаблону пути, должно быть привязано к телу запроса.

❸ Добавляем привязку gRPC/HTTP в метод `getProduct`. Здесь мы указываем HTTP-метод GET с шаблоном URL-пути `/v1/product/{value}` и `ProductID` в качестве передаваемого параметра.

Есть еще несколько важных правил относительно привязки gRPC-сервисов к HTTP-ресурсам. Некоторые из них перечислены ниже. Более подробная информация о связывании HTTP с gRPC приведена в документации Google API (oreil.ly/iYyZC).

- Каждая привязка должна определять шаблон URL-пути и HTTP-метод.
- Шаблон пути в gRPC-запросе может занимать одно или несколько полей с неповторяемым примитивным типом.
- Любое поле запроса, которое не указано в шаблоне пути, автоматически становится параметром HTTP-запроса (при условии, что у HTTP-запроса нет тела).
- Поля, привязанные к URL-параметрам, могут быть сообщениями или иметь примитивный тип. При этом в первом случае они должны быть неповторяемыми.
- Повторяемые поля в URL становятся параметрами запроса такого вида: `...?param=A&param=B`.

- Каждое поле сообщения привязывается к отдельному параметру — например, `...?foo.a=A&foo.b=B&foo.c=C`.

Определение сервиса нужно скомпилировать с помощью компилятора Protocol Buffers и сгенерировать исходный код обратного прокси-сервера. Обсудим генерацию кода и реализацию сервера в контексте языка Go.

Перед компиляцией необходимо установить несколько зависимостей. Загрузите нужные пакеты с помощью таких команд:

```
go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-swagger
go get -u github.com/golang/protobuf/protoc-gen-go
```

После загрузки пакетов выполните следующую команду, чтобы скомпилировать определение сервиса (`product_info.proto`) и сгенерировать заглушку:

```
protoc -I/usr/local/include -I. \
-I$GOPATH/src \
-I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
--go_out=plugins=grpc:. \
product_info.proto
```

Данная команда сгенерирует в том же каталоге заглушку `product_info.pb.go`. Кроме того, нужно создать обратный прокси-сервер для поддержки REST-клиентов. Чтобы это сделать, можно воспользоваться дополнением grpc-gateway, которое поддерживает компилятор Go.

Дополнение grpc-gateway доступно только для Go. Это значит, что мы не можем скомпилировать и сгенерировать обратный прокси-сервер для gRPC-шлюза на других языках.

Сгенерируем обратный прокси-сервер из определения сервиса, используя следующую команду:

```
protoc -I/usr/local/include -I. \
-I$GOPATH/src \
-I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
--grpc-gateway_out=logtostderr=true:. \
product_info.proto
```

В результате в том же каталоге будет создан обратный прокси-сервер `product_info.pb.gw.go`.

Теперь создадим для HTTP-сервера прослушивающую конечную точку и запустим обратный прокси-сервер, который мы создали только что. В листинге 8.2 показано, как создать новый экземпляр сервера и зарегистрировать в нем сервис для обработки входящих HTTP-запросов.

Листинг 8.2. Обратный прокси-сервер с поддержкой HTTP на языке Go

```
package main

import (
  "context"
  "log"
  "net/http"

  "github.com/grpc-ecosystem/grpc-gateway/runtime"
  "google.golang.org/grpc"

  gw "github.com/grpc-up-and-running/samples/ch08/grpc-gateway/go/gw" ❶
)

var (
  grpcServerEndpoint = "localhost:50051" ❷
)

func main() {
  ctx := context.Background()
  ctx, cancel := context.WithCancel(ctx)
  defer cancel()

  mux := runtime.NewServeMux()
  opts := []grpc.DialOption{grpc.WithInsecure()}
  err := gw.RegisterProductInfoHandlerFromEndpoint(ctx, mux,
      grpcServerEndpoint, opts) ❸
  if err != nil {
     log.Fatalf("Fail to register gRPC gateway service endpoint: %v", err)
  }

  if err := http.ListenAndServe(":8081", mux); err != nil { ❹
     log.Fatalf("Could not setup HTTP endpoint: %v", err)
  }
}
```

❶ Импортируем пакет со сгенерированным кодом обратного прокси-сервера.

❷ Указываем URL gRPC-сервера. Убедитесь в том, что ваш сервер работает корректно и доступен по этому пути. Здесь мы используем gRPC-сервис, который создали в главе 2.

