O'REILLY®

TURING 图灵程序设计丛书

gRPC 与云原生应用开发

以Go和Java为例

gRPC: Up and Running

[斯里兰卡] 卡山·因德拉西里
丹尼什·库鲁普 著
张卫滨 译

中国工信出版集团
人民邮电出版社 POSTS & TELECOM PRESS

版权信息

书名：gRPC与云原生应用开发：以Go和Java为例

作者：[斯里兰卡] 卡山 • 因德拉西里　丹尼什 • 库鲁普

译者：张卫滨

ISBN：978-7-115-55498-7

版权声明
O'Reilly Media, Inc. 介绍
    业界评论
译者序
前言
    出版说明
    读者对象
    内容结构
    使用示例代码
    排版约定
    O'Reilly在线学习平台（O'Reilly Online Learning）
    联系我们
    致谢
    更多信息
第 1 章　gRPC 入门
    1.1　gRPC的定义
        1.1.1　服务定义
        1.1.2　gRPC服务器端
        1.1.3　gRPC客户端
        1.1.4　客户端–服务器端的消息流
    1.2　进程间通信技术的演化
        1.2.1　传统的RPC
        1.2.2　SOAP
        1.2.3　REST
        1.2.4　gRPC的起源
        1.2.5　选择gRPC的原因
        1.2.6　gRPC与其他协议的对比：Thrift和GraphQL
    1.3　现实世界中的gRPC

1.3.1 Netflix
1.3.2 etcd
1.3.3 Dropbox
1.4 小结
第 2 章 开始使用 gRPC
2.1 创建服务定义
2.1.1 定义消息类型
2.1.2 定义服务类型
2.2 实现
2.2.1 开发服务
2.2.2 开发gRPC客户端
2.3 构建和运行
2.3.1 构建Go服务器端应用程序
2.3.2 构建Go客户端应用程序
2.3.3 运行Go服务器端应用程序和客户端应用程序
2.3.4 构建Java服务器端应用程序
2.3.5 构建Java客户端应用程序
2.3.6 运行Java服务器端应用程序和客户端应用程序
2.4 小结
第 3 章 gRPC 的通信模式
3.1 一元RPC模式
3.2 服务器端流RPC模式
3.3 客户端流RPC模式
3.4 双向流RPC模式
3.5 使用gRPC实现微服务通信
3.6 小结
第 4 章 gRPC 的底层原理
4.1 RPC流

4.2 使用protocol buffers编码消息
编码技术
4.3 基于长度前缀的消息分帧
4.4 基于HTTP/2的gRPC
4.4.1 请求消息
4.4.2 响应消息
4.4.3 理解gRPC通信模式中的消息流
4.5 gRPC实现架构
4.6 小结
第 5 章 gRPC：超越基础知识
5.1 拦截器
5.1.1 服务器端拦截器
5.1.2 客户端拦截器
5.2 截止时间
5.3 取消
5.4 错误处理
5.5 多路复用
5.6 元数据
5.6.1 创建和检索元数据
5.6.2 发送和接收元数据：客户端
5.6.3 发送和接收元数据：服务器端
5.6.4 命名解析器
5.7 负载均衡
5.7.1 负载均衡器代理
5.7.2 客户端负载均衡
5.7.3 压缩
5.8 小结
第 6 章 安全的 gRPC

6.1 使用TLS认证gRPC通道
6.1.1 启用单向安全连接
6.1.2 启用mTLS保护的连接
6.2 对gRPC调用进行认证
6.2.1 使用basic认证
6.2.2 使用OAuth 2.0
6.2.3 使用JWT
6.2.4 使用基于令牌的谷歌认证
6.3 小结
第 7 章 在生产环境中运行 gRPC
7.1 测试gRPC应用程序
7.1.1 测试gRPC服务器端
7.1.2 测试gRPC客户端
7.1.3 负载测试
7.1.4 持续集成
7.2 部署
7.2.1 部署到Docker上
7.2.2 部署到Kubernetes上
7.3 可观察性
7.3.1 度量指标
7.3.2 日志
7.3.3 跟踪
7.4 调试和问题排查
启用额外日志
7.5 小结
第 8 章 gRPC 的生态系统
8.1 gRPC网关
8.2 gRPC的HTTP/JSON转码

8.3 gRPC服务器端反射协议
8.4 gRPC中间件
8.5 健康检查协议
8.6 gRPC健康探针
8.7 其他生态系统项目
8.8 小结
关于作者
关于封面

版权声明

O'Reilly Media, Inc. 介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

译者序

随着微服务和云原生相关技术的发展，应用程序的架构模式已从传统的单体架构或分层架构转向了分布式的计算架构。尽管分布式架构本身有一定的开发成本和运维成本，但它所带来的收益是显而易见的。

在实现分布式应用程序时，我们必须考虑两个因素：网络协议和传输载荷的编码。从最早的 RMI+Java 原生序列化，到 HTTP+JSON/XML，业界一直在尝试寻找一种兼具高效性和可读性的方案。虽然在实践微服务的过程中，大家普遍愿意采用更具语义化的 HTTP+JSON 形式，但这种形式有其自身的局限性，比如其网络传输载荷低效、接口规范松散等。正是在这样的背景下，gRPC 应运而生，借助优异的性能和谷歌的大力推广，gRPC 受到众多大厂青睐。目前，gRPC 已经成为云原生计算基金会的孵化项目，并被广泛应用于众多开源项目和企业级项目中。

gRPC 最大的特点是高性能，HTTP/2+protocol buffers 的组合使其在性能方面具备了天然的优势，这也是 gRPC 广受欢迎的重要原因。但是，相对于更成熟稳定的 HTTP+JSON 组合，gRPC 的风险不容低估，比如其协议不够稳定、社区相对较小等，这些都是在做技术选型的时候需要考虑的重要因素。正如我们所熟知的，从来就没有“银弹”，作为技术从业者，我们只能去分析和对比各种可用的技术，根据自身需求选择最合适的技术方案。

本书由浅入深，介绍了 gRPC 相关技术，从通信模式到消息编码，从服务跟踪到容器化部署，并且书中的所有示例都提供了 Java 语言和 Go 语言的两种实现。不管你是只想了解这项新技术，还是想为自己的项目寻找新方案，相信都能从本书中找到感兴趣的话题。

在本书的翻译和后期编辑过程中，我们做了很多斟酌和修正工作，力争本书内容准确。但是限于时间和译者的知识水平，书中难免有错误之处，欢迎广大读者指正，请通过 levinzhang1981@126.com 与我联系。

希望本书对你有所帮助，祝阅读愉快！

前言

如今，软件应用程序会经常通过计算机网络，借助进程间通信技术实现彼此间的连接。gRPC 是一种基于高性能 RPC（远程过程调用）的现代进程间通信风格，适用于构建分布式应用程序和微服务。随着微服务和云原生应用程序的出现，gRPC 的采用率正在呈指数级增长。

出版说明

随着 gRPC 的采用率不断增长，开发人员需要一本全面的 gRPC 书，能够将其作为终极指南应用于 gRPC 应用程序开发周期的各个阶段。现在，随处可见很多关于 gRPC 的资源和代码示例，如文档、博客、文章以及会议演讲等，但缺乏一个指导我们开发 gRPC 应用程序的综合资源。另外，没有任何资源来阐述 gRPC 协议的内部原理及其底层的运行方式。

我们编写本书的目的就是解决上述问题，并让读者对 gRPC 有一个全面的了解，让读者明白它与常规的进程间通信技术有什么区别，现实世界中的 gRPC 通信模式是什么，如何使用 Go 和 Java 构建 gRPC 应用程序，它的底层是如何运行的，如何在生产环境中运行 gRPC 应用程序，gRPC 如何与 Kubernetes 协作，以及其生态系统中的其他内容。

读者对象

本书最直接的读者是使用不同的进程间通信技术构建分布式应用程序和微服务的开发人员。在构建这样的应用程序和服务时，开发人员需要学习 gRPC 的基础知识，以及何时和如何将其用于服务间通信、在生产环境中运行 gRPC 服务的最佳实践。同时，对于采用微服务或云原生架构并设计服务间通信的架构师来说，也能从本书中受益匪浅，这是因为书中对比了 gRPC 和其他的技术，指出了何时应该使用它、何时应该避免使用它。

我们假定开发人员和架构师都对分布式计算的基础，如进程间通信技术、面向服务的架构（SOA）和微服务，有基本的了解。

内容结构

我们以实际用例阐述理论、概念的方式编排了本书的内容。全书广泛使用 Go 和 Java 编写示例代码，帮助你实际运用每个概念。我们将本书分为 8 章。

第 **1** 章　**gRPC** 入门

你将了解 gRPC 的基础知识，并将它与 REST、GraphQL 和其他 RPC 技术等类似的进程间通信风格进行对比。

第 **2** 章　开始使用 **gRPC**

你将初次体验使用 Go 或 Java 构建完整的 gRPC 应用程序。

第 **3** 章　**gRPC** 的通信模式

在这一章中，我们将使用真实的示例探索 gRPC 通信模式。

第 **4** 章　**gRPC** 的底层原理

如果你是 gRPC 高级用户，并且对 gRPC 的底层原理感兴趣，那么可以通过这一章来学习这些知识。这一章讲述在服务器端和客户端之间进行 gRPC 通信的步骤以及如何通过网络实现。

第 **5** 章　**gRPC**：超越基础知识

这一章讲述 gRPC 的一些非常重要的高级特性，如拦截器、截止时间、元数据、多路复用、负载均衡等。

第 **6** 章　安全的 **gRPC**

你将全面理解如何保护通信通道、如何认证以及如何控制用户对 gRPC 应用程序的访问。

第 **7** 章　在生产环境中运行 **gRPC**

你将了解 gRPC 应用程序的整个开发生命周期。我们将讨论测试 gRPC 应用程序、与 CI/CD 集成、在 Docker 和 Kubernetes 上部署与运行以及观察 gRPC 应用程序。

第 **8** 章　**gRPC** 的生态系统

在这一章中，我们将讨论围绕 gRPC 所构建的有用的支撑组件。在使用 gRPC 构建真正的应用程序时，这些项目中的大多数是有用的。

使用示例代码

本书的所有代码和补充材料都可以通过 https://grpc-up-and-running.github.io 下载[1]。我们强烈推荐你在阅读本书的时候尝试仓库中的这些示例，这样做能够让你更好地理解正在学习的概念。

> [1]也可前往本书的图灵社区页面下载。——编者注

我们会持续维护示例代码，以便提供最新版本的库、依赖项和开发工具。你偶尔可能会发现书中的示例代码和仓库中的示例代码有所不同。如果你发现示例代码的相关问题或者有对其改进的建议，我们强烈建议发送 pull request（PR）。

本书旨在帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书提供的示例代码。除非需要复制大量代码，否则无须联系我们获得许可。比如，使用本书中的几个代码片段编写程序无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将本书中的大量示例代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明通常包括书名、作者、出版社和 ISBN，比如“*gRPC: Up and Running* by Kasun Indrasiri and Danesh Kuruppu (O'Reilly). Copyright 2020 Kasun Indrasiri and Danesh Kuruppu, 978-1-492-05833-5”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

排版约定

本书采用了以下排版约定。

黑体

表示新术语或重点强调的内容。

等宽字体（`constant width`）

表示程序片段，以及正文中出现的变量、函数名、数据类型、环境变量、语句和关键字等。

等宽粗体（**`constant width bold`**）

表示应该由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

O'Reilly在线学习平台（O'Reilly Online Learning）

40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 在线学习平台使你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 https://www.oreilly.com。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息[2]。本书的网页是 https://oreil.ly/gRPC_Up_and_Running。

[2]也可以通过图灵社区本书主页下载示例代码或提交中文版勘误。——编者注

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

感谢本书的技术审校人，他们是 Julien Andrieux、Tim Raymond 和 Ryan Michela。同时，感谢本书的开发编辑 Melissa Potter 的指导和支持，以及策划编辑 Ryan Shaw 给予的所有支持。最后，感谢整个 gRPC 社区创建了这样一个伟大的开源项目。

更多信息

扫描下方二维码，即可获取电子书相关信息及读者群通道入口。

第 1 章　gRPC 入门

现代软件应用程序很少是孤立运行的，相反，它们会通过计算机网络连接在一起，并且以互相传递消息的方式进行通信和协调。因此，现代软件系统是分布式软件应用程序的集合，这些应用程序在不同的网络位置运行，并且运用不同的通信协议在彼此间传递消息。例如，一个在线零售软件系统会由多个分布式应用程序组成，如订单管理应用程序、商品目录应用程序和数据库等。为了实现在线零售系统的业务功能，这些分布式应用程序需要相互连接。

微服务架构

微服务架构将软件应用程序构建为一组独立、自治（独立开发、部署和扩展）、松耦合、面向业务能力[1]的服务[2]。

[1]业务能力有别于业务功能。“能力”一词更强调软件系统所具有的能力，而不仅仅是完成指定的任务。——译者注

[2]参见卡山 • 因德拉西里等人所著的 *Microservices for the Enterprise*。

随着微服务架构和云原生架构的出现，为多种业务能力所构建的传统软件系统被进一步拆分为一组细粒度、自治和面向业务能力的实体，也就是微服务。因此，基于微服务的软件系统也需要借助进程间（或服务间、应用程序间）通信技术，将这些微服务通过网络连接起来。比如，对于一个采用微服务架构实现的在线零售系统，我们会发现它有多个互相连接的微服务，如订单管理、搜索、结账、配送等。与传统应用程序不同，微服务的细粒度特性使得网络通信连接的数量陡增。因此，不管采用哪种架构风格（传统架构或微服务架构），进程间通信技术都是现代分布式软件应用程序的重要组成部分。

进程间通信通常会采用消息传递的方式来实现，要么是同步的请求–响应风格，要么是异步的事件驱动风格。在同步通信风格中，客户端进程通过网络发送请求消息到服务器进程，并等待响应消息。在异步的事件驱动风格中，进程间会通过异步消息传递进行通信，这个过程会用到一

个中介，也就是事件代理（event broker）。我们可以根据业务场景，选择希望实现的通信模式。

当为现代云原生应用程序和微服务实现同步的请求–响应风格的通信时，最常见和最传统的方式就是将它们构建为 RESTful 服务。也就是说，将应用程序或服务建模为一组资源，这些资源可以通过 HTTP 的网络调用进行访问和状态变更。但是，对大多数使用场景来说，使用 RESTful 服务来实现进程间通信显得过于笨重、低效并且易于出错。我们通常需要扩展性强、松耦合的进程间通信技术，该技术比 RESTful 服务更高效。这也就是 gRPC 的优势所在，gRPC 是构建分布式应用程序和微服务的现代进程间通信风格（本章稍后会对比 gRPC 和 RESTful 服务）。gRPC 主要采用同步的请求–响应风格进行通信，但在建立初始连接后，它完全可以以异步模式或流模式进行操作。

本章将介绍 gRPC 的定义以及发明这项进程间通信协议的主要动机，其间会借助一些实际的应用场景来深入探讨 gRPC 的核心构成要素。另外，本章还涉及进程间通信技术本身及其演化过程，熟悉这一点非常重要，有助于理解 gRPC 试图解决的关键问题。本章将逐一介绍这些技术，并对它们进行对比。下面先来看一下 gRPC 的定义。

1.1 gRPC的定义

gRPC[3] 是一项进程间通信技术，可以用来连接、调用、操作和调试分布式异构应用程序。就像调用本地函数一样，整个过程操作起来很简单。

[3]在每个 gRPC 发布版本中，字母 g 的含义都不同。比如 1.1 版本的 g 代表 good（优秀），1.2 版本的 g 代表 green（绿色）。——译者注

在开发 gRPC 应用程序时，先要定义服务接口，其中应包含如下信息：消费者消费服务的方式、消费者能够远程调用的方法以及调用这些方法所使用的参数和消息格式等。在服务定义中所使用的语言叫作接口定义语言（interface definition language，IDL）。

借助服务定义，可以生成服务器端代码，也就是服务器端骨架 [4]，它通过提供低层级的通信抽象简化了服务器端的逻辑。同时，还可以生成客户端代码，也就是客户端存根，它使用抽象简化了客户端的通信，为不同的编程语言隐藏了低层级的通信。就像调用本地函数那样，客户端能够远程调用我们在服务接口定义中所指定的方法。底层的 gRPC 框架处理所有的复杂工作，通常包括确保严格的服务契约、数据序列化、网络通信、认证、访问控制、可观察性等。

