Оглавление

Введение 11
Зачем мы написали эту книгу 11
Целевая аудитория 12
Структура издания 12
Использование примеров кода 13
Условные обозначения 14
Благодарности 15
От издательства 15

Глава 1. Введение в gRPC 16
Что такое gRPC 18
Определение сервиса 19
gRPC-сервер 21
gRPC-клиент 23
Обмен сообщениями между клиентом и сервером 24
Эволюция межпроцессного взаимодействия 24
Традиционные подходы к RPC 24
SOAP 25
REST 25
Появление gRPC 27

Почему стоит выбрать gRPC ... 28
Сравнение gRPC с другими протоколами: GraphQL и Thrift ... 31
gRPC в реальных условиях ... 33
Netflix ... 33
etcd ... 34
Dropbox ... 35
Резюме ... 35

Глава 2. Начало работы с gRPC ... 37
Определение сервиса ... 38
Определение сообщений ... 39
Определение сервисов ... 40
Реализация ... 43
Разработка сервиса ... 44
Разработка gRPC-клиента ... 55
Сборка и запуск ... 59
Сборка сервера, написанного на Go ... 60
Сборка клиента, написанного на Go ... 60
Запуск сервера и клиента, написанных на Go ... 61
Сборка сервера, написанного на Java ... 61
Сборка клиента, написанного на Java ... 61
Запуск сервера и клиента, написанных на Java ... 62
Резюме ... 62

Глава 3. Методы взаимодействия на основе gRPC ... 64
Простой (унарный) RPC ... 64
Потоковый RPC на стороне сервера ... 67
Потоковый RPC на стороне клиента ... 71
Двунаправленный потоковый RPC ... 74
Взаимодействие микросервисов на основе gRPC ... 80
Резюме ... 82

Глава 4. Внутреннее устройство gRPC ... 83
Процесс передачи сообщений в RPC ... 84
Кодирование сообщений с помощью Protocol Buffers ... 86
Методики кодирования ... 90
Обрамление сообщений с префиксом длины ... 93
gRPC поверх HTTP/2 ... 95
Запрос ... 96
Ответ ... 98
Передача сообщений с помощью разных методов взаимодействия на основе gRPC ... 100
Практическая реализация архитектуры gRPC ... 104
Резюме ... 105

Глава 5. gRPC: расширенные возможности ... 106
Перехватчики ... 106
Серверные перехватчики ... 107
Клиентские перехватчики ... 112
Крайние сроки ... 116
Механизм отмены ... 120
Обработка ошибок ... 121
Мультиплексирование ... 126
Метаданные ... 128
Создание и извлечение метаданных ... 129
Отправка и получение метаданных на стороне клиента ... 130
Отправка и получение метаданных на стороне сервера ... 132
Сопоставление имен ... 134
Балансировка нагрузки ... 135
Прокси-сервер для балансировки нагрузки ... 136
Балансировка нагрузки на стороне клиента ... 137
Сжатие ... 139
Резюме ... 140

Глава 6. Безопасность в gRPC.... 141
Аутентификация gRPC-канала с помощью TLS 141
Однонаправленное защищенное соединение.... 142
Включение безопасного соединения mTLS 146
Аутентификация вызовов в gRPC.... 151
Использование базовой аутентификации 152
Использование OAuth 2.0 157
Использование JWT.... 161
Аутентификация в Google Cloud с использованием токенов 162
Резюме.... 163

Глава 7. Использование gRPC в промышленных условиях 165
Тестирование gRPC-приложений 165
Тестирование gRPC-сервера 165
Тестирование gRPC-клиента 167
Нагрузочное тестирование 169
Непрерывная интеграция 170
Развертывание 170
Развертывание в Docker 171
Развертывание в Kubernetes.... 173
Наблюдаемость 180
Метрики 180
Журнальные записи 190
Трассировка 191
Отладка и устранение неполадок 195
Резюме.... 196

Глава 8. Экосистема gRPC 198
gRPC-шлюз 198
Перекодирование из HTTP/JSON в gRPC 206
Протокол отражения gRPC-сервера 207

gRPC Middleware 210
Протокол для проверки работоспособности 213
grpc_health_probe 215
Другие проекты экосистемы gRPC 217
Резюме 217

Об авторах 219

Об обложке 220

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.