❸ Регистрируем прокси-обработчик для конечной точки gRPC-сервера. Во время выполнения мультиплексор сопоставляет HTTP-запрос с шаблоном и вызывает подходящий обработчик.

❹ Начинаем принимать запросы на порте 8081.

Теперь мы можем запустить оба сервера, gRPC и HTTP, и проверить, как они работают. В данном случае gRPC-сервер прослушивает порт 50 051, а HTTP-сервер доступен на порте 8081.

Выполним несколько HTTP-запросов с помощью curl и посмотрим, что получится.

1. Добавим новый товар в сервис `ProductInfo`.

```
$ curl -X POST http://localhost:8081/v1/product
        -d '{"name": "Apple", "description": "iphone7", "price": 699}'
"38e13578-d91e-11e9"
```

2. Получим существующий товар по `ProductID`:

```
$ curl http://localhost:8081/v1/product/38e13578-d91e-11e9

{"id":"38e13578-d91e-11e9","name":"Apple","description":"iphone7",
"price":699}
```

3. gRPC-шлюз также поддерживает генерацию определения обратного прокси-сервера в формате Swagger. Для этого можно использовать следующую команду:

```
protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/\
  third_party/googleapis \
  --swagger_out=logtostderr=true:. \
  product_info.proto
```

4. Эта команда сгенерирует в том же каталоге определение обратного прокси-сервера (product_info.swagger.json). В случае с нашим сервисом `ProductInfo` оно будет выглядеть следующим образом:

```
{
 "swagger": "2.0",
 "info": {
   "title": "product_info.proto",
```

```
    "version": "version not set"
  },
  "schemes": [
    "http",
    "https"
  ],
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json"
  ],
  "paths": {
    "/v1/product": {
      "post": {
        "operationId": "addProduct",
        "responses": {
          "200": {
            "description": "A successful response.",
            "schema": {
              "type": "string"
            }
          }
        },
        "parameters": [
          {
            "name": "body",
            "in": "body",
            "required": true,
            "schema": {
              "$ref": "#/definitions/ecommerceProduct"
            }
          }
        ],
        "tags": [
          "ProductInfo"
        ]
      }
    },
    "/v1/product/{value}": {
      "get": {
        "operationId": "getProduct",
        "responses": {
          "200": {
            "description": "A successful response.",
            "schema": {
```

```
              "$ref": "#/definitions/ecommerceProduct"
            }
          }
        },
        "parameters": [
          {
            "name": "value",
            "description": "The string value.",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "tags": [
          "ProductInfo"
        ]
      }
    }
  },
  "definitions": {
    "ecommerceProduct": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string"
        },
        "name": {
          "type": "string"
        },
        "description": {
          "type": "string"
        },
        "price": {
          "type": "number",
          "format": "float"
        }
      }
    }
  }
}
```

Мы реализовали для нашего gRPC-сервиса обратный прокси-сервер с поддержкой HTTP, используя gRPC-шлюз. Таким образом, мы можем открыть доступ к нашему gRPC-серверу для HTTP-клиентов. Подробности о реализации этого шлюза можно найти в репозитории grpc-gateway (oreil.ly/rN1WK).

Ранее уже упоминалось, что gRPC-шлюз поддерживается только в Go. Но существует похожая концепция, известная как перекодирование HTTP/JSON, и о ней мы поговорим в следующем разделе.

Перекодирование из HTTP/JSON в gRPC

Перекодирование — процесс преобразования HTTP-вызовов в формате JSON в RPC-вызовы с последующей их передачей gRPC-сервису. Это может пригодиться в случае, если клиентскому приложению, которое не поддерживает gRPC, нужно предоставить доступ к gRPC-сервису по HTTP с помощью JSON. Выполнить такое перекодирование поможет библиотека под названием grpc-httpjson-transcoding, написанная на C++; на сегодняшний день она применяется в Istio (oreil.ly/vWllM) и API Google (oreil.ly/KR5_X).

Прокси-сервер Envoy (oreil.ly/33hyY) тоже поддерживает перекодирование за счет предоставления для gRPC-сервисов интерфейса HTTP/JSON. По аналогии с gRPC-шлюзом нужно предоставить определение сервиса с привязками к HTTP-ресурсам. Envoy использует те же правила привязывания, которые описаны в документации Google API (oreil.ly/H6ysW). Поэтому определение сервиса, которое мы отредактировали в листинге 8.1, можно применять и для перекодирования в HTTP/JSON.