[4]这里的“骨架”和“存根”都是代理。服务器端代理叫作“骨架”（skeleton），客户端代理叫作“存根”（stub）。——编者注

为了理解 gRPC 的基本概念，我们来看一个使用 gRPC 实现微服务的实际场景。假设我们正在构建一个在线零售应用程序，该应用程序由多个微服务组成。如图 1-1 所示，假设我们要构建一个微服务来展现在线零售应用程序中可售商品的详情（第 2 章会从零开始构建该场景）。将 `ProductInfo` 服务建模为 gRPC 服务，通过网络对外暴露。

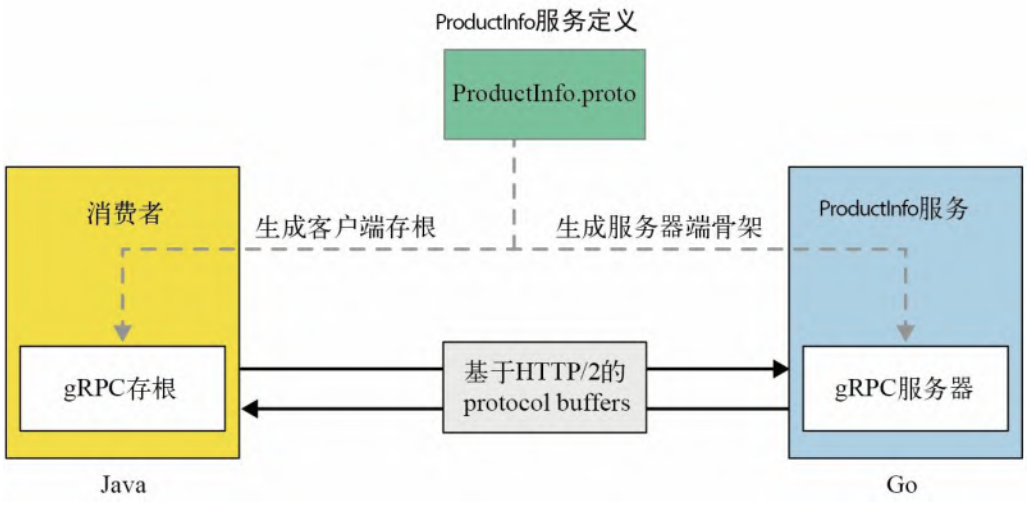

图 **1-1**：基于 **gRPC** 的微服务及其消费者

服务定义是在 ProductInfo.proto 文件中声明的，服务器端和客户端都会使用该文件来生成代码。这里假设 `ProductInfo` 服务使用 Go 语言来实现，消费者使用 Java 语言来实现，两者之间的通信则通过 HTTP/2 来进行。

接下来深入了解 gRPC 通信的细节。构建 gRPC 服务的第一步是创建服务接口定义，其中包括该服务暴露的方法及其输入参数和返回类型。下面介绍服务定义的细节。

1.1.1 服务定义

gRPC 使用 protocol buffers 作为 IDL 来定义服务接口。protocol buffers 是语言中立、平台无关、实现结构化数据序列化的可扩展机制[5]。服务接口定义在 proto 文件中指定，也就是在扩展名为“.proto”的普通文本文件中。我们要按照普通的 protocol buffers 格式来定义 gRPC 服务，并将 RPC 方法参数和返回类型指定为 protocol buffers 消息。因为服务定义是 protocol buffers 规范的扩展，所以可以借助特殊的 gRPC 插件来根据 proto 文件生成代码。

> [5]第 4 章会详细介绍 protocol buffers 的基本原理，现在可将其看作一种数据序列化机制。

在示例场景中，`ProductInfo` 服务接口可以使用代码清单 1-1 中的 protocol buffers 来定义，其中涉及远程方法调用、相关输入参数和输出

参数以及这些参数的类型定义（或消息格式），这些也是 ProductInfo 服务定义的组成部分。

代码清单 **1-1**　使用 protocol buffers 来定义 ProductInfo 服务（gRPC）

```
// 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 服务定义首先声明所使用的 protocol buffers 版本（proto3）。

❷ 用来防止协议消息类型之间发生命名冲突的包名，该包名也会用来生成代码。

❸ 定义 gRPC 服务的接口。

❹ 添加商品的远程方法，该方法会返回商品 ID 作为响应。

❺ 基于商品 ID 获取商品的远程方法。

❻ 定义 Product 的消息格式或类型。

❼ 保存商品 ID 的字段（名–值对），具有唯一的字段编号，该编号用来在二进制格式消息中识别字段。 ❽ 用于商品标识号的用户定义类型。

服务就是可被远程调用的一组方法，比如 addProduct 方法和 getProduct 方法。每个方法都有输入参数和返回类型，既可以被定义为服务的一部分，也可以导入 protocol buffers 定义中。

输入参数和返回参数既可以是用户定义类型，比如 Product 类型和 ProductID 类型，也可以是服务定义中已经定义好的 protocol buffers 已知类型。这些类型会被构造成消息，每条消息都是包含一系列名–值对信息的小型逻辑记录，这些名–值对叫作字段。这些字段都是具有唯一编号的名–值对（如 string id = 1），在二进制形式消息中，可以用编号来识别相应字段。

该服务定义会被用来构建 gRPC 应用程序的服务器端和客户端。1.1.2 节将介绍 gRPC 服务器端的实现细节。

1.1.2 gRPC服务器端

一旦服务定义准备就绪，就可以使用 protocol buffers 编译器 protoc 来生成服务器端和客户端的代码了。借助 gRPC 的 protocol buffers 插件，可以生成 gRPC 服务器端代码、客户端代码以及常规的 protocol buffers 代码，从而填充、序列化和检索消息类型。

在服务器端，需要实现该服务定义，并运行 gRPC 服务器来处理客户端的调用。因此，为了让服务器端的 ProductInfo 服务完成其任务，需要先做以下两件事情。

01. 通过重载服务基类，实现所生成的服务器端骨架的逻辑。
02. 运行 gRPC 服务器，监听来自客户端的请求并返回服务响应。

要实现服务逻辑，首先要根据服务定义生成服务器端骨架。例如，在代码清单 1-2 中，可以看到使用 Go 语言为 ProductInfo 服务所生成的远程函数。在这些远程函数体中，我们可以实现每个函数的逻辑。

代码清单 **1-2**　使用 Go 语言为 ProductInfo 服务实现 gRPC 服务器端逻辑

```
import (
  ...
  "context"
```

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

// 使用Go语言的ProductInfo实现

// 添加商品的远程方法
func (s *server) AddProduct(ctx context.Context, in *pb.Product) (
      *pb.ProductID, error) {
   // 业务逻辑
}

// 获取商品的远程方法
func (s *server) GetProduct(ctx context.Context, in *pb.ProductID) (
     *pb.Product, error) {
   // 业务逻辑
}
```

当服务实现准备就绪之后，接下来需要运行 gRPC 服务器，从而监听来自客户端的请求，将这些请求分发到服务实现，并将服务的响应返回到客户端。代码清单 1-3 展示了 ProductInfo 服务用例，其中使用 Go 语言来编写 gRPC 服务器实现。另外，这里打开了一个 TCP 端口并启动了 gRPC 服务器，同时在该服务器上注册了 ProductInfo 服务。

代码清单 **1-3**　使用 Go 语言为 ProductInfo 服务运行 gRPC 服务器

```
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)
  }
}
```

这就是服务器端要做的所有事情了。接下来看一下 gRPC 的客户端实现。

1.1.3　gRPC客户端

与服务器端类似，可以使用服务定义生成客户端存根。客户端存根提供了与服务器端类似的方法，供客户端代码进行调用。客户端存根会将这些方法转换成对服务器端的远程函数网络调用。由于 gRPC 服务定义是语言中立的，能够为所支持的任意语言（通过第三方实现）生成客户端和服务器端，因此对于 `ProductInfo` 服务用例来说，虽然我们的服务器端使用 Go 语言来实现，但是仍可以生成使用 Java 语言的客户端存根。代码清单 1-4 展示了用 Java 语言编写的代码，尽管所使用的编程语言不同，但可以看到，客户端实现仅涉及几个简单的步骤，包括建立与远程服务器的连接、将客户端存根与连接关联到一起，以及使用客户端存根调用远程方法。

代码清单 **1-4**　调用服务远程方法的 gRPC 客户端

```
// 使用远程服务器地址创建通道
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8080
   .usePlaintext(true)
   .build();

// 使用通道初始化阻塞式存根
ProductInfoGrpc.ProductInfoBlockingStub stub =
       ProductInfoGrpc.newBlockingStub(channel);

// 使用阻塞式存根调用远程方法
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());
```

现在我们已经理解了 gRPC 的关键概念，接下来详细了解 gRPC 客户端–服务器端的消息流。

1.1.4　客户端–服务器端的消息流

当调用 gRPC 服务时，客户端的 gRPC 库会使用 protocol buffers，并将 RPC 的请求编排（marshal）为 protocol buffers 格式，然后将其通过 HTTP/2 进行发送。在服务器端，请求会被解排（unmarshal），对应的过程调用会使用 protocol buffers 来执行。响应会遵循类似的执行流，从

服务器端发送到客户端。gRPC 会使用 HTTP/2 来进行有线传输，HTTP/2 是一个高性能的二进制消息协议，支持双向的消息传递。第 4 章将结合 protocol buffers 进一步讨论有关 gRPC 客户端和服务器端消息流的细节，以及 gRPC 对 HTTP/2 的使用方式。

编排是将参数和远程函数打包的过程，解排则是解包消息到对应的方法调用的过程。

在进一步了解和研究 gRPC 之前，还有一点很重要，那就是了解不同的进程间通信技术，以及它们随时间推移的演化过程。

1.2 进程间通信技术的演化

随着时间的推移，进程间通信技术发生了巨大的变化。各种各样的新技术不断涌现，以满足现代化的需求并提供更好、更高效的开发体验。因此，了解进程间通信技术是如何演化的以及 gRPC 是如何形成的就显得非常重要了。接下来介绍最常用的进程间通信技术，并尝试将它们与 gRPC 进行比较。

1.2.1 传统的RPC

在构建客户端–服务器端应用程序方面，RPC 是很流行的进程间通信技术。借助 RPC，客户端能够像调用本地方法那样远程调用某个方法的功能。早期有一些很流行的 RPC 实现，比如通用对象请求代理体系结构（common object request broker architecture，CORBA）和 Java 远程方法调用（remote method invocation，RMI），它们都用来构建和连接服务或应用程序。但是，大多数传统的 RPC 实现极其复杂，因为它们构建在 TCP 这样的通信协议之上，而这会妨碍互操作性，并且它们还有大量的规范限制。

1.2.2 SOAP

鉴于 CORBA 等传统 RPC 实现的局限性，简单对象访问协议（simple object access protocol，SOAP）应运而生，并且得到了微软、IBM 等企业的大力推广。SOAP 是面向服务的架构（service-oriented architecture，SOA）中的标准通信技术，用于在服务（在 SOA 中通常叫作 Web 服务）之间交换基于 XML 的结构化数据，并且能够基于任意的底层通信协议进行通信，其中最常用的协议是 HTTP。

通过 SOAP，可以定义服务接口、服务的操作以及调用这些操作的 XML 消息格式。SOAP 曾是一项非常流行的技术，但其消息格式的复杂性以及围绕 SOAP 所构建的各种规范的复杂性，妨碍了构建分布式应用程序的敏捷性。因此，在现代分布式应用程序开发中，SOAP Web 服务被认为是一种遗留技术。大多数现有的分布式应用程序采用 REST 架构风格，而非 SOAP。

1.2.3 REST

描述性状态迁移（representational state transfer，REST）架构风格起源于 Roy Fielding 的博士论文。Fielding 是 HTTP 规范的主要作者之一，也是 REST 架构风格的创始人。REST 是面向资源的架构（resource-oriented architecture，ROA）的基础，在这种架构中，需要将分布式应用程序建模为资源集合，访问这些资源的客户端可以变更这些资源的状态（创建、读取、更新或删除）。

REST 的通用实现是 HTTP，通过 HTTP，可以将 RESTful Web 应用程序建模为能够通过唯一标识符访问的资源集合。应用于资源的状态变更操作会采用 HTTP 动词（GET、POST、PUT、DELETE、PATCH 等）的形式，资源的状态会以文本的格式来表述，如 JSON、XML、HTML、YAML 等。

实际上，通过 HTTP 和 JSON 将应用程序构建为 REST 架构风格已成为构建微服务的标准方法。但是，随着微服务的数量及其网络交互的激增，RESTful 服务已经无法满足现代化的需求了。下面介绍 RESTful 服务的 3 个主要局限性，这些局限性妨碍了其作为消息协议在现代微服务应用程序中的运用。

01. 基于文本的低效消息协议

    从本质上来讲，RESTful 服务建立在基于文本的传输协议（如 HTTP 1.*x*）之上，并且会使用人类可读的文本格式，如 JSON。但是，在进行服务与服务之间的通信时，通信双方都不需要这种人类可读的文本化格式，这时使用这种格式非常低效。

    客户端应用程序（源）生成需要发送给服务器的二进制内容，然后需要将二进制结构转换成文本（如果使用 HTTP 1.*x*，就只能发送文本化消息），并通过网络以文本的形式（借助 HTTP）发送到另一台机器上，这台机器需要在服务器端（目标）解析文本并将其转换回二进制结构。其实，我们也可以很轻松地发送映射服务和消费者业务逻辑的二进制内容，采用 JSON 格式主要是因为它是“人类可读的”，相对来说易于使用。这涉及工具选择问题，而不是二进制协议问题。

02. 应用程序之间缺乏强类型接口

随着越来越多的服务要通过网络进行交互，而且这些服务使用完全不同的语言来构建，缺乏明确定义和强类型的服务接口成了使用 RESTful 服务的主要阻碍。RESTful 中现有的各种服务定义技术（如 OpenAPI/Swagger 等）都是事后的补救措施，并没有与底层的架构风格或消息协议紧密集成在一起。

在构建这种分散的应用程序时，会遇到很多的不兼容、运行时错误和互操作等问题。例如，在开发 RESTful 服务时，应用程序之间并不需要共享服务定义和类型定义的信息。但是，在开发 RESTful 应用程序时，我们要么通过网络查看文本格式，要么使用第三方 API 定义技术（如 OpenAPI）。因此，现在非常重要的任务就是拥有现代化的强类型服务定义技术以及框架，从而为多语言技术生成核心的服务器端代码和客户端代码。

03. **REST** 架构风格难以强制实施

REST 架构风格有很多“好的实践”，只有遵循这些实践，才能构建出真正的 RESTful 服务。但是，由于它们并没有作为实现协议（比如 HTTP）的一部分进行强制的要求，因此在实现阶段，这些实践很难实施。事实上，大多数自称 RESTful 的服务并没有遵循基础的 REST 架构风格，也就是说，这些所谓的 RESTful 服务不过是通过网络公开的 HTTP 服务。因此，开发团队必须花费大量时间来维护 RESTful 服务的一致性和纯度。

鉴于进程间通信技术在构建现代云原生应用程序时所存在的这些限制，人们开始寻求更好的消息协议。

1.2.4 gRPC的起源

长期以来，谷歌有一个名为 Stubby 的通用 RPC 框架，用来连接成千上万的微服务，这些微服务跨多个数据中心并且使用完全不同的技术来构建。Stubby 的核心 RPC 层每秒能处理数百亿次的互联网请求。Stubby 有许多很棒的特性，但无法标准化为业界通用的框架，这是因为它与谷歌内部的基础设施耦合得过于紧密。

2015 年，谷歌发布了开源 RPC 框架 gRPC，这个 RPC 基础设施具有标准化、可通用和跨平台的特点，旨在提供类似 Stubby 的可扩展性、性能和功能，但它主要面向社区。

在此之后，gRPC 的受欢迎程度陡增，很多大型公司大规模采用了 gRPC，如 Netflix、Square、Lyft、Docker、CoreOS 和思科。接着，gRPC 加入了云原生计算基金会（Cloud Native Computing Foundation，CNCF），这是最受欢迎的开源软件基金会之一，它致力于让云原生计算具备通用性和可持续性。gRPC 从 CNCF 生态系统项目中获得了巨大的发展动力。