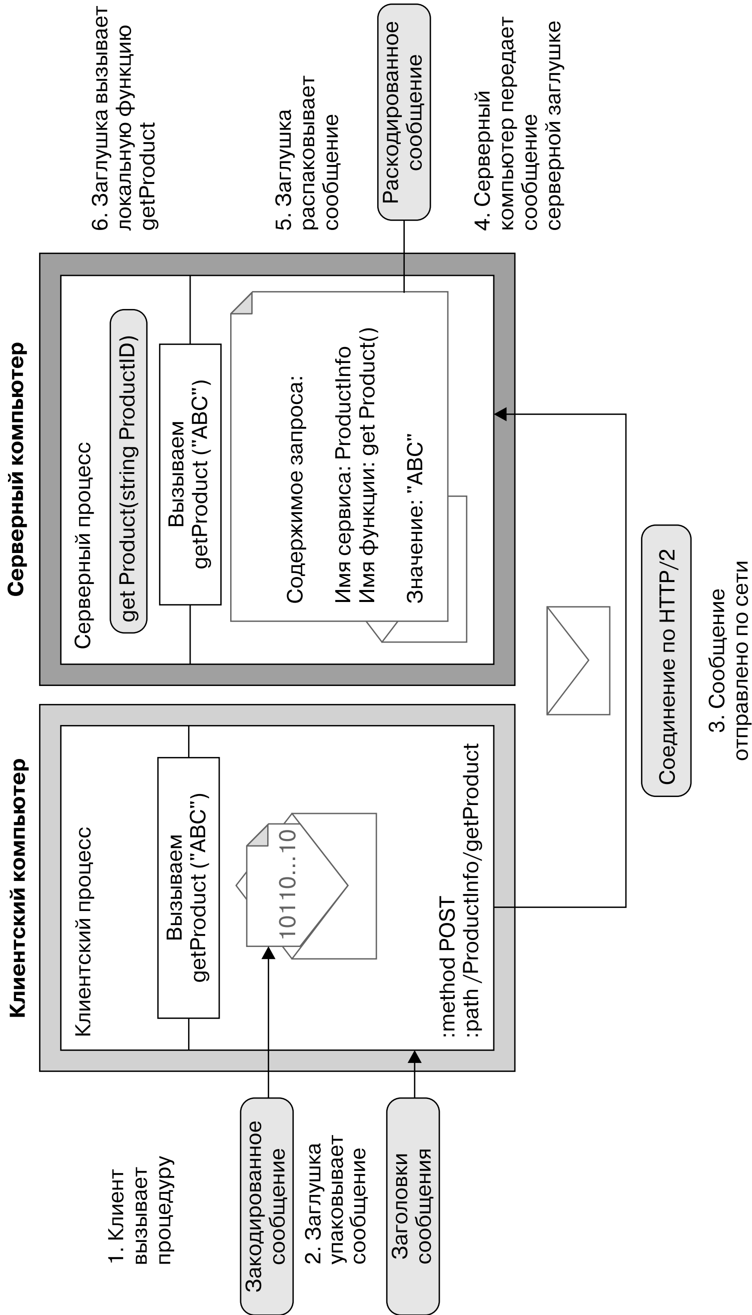


Рис. 4.1. Как выполняется удаленный вызов процедуры по сети

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.