Например, метод `getProduct` сервиса `ProductInfo` определен в proto-файле с указанием типов запроса и ответа:

```
rpc getProduct(google.protobuf.StringValue) returns (Product) {
     option (google.api.http) = {
         get:"/v1/product/{value}"
     };
}
```

Если клиент вызовет этот метод, отправив GET-запрос по URL http://localhost:8081/v1/product/2, то прокси-сервер создаст переменную `google.protobuf.StringValue` со значением `2` и передаст ее gRPC-методу `getProduct()`. В результате сервис вернет запрошенный товар (`Product`) с ID 2, который прокси-сервер затем преобразует в формат JSON и отправит клиенту.

Итак, мы обсудили перекодирование в HTTP/JSON. В следующем разделе поговорим еще об одной важной концепции — об отражении gRPC-сервера.

Протокол отражения gRPC-сервера

Отражение сервера — сервис, который размещен на gRPC-сервере и предоставляет сведения о его публично доступных gRPC-сервисах. Проще говоря, отражение сервера предоставляет клиентскому приложению определения сервисов, зарегистрированных на этом сервере. Таким образом, для взаимодействия с сервисами клиенту не нужны их предкомпилированные определения.

Как уже обсуждалось в главе 2, чтобы подключиться к gRPC-сервису и взаимодействовать с ним, клиентское приложение должно знать его определение. Поэтому нам сначала нужно скомпилировать определение сервиса и затем сгенерировать из него соответствующую клиентскую заглушку. Дальше мы можем вызывать методы данной заглушки из клиентского кода. Отражение gRPC-сервера позволяет обойтись без всего этого.

Отражение сервиса может пригодиться при создании утилиты командной строки для отладки gRPC-сервера. Вместо определений сервисов такой утилите можно предоставить метод и текстовые параметры. Данный метод переведет параметры в двоичный вид, отправит их серверу и вернет пользователю ответ в формате, понятном человеку. Прежде чем применять отражение сервисов, его сначала нужно включить на серверной стороне. В листинге 8.3 показано, как это делается.

Листинг 8.3. Включение отражения для gRPC-сервера на языке Go

```
package main

import (
  ...

  pb "productinfo/server/ecommerce"
  "google.golang.org/grpc"
  "google.golang.org/grpc/reflection" ❶
)

func main() {
  lis, err := net.Listen("tcp", port)
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
```

```
  s := grpc.NewServer()
  pb.RegisterProductInfoServer(s, &server{})
  reflection.Register(s) ❷
  if err := s.Serve(lis); err != nil {
     log.Fatalf("failed to serve: %v", err)
  }
}
```

❶ Импортируем пакет `reflection` для доступа к API отражения.

❷ Регистрируем отражающий сервис на gRPC-сервере.

Включив отражение для своего сервера, вы можете использовать его с помощью своей утилиты gRPC CLI.

Утилита gRPC CLI находится в репозитории gRPC. Она имеет много возможностей, таких как вывод списка сервисов и методов, доступных на сервере, и отправка/получение удаленных вызовов с метаданными. На сегодняшний день ее необходимо собирать из исходного кода. Подробности о сборке и использовании утилиты gRPC CLI можно найти в ее репозитории (oreil.ly/jYl0h).

Собрав утилиту gRPC CLI из исходного кода (github.com/grpc/grpc), вы можете использовать ее для обращения к сервисам. Покажем это на примере нашего сервиса для управления товарами, который мы создали в главе 2. После запуска gRPC-сервера вы можете воспользоваться gRPC CLI для получения информации о сервисе `ProductInfo`.

Эта утилита поддерживает следующие действия.

- *Вывод списка сервисов.* Выполните команду, указанную ниже, чтобы получить список всех сервисов, доступных по адресу localhost:50051:

```
$ ./grpc_cli ls localhost:50051

Output:
ecommerce.ProductInfo
grpc.reflection.v1alpha.ServerReflection
```

- *Вывод подробностей о сервисе.* Чтобы получить информацию о сервисе, передайте его полное имя (в формате `<пакет>.<сервис>`) следующей команде:

```
$ ./grpc_cli ls localhost:50051 ecommerce.ProductInfo -l

Output:
package: ecommerce;
service ProductInfo {
rpc addProduct(ecommerce.Product) returns
(google.protobuf.StringValue) {}
rpc getProduct(google.protobuf.StringValue) returns
(ecommerce.Product) {}
}
```