下面看一下相对于传统进程间通信协议，选择使用 gRPC 的一些关键原因。

1.2.5 选择gRPC的原因

gRPC 是一种支持互联网规模的进程间通信技术，可以弥补传统进程间通信技术的大多数缺点。鉴于 gRPC 所带来的收益，越来越多的现代应用程序和服务器将其进程间通信协议替换成了 gRPC。在面对如此众多的可选方案时，为什么选择 gRPC 作为通信协议呢？下面详细介绍 gRPC 的关键优势。

01. **gRPC** 的优势

gRPC 的优势是它被越来越多的人所采用的关键所在，主要有以下几个方面。

提供高效的进程间通信

gRPC 没有使用 JSON 或 XML 这样的文本化格式，而是使用一个基于 protocol buffers 的二进制协议与 gRPC 服务和客户端通信。同时，gRPC 在 HTTP/2 之上实现了 protocol buffers，从而能够更快地处理进程间通信。这样一来，gRPC 就变成了最高效的进程间通信技术之一。

具有简单且定义良好的服务接口和模式

gRPC 为应用程序开发提供了一种契约优先的方式。也就是

说，首先必须定义服务接口，然后才能去处理实现细节。因此，与 RESTful 服务定义中的 OpenAPI/Swagger 和 SOAP Web 服务中的 WSDL 不同，gRPC 提供了简单但一致、可靠且可扩展的应用程序开发体验。

属于强类型

因为使用 protocol buffers 来定义 gRPC 服务，所以 gRPC 服务契约清晰定义了应用程序间进行通信所使用的类型。这样一来，在构建跨多个团队和技术类型的云原生应用程序时，对于其所产生的大多数运行时错误和互操作错误，可以通过静态类型来克服，因此分布式应用程序的开发更加稳定。

支持多语言

gRPC 支持多种编程语言。基于 protocol buffers 的服务定义是语言中立的。因此，我们可以选择任意一种语言，它们都能与现有的 gRPC 服务或客户端进行互操作。

支持双工流

gRPC 在客户端和服务器端都提供了对流的原生支持，这些功能都被整合到了服务定义本身之中。因此，开发流服务或流客户端变得非常容易。与传统的 RESTful 服务消息风格相比，gRPC 的关键优势就是能够同时构建传统的请求–响应风格的消息以及客户端流和服务器端流。

具备内置的商业化特性

gRPC 提供了对商业化特性的内置支持，如认证、加密、弹性（截止时间和超时）、元数据交换、压缩、负载均衡、服务发现等（第 5 章会讨论这些功能）。

与云原生生态系统进行了集成

gRPC 是 CNCF 的一部分，大多数现代框架和技术对 gRPC 提供了原生支持。例如，CNCF 下的很多项目（如 Envoy）支持使用 gRPC 作为通信协议。另外，对于横切性的特性，比如度量指标和

监控，gRPC 也得到了大多数工具的支持，比如使用 Prometheus 来监控 gRPC 应用程序。

业已成熟并被广泛采用

通过在谷歌进行的大量实战测试，gRPC 已发展成熟。许多大型科技公司采用了 gRPC，如 Square、Lyft、Netflix、Docker、CoreOS 和思科等。

与其他技术一样，gRPC 也存在一定的劣势。在开发应用程序时，了解这些方面非常有用。

02. **gRPC** 的劣势

下面介绍 gRPC 的一些劣势，在选择它来构建应用程序时，需要注意以下 3 点。

gRPC 可能不太适合面向外部的服务

大多数的外部消费者可能对 gRPC、REST 或 HTTP 等协议很陌生。因此，如果希望将应用程序或服务通过互联网暴露给外部客户端，gRPC 可能不是最适合的协议。gRPC 服务具有契约驱动、强类型的特点，这可能会限制我们向外部暴露的服务的灵活性，同时消费者的控制权会削弱很多（这与 1.2.6 节讨论的 GraphQL 协议有所不同）。按照设计，gRPC 网关将是克服该问题的解决方案。第 8 章会对此进行详细讨论。

巨大的服务定义变更是复杂的开发流程

在现代的服务间通信场景中，模式修改很常见。如果出现巨大的 gRPC 服务定义变更，通常需要重新生成客户端代码和服务器端代码。这需要整合到现有的持续集成过程中，可能会让整个开发生命周期复杂化。但是，大多数 gRPC 服务定义的变更可以在不破坏服务契约的情况下完成，而且只要不引入破坏性的变更，gRPC 就可以与使用不同版本 proto 的客户端和服务器端进行交互。因此，大多数情况并不需要重新生成代码。

gRPC 生态系统相对较小

与传统的 REST 或 HTTP 等协议相比，gRPC 的生态系统依然相对较小。浏览器和移动应用程序对 gRPC 的支持依然处于初级阶段。

在开发应用程序时，必须注意这些方面的问题。由此可以看到，gRPC 并不是适用于所有进程间通信需求的万能技术。相反，你需要评估业务场景和需求，选择适当的消息协议。第 8 章会讨论其中的一些指导原则。

如前所述，目前有很多新兴的进程间通信技术。因此，有一点非常重要，那就是了解如何将 gRPC 与在现代应用程序开发中流行的类似技术进行对比，从而为服务选择最合适的协议。

1.2.6 gRPC与其他协议的对比：Thrift和GraphQL

前面详细讨论了 RESTful 服务的局限性，正是这些局限性为 gRPC 的诞生奠定了基础。无独有偶，还有很多新兴的进程间通信技术，它们的问世也是为了满足相同的需求。下面看一下目前较为流行的技术，并将之与 gRPC 进行对比。

01. **Thrift**

Apache Thrift（以下简称 Thrift）是与 gRPC 类似的 RPC 框架，最初由 Facebook 开发，后来被捐赠给了 Apache。它有自己的接口定义语言并提供了对多种编程语言的支持。Thrift 可以在定义文件中定义数据类型和服务接口。Thrift 编译器以服务定义作为输入，能够生成客户端代码和服务器端代码。Thrift 的传输层为网络 I/O 提供了抽象，并将 Thrift 从系统的其他组成部分中解耦出来，这意味着 Thrift 可以在任意传输实现上运行，如 TCP、HTTP 等。

如果将 Thrift 和 gRPC 进行对比，可以发现它们遵循相同的设计理念和使用目标。但是，两者之间也有一些重要的区别。

传输方面

相对于 Thrift，gRPC 的倾向性更强，它为 HTTP/2 提供了一流

的支持。gRPC 基于 HTTP/2 的实现充分利用了该协议的功能，从而实现了高效率并且能够支持像流这样的消息模式。

流方面

gRPC 服务定义原生支持双向流（客户端和服务器端），它本身便是服务定义的一部分。

采用情况和社区资源方面

从采用情况来看，gRPC 的势头似乎更好，它已围绕 CNCF 项目成功构建了一个良好的生态系统。同时，gRPC 的社区资源非常丰富，比如良好的文档、外部的演讲以及示例。因此，相对于 Thrift，采用 gRPC 会更顺利一些。

性能方面

虽然目前还没有 gRPC 和 Thrift 对比的官方结果，但一些在线资源对比了两者的性能，结果显示 Thrift 的数据表现更好。然而，gRPC 的绝大多数发布版本经过了大量的性能测试。因此，性能问题不太可能是选择 Thrift 而非 gRPC 的决定性因素。同时，一些其他 RPC 框架提供了类似的功能，但不管怎样，gRPC 是目前最标准、最具交互性和采用范围最广的 RPC 技术，处于领先地位。

02. **GraphQL**

GraphQL 是另一项越来越流行的进程间通信技术，该项目由 Facebook 发起并通过开源进行了标准化。它是一门针对 API 的查询语言，并且是基于既有数据满足这些查询的运行时。GraphQL 为传统的客户端–服务器端通信提供了一种完全不同的方法，该方法允许客户端定义希望获得的数据、获取数据的方式以及数据格式。gRPC 则有针对远程方法的特定契约，借此实现客户端和服务器端之间的通信。

GraphQL 更适合面向外部的服务或 API，它们被直接暴露给消费者。在这种情况下，消费者需要对来自服务器端的数据有更多的控制权。以在线零售应用程序场景为例，假设 `ProductInfo` 服务的

消费者只需要关于商品的特定信息，而不是商品属性的完整集合，而且他们希望能有一种方法来指定想要的信息，那么我们可以借助 GraphQL 来建模一个服务，允许消费者使用 GraphQL 查询语言来查询服务并获取想要的信息。

在 GraphQL 和 gRPC 的大多数使用场景中，GraphQL 用于面向外部的服务或 API，而支撑 API 的内部服务则使用 gRPC 来实现。

接下来看一些实际的 gRPC 采用者及其使用场景。

1.3 现实世界中的gRPC

任何进程间通信协议的成功在很大程度上都依赖于行业范围内的采用情况，以及项目背后的用户社区和开发人员社区。gRPC 已广泛地应用于微服务和云原生应用程序的构建。下面介绍 gRPC 的一些主要的成功案例。

1.3.1 Netflix

Netflix 是一家基于订阅的视频流公司，也是大规模实践微服务架构的先驱之一，其中所有视频流功能都是通过面向外部的托管服务（或 API）提供给消费者的，有数百个后端服务支撑着其 API。因此，在 Netflix 使用场景中，进程间（或服务间）通信是最重要的方面之一。在微服务实现的初期，Netflix 使用基于 HTTP/1.1 的 RESTful 服务构建了自己的进程间通信技术栈，这支撑了 Netflix 产品大约 98% 的业务场景。

但是，在互联网规模的运维中，Netflix 发现了 RESTful 服务方式的一些局限性。对于 RESTful 微服务的消费者，他们会首先探查 RESTful 服务的资源及其所需的消息格式，然后开始进行编写。这种方式非常耗时，影响了开发人员的效率，并且增加了代码出错的风险。由于缺乏统一的服务接口定义，服务的实现和消费也面临着挑战，因此 Netflix 最初想构建一个内部的 RPC 框架来突破这些限制，但在评估了可用的技术栈后，它选择了 gRPC 作为其服务间通信技术。在评估的过程中，Netflix 发现 gRPC 最实用的地方就是将所有需要的职责封装到一个易于使用的包中。

在采用 gRPC 之后，Netflix 见证了开发效率的巨大提升。举例来说，对于每个客户端，几百行的自定义代码被替换成了只有两三行的 proto 配置代码。原来创建客户端可能需要两三周的时间，现在使用 gRPC 几分钟就能完成。平台的稳定性也得到了很大的提升，这是因为大多数商业化特性不再需要手动编写代码，而且现在有了一种全面且安全的方式来定义服务接口。得益于 gRPC 所带来的性能提升，Netflix 整个平台的延迟也有所减少。由于大多数的进程间通信场景已采用了 gRPC，因此对于一些为使用 REST 和 HTTP 等协议的进程间通信而构建的内部项目（如 Ribbon），Netflix 似乎已将其设置为维护模式（不在积极开发

中），并转而使用 gRPC。

1.3.2 etcd

etcd 是可靠的分布式键–值存储设施，可用于存储分布式系统中最关键的数据。它是 CNCF 中最受欢迎的开源项目之一，被 Kubernetes 等其他开源项目广泛采用。gRPC 成功的关键因素在于其 API 具有简单、定义良好、易于使用且面向用户的特点，etcd 使用了 gRPC 面向用户的 API，从而充分发挥了 gRPC 的作用。

1.3.3 Dropbox

Dropbox 是一个文件托管服务，提供了云存储、文件同步、个性化云和客户端软件等功能。Dropbox 运行了数百个多语言微服务，它们之间每秒交换数百万个请求。最初，它使用了多个 RPC 框架，包括一个自建的 RPC 框架、Thrift 和一个遗留的 RPC 框架。其中，自建的 RPC 框架使用自定义的协议来实现手动序列化和反序列化，遗留的 RPC 框架则基于 HTTP/1.1，并使用了经过 protobuf 编码的消息。

最终，Dropbox 放弃了上述所有方案，转而选择 gRPC，这样也能够重用消息格式中一些现有的 protocol buffers 定义。Dropbox 创建了基于 gRPC 的 RPC 框架，即 Courier，这并不是新的 RPC 协议，而是集成 gRPC 与 Dropbox 现有基础设施的项目。Dropbox 对 gRPC 进行了扩充，来满足其在认证、授权、服务发现、服务统计、事件日志和跟踪工具等方面的特定需求。

从这些成功案例可以看到，作为一种进程间消息传递协议，gRPC 操作简单，能够大幅提高生产率和可靠性，并且可以针对互联网规模扩展和运维。这里只列举了一些知名的 gRPC 早期采用者，但 gRPC 的使用场景和采用者正在不断增多。

1.4 小结

现代软件应用程序或服务很少孤立地存在，连接它们的进程间通信技术是现代分布式软件应用程序最重要的方面之一。gRPC 是一个可扩展、松耦合、类型安全的解决方案，相对于基于 REST 或 HTTP 等协议的传统方式，gRPC 能够实现更高效的进程间通信。另外，它可以像调用本地方法那样，通过 HTTP/2 等网络传输协议，轻松地连接、调用、操作和调试分布式异构应用程序。

gRPC 也可以看作传统 RPC 的演化结果，它成功克服了传统方案的局限性。为了满足进程间通信的需求，许多互联网公司采用了 gRPC，其中最常见的就是将它用于构建组织内部的服务间通信。

掌握本章内容会为学习后续章节打下良好的基础，从而帮助你深入了解 gRPC 通信的各个方面。第 2 章会将本章介绍的内容付诸实践，从头开始构建真正的 gRPC 应用程序。

第 2 章　开始使用 gRPC

关于 gRPC 的理论知识已经介绍得差不多了，接下来根据第 1 章介绍的知识从头构建真正的 gRPC 应用程序。本章将分别使用 Go 和 Java 构建简单的 gRPC 服务以及调用该服务的客户端应用程序。在此过程中，我们将学习如何使用 protocol buffers 声明 gRPC 服务定义、生成服务器端骨架和客户端存根、实现服务的业务逻辑、在 gRPC 服务器上运行我们实现的服务并通过 gRPC 客户端应用程序调用该服务。

本章继续使用第 1 章的在线零售系统，并在这个系统中构建一个管理零售商店中所有商品的服务。该服务可以进行远程访问，服务的消费者能够向系统添加新的商品，并可以根据所提供的商品 ID 检索商品详情。我们将使用 gRPC 建模该服务及其消费者。你可以使用任意喜欢的语言来实现这些功能，但本章将使用 Go 和 Java 来实现。

可以通过本书的源代码仓库，尝试本示例的 Go 实现和 Java 实现。

图 2-1 展示了 `ProductInfo` 服务的客户端–服务器端对于每一种方法调用的通信模式。服务器端托管了一个 gRPC 服务，该服务提供了两个远程方法：`addProduct(Product)` 和 `getProduct(ProductID)`。客户端可以调用这两个方法中的任意一个。

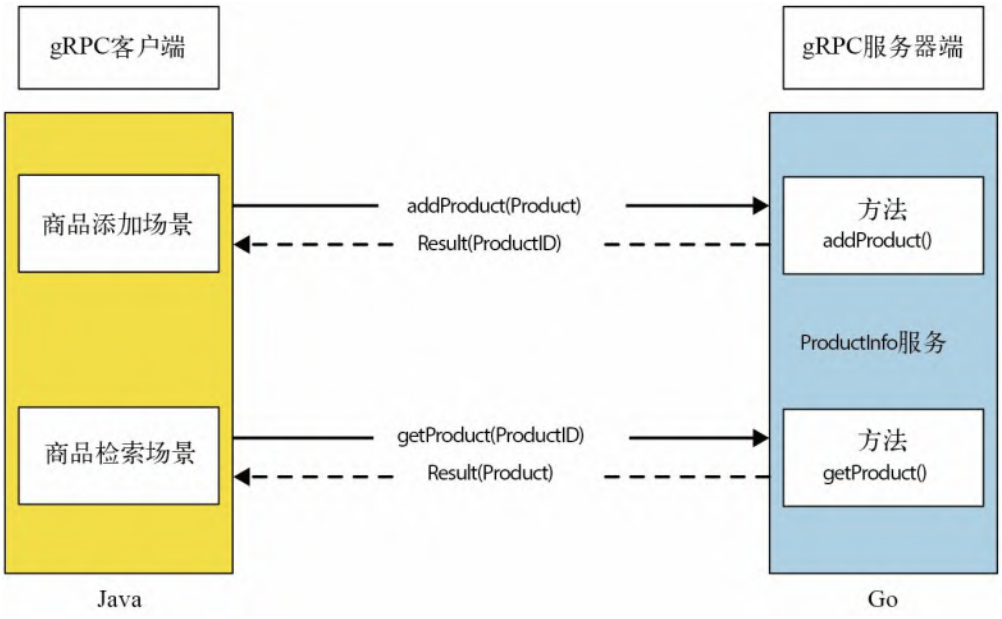

图 **2-1：ProductInfo** 服务的客户端–服务器端交互

接下来从创建 `ProductInfo` gRPC 服务的服务定义开始构建该示例。

2.1 创建服务定义

如第 1 章所述，在开发 gRPC 应用程序时，要先定义服务接口，其中包含允许远程调用的方法、方法参数以及调用这些方法所使用的消息格式等。这些服务定义都以 protocol buffers 定义的形式进行记录，也就是 gRPC 中所使用的接口定义语言。