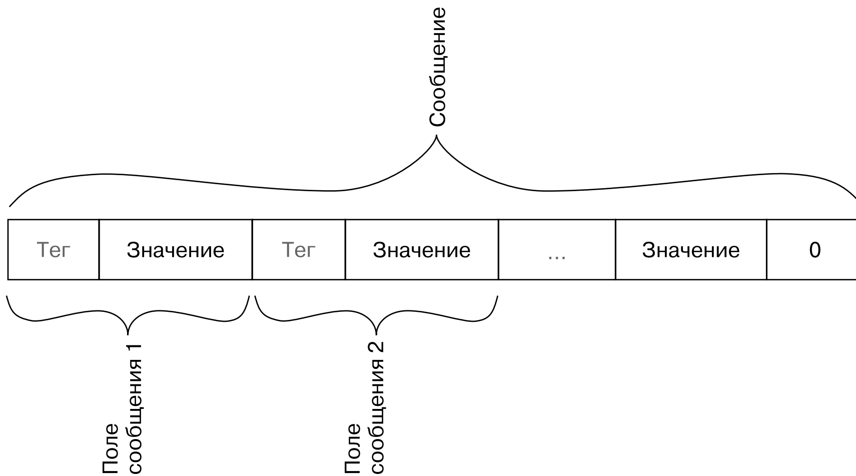


Рис. 4.2. Байтовый поток, закодированный с помощью Protocol Buffers

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.

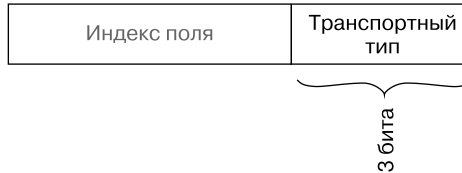


Рис. 4.3. Структура значения тега

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).

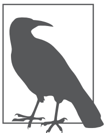
Но прежде, чем отправлять наше закодированное сообщение по сети, его необходимо обрамить.

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.

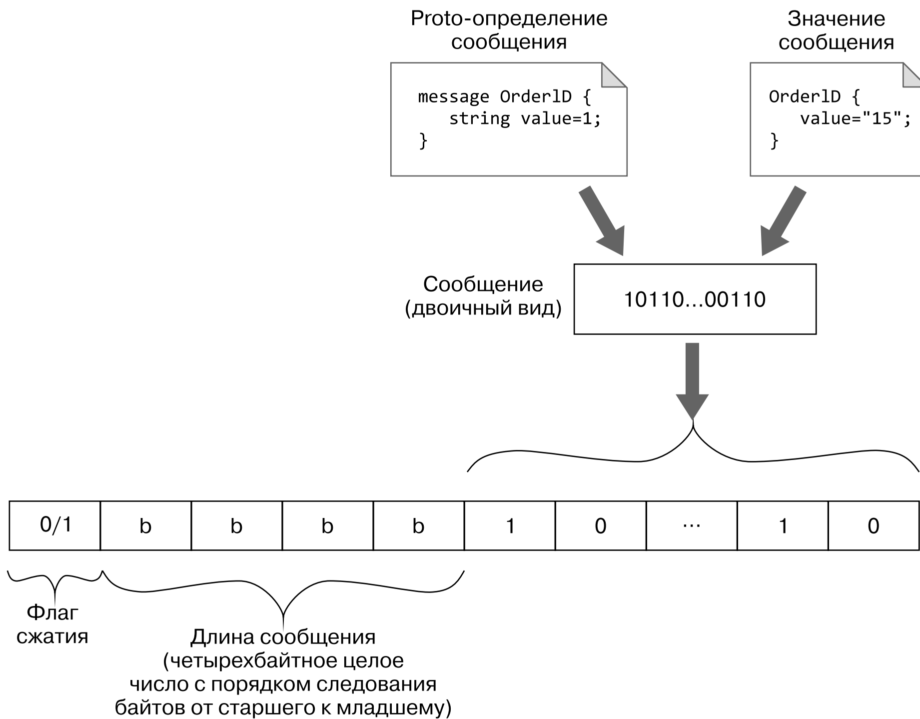


Рис. 4.4. Обрамление с префиксом длины в gRPC-сообщении

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.