- *Вывод подробностей о методе.* Чтобы получить информацию о методе, передайте его полное имя (в формате `<пакет>.<сервис>.<метод>`) следующей команде:

```
$ ./grpc_cli ls localhost:50051 ecommerce.ProductInfo.addProduct -l

Output:
rpc addProduct(ecommerce.Product) returns
(google.protobuf.StringValue) {}
```

- *Вывод структуры сообщения.* Чтобы получить информацию о структуре сообщения, передайте полное имя его типа (в формате `<пакет>.<тип>`) следующей команде:

```
$ ./grpc_cli type localhost:50051 ecommerce.Product

Output:
message Product {
string id = 1[json_name = "id"];
string name = 2[json_name = "name"];
string description = 3[json_name = "description"];
float price = 4[json_name = "price"];
}
```

- *Вызов удаленных методов.* Выполните следующие команды, чтобы послать серверу удаленные вызовы и получить ответы:

  1) вызовите метод `addProduct` из сервиса `ProductInfo`:

```
$ ./grpc_cli call localhost:50051 addProduct "name:
             'Apple', description: 'iphone 11', price: 699"

Output:
connecting to localhost:50051
value: "d962db94-d907-11e9-b49b-6c96cfe0687d"
Rpc succeeded with OK status
```

2) вызовите метод `getProduct` из сервиса `ProductInfo`:

```
$ ./grpc_cli call localhost:50051 getProduct "value:
              'd962db94-d907-11e9-b49b-6c96cfe0687d'"

Output:
connecting to localhost:50051
id: "d962db94-d907-11e9-b49b-6c96cfe0687d"
name: "Apple"
description: "iphone 11"
price: 699
Rpc succeeded with OK status
```

Теперь мы можем включить отражение для gRPC-сервера на языке Go и обратиться к нему с помощью утилиты gRPC CLI. То же действие доступно и на gRPC-сервере, написанном на Java; соответствующие примеры можно найти в репозитории с исходным кодом.

Обсудим еще один интересный механизм: gRPC Middleware.

gRPC Middleware

Если не углубляться в детали, то gRPC Middleware *(промежуточный слой)* — это программный механизм распределенной системы, который используется для направления клиентских запросов к внутреннему серверу путем соединения разных компонентов. Проект gRPC Middleware (oreil.ly/EqnCQ) позволяет выполнять код до и после работы gRPC-сервера и клиентского приложения.

На самом деле промежуточный слой в gRPC основан на *перехватчиках*, с которыми мы познакомились в главе 5. gRPC Middleware — коллекция перехватчиков, вспомогательных методов и утилит на языке Go, необходимых для создания gRPC-приложений. С ее помощью можно сформировать цепочку перехватчиков и использовать их на клиентской или серверной стороне. Кроме того, этот проект вбирает в себя различные компоненты с возможностью многократного использования, основанные на перехватчиках, — например, аутентификация, журналирование, проверка сообщений, повторные вызовы, мониторинг и т. д. Типичный пример работы с пакетом gRPC Middleware показан в листинге 8.4. Здесь он служит для применения нескольких перехватчиков в ходе унарного и потокового обмена сообщениями.

Листинг 8.4. Создание цепочки перехватчиков на стороне сервера с помощью gRPC Middleware