第 3 章会进一步讨论针对不同消息模式的服务定义技术。另外，第 4 章会具体介绍 protocol buffers 和 gRPC 实现。

在确定了服务的业务功能之后，就可以定义服务接口来满足业务需要了。在本示例中，可以看到 `ProductInfo` 服务有两个远程方法，即 `addProduct(Product)` 和 `getProduct(ProductID)`，并且这两个方法都会接受或返回两个消息类型（`Product` 和 `ProductID`）。

接下来以 protocol buffers 定义的形式声明这些服务定义。protocol buffers 可以定义消息类型和服务类型，其中消息包含字段，每个字段由其类型和唯一索引值进行定义；服务则包含方法，每个方法由其类型、输入参数和输出参数进行定义。下面深入学习如何定义消息类型和服务类型。

2.1.1 定义消息类型

消息（message）是客户端和服务器端交换的数据结构。正如图 2-1 所示，`ProductInfo` 用例有两个消息类型：一个是商品信息（`Product`），用于在系统中添加新的商品，另外当检索特定的商品时，它也会用作返回值；另一个消息类型是商品的唯一标识（`ProductID`），用于从系统中检索特定商品，当添加新商品时，它也会用作返回值。

`ProductID`

`ProductID` 是商品的唯一标识，可以是字符串类型的值。我们也可以自定义包含字符串字段的消息类型，抑或使用 protocol buffers 库所提

供的较为流行的消息类型 google.protobuf.StringValue。本例将自定义包含字符串字段的消息类型。ProductID 消息类型的定义如代码清单 2-1 所示。

代码清单 **2-1**　ProductID 消息类型的 protocol buffers 定义

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

Product

Product 是自定义消息类型，代表在线零售应用程序中的商品应该具有的数据。它包含一组字段，这些字段代表每个商品所关联的数据。假设 Product 消息类型有如下字段。

id

商品的唯一标识符。

name

商品的名称。

description

商品的描述。

price

商品的价格。

这样就可以使用 protocol buffers 来自定义消息类型，如代码清单 2-2 所示。

代码清单 **2-2**　Product 消息类型的 protocol buffers 定义

```
message Product {
   string id = 1;
   string name = 2;
```

```
   string description = 3;
   float price = 4;
}
```

这里为每个消息字段所分配的数字用来在消息中标识该字段。因此，在同一个消息定义中，不能为两个字段设置相同的数字。第 4 章会进一步介绍 protocol buffers 消息定义技术，并阐述为每个字段提供唯一数字的原因。就现在来讲，可以将其视为定义 protocol buffers 消息的一个规则。

protocol buffers 库为众所周知的类型提供了 `protobuf` 消息类型的集合。因此，可以在服务定义中重复使用它们，而无须再次定义这样的类型。可以在 protocol buffers 文档中详细了解这些消息类型。

目前已完成了 `ProductInfo` 服务消息类型的定义，下面来看一下服务接口的定义。

2.1.2 定义服务类型

服务（service）是暴露给客户端的远程方法集合。在示例中，`ProductInfo` 服务有两个远程方法：`addProduct(Product)` 和 `getProduct(ProductID)`。按照 protocol buffers 的规则，远程方法只能有一个参数，并只能返回一个值。如果需要像 `addProduct` 方法那样给方法传递多个值，就要定义一个消息类型，并对所有的值进行分组，就像在 `Product` 消息类型中所做的那样。

`addProduct`

在系统中创建新的 `Product`。该方法需要商品的详细信息作为输入，并且如果操作成功完成，就会返回新创建商品的标识符数字。代码清单 2-3 展示了 `addProduct` 方法的定义。

代码清单 **2-3** `addProduct` 方法的 protocol buffers 定义

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

getProduct

检索商品信息。该方法需要 ProductID 作为输入，如果系统中存在相应商品，就会返回 Product 的详情。代码清单 2-4 展示了 getProduct 方法的定义。

代码清单 **2-4** getProduct 方法的 protocol buffers 定义

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

将消息和服务组合到一起，就有了 ProductInfo 用例的完整 protocol buffers 定义，如代码清单 2-5 所示。

代码清单 **2-5** ProductInfo 服务使用 protocol buffers 的 gRPC 服务定义

```
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 服务定义首先要指定所使用的 protocol buffers 版本（proto3）。

❷ 为了避免协议消息类型之间的命名冲突，这里使用了包名，它也会用于生成代码。

❸ 服务接口的定义。

❹ 用于添加商品的远程方法，它会返回商品 ID 作为响应。

❺ 基于商品 ID 获取商品的远程方法。

❻ `Product` 消息类型（格式）的定义。

❼ 用来保存商品 ID 的字段（名–值对），使用唯一的数字来标识二进制消息格式中的各个字段。

❽ `ProductID` 消息类型（格式）的定义。

在 protocol buffers 定义中，可以指定包名（如 ecommerce），这样做能够避免在不同的项目间出现命名冲突。当使用这个包属性生成服务或客户端代码时，除非明确指明了不同的包名，否则将为对应的编程语言生成相同的包。当然，该语言需要支持包的概念，本例代码就是用它来编写的。在定义包名的时候，还可以使用版本号，如 ecommerce.v1 和 ecommerce.v2。这样一来，未来对 API 的主要变更就可以在相同的代码库中共存。

对于 IntelliJ IDEA、Eclipse、VSCode 等常用的集成开发环境，现在有支持 protocol buffers 的插件。可以将插件安装到集成开发环境中，这样就能很容易地为服务创建 protocol buffers 定义了。

还有一个过程需要注意，那就是从其他 proto 文件中进行导入。如果需要使用其他 proto 文件中定义的消息类型，那么可以将它们导入本例的 protocol buffers 定义中。如果要使用 wrappers.proto 文件中的 `StringValue` 类型（`google.protobuf.StringValue`），就可以按照如下方式在定义中导入 google/protobuf/wrappers.proto 文件：

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

package ecommerce;
...
```

在完成服务定义的规范之后，就可以处理 gRPC 服务和客户端的实现了。

2.2 实现

接下来要实现该 gRPC 服务，它包含了我们在服务定义中所声明的远程方法。这些方法会通过服务器端暴露出来，gRPC 客户端会连接到服务器端，并调用这些远程方法。

如图 2-2 所示，首先需要编译 `ProductInfo` 服务定义，然后生成选定语言的源代码。gRPC 为所有流行语言都提供了便于使用的支持，如 Java、Go、Python、Ruby、C、C++、Node 等。在实现服务和客户端时，可以任意选择要使用的语言。gRPC 还能跨语言和跨平台运行，这意味着在应用程序中，可以使用某种语言来编写服务器端，而使用另外一种语言来编写客户端。这里将同时使用 Go 语言和 Java 语言来编写客户端和服务器端，你可以任意选择喜欢的实现。

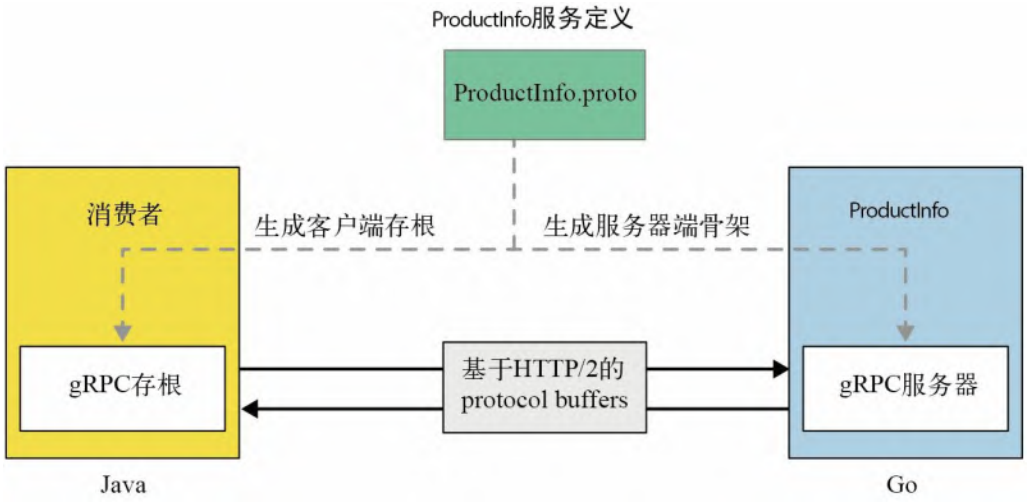

图 **2-2**：基于服务定义的微服务和消费者

为了根据服务定义生成源代码，可以使用 protocol buffers 编译器来手动编译 proto 文件，也可以使用像 Bazel、Maven 或 Gradle 这样的自动化构建工具。在构建项目时，这些自动化构建工具已有一组预先定义的代码生成规则，但对生成 gRPC 服务和客户端的源代码来说，与现有的构建工具集成通常会更为容易。

本章示例将使用 Gradle 来构建 Java 应用程序，并使用 Gradle protocol buffers 插件来生成服务和客户端代码，同时使用 protocol buffers 编译器

来生成 Go 应用程序的代码。

接下来将使用 Go 语言和 Java 语言生成 gRPC 服务器端和客户端。在此之前，必须确保本地计算机已安装了 Java 7 或更高版本以及 Go 1.11 或更高版本。

2.2.1 开发服务

在生成服务器端骨架时，将得到建立 gRPC 连接、相关消息类型和接口的基础代码。实现服务的任务就是实现代码生成阶段所得到的接口。下面首先从实现 Go 服务开始，然后再看一下如何使用 Java 语言实现相同的服务。

01. 使用 **Go** 语言实现 **gRPC** 服务

    实现 Go 服务分为 3 步：首先，生成服务定义的存根文件；其次，实现该服务中远程方法的业务逻辑；最后，创建服务器，监听特定的端口并注册该服务，从而接受来自客户端的请求。我们从创建一个新的 Go 模块开始，这里会创建一个新的模块以及该模块中的子目录。`productinfo/service` 模块用来存放服务代码，子目录（ecommerce）用来保存自动生成的存根文件。然后，在 productinfo 目录下创建名为 service 的子目录。进入 service 子目录并执行如下命令来创建 `productinfo/service` 模块：

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

    创建完模块并在模块内创建完子目录之后，将得到如下所示的模块结构：

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

    我们还需要更新 go.mod 文件的依赖项，具体的版本如下所示。

```
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
)
```

Go 1.11 引入了名为模块（module）的新概念，可以让开发人员在 GOPATH 之外构建和运行 Go 项目。要创建 Go 模块，需要在 $GOPATH/src 之外的任意地方创建新目录，然后进入该目录，使用模块名来执行如下命令，从而初始化模块：

```
go mod int <module_name>
```

当模块完成初始化之后，模块的根目录会创建 go.mod 文件。接下来就可以在模块中创建 Go 源代码文件并进行构建。Go 语言将使用 go.mod 文件中所列的特定依赖模块的版本来解析导入项。

生成客户端存根或服务器端骨架。现在，使用 protocol buffers 编译器来手动生成客户端存根或服务器端骨架。为了实现这一点，需要满足下列先决条件。

- 从 GitHub 的发布页面下载并安装最新的 protocol buffers 编译器（版本 3）。

在下载编译器时，选择适合所用平台的编译器。假设正在使用 64 位 Linux 机器，同时需要获得版本为 *x.x.x* 的 protocol buffers 编译器，就需要下载 protoc-x.x.x-linux-x86_64.zip 文件。

- 使用如下命令安装 gRPC 库。

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

- 使用如下命令安装 Go 语言的 protoc 插件。

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

当满足这些先决条件之后，就可以通过执行如下所示的 protoc 命令为服务定义生成代码了：

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

❶ 指定源 proto 文件和依赖的 proto 文件的目录路径（通过 --proto_path 或 -I 命令行标记来指定）。如果不指定该值，则将使用当前目录作为源目录。在这个目录下，需要根据包名来存放依赖的 proto 文件。

❷ 指定希望编译的 proto 文件路径。编译器将阅读该文件并生成输出的 Go 文件。

❸ 指定生成的代码要存放的目标目录。

当执行该命令时，在模块的给定子目录下（ecommerce）会生成一个存根文件（product_info.pb.go）。获得这个存根文件后，需要使用生成的代码来实现业务逻辑。

实现业务逻辑。首先需要在 Go 模块（productinfo/service）中创建名为 productinfo_service.go 的 Go 文件，然后实现如代码清单 2-6 所示的远程方法。

代码清单 **2-6**　使用 Go 语言编写的 ProductInfo 服务的 gRPC 服务实现

```
package main

import (
  "context"
  "errors"
  "log"

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

```
)

// 用来实现ecommerce/product_info的服务器
type server struct{ ❷
  productMap map[string]*pb.Product
}

// 实现ecommerce.AddProduct的AddProduct方法
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()

}

// 实现ecommerce.GetProduct的GetProduct方法
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.",

}
```

❶ 导入刚刚通过 protobuf 编译器所生成的代码所在的包。

❷ server 结构体是对服务器的抽象。可以通过它将服务方法附加到服务器上。

❸ AddProduct 方法以 Product 作为参数并返回一个 ProductID。Product 和 ProductID 结构体定义在 product_info.pb.go 文件中，该文件是通过 product_info.proto 定义自动生成的。

❹ GetProduct 方法以 ProductID 作为参数并返回 Product。

❺ 这两个方法都有一个 Context 参数。Context 对象包含一些元数据，比如终端用户授权令牌的标识和请求的截止时间。这些元数据会在请求的生命周期内一直存在。

❻ 这两个方法都会返回一个错误以及远程方法的返回值（方法有多种返回类型）。这些错误会传播给消费者，用来进行消费者端的错误处理。

这样就实现了 ProductInfo 服务的业务逻辑。接下来可以创建简单的服务器，来托管该服务并接受来自客户端的请求。

创建 **Go** 服务器。要用 Go 语言创建服务器，需要在相同的 Go 包（productinfo/service）中创建名为 main.go 的新 Go 文件，并实现如代码清单 2-7 所示的 main 方法。

代码清单 **2-7**　使用 Go 语言编写的托管 ProductInfo 服务的 gRPC 服务器实现

```
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 导入通过 protobuf 编译器所生成的代码所在的包。

❷ 希望由 gRPC 服务器所绑定的 TCP 监听器在给定的端口（50051）上创建。

❸ 通过调用 gRPC Go API 创建新的 gRPC 服务器实例。

❹ 通过调用生成的 API，将之前生成的服务注册到新创建的 gRPC 服务器上。

❺ 在指定端口（50051）上开始监听传入的消息。

现在，我们已通过 Go 语言为业务场景构建了 gRPC 服务。同时，我们创建了简单的服务器，该服务器将暴露服务方法，并接收来自 gRPC 客户端的消息。

如果你更喜欢使用 Java 语言，那么也可以使用该语言来构建相同的服务。该实现过程与 Go 语言的非常类似，接下来将使用 Java 语言再次构建服务。如果你对如何使用 Go 语言构建客户端更感兴趣，那么可以直接阅读 2.2.2 节。

02. 使用 **Java** 语言实现 **gRPC** 服务

在创建 Java gRPC 项目时，最佳的实现方式是使用现有的构建工具，如 Gradle、Maven 或 Bazel，它们能够管理所有的依赖项和代码生成功能。在示例中，我们使用 Gradle 来管理项目，同时讨论如何使用 Gradle 来创建 Java 项目，以及如何实现服务中所有远程方法的业务逻辑。最后，创建服务器并注册服务，从而接受来自客户端的请求。

 Gradle 是一个自动化构建工具，它支持多种语言，包括 Java、Scala、Android、C、C++ 和 Groovy，并且与 Eclipse 和

IntelliJ IDEA 等开发工具紧密集成。可以按照其官网页面给出的步骤在自己的机器上安装 Gradle。

搭建 **Java** 项目。首先来创建一个 Gradle Java 项目（`product-info-service`）。在创建完项目之后，会得到下面这样的项目结构：

```
product-info-service

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

在 src/main 目录下，创建 proto 目录，并将 `ProductInfo` 服务定义文件（.proto 文件）放到 proto 目录下。

然后，需要更新 build.gradle 文件，为 Gradle 添加依赖项和 protobuf 插件。更新后的 build.gradle 文件内容如代码清单 2-8 所示。

代码清单 **2-8**　适用于 gRPC Java 项目的 Gradle 配置

```
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 : zipTre
   }
}

apply plugin: 'application'
```

```
startScripts.enabled = false
```

❶ Gradle 项目所使用的 gRPC Java 库的版本。

❷ 该项目所使用的外部依赖项。

❸ 该项目所使用的 Gradle protobuf 插件的版本。如果 Gradle 版本低于 2.12，那么需要使用 0.7.5 版本的插件。

❹ 在 protobuf 插件中，需要指定 protobuf 编译器的版本和 protobuf Java 可执行包的版本。

❺ 这会告知 IntelliJ IDEA、Eclipse 或 NetBeans 等集成开发环境有关生成代码的信息。

❻ 配置运行应用程序所使用的主类。然后，运行下面的命令来构建库并根据 protobuf 构建插件来生成存根代码：

```
$ ./gradle build
```

现在 Java 项目已包含生成的代码。接下来实现服务接口并为远程方法添加业务逻辑。

实现业务逻辑。首先，在 src/main/java 源代码目录下创建 Java 包（ecommerce），并在包中创建 Java 类（`ProductInfoImpl.java`）。然后，实现如代码清单 2-9 所示的远程方法。

代码清单 **2-9**　使用 Java 语言编写的 `ProductInfo` 服务的 gRPC 服务实现

```
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.ProductInfoImplBa

   private Map productMap = new HashMap<String, ProductInfoOuterClass.

   @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_FOU
       }
  }
}
```

❶ 扩展插件生成的抽象类（ProductInfoGrpc.ProductInfoImplBase）。这样一来，就能够为服务定义文件中所定义的 addProduct 方法和 getProduct 方法添加业务逻辑了。

❷ addProduct 方法接受 Product 类（ProductInfoOuterClass.Product）作为参数。Product 类从服务定义中生成，并在 ProductInfoOuterClass 类中进行定义。

❸ getProduct 方法接受 ProductID 类

（ProductInfoOuterClass.ProductID）作为参数。ProductID 类从服务定义中生成，并在 ProductInfoOuterClass 类中进行定义。

❹ responseObserver 对象用来发送响应给客户端并关闭流。

❺ 发送响应给客户端。

❻ 通过关闭流终结客户端调用。

❼ 发送错误给客户端。

至此，我们就使用 Java 实现了 ProductInfo 服务的业务逻辑。接下来创建一台简单的服务器，使用它来托管服务并接受来自客户端的请求。

创建 **Java** 服务器。为了将服务暴露出去，我们需要创建一个 gRPC 服务器实例，并将 ProductInfo 服务注册到该服务器上。该服务器将监听指定端口，并将所有的请求分派给相关的服务。这里需要在包中创建一个主类（ProductInfoServer.java），如代码清单 2-10 所示。

代码清单 **2-10**　使用 Java 语言编写的托管 ProductInfo 服务的 gRPC 服务器实现

```
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 服务器实例在端口 50051 创建。我们希望服务器绑定到该端口，并监听传入的消息。另外，在该服务器上添加了 `ProductInfo` 服务实现。

❷ 添加运行时的关闭 hook，这会在 JVM 关闭时关闭 gRPC 服务器。

❸ 在方法的最后，服务器的线程会一直保持，直到服务器终止。

现在，我们已经用两种语言实现了 gRPC 服务，接下来可以开始实现 gRPC 客户端了。

2.2.2 开发**gRPC**客户端

gRPC 服务实现已准备就绪，下面讨论如何创建应用程序来与该服务器对话。这里先从根据服务定义生成客户端存根开始。基于生成的客户端存根，可以创建简单的 gRPC 客户端，使其连接前面创建的 gRPC 服务器，并调用服务器所提供的远程方法。

本示例将同时使用 Java 和 Go 这两种语言来编写客户端应用程序。但是，这并不代表在实际创建服务器和客户端时，必须使用相同的语言，也不代表它们必须在相同的平台上才能运行。由于 gRPC 能够跨语言和跨平台运行，因此你可以使用 gRPC 所支持的任意语言来创建它们。下面先来看 Go 语言的实现，如果你对 Java 实现更感兴趣，可以跳到第 2 小节，直接学习 Java 客户端。

01. 实现 **gRPC** 的 **Go** 客户端

首先创建新的 Go 模块（productinfo/client），并在该模块中创建子目录（ecommerce）。然后，为了实现 Go 客户端应用程序，还需要像实现 Go 服务那样生成存根文件。2.2.1 节介绍了如何生成存根文件，因为遵循相同的步骤来创建相同的文件（product_info.pb.go），所以这里不再赘述。

在 Go 模块中（productinfo/client），创建名为 productinfo_client.go 的新 Go 文件，并实现调用远程方法的主方法，如代码清单 2-11 所示。

代码清单 **2-11**　使用 Go 语言编写的 gRPC 客户端应用程序

```
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 wit
              Ultra Wide and Night mode.`
  price := float32(1000.0)
  ctx, cancel := context.WithTimeout(context.Background(), time.Second
  defer cancel()
  r, err := c.AddProduct(ctx,
```

```
        &pb.Product{Name: name, Description: description, Price: pric
  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 导入 protobuf 编译器生成代码所在的包。

❷ 根据提供的地址（`localhost: 50051`）创建到服务器端的连接。这里创建了一个客户端和服务器端之间的连接，但它目前不安全。

❸ 传递连接并创建存根文件。这个实例包含可调用服务器的所有远程方法。

❹ 创建 `Context` 以传递给远程调用。这里的 `Context` 对象包含一些元数据，如终端用户的标识、授权令牌以及请求的截止时间，该对象会在请求的生命周期内一直存在。

❺ 使用商品的详情信息调用 `AddProduct` 方法。如果操作成功完成，就会返回一个商品 ID，否则将返回一个错误。

❻ 使用商品 ID 来调用 `GetProduct` 方法。如果操作成功完成，将返回商品详情，否则会返回一个错误。

❼ 所有事情都完成后，关闭连接。

现在，我们已使用 Go 语言构建了 gRPC 客户端，接下来使用 Java 语言来创建客户端。这里只是用不同的方式实现相同的目标，也就是说，如果你对使用 Java 构建 gRPC 感兴趣，那么可以继续往下阅读，否则可以跳过这一部分，直接阅读 2.3 节。

02. 实现 **gRPC** 的 **Java** 客户端

为了创建 Java 客户端应用程序，需要搭建一个 Gradle 项目（`product-info-client`），并且要像实现 Java 服务那样使用 Gradle 插件来生成类。请按照 2.2.1 节中的步骤搭建 Java 客户端项目。

在通过 Gradle 构建工具生成项目的客户端存根代码之后，接下来在 ecommerce 包中创建名为 `ProductInfoClient` 的新类，并添加代码清单 2-12 所示的内容。

代码清单 **2-12**　使用 Java 语言编写的 gRPC 客户端应用程序

```
package ecommerce;

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

import java.util.logging.Logger;

/**
* ProductInfo服务的gRPC客户端示例
*/
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(productI
      System.out.println(product.toString());
      channel.shutdown(); ❺
```

```
  }
}
```

❶ 创建 gRPC 通道并指定希望连接的服务器地址和端口。这里希望连接在本地机器上运行并监听端口 50051 的服务器。同时启用了明文（plaintext），这意味着在客户端和服务器端之间建立的连接不安全。

❷ 使用新建的通道来创建客户端存根代码，其中包括两种类型：一种是 `BlockingStub`，它会一直等待，直到接收到服务器的响应为止；另一种是 `NonBlockingStub`，它不会等待服务器的响应，而会注册一个观察者（observer）来接收响应。本例使用的是`BlockingStub`，这使客户端更加简单。

❸ 使用商品详情调用 `addProduct` 方法。如果操作成功完成，会返回商品 ID。

❹ 使用商品 ID 调用 `getProduct` 方法。如果操作成功完成，会返回商品详情。

❺ 在所有的事情都完成后，关闭连接。这样一来，当应用程序所使用的网络资源用完后，就能够安全回收。

这样就完成了 gRPC 客户端的开发。接下来实现客户端和服务器之间的对话。

2.3 构建和运行

现在该运行刚刚创建的 gRPC 服务器端应用程序和客户端应用程序了。可以在本地机器、虚拟机、Docker 或 Kubernetes 上部署并运行 gRPC 应用程序，本节将讨论如何在本地机器上构建和运行 gRPC 服务器端应用程序和客户端应用程序。

第 7 章介绍如何在 Docker 和 Kubernetes 上部署和运行 gRPC 应用程序。

接下来在本地机器上运行刚才构建的 gRPC 服务器端应用程序和客户端应用程序。因为服务器端应用程序和客户端应用程序是使用两种语言编写的，所以这里会对应不同语言分别构建。

2.3.1 构建Go服务器端应用程序

在实现 Go 服务时，工作空间中最终的包结构如下所示：

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

可以通过构建服务来生成服务的二进制文件（bin/server）。要进行构建，首先进入 Go 模块根目录（productinfo/service），然后执行如下的 shell 命令：

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

当构建成功之后，就会在 bin 目录下创建一个可执行文件（bin/server）。

接下来构建 Go 客户端应用程序。

2.3.2 构建Go客户端应用程序

在实现 Go 客户端应用程序时，工作空间中的包结构如下所示：

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

可以按照与构建 Go 服务同样的 shell 命令来构建客户端代码：

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

当构建成功后，就会在 bin 目录下创建一个可执行文件（bin/client）。下一步就是执行这些文件了。

2.3.3 运行Go服务器端应用程序和客户端应用程序

我们已经构建了服务器端应用程序和客户端应用程序。接下来在不同的终端上运行它们，并使其进行对话：

```
// 运行服务器端
$ bin/server
2019/08/08 10:17:58 Starting gRPC listener on port :50051

// 运行客户端
$ 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
```

接下来构建 Java 服务器端应用程序。

2.3.4 构建Java服务器端应用程序

因为我们是以 Gradle 项目的形式来实现 Java 服务的，所以可以很容易地使用如下命令来构建项目：

```
$ gradle build
```

当构建成功之后，就会在 build/libs 目录下创建一个可执行文件（server.jar）。

2.3.5 构建Java客户端应用程序

与服务类似，我们可以使用如下命令很容易地构建项目：

```
$ gradle build
```

当构建成功之后，就会在 build/libs 目录下创建一个可执行文件（client.jar）。

2.3.6 运行Java服务器端应用程序和客户端应用程序

现在已经构建了基于 Java 语言的服务器端应用程序和客户端应用程序。接下来运行这些应用程序：

```
$ 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
```

现在，我们已经在本地机器上成功构建并运行了示例。在成功运行服务器端应用程序和客户端应用程序后，客户端应用程序首先会使用商品详情调用 `addProduct` 方法，并且会接收到新创建商品的标识符作为响应，然后使用商品标识符调用 `getProduct` 方法，来检索新创建的商品详情。本章在前面提到，要实现服务器端应用程序与客户端应用程序的通信，无须使用相同的语言。也就是说，我们完全可以畅通无阻地运行

gRPC Java 服务器端应用程序和 Go 客户端应用程序。

2.4 小结

在开发 gRPC 应用程序时，首先要使用 protocol buffers 定义服务接口，protocol buffers 是语言中立、平台无关、可扩展的结构化数据序列化机制。然后，为选择的编程语言生成服务器端代码和客户端代码，这种方式提供了较低层级通信细节的抽象，从而简化了服务器端和客户端的逻辑。也就是说，在服务器端，需要为远程暴露的方法实现逻辑，并让 gRPC 服务器在运行的时候绑定该服务。在客户端，需要连接 gRPC 服务器，并使用生成的客户端代码调用远程方法。

本章旨在帮助你获得开发和运行 gRPC 服务器端应用程序和客户端应用程序的实际经验，从而构建真正的 gRPC 应用程序。不管你使用哪种语言，构建 gRPC 应用程序都遵循类似的步骤。因此，第 3 章将进一步扩展你所学到的概念和技术，来构建真实的用例。

第 3 章　gRPC 的通信模式

第 1 章和第 2 章介绍了 gRPC 进程间通信技术的基础知识，其中还涉及构建简单的 gRPC 应用程序。到目前为止，我们已经完成了定义服务接口、实现服务、运行 gRPC 服务器以及通过 gRPC 客户端应用程序远程调用服务等操作。客户端和服务器端之间的通信模式是简单的请求–响应风格的通信，这里每个请求都会得到一个响应。但是，借助 gRPC，可以实现不同的进程间通信模式（也称 RPC 风格），而不仅仅是简单的请求–响应模式。

本章将讨论 gRPC 应用程序的 4 种基础通信模式：一元 RPC、服务器端流 RPC、客户端流 RPC 以及双向流 RPC。在这个过程中，我们会使用一些真实用例来展示每种模式，使用 gRPC IDL 进行服务定义，并使用 Go 语言来实现服务和客户端。

用 **Go** 和 **Java** 编写的代码示例

为了保持一致性，本章的所有代码示例都是使用 Go 语言编写的。但是，如果你是 Java 开发人员，也能在本书的源代码仓库中找到相同用例的完整 Java 代码示例。

3.1 一元RPC模式

我们从最简单的 RPC 风格开始讨论 gRPC 通信模式。一元 RPC 模式也被称为简单 RPC 模式。在该模式中，当客户端调用服务器端的远程方法时，客户端发送请求至服务器端并获得一个响应，与响应一起发送的还有状态细节以及 trailer 元数据。事实上，这也是第 1 章和第 2 章所介绍的通信模式。接下来看一个真实的用例，来进一步了解一元 RPC 模式。

假设需要为基于 gRPC 的在线零售应用程序构建 `OrderManagement` 服务，并在该服务中实现 `getOrder` 方法。借助该方法，客户端可以通过订单 ID 检索已有的订单。如图 3-1 所示，客户端发送一个带有订单 ID 的请求，服务器端给出响应，响应中包含订单的信息。因此，它遵循一元 RPC 模式。

图 **3-1**：一元 **RPC** 模式

下面来实现这种模式。第一步就是为 `OrderManagement` 服务及其 `getOrder` 方法创建服务定义。如代码清单 3-1 所示，可以使用 protocol buffers 进行服务定义，`getOrder` 远程方法接受一个订单 ID 的请求，并且会给出一个包含 `Order` 消息的响应。在本用例中，`Order` 消息具有描述订单所需的结构。

代码清单 **3-1** `OrderManagement` 服务定义，服务中的 `getOrder` 方法遵循一元 RPC 模式

```
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 导入这个包，从而使用常见的类型，如 StringValue。

❷ 检索订单的远程方法。

❸ 定义 Order 类型。

❹ 使用 repeated 表明这个字段在消息中可以重复出现任意次，包括 0 次。在这里，一条订单消息可以有任意数量的条目。

然后，借助 gRPC 服务定义的 proto 文件，就可以生成服务器端骨架代码并实现 GetOrder 方法的逻辑了。代码清单 3-2 展示了 OrderManagement 服务的 Go 实现。作为 GetOrder 方法的输入，单个订单 ID（String）用来组成请求，这样做可以很容易地在服务器端找到订单并以 Order 消息（Order 结构体）的形式进行响应。Order 消息可以和 nil 错误一起返回，从而告诉 gRPC，我们已经处理完 RPC，可以将 Order 返回到客户端了。

代码清单 **3-2**　使用 Go 语言编写的 OrderManagement 服务的 GetOrder 方法实现

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

```
}
```

第 4 章将介绍关于 gRPC 服务器端和客户端完整消息流的更多细节。除了在服务定义中为 GetOrder 方法所指定的参数，可以看到，在 OrderManagement 服务的 Go 实现中，还有一个 Context 参数被传递到了方法中。Context 包含一些用于控制 gRPC 行为的构造，比如截止时间和取消功能。第 5 章会详细讨论这些概念。