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

orderMgtServer := grpc.NewServer(
    grpc.Unaryinterceptor(grpc_middleware.ChainUnaryServer( ❶
        grpc_ctxtags.UnaryServerinterceptor(),
        grpc_opentracing.UnaryServerinterceptor(),
        grpc_prometheus.UnaryServerinterceptor,
        grpc_zap.UnaryServerinterceptor(zapLogger),
        grpc_auth.UnaryServerinterceptor(myAuthFunction),
        grpc_recovery.UnaryServerinterceptor(),
    )),
    grpc.Streaminterceptor(grpc_middleware.ChainStreamServer( ❷
        grpc_ctxtags.StreamServerinterceptor(),
        grpc_opentracing.StreamServerinterceptor(),
        grpc_prometheus.StreamServerinterceptor,
        grpc_zap.StreamServerinterceptor(zapLogger),
        grpc_auth.StreamServerinterceptor(myAuthFunction),
        grpc_recovery.StreamServerinterceptor(),
    )),
    )
```

❶ Создаем на сервере цепочку унарных перехватчиков.

❷ Создаем на сервере цепочку потоковых перехватчиков.

Перехватчики вызываются в порядке их регистрации с помощью gRPC Middleware. Этот проект также предлагает готовые перехватчики для решения распространенных задач. Некоторые из них перечислены ниже.

- *Аутентификация:*
  - `grpc_auth` — расширяемый (с помощью `AuthFunc`) промежуточный слой для аутентификации.
- *Журналирование:*
  - `grpc_ctxtags` — библиотека, которая добавляет в контекст хеш-таблицу Tag с содержимым тела запроса;
  - `grpc_zap` — интеграция библиотеки для ведения журнала zap в gRPC-обработчики;
  - `grpc_logrus` — интеграция библиотеки для ведения журнала logrus в gRPC-обработчики.

- *Мониторинг:*
  - `grpc_prometheus` — промежуточный слой для мониторинга клиентской и серверной сторон на основе Prometheus;
  - `grpc_opentracing` — клиентские и серверные перехватчики OpenTracing с поддержкой потоковой передачи и тегов, возвращаемых обработчиками.
- *Клиент:*
  - `grpc_retry` — клиентский промежуточный слой с универсальным механизмом для повторной отправки gRPC-запросов.
- *Сервер:*
  - `grpc_validator` — проверка входящих сообщений на основе кода, сгенерированного из параметров в proto-файле;
  - `grpc_recovery` — предотвращение отказа приложения и вывод gRPC-ошибки;
  - `ratelimit` — ограничение частоты gRPC-запросов с помощью собственного ограничителя.

На стороне клиента gRPC Middleware используется точно таким же образом. В листинге 8.5 показан пример кода с цепочкой клиентских перехватчиков.

Листинг 8.5. Создание цепочки перехватчиков на стороне клиента с помощью gRPC Middleware

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

clientConn, err = grpc.Dial(
   address,
     grpc.WithUnaryinterceptor(grpc_middleware.ChainUnaryClient(
          monitoringClientUnary, retryUnary)), ❶
     grpc.WithStreaminterceptor(grpc_middleware.ChainStreamClient(
          monitoringClientStream, retryStream)), ❷
)
```

❶ Цепочка клиентских унарных перехватчиков.

❷ Цепочка клиентских потоковых перехватчиков.

Как и на серверной стороне, перехватчики выполняются в порядке их регистрации в клиенте.

Далее мы поговорим о том, как открыть доступ к информации о работоспособности gRPC-сервера. В высокодоступных системах это нужно для того, чтобы периодически проверять состояние сервера и при необходимости принимать меры по минимизации ущерба.

Протокол для проверки работоспособности

В состав gRPC входит протокол проверки работоспособности (Health Checking API), с помощью которого потребители могут получать информацию о состоянии сервера. Сервер считается *неработоспособным*, если не готов обрабатывать удаленные вызовы или вовсе не отвечает на запрос о проверке состояния. Получив информацию о *неработоспособности* сервера или не дождавшись ответа, клиент может принять соответствующие меры.

Health Checking Protocol определяет API, основанный на gRPC. В качестве механизма проверки работоспособности (со стороны клиента или другой управляющей системы, такой как балансировщик нагрузки) используется gRPC-сервис. В листинге 8.6 показано стандартное определение интерфейса для проверки работоспособности в gRPC.

Листинг 8.6. Определение gRPC-сервиса для Health Checking API

```
syntax = "proto3";

package grpc.health.v1;

message HealthCheckRequest { ❶
  string service = 1;
}

message HealthCheckResponse { ❷
  enum ServingStatus {
    UNKNOWN = 0;
    SERVING = 1;
    NOT_SERVING = 2;
  }
  ServingStatus status = 1;
}

service Health {
  rpc Check(HealthCheckRequest) returns (HealthCheckResponse); ❸

  rpc Watch(HealthCheckRequest) returns (stream HealthCheckResponse); ❹
}
```

❶ Структура запроса для проверки работоспособности.

❷ Ответ на проверку работоспособности с состоянием обслуживания.

❸ Клиент может запросить состояние работоспособности сервера, вызвав метод `Check`.

❹ Для проверки работоспособности в потоковом режиме клиент может воспользоваться методом `Watch`.

Реализация gRPC-сервиса для проверки работоспособности похожа на традиционную. Во многих случаях она работает на том же gRPC-сервере, что и обычные, бизнес-ориентированные сервисы (для этого используется мультиплексирование, которое мы обсудили в главе 5). То есть проверка работоспособности ничем не отличается от обычного удаленного вызова. Результаты часто имеют развитую семантику и содержат такие подробности, как состояние каждого отдельного сервиса. Сервис, реализующий проверку работоспособности, может повторно использовать и полностью контролировать любую информацию, присутствующую на сервере.