现在来实现客户端的逻辑，从而远程调用 GetOrder 方法。与服务器端的实现一样，可以为自己喜欢的语言生成代码来创建客户端存根，然后使用该存根调用服务，代码清单 3-3 使用 Go gRPC 客户端调用 OrderManagement 服务。当然，首先要创建到服务器端的连接并初始化调用服务的客户端存根。然后，就可以调用客户端存根的 GetOrder 方法，从而实现对远程方法的调用。这时会得到一个 Order 消息作为响应，其中包含服务定义中使用 protocol buffers 所定义的订单信息。

代码清单 **3-3**　使用 Go 语言调用远程 GetOrder 方法的客户端实现

```
// 建立到服务器端的连接.
...
orderMgtClient := pb.NewOrderManagementClient(conn)
...

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

这种一元 RPC 模式非常容易实现，适用于大多数进程间通信用例。在多种语言间，实现方式都是非常类似的，本书的示例代码仓库提供了 Go 和 Java 的源代码。

现在，我们已经对一元 RPC 模式有了大致的了解，接下来看一下服务器端流 RPC 模式。

3.2 服务器端流RPC模式

在一元 RPC 模式中，gRPC 服务器端和 gRPC 客户端在通信时始终只有一个请求和一个响应。在服务器端流 RPC 模式中，服务器端在接收到客户端的请求消息后，会发回一个响应的序列。这种多个响应所组成的序列也被称为“流”。在将所有的服务器端响应发送完毕之后，服务器端会以 trailer 元数据的形式将其状态发送给客户端，从而标记流的结束。

下面通过一个真实的用例来进一步了解服务器端流。在 `OrderManagement` 服务中，假设需要实现一个订单搜索功能，利用该功能，只要提供一个搜索词就能得到匹配的结果，如图 3-2 所示。`OrderManagement` 服务不会将所有匹配的订单一次性地发送给客户端，而是在找到匹配的订单时，逐步将其发送出去。这意味着当订单服务的客户端发出一个请求之后，会接收到多条响应消息。

图 **3-2**：服务器端流 **RPC** 模式

现在，在 `OrderManagement` 服务的 gRPC 服务定义中新增 `searchOrders` 方法。如代码清单 3-4 所示，`searchOrders` 方法定义与代码清单 3-1 中的 `getOrder` 方法非常类似，但是在服务定义的 proto 文件中，我们通过使用 `returns (stream Order)` 将返回参数指定为订单的流。

代码清单 **3-4** 使用服务器端流 RPC 模式的服务定义

```
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 通过返回 Order 消息的 stream 定义服务器端流。

通过服务定义，可以生成服务器端的代码，然后通过实现所生成的接口，就可以为 OrderManagement 服务的 searchOrders 方法构建逻辑了。在代码清单 3-5 所示的 Go 实现中，SearchOrders 方法有两个参数，分别是字符串类型的 searchQuery 和用来写入响应的特殊参数 OrderManagement_SearchOrdersServer。OrderManagement_Searc 是流的引用对象，可以写入多个响应。这里的业务逻辑是找到匹配的订单，并通过流将其依次发送出去。当找到新的订单时，使用流引用对象的 Send(...) 方法将其写入流。一旦所有响应都写到了流中，就可以通过返回 nil 来标记流已经结束，服务器端的状态和其他 trailer 元数据会发送给客户端。

代码清单 **3-5**　使用 Go 语言编写的 SearchOrders 方法的 OrderManagement 服务实现

```
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
                                // 在流中发送匹配的订单
```

```
                                err := stream.Send(&order) ❷
                                if err != nil {
                                   return fmt.Errorf(
                                            "error sending message to strea
                                                    err) ❸
                                }
                                log.Print("Matching Order Found : " + key)
                                break
                        }
                }
        }
        return nil
}
```

❶ 查找匹配的订单。

❷ 通过流发送匹配的订单。

❸ 检查在将消息以流的形式发送给客户端的过程中可能出现的错误。

客户端的远程方法调用和一元 RPC 模式中的非常类似。但是，因为服务器端往流中写入了多个响应，所以这里必须处理多个响应。因此，我们在 gRPC 客户端的 Go 语言实现中使用 Recv 方法从客户端流中检索消息，并且持续检索，直到流结束为止，如代码清单 3-6 所示。

代码清单 **3-6**　使用 Go 语言编写的 SearchOrders 方法的 OrderManagement 客户端实现

```
// 建立到服务器端的连接
...
        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
                }
        // 处理可能出现的错误
                log.Print("Search Result : ", searchOrder)
     }
```

❶ SearchOrders 方法返回 OrderManagement_SearchOrdersClient 的客户端流，它有一个名为 Recv 的方法。

❷ 调用客户端流的 Recv 方法，逐个检索 Order 响应。

❸ 当发现流结束的时候，Recv 会返回 io.EOF。

下面看一下客户端流 RPC 模式，它恰好与服务器端流 RPC 模式相反。

3.3 客户端流RPC模式

在客户端流 RPC 模式中，客户端会发送多个请求给服务器端，而不再是单个请求。服务器端则会发送一个响应给客户端。但是，服务器端不一定要等到从客户端接收到所有消息后才发送响应。基于这样的逻辑，我们可以在接收到流中的一条消息或几条消息之后就发送响应，也可以在读取完流中的所有消息之后再发送响应。

现在进一步扩展 `OrderManagement` 服务，从而更好地理解客户端流 RPC 模式。假设希望在 `OrderManagement` 服务中添加新的 `updateOrders` 方法，从而更新一个订单集合，如图 3-3 所示。在这里，我们想以消息流的形式发送订单列表到服务器端，服务器端会处理这个流并发送一条带有已更新订单状态的消息给客户端。

图 **3-3**：客户端流 **RPC** 模式

然后，可以将 `updateOrders` 方法添加到 `OrderManagement` 服务的服务定义文件中，如代码清单 3-7 所示。只需使用 `stream Order` 作为 `updateOrders` 方法的参数，就能表明 `updateOrders` 会接收来自客户端的多条消息作为输入。因为服务器端只发送一个响应，所以返回值是单一的字符串消息。

代码清单 **3-7**　具有客户端流 RPC 功能的服务定义

```
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;
}
```

当更新完服务定义文件之后，就可以生成服务器端和客户端的代码了。在服务器端，需要实现 OrderManagement 服务中所生成的 updateOrders 方法接口。在代码清单 3-8 所示的 Go 实现中，UpdateOrders 方法有一个 OrderManagement_UpdateOrdersServer 参数，它是客户端传入消息流的引用对象。因此，可以通过调用该对象的 Recv 方法来读取消息。根据业务逻辑，可以读取其中一些消息，也可以读取所有的消息。服务只需调用 OrderManagement_UpdateOrdersServer 对象的 SendAndClose 方法就可以发送响应，它同时也标记服务器端消息终结了流。如果要提前停止读取客户端流，那么服务器端应该取消客户端流，这样客户端就知道停止生成消息了。

代码清单 **3-8**　使用 Go 语言编写的 UpdateOrders 方法的 OrderManagement 服务实现

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

        ordersStr := "Updated Order IDs : "
        for {
                order, err := stream.Recv() ❶
                if err == io.EOF { ❷
                        // 完成读取订单流
                        return stream.SendAndClose(
                                &wrapper.StringValue{Value: "Orders process
                                + ordersStr})
                 }
```

```
                // 更新订单
                orderMap[order.Id] = *order

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

❶ 从客户端流中读取消息。

❷ 检查流是否已经结束。

下面来看这个客户端流用例的客户端实现。如代码清单 3-9 中的 Go 实现所示，客户端可以通过客户端流引用，借助 updateStream.Send 方法发送多条消息。一旦所有消息都以流的形式发送出去，客户端就可以将流标记为已完成，并接收来自服务器端的响应。这是通过流引用的 CloseAndRecv 方法实现的。

代码清单 **3-9**　使用 Go 语言编写的 UpdateOrders 方法的 OrderManagement 客户端实现

```
// 建立到服务器端的连接
...
        c := pb.NewOrderManagementClient(conn)
...
updateStream, err := client.UpdateOrders(ctx) ❶

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

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

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

   // 更新订单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 调用 `UpdateOrders` 远程方法。

❷ 处理与 `UpdateOrders` 相关的错误。

❸ 通过客户端流发送订单更新的请求。

❹ 处理在发送消息到流时发生的错误。

❺ 关闭流并接收响应。

当调用这个方法后，会收到服务的响应消息。现在，我们对服务器端流 RPC 模式和客户端流 RPC 模式都有了非常好的了解。接下来将介绍双向流 RPC 模式，它是前面讨论的不同 RPC 风格的一种组合。

3.4 双向流RPC模式

在双向流 RPC 模式中，客户端以消息流的形式发送请求到服务器端，服务器端也以消息流的形式进行响应。调用必须由客户端发起，但在此之后，通信完全基于 gRPC 客户端和服务器端的应用程序逻辑。下面通过一个示例来进一步了解双向流 RPC 模式。如图 3-4 所示，在 `OrderManagement` 服务用例中，假设需要一个订单处理功能，通过该功能，用户可以发送连续的订单集合（订单流），并根据投递地址对它们进行组合发货，也就是说，订单要根据投递目的地进行组织和发货。

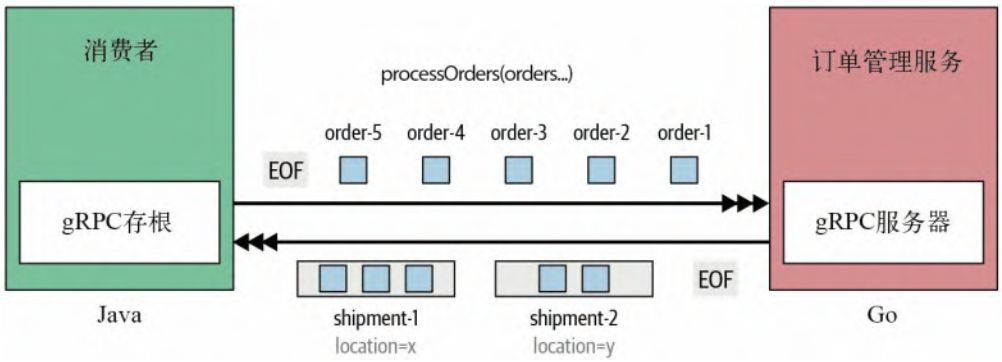

图 **3-4**：双向流 **RPC** 模式

可以看到，这个业务用例的关键步骤如下所示。

- 客户端应用程序通过建立与服务器端的连接并发送调用元数据（头信息）初始化业务用例。
- 一旦连接成功建立，客户端应用程序就发送连续的订单 ID 集合，这些订单需要由 `OrderManagement` 服务进行处理。
- 每个订单 ID 以独立的 gRPC 消息的形式发送至服务器端。
- 服务会处理给定订单 ID 所对应的每个订单，并根据订单的投递位置将它们组织到发货组合中。
- 每个发货组合可能会包含多个订单，它们应该被投递到相同的目的地。
- 订单是成批处理的。当达到指定的批次大小时，当前创建的所有发货组合都会被发送至客户端。
- 假设流中有 4 个订单，其中有 2 个订单要发送至位置 `X`，另外两个要发送至位置 `Y`，则可以将其表示为 `X`、`Y`、`X`、`Y`。如果批次大小

为 3，那么所创建的订单发货组合会是 [X, X]、[Y] 和 [Y]。这些发货组合也会以流的形式发送至客户端。

这个业务用例的核心理念就是一旦调用 RPC 方法，那么无论是客户端还是服务器端，都可以在任意时间发送消息。这也包括来自任意一端的流结束标记。

下面看一下上述用例的服务定义。如代码清单 3-10 所示，可以定义一个 processOrders 方法，该方法接受一个字符串流作为方法参数，代表了订单流 ID 并且以 CombinedShipment 流作为方法的返回值。因此，通过将方法参数和返回参数均声明为 stream，可以定义双向流的 RPC 方法。发货组合的消息也是通过服务定义声明的，它包含了订单元素的列表。

代码清单 **3-10**　具有双向流 RPC 功能的服务定义

```
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 在双向流 RPC 模式中，将方法参数和返回参数均声明为 stream。

❷ Order 消息的结构。

❸ CombinedShipment 消息的结构。

接下来，就可以根据更新后的服务定义生成服务器端的代码了。服务应该实现 OrderManagement 服务中的 processOrders 方法。如代码清单 3-11 所示，在 Go 实现中，ProcessOrders 方法有一个 OrderManagement_ProcessOrdersServer 参数，它是客户端和服务器端之间消息流的对象引用。借助这个流对象，服务器端可以读取客户端以流的方式发送的消息，也能写入服务器端的流消息并返回给客户端。传入的消息流可以通过该引用对象的 Recv 方法来读取。在 ProcessOrders 方法中，服务可在持续读取传入消息流的同时，使用 Send 方法将消息写入同一个流中。


为了便于演示，代码清单 3-11 没有展示完整的逻辑。不过，可以通过本书的源代码仓库找到完整的代码示例。

代码清单 **3-11** 使用 Go 语言编写的 ProcessOrders 方法的 OrderManagement 服务实现

```
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
                }

                // 基于目的地位置，
                // 将订单组织到发货组合中的逻辑
                ...
```

```
                //

                if batchMarker == orderBatchSize { ❺
                        // 将组合后的订单以流的形式分批发送至客户端
                        for _, comb := range combinedShipmentMap {
                                // 将发货组合发送到客户端
                                stream.Send(&comb)      ❻
                        }
                        batchMarker = 0
                        combinedShipmentMap = make(
                                map[string]pb.CombinedShipment)
                } else {
                        batchMarker++
                }
        }
}
```

❶ 从传入的流中读取订单 ID。

❷ 持续读取，直到流结束为止。

❸ 当流结束时，将所有剩余的发货组合发送给客户端。

❹ 通过返回 `nil` 标记服务器端流已经结束。

❺ 按批次处理订单。当达到该批次的规模时，将所有已创建的发货组合以流的形式发送给客户端。

❻ 将发货组合写入流中。

这里是基于订单 ID 来处理传入的订单的，当创建新的发货组合后，服务会将其写入相同的流中。这与客户端流 RPC 模式不同，当时服务通过 `SendAndClose` 方法写入流并将其关闭。当发现客户端流已经结束时，发送 `nil` 标记服务器端流的结束。

如代码清单 3-12 所示，客户端实现与之前的示例非常相似。当客户端通过 `OrderManagement` 对象调用 `ProcessOrders` 方法时，它会得到一个对流的引用（`streamProcOrder`），这个引用可以用来发送消息到服务器端，也能读取来自服务器端的消息。

代码清单 **3-12**　使用 Go 语言编写的 `ProcessOrders` 方法的

OrderManagement 客户端实现

```
// 处理订单
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.OrdersL
        }
        <-c
}
```

❶ 调用远程方法并获取流引用，以便在客户端写入和读取。

❷ 向服务发送消息。

❸ 创建 Goroutines 所使用的通道。

❹ 使用 Goroutines 调用函数，以便并行读取来自服务的消息。

❺ 模拟向服务发送消息的延迟。

❻ 为客户端流标记流的结束（订单 ID）。

❼ 在客户端读取服务的消息。

❽ 该条件探测流是否已经结束。

客户端可以在任意时间发送消息给服务并关闭流。读取消息也是同样的道理。前面的示例使用了 Go 语言中的 Goroutines，在两个并发线程中执行客户端的消息写入逻辑和消息服务逻辑。

Goroutines

在 Go 语言中，Goroutines 是能够与其他函数或方法并行运行的函数或方法，可以将它们视为轻量级的线程。

客户端可以并发读取和写入同一个流，输入流和输出流可以独立进行操作。这里所展示的是稍微复杂的示例，它展现了双向流 RPC 模式的威力。流的操作完全独立，客户端和服务器端可以按照任意顺序进行读取和写入，理解这一点非常重要。一旦建立连接，客户端和服务器端之间的通信模式就完全取决于客户端和服务器端本身。

目前本书已经讨论了所有可能的通信模式，可以使用它们实现基于 gRPC 的应用程序之间的交互。至于具体选择哪种通信模式，并没有硬性的规定，但是最好的办法就是分析业务用例，并据此选择最合适的模式。