Вернемся к серверному интерфейсу из листинга 8.6. Клиент может вызвать метод `Check` (при желании указав имя сервиса в качестве дополнительного параметра) для проверки работоспособности отдельного сервиса или всего сервера.

Вдобавок клиент может воспользоваться методом `Watch`, чтобы сделать то же самое в потоковом режиме. В таком случае происходит потоковый обмен сообщениями; это значит, что после вызова данного метода сервер начнет отправлять клиенту сообщения при каждом изменении текущего состояния.

Ниже перечислены ключевые аспекты Health Checking Protocol.

- Каждый сервис, состояние которого мы хотим сделать доступным для проверки, нужно зарегистрировать вручную. Получить общее состояние сервера можно с помощью удаленного вызова без указания имени сервиса.
- Каждый запрос для проверки работоспособности, отправляемый клиентом, должен иметь крайний срок; сервер, который не успевает ответить до его окончания, будет считаться неработоспособным.
- При выполнении каждой проверки работоспособности клиент может указать имя сервиса (или оставить его пустым). В случае нахождения этого

имени в реестре ответ `HealthCheckResponse` должен содержать статус HTTP OK и поле `status` с состоянием соответствующего сервиса (`SERVING` или `NOT_SERVING`). Если сервис не найден в реестре сервера, то ответ должен иметь состояние `NOT_FOUND`.

- Если клиенту нужно узнать общее состояние сервера, а не отдельного сервиса, то он может отправить запрос с пустым строковым параметром.
- Если сервер не предоставляет API для проверки работоспособности, то эти обязанности ложатся на сам клиент.

К сервисам для проверки работоспособности обращаются как другие gRPC-потребители, так и промежуточные подсистемы, такие как балансировщики нагрузки или прокси-серверы. Вместо того чтобы реализовывать клиент с нуля, можно воспользоваться уже готовой реализацией с поддержкой проверки работоспособности, такой как `grpc_health_probe`.

grpc_health_probe

Проектом `grpc_health_probe` (oreil.ly/I84Ui) занимается сообщество. Он позволяет проверять состояние работоспособности сервера, если тот предоставляет его в виде сервиса, реализующего Health Checking Protocol. Это универсальный клиент, способный взаимодействовать со стандартным gRPC-сервисом для проверки работоспособности. Вы можете использовать исполняемый файл `grpc_health_probe` в качестве утилиты командной строки, как показано ниже:

```
$ grpc_health_probe -addr=localhost:50051 ❶

healthy: SERVING

$ grpc_health_probe -addr=localhost:50052 -connect-timeout 600ms \
 -rpc-timeout 300ms ❷

failed to connect service at "localhost:50052": context deadline exceeded
exit status 2
```

❶ Запрос для проверки работоспособности локального gRPC-сервера, доступного на порте 50 051.

❷ Запрос для проверки работоспособности с несколькими дополнительными сетевыми параметрами.

Как можно видеть в предыдущем выводе командной строки, `grpc_health_probe` выполняет удаленный вызов, направленный к `/grpc.health.v1.Health/Check`. Если ответ имеет состояние `SERVING`, то утилита `grpc_health_probe` успешно завершает работу; в противном случае она возвращает ненулевой код выхода.

Если ваше gRPC-приложение выполняется в Kubernetes, то вы можете определить соответствующие проверки работоспособности и готовности (oreil.ly/a7bOC) для pod-оболочек своего gRPC-сервера.

Для этого gRPC-команду проверки работоспособности можно упаковать в образ Docker, как показано в следующем фрагменте файла Dockerfile:

```
RUN GRPC_HEALTH_PROBE_VERSION=v0.3.0 && \
    wget -qO/bin/grpc_health_probe \
    https://github.com/grpc-ecosystem/grpc-health-probe/releases/download/
            ${GRPC_HEALTH_PROBE_VERSION}/grpc_health_probe-linux-amd64 && \
    chmod +x /bin/grpc_health_probe
```

Затем в спецификации pod-оболочки вашего объекта развертывания Kubernetes можно определить `livenessProbe` и/или `readinessProbe`, как показано ниже:

```
spec:
  containers:
  - name: server
    image: "kasunindrasiri/grpc-productinfo-server"
    ports:
    - containerPort: 50051
    readinessProbe:
      exec:
        command: ["/bin/grpc_health_probe", "-addr=:50051"] ❶
      initialDelaySeconds: 5
    livenessProbe:
      exec:
        command: ["/bin/grpc_health_probe", "-addr=:50051"] ❷
      initialDelaySeconds: 10
```

❶ Указываем `grpc_health_probe` в качестве проверки готовности.