在结束关于 gRPC 通信模式的讨论之前，还有一个重要的方面需要了

解，即 gRPC 是如何应用于微服务通信的。

3.5 使用gRPC实现微服务通信

gRPC 的主要用途之一就是实现微服务以及服务之间的通信。在微服务的服务间通信中，gRPC 会与其他通信协议一同使用，并且 gRPC 服务通常会实现为多语言服务（由不同的语言实现）。为了进一步理解该技术，下面来看在线零售系统这样一个真实的场景，如图 3-5 所示，它是对前述内容的扩展。

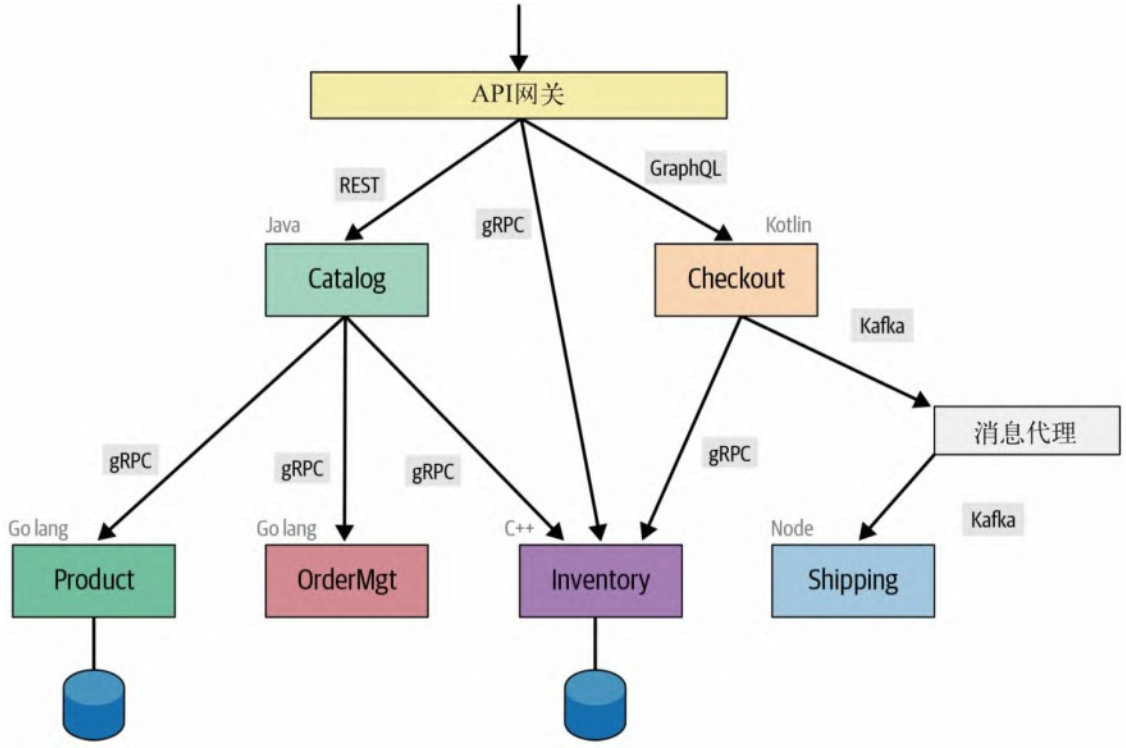

图 **3-5**：使用 **gRPC** 和其他协议的通用微服务部署模式

该场景中有许多微服务，每个微服务都面向在线零售系统的特定业务能力。有一些服务的实现形式是 gRPC 服务，如 `Product` 服务；另外还有一些组合服务，如 `Catalog` 服务，它会调用底层的服务来构建其业务能力。如第 1 章所述，大多数同步消息可以使用 gRPC 来传递。如果有特定的异步消息场景，可能需要持久化消息，那么就可以使用事件代理或消息代理，如 Kafka、Active MQ、RabbitMQ 和 NATS。当需要将特定的业务功能暴露到外部时，可以使用传统的基于 REST 或 OpenAPI 的服务或者 GraphQL 服务。因此，`Catalog` 和 `Checkout` 等服务消费

基于 gRPC 的后端服务，同时暴露基于 REST 或 GraphQL 的外部接口。

在大多数实际用例中，这些面向外部的服务是通过 API 网关暴露的。这里可以应用各种非功能性的能力，如安全性、节流、版本化等。大多数这样的 API 使用像 REST 或 GraphQL 这样的协议，但还有一种可能，这种情况不太常见，那就是只要 API 网关支持暴露 gRPC 接口，gRPC 就可以作为对外的接口。API 网关实现了横切性的功能，如认证、日志、版本化、节流和负载均衡。通过组合使用 API 网关与 gRPC API，可以将这些功能部署到核心 gRPC 服务之外。这种架构还有另外一个重要方面，那就是可以使用多种编程语言，但共享相同的服务契约，比如通过相同的 gRPC 服务定义来生成代码。这样一来，便可以根据服务的业务能力来选择适当的实现技术。

3.6 小结

gRPC 提供了一组不同的 RPC 通信风格，用于在基于 gRPC 的应用程序之间构建进程间通信。本章探讨了 4 种主要的通信模式，其中一元 RPC 模式是最基本的一种模式，它是一种非常简单的请求–响应式 RPC；服务器端流 RPC 模式可以在第一次调用远程方法后从服务向消费者发送多条消息；客户端流 RPC 模式可以从客户端向服务发送多条消息；双向流 RPC 模式有一点复杂，其中流的操作是完全独立的，客户端和服务器端可以按照任意顺序进行读取和写入。另外，本章深入研究了如何通过一些真实的用例来实现这些模式。

本章内容对实现任何 gRPC 用例都非常有用，你可以根据实际情况选择最合适的通信模式。虽然本章深入探讨了 gRPC 通信模式，但并没有涉及对用户透明的底层通信细节。第 4 章将介绍在使用基于 gRPC 的进程间通信时低级通信的实现方式。

第 4 章　gRPC 的底层原理

如前几章所述，gRPC 应用程序使用 RPC 通过网络进行通信。对于实现 RPC 的底层细节、所使用的消息编码技术以及在网络中的运行方式，这些方面都无须 gRPC 应用程序开发人员担心，只需使用服务定义来生成所选语言对应的服务器端代码和客户端代码即可。所有的底层通信细节都是由所生成的代码实现的，你需要做的就是处理高层级的抽象。但是，在构建基于 gRPC 的复杂系统并在生产环境中运行它们时，从根本上了解 gRPC 的工作原理十分重要。

本章将探索 gRPC 通信流的实现方式、所使用的编码技术以及 gRPC 中的底层网络通信技术的使用方法等，介绍涉及客户端调用给定 RPC 的消息流，并探讨其他相关问题，包括如何将其编排为网络上的 gRPC 调用、如何使用网络通信协议、如何在服务器端解排，以及如何调用对应的服务和远程函数等。

另外，本章将 protocol buffers 作为 gRPC 的编码技术，将 HTTP/2 作为 gRPC 的通信协议，并介绍它们的实现方法，最后研究 gRPC 实现架构以及围绕它所构建的语言支持栈。尽管对大多数 gRPC 应用程序来说，这里要讨论的底层细节作用有限，但在设计复杂的 gRPC 应用程序或设法调试现有的应用程序时，理解底层通信细节很有帮助。

4.1 RPC流

在 RPC 系统中，服务器端会实现一组可以远程调用的方法。客户端会生成一个存根，该存根为服务器端的方法提供抽象。这样一来，客户端应用程序可以直接调用存根方法，进而调用服务器端应用程序的远程方法。

以第 2 章所讨论的 `ProductInfo` 服务为例，下面看一下 RPC 如何通过网络来运行。在 `ProductInfo` 服务中，我们实现了一个 `getProduct` 方法，借助该方法，客户端可以通过提供商品 ID 来获取商品详情。图 4-1 展示了客户端在调用远程方法时所涉及的操作。

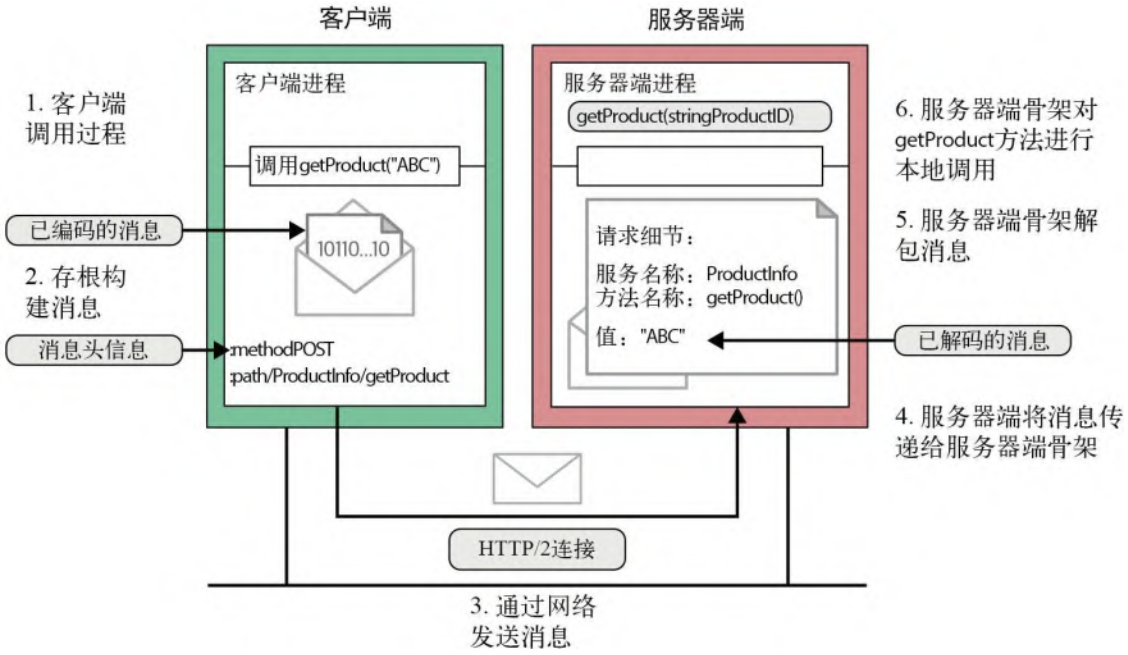

图 **4-1**：通过网络实现 **RPC**

如图 4-1 所示，当客户端通过生成的存根调用 `getProduct` 方法时，可以看出如下几个关键的步骤。

01. 客户端进程通过生成的存根调用 `getProduct` 方法。
02. 客户端存根使用已编码的消息创建 HTTP POST 请求。在 gRPC 中，所有的请求都是 HTTP POST 请求，并且 content-type 前缀为 `application/grpc`。要调用的远程方法

（/ProductInfo/getProduct）是以单独的 HTTP 头信息的形式发送的。

03. HTTP 请求消息通过网络发送到服务器端。
04. 当接收到消息后，服务器端检查消息头信息，从而确定需要调用的服务方法，然后将消息传递给服务器端骨架。
05. 服务器端骨架将消息字节解析成特定语言的数据结构。
06. 借助解析后的消息，服务发起对 getProduct 方法的本地调用。

服务方法的响应经过编码后被发送回客户端。响应消息会遵循我们在客户端上所观察到的相同过程（响应→编码→线路上的 HTTP 响应），该消息会被解包，它的值将返回给等待的客户端进程。

这些步骤与大多数 RPC 系统非常类似，如 CORBA、Java RMI 等。这里，gRPC 的主要区别在于消息的编码方式，如图 4-1 所示。在消息编码方面，gRPC 使用了 protocol buffers。protocol buffers 是一个语言中立、平台无关、实现结构化数据序列化的可扩展机制。只需定义数据该如何进行结构化，就可以使用专门生成的源代码，轻松地在各种数据流之间写入和读取结构化数据。

接下来深入理解 gRPC 如何使用 protocol buffers 编码消息。

4.2 使用protocol buffers编码消息

如前文所述，gRPC 使用 protocol buffers 编写服务定义。使用 protocol buffers 定义服务，具体包括定义服务中的远程方法以及希望通过网络发送的消息。以 `ProductInfo` 服务中的 `getProduct` 方法为例，该方法接受 `ProductID` 消息作为输入参数，并返回 `Product` 消息。这里可以将输入和输出的消息结构使用 protocol buffers 进行定义，如代码清单 4-1 所示。

代码清单 **4-1** `getProduct` 方法的服务定义

```
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;
}
```

如代码清单 4-1 所示，由于 `ProductID` 消息带有唯一的商品 ID，因此它只有一个字符串类型的字段，`Product` 消息则具有表示商品所需的结构。正确地定义消息非常重要，这决定了消息该如何进行编码。本节稍后将讨论在编码消息时如何使用消息定义。

有了消息定义之后，接下来看一下如何编码消息，并生成与之对等的字节内容。在正常情况下，这是由消息定义生成的源代码处理的。所有支持的语言都有自己的编译器来生成源代码，应用程序开发人员则需要将消息定义传递进去，从而生成读取消息和写入消息的源代码。

假设需要根据商品 ID（15）来获取商品详情，那么可以创建一个值为 15 的消息对象，并将其传递给 getProduct 方法。如下的代码片段展示了如何创建值为 15 的 ProductID 消息，并将其传递给 getProduct 方法，从而获取商品详情：

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

这个代码片段是用 Go 语言编写的，ProductID 消息的定义位于生成的代码之中。我们创建了一个 ProductID 实例，并将它的值设置为 15。Java 语言的实现与之类似，下面使用生成的方法来创建 ProductID 实例：

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

在接下来要讨论的 ProductID 消息结构中，有一个名为 value 的字段，并且字段索引为 1。当创建 value 值为 15 的消息实例时，对应的字节内容会包含一个用于 value 字段的标识符，随后是其编码后的值。字段的标识符也被称为标签（tag）：

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

这个字节内容的结构如图 4-2 所示，其中每个字段包含一个字段标识符及其编码后的值。

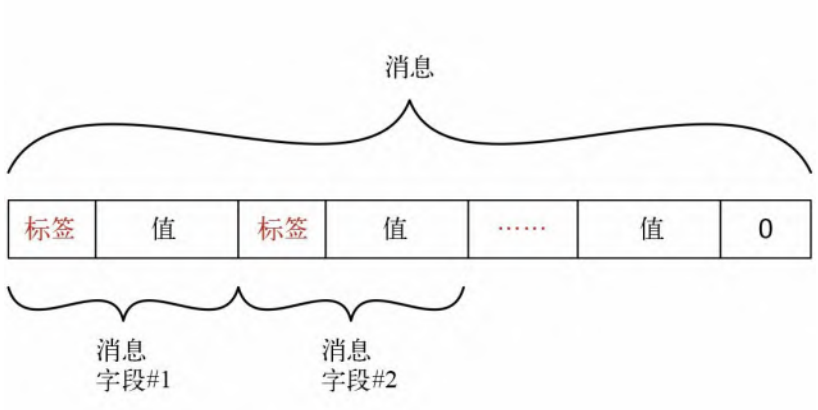

图 **4-2**：**protocol buffers** 编码后的字节流

标签由两个值构成：字段索引和线路类型（wire type）。字段索引就是在 proto 文件中定义消息时，为每个消息字段所设置的唯一数字。线路类型是基于字段类型的，也就是能够为字段输入值的数据类型。线路类型会提供信息来确定值的长度。表 4-1 展示了线路类型如何映射为字段类型，这些都是预定义的线路类型和字段类型的映射。可以参考 protocol buffers 编码的官方文档来获取关于映射的更多信息。

表**4-1**：可用的线路类型及其对应的字段类型

线路类型	分类	字段类型
0	Varint	int32、int64、uint32、uint64、sint32、sint64、bool、enum
1	64 位	fixed64、sfixed64、double
2	基于长度分隔	string、bytes、嵌入式消息、打包的 repeated 字段
3	起始组	groups（已废弃）
4	结束组	groups（已废弃）
5	32 位	fixed32、sfixed32、float

了解了特定字段的字段索引和线路类型后，就可以使用下面的公式来确定其标签的值。这里将表示字段索引的二进制左移 3 位并与表示线路类型的值进行按位或操作：

```
Tag value = (field_index << 3) | wire_type
```