❷ Указываем `grpc_health_probe` в качестве проверки работоспособности.

После того как вы определите проверки работоспособности и готовности, используя `grpc_health_probe`, Kubernetes сможет принимать решения, в зависимости от состояния вашего gRPC-сервера.

Другие проекты экосистемы gRPC

Экосистема gRPC содержит много других проектов, которые могут пригодиться при разработке приложений. В частности, благодаря возможности подключать к Protocol Buffers пользовательские дополнения начали пользоваться популярностью такие проекты, как *protoc-gen-star* (PG*; https://oreil.ly/9eRq8). Кроме того, библиотеки наподобие *protoc-gen-validate* (PGV; https://oreil.ly/KlGy7) предлагают дополнения, которые генерируют код для проверки сообщений на разных языках. В экосистеме gRPC продолжают появляться новые инструменты, реализующие различные аспекты разработки gRPC-приложений.

На этом мы завершаем наше обсуждение экосистемы gRPC-компонентов. Необходимо понимать, что данные инструменты не являются частью проекта gRPC. Прежде чем использовать их в промышленных условиях, их следует основательно проанализировать. К тому же предоставленная здесь информация может меняться: одни проекты могут устареть, другие — стать общепринятыми; кроме того, в экосистеме gRPC могут появиться совершенно новые инструменты.

Резюме

Как видите, хотя рассмотренные здесь проекты и не являются частью основной реализации gRPC, они могут быть довольно полезны для разработки и использования gRPC-приложений в реальных условиях. Это проекты построены вокруг gRPC и призваны преодолеть проблемы или ограничения, которые возникают в ходе создания промышленных систем с помощью данного протокола. Например, при переводе сервисов с REST на gRPC необходимо позаботиться о существующих REST-клиентах. Для этого были созданы такие механизмы, как перекодирование из HTTP/JSON и gRPC-шлюз, благодаря которым сервисы могут обращаться к одному и тому же сервису и по REST, и по gRPC. Точно так же в целях преодоления ограничений, присущих тестированию gRPC-сервисов с помощью утилит командной строки, была придумана концепция отражения.

Учитывая высокую популярность gRPC в облачно-ориентированном мире и тот факт, что разработчики постепенно переходят на этот протокол с REST, в будущем мы увидим еще больше подобных проектов.

Поздравляем! Вы наконец дочитали книгу и познакомились практически со всеми этапами цикла разработки gRPC-приложений, а также рассмотрели многочисленные примеры кода на языках Go и Java. Надеемся, представленный материал послужит прочным фундаментом для использования gRPC в качестве технологии межпроцессного взаимодействия в ваших приложениях и микросервисах. То, чему вы здесь научились, поможет вам быстро создавать gRPC-приложения, хорошо ориентироваться в том, как они сосуществуют с другими системами, и запускать их в промышленных условиях.

Что ж, пришло время двигаться дальше. Попробуйте создать настоящие приложения, опираясь на материал, изложенный в нашей книге. Многие возможности gRPC зависят от выбранного вами языка программирования; как следствие, некоторые методики, свойственные тому или иному языку, вам придется изучить самостоятельно. Кроме того, нужно отметить, что экосистема gRPC растет экспоненциально, поэтому будет полезно следить за новейшими технологиями и фреймворками, поддерживающими данный протокол. Вперед, навстречу открытиям!

Об авторах

Касун Индрасири — архитектор программных систем, имеющий богатый опыт в области микросервисной и облачно-ориентированной архитектур, а также корпоративной интеграции. Занимает должность директора по архитектуре интеграции в WSO2 и отвечает за разработку WSO2 Enterprise Integrator. Касун написал книгу *Microservices for Enterprise* (Apress, 2018). Кроме того, он выступал на нескольких конференциях, включая O'Reilly Software Architecture Conference 2019 в Сан-Хосе и GOTO Con 2019 в Чикаго, а также на конференции WSO2. Касун живет в Сан-Хосе, штат Калифорния; организовал одну из крупнейших встреч для специалистов по микросервисам в районе залива Сан-Франциско, Silicon Valley Microservices, APIs and Integration.

Данеш Куруппу — ведущий разработчик в WSO2, работает в области корпоративной интеграции и микросервисных технологий. Как проектировщик и разработчик он руководит внедрением поддержки gRPC в открытый облачно-ориентированный язык программирования Ballerina. Является участником сообщества и ключевым соавтором таких проектов, как WSO2 Microservices Framework for Java и WSO2 Governance Registry.

Об обложке

На обложке изображена морская чернеть *(Aythya marila)*, представитель семейства утиных. Размножается весной и летом в приполярной тундре Аляски, Канады и Европы, а затем мигрирует на юг, чтобы перезимовать на побережьях Северной Америки, Европы и Азии.

У самцов желтые глаза, ярко-синий клюв, черная голова с заметным темно-зеленым отливом, белые бока и рябая спина с частым чередованием серых и белых перьев. У самок менее яркий окрас, что защищает их в период гнездования: бледно-синий клюв, над которым находится небольшое белое пятно, а также коричневые голова и тело. Эти утки в среднем достигают около 50 см в длину, имеют размах крыльев 75 см и обычно весят 900 г.

Особи морской чернети спариваются весной. Самка откладывает в среднем восемь яиц в земляное гнездо, выстланное ее собственным пухом. Утята покидают гнездо сразу после вылупления и способны себя прокормить с самого рождения. Летать они начинают только спустя сорок, а то и больше дней и в этот период, несмотря на защиту матери, являются легкой добычей для хищных птиц и наземных хищников, таких как лисы.

Чернеть относится к так называемым *нырковым уткам*; она может добывать пищу не только на земле и на плаву, но и под водой. Как и у других нырковых уток, лапки у чернети сдвинуты ближе к задней части ее компактного туловища, что помогает ей грести во время ныряния. Морская чернеть может нырять на глубину до шести метров и находиться под водой около минуты, благодаря чему ей доступно больше еды, чем большинству других нырковых уток.

Несмотря на сокращение популяции в последние 40 лет, в Красной книге эти птицы помечены как «вид, вызывающий наименьшие опасения».

Многие животные, изображенные на обложках издательства O'Reilly, находятся под угрозой исчезновения: все они важны для нашей планеты.

Титульная цветная иллюстрация создана Карен Монтгомери на основе черно-белой гравюры из книги *British Birds*.

Касун Индрасири, Данеш Курупу

gRPC: запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes

Перевел с английского А. Павлов

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2021. Наименование: книжная продукция. Срок годности: не ограничен.

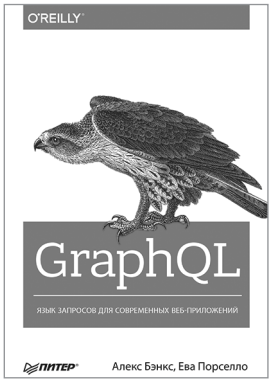
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 26.01.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 18,060. Тираж 500. Заказ 0000.

Алекс Бэнкс , Ева Порселло

GRAPHQL: ЯЗЫК ЗАПРОСОВ ДЛЯ СОВРЕМЕННЫХ ВЕБ-ПРИЛОЖЕНИЙ



GraphQL — это язык запросов, альтернативный REST и ситуативным архитектурам веб-сервисов, самая революционная технология извлечения данных со времен Ajax. Точно как React изменил взгляд веб-разработчика на создание пользовательских интерфейсов, GraphQL полностью изменит практику передачи данных по HTTP. Это практическое руководство поможет вам приступить к работе с языком GraphQL.

Рауль-Габриэль Урма, Марио Фуско, Алан Майкрофт

СОВРЕМЕННЫЙ ЯЗЫК JAVA. ЛЯМБДА-ВЫРАЖЕНИЯ, ПОТОКИ И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Преимущество современных приложений — в передовых решениях, включающих микросервисы, реактивные архитектуры и потоковую обработку данных. Лямбда-выражения, потоки данных и долгожданная система модулей платформы Java значительно упрощают их реализацию. Пришло время повысить свою квалификацию и встретить любой вызов во всеоружии!

Книга поможет вам овладеть новыми возможностями современных дополнений, таких как API Streams и система модулей платформы Java. Откройте для себя новые подходы к конкурентности и узнайте, как концепции функциональности улучшают работу с кодом.