图 4-3 展示了字段索引和线路类型在标签值中的排列方式。

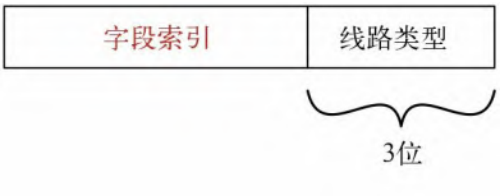

图 **4-3**：标签值的结构

可以通过前面的例子来进一步了解标签值这个术语。`ProductID` 有一个字符串字段，该字段的索引为 `1`，并且字符串的线路类型为 `2`。在将其转换为二进制表述时，字段索引将是 `00000001`，线路类型将是 `00000010`。将这些值代入公式，可以按照如下的方式得到值为 `10` 的标签值：

```
Tag value = (00000001 << 3) | 00000010
          = 000 1010
```

下一步就是编码消息字段的值。protocol buffers 使用不同的编码技术来编码不同类型的数据。对于字符串值，protocol buffers 会使用 UTF-8 对值进行编码；对于 `int32` 字段类型的整型值，它会使用名为 Varint 的编码技术。下面的小节将详细讨论不同的编码技术以及何时使用这些技术。下面看一下如何对字符串值进行编码，从而完成该示例。

在 protocol buffers 编码中，字符串值会使用 UTF-8 编码技术来进行编码。UTF（Unicode Transformation Format）使用 8 位的块来表示一个字符。它是一种长度可变的字符编码技术，也是 Web 页面和电子邮件首选的编码技术。

在这个示例的 `ProductID` 消息中，`value` 字段的值为 `15`，`15` 对应的 UTF-8 编码值为 `\x31\x35`。换句话说，表述编码值所需的 8 位块的数量并不是固定的，它会根据消息字段值的变化而变化。在这里，它会有 2 块。因此，我们需要在编码值之前传递编码值的长度，也就是编码值所要跨的块数。编码值为 `15` 的十六进制表示如下所示：

```
A 02 31 35
```

在这里，右侧的两字节是 `15` 的 UTF-8 编码值。值 `0x02` 表示编码后的字符串值在 8 位块中的长度。

当消息编码后，标签和值会连接到一个字节流中。图 4-2 展示了如何在消息有多个字段的情况下，将字段值安排成字节流。流的结束会通过发送值为 `0` 的标签来进行标记。

现在，我们已经使用 protocol buffers 完成了对带有字符串字段的简单消息进行编码。protocol buffers 支持各种字段类型，有些字段类型有不同的编码机制。下面概述 protocol buffers 所使用的编码技术。

编码技术

protocol buffers 支持很多种编码技术，它会根据数据类型使用不同的编码技术。例如，字符串值会使用 UTF-8 字符编码，`int32` 则会使用名为 Varint 的技术进行编码。在设计消息定义时，了解各种数据类型对应的编码技术很重要，这样做能够为每个消息字段设置最合适的数据类型，从而让消息能够在运行时高效编码。

protocol buffers 所支持的字段类型被分成了不同的组，每组使用不同的技术来编码值。下面列出了 protocol buffers 中的几种常用的编码技术。

01. **Varint** 类型

    Varint（可变长度整数）是使用单字节或多字节来序列化整数的方法。它基于这样一种思想：由于大多数数字并非均匀分布，因此为每个值所分配的字节数量不是固定的，而是依赖于具体的值。如表 4-1 所示，像 `int32`、`int64`、`uint32`、`uint64`、`sint32`、`sint64`、`bool` 和 `enum` 这样的字段类型属于 Varint 类型，并且会按照 Varint 进行编码。表 4-2 展示了在 Varint 分类下的字段类型以及每个类型的用途。

    表**4-2**：字段类型定义

字段类型	定义
`int32`	表示有符号整数的值类型，值的范围是–2 147 483 648 ～ 2 147 483 647。注意，这种类型在编码负数时效率较低
`int64`	表示有符号整数的值类型，值的范围是–9 223 372 036 854 775 808 ～ 9 223 372 036 854 775 807。注意，这种类型在编码负数时效率较低

uint32	表示无符号整数的值类型，值的范围是 0 ～ 4 294 967 295
uint64	表示无符号整数的值类型，值的范围是 0 ～ 18 446 744 073 709 551 615
sint32	表示有符号整数的值类型，值的范围是–2 147 483 648 ～ 2 147 483 647。相比常规的 int32，这种类型能够更高效地编码负数（续）
sint64	表示有符号整数的值类型，值的范围是–9 223 372 036 854 775 808 ～ 9 223 372 036 854 775 807。相比常规的 int64，这种类型能够更高效地编码负数
bool	表示只有两种可能值的值类型，通常用于表示 true 或 false
enum	表示一组命名值的值类型

在 Varint 中，除了最后的字节，其他所有字节都会设置最高有效位（most significant bit，MSB），表明后面还有字节。每字节中较低的 7 位用来存储数字的二进制补码形式。同时，最低有效组放在前面，这意味着我们要在低阶组中添加延续位。

02. 有符号整数类型

有符号整数是能够表示正整数值和负整数值的类型。像 sint32 和 sint64 这样的字段类型就是有符号整数。对于有符号类型，会使用 zigzag 编码来将有符号整数转换成无符号整数。随后，无符号整数会使用前面的 Varint 编码技术来进行编码。

在 zigzag 编码中，有符号整数会将负整数和正整数以“之”字形的方式映射为无符号整数。表 4-3 展示了如何使用 zigzag 编码实现映射。

表**4-3**：针对有符号整数使用**zigzag**编码

原始值	映射值
0	0
–1	1
1	2
–2	3
2	4

如表 4-3 所示，映射值 0 依然对应原始值 0，其他值则按照“之”字形的方式匹配为正数。原始的负值匹配为奇数正值，原始的正值则匹配为偶数正值。通过 zigzag 编码后，不管原始值的符号是什么，得到的都是正数。在得到正数之后，就可以使用 Varint 对值进行编码。

对于负整数，推荐使用像 `sint32` 和 `sint64` 这样的有符号整数类型，这是因为如果使用像 `int32` 或 `int64` 这样的常规类型，就意味着使用 Varint 编码将负值转换成二进制值，但这比转换正值要使用更多的字节。因此，有效编码负数的方式就是将负数转换成正数，并对正数进行编码。在像 `sint32` 这样的有符号整数类型中，负数首先会使用 zigzag 编码转换成正数，然后再使用 Varint 进行编码。

03. 非 **Varint** 类型

非 Varint 类型恰好与 Varint 类型相反。它们分配固定数量的字节，字节数与实际值没有关系。protocol buffers 有两个线路类型属于非 Varint 类型，其中一个用来表示 64 位的数据类型，如 `fixed64`、`sfixed64` 和 `double`；另一个用来表示 32 位的数据类型，如 `fixed32`、`sfixed32` 和 `float`。

04. 字符串类型

在 protocol buffers 中，字符串类型属于基于长度分隔（length-delimited）的线路类型，这意味着首先会有一个经过 Varint 编码的长度值，随后才是指定数量的字节数据。字符串值会使用 UTF-8 字符编码格式来进行编码。

以上就是编码常用数据类型所使用的技术。在 protocol buffers 的官网上，可以找到关于 protocol buffers 编码的详细介绍。

我们现在已经使用 protocol buffers 对消息进行了编码，接下来先将消息分帧，再通过网络将消息发送至服务器端。

4.3 基于长度前缀的消息分帧

通常，消息分帧（message-framing）会构建消息和通信，以便于目标受众很容易地提取信息。对 gRPC 通信来说，情况同样如此。一旦获取了要发送给另一方的已编码数据，就需要以对方易于提取信息的方式打包数据。为了打包要通过网络发送的信息，gRPC 使用了名为长度前缀分帧（length-prefix framing）的消息分帧技术。

长度前缀分帧是指在写入消息本身之前，写入长度信息，来表明每条消息的大小。如图 4-4 所示，已编码的二进制消息前面分配了 4 字节来指明消息的大小。在 gRPC 通信中，每条消息都有额外的 4 字节用来设置其大小。消息大小是一个有限的数字，为其分配 4 字节来表示消息的大小，也就意味着 gRPC 通信可以处理大小不超过 4GB 的所有消息。

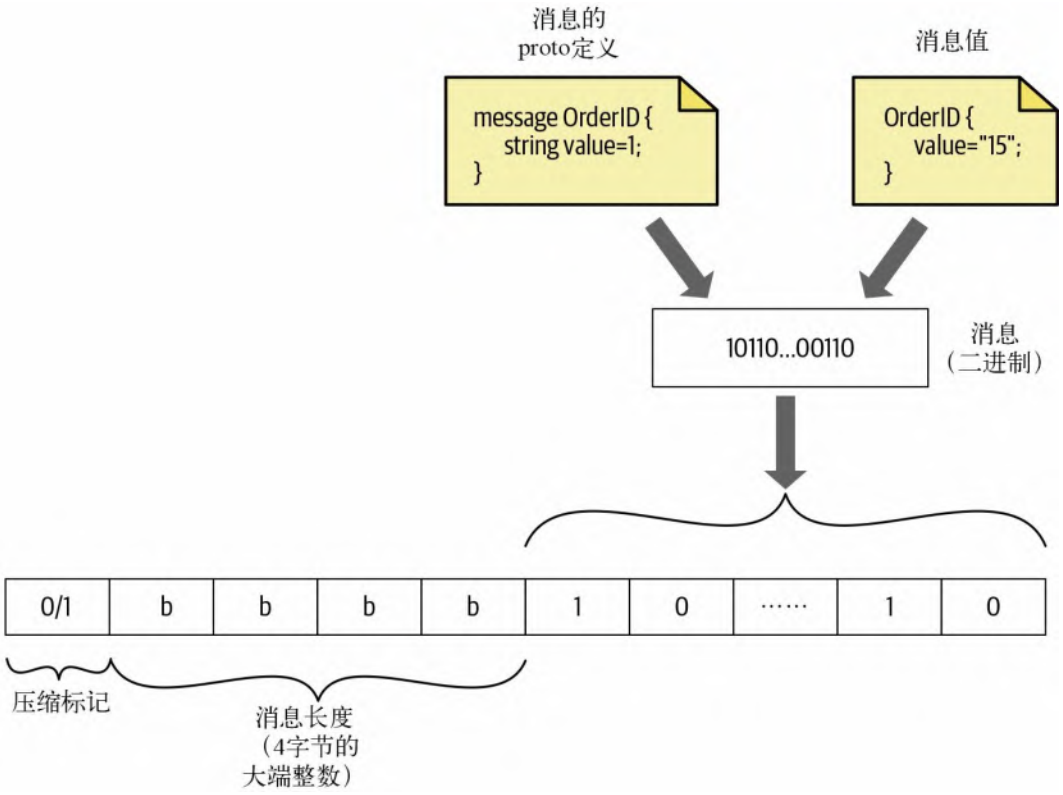

图 **4-4**：使用长度前缀分帧技术的 **gRPC** 消息帧

如图 4-4 所示，当消息使用 protocol buffers 编码时，我们会得到二进制

格式的消息。然后，计算二进制内容的大小，并以大端（big-endian）格式将其添加到二进制内容的前面。

大端是一种在系统或消息中对二进制数据进行排序的方式。在大端格式中，序列中的最高有效位（2 的最大乘方）存储在最低的存储地址上。

除了消息的大小，帧中还有单字节的无符号整数，用来表明数据是否进行了压缩。假设压缩标记值为 1，这代表二进制数据使用 Message-Encoding 头信息中声明的机制进行了压缩，该信息会作为 HTTP 传输头信息中的一项。假设值为 0，则代表消息字节没有进行压缩。4.4 节将详细讨论 gRPC 通信所支持的 HTTP 头信息。

现在，消息已经分帧完成，可以通过网络将其发送给收件方了。对于客户端的请求消息，收件方是服务器；而对于响应消息，收件方则是客户端。在收件方一侧，当收到消息之后，首先要读取其第一字节，来检查该消息是否经过压缩。然后，收件方读取接下来的 4 字节，以获取编码二进制消息的大小，接着就可以从流中精确地读取确切长度的字节了。对于简单的消息，只需处理一条以长度为前缀的消息；而对于流消息，就会有多条以长度为前缀的消息要处理。

我们大致理解了如何准备消息，以便于通过网络发送给收件方。4.4 节将讨论 gRPC 如何通过网络发送这些以长度为前缀的消息。目前，gRPC 核心支持 3 种传输实现：HTTP/2、Cronet 和进程内（in-process）。在这 3 种实现中，最常见的是 HTTP/2。下面看一下 gRPC 如何利用 HTTP/2 网络高效发送消息。

4.4 基于HTTP/2的gRPC

HTTP/2 是互联网协议 HTTP 的第 2 个主版本。之所以引入它，是为了解决以前版本（HTTP/1.1）在安全性、速度等方面所遇到的问题。HTTP/2 支持 HTTP/1.1 所有的核心特性，只不过实现方式更高效。因此，使用 HTTP/2 编写的应用程序更快、更简单，也更健壮。

gRPC 使用 HTTP/2 作为其传输协议，实现通过网络发送消息。这也是 gRPC 能够成为高性能 RPC 框架的原因之一。接下来探索 gRPC 和 HTTP/2 的关系。

在 HTTP/2 中，客户端和服务器端的所有通信都是通过一个 TCP 连接完成的，这个连接可以传送任意数量的双向字节流。为了理解 HTTP/2 的过程，最好熟悉下面这些重要术语。

- 流（stream）：在一个已建立的连接上的双向字节流。一个流可以携带一条或多条消息。
- 帧（frame）：HTTP/2 中最小的通信单元。每一帧都包含一个帧头，它至少要标记该帧所属的流。
- 消息（message）：完整的帧序列，映射为一条逻辑上的 HTTP 消息，由一帧或多帧组成。这样的话，允许消息进行多路复用，客户端和服务器端能够将消息分解成独立的帧，交叉发送它们，然后在另一端进行重新组合。

如图 4-5 所示，gRPC 通道代表一个到端点的连接，也就是一个 HTTP/2 连接。当客户端应用程序创建 gRPC 通道的时候，它会在幕后创建一个到服务器端的 HTTP/2 连接。在通道创建完成之后，就可以重用它来发送多个到服务器端的远程调用。这些远程调用会映射为 HTTP/2 中的流。远程调用中的消息以 HTTP/2 帧的形式进行发送，帧可能会携带一条 gRPC 长度前缀的消息，也可能在 gRPC 消息非常大的情况下，一条消息跨多帧。

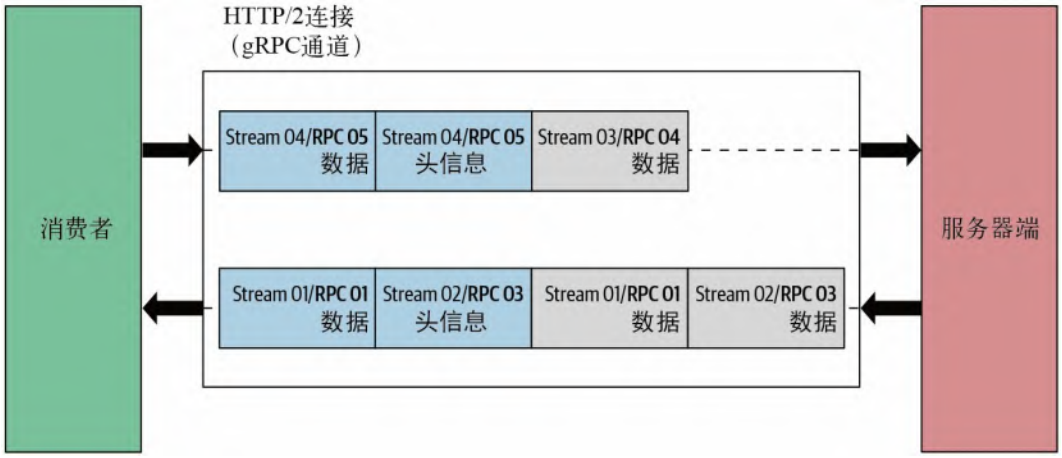

图 **4-5**：**gRPC** 语义与 **HTTP/2** 之间的关系

4.3 节讨论了基于长度前缀的消息分帧。当把这些消息以请求消息或响应消息的形式通过网络进行发送时，除了消息本身，还要发送额外的头信息。下面讨论如何组织请求消息和响应消息，以及针对每条消息所要传递的头信息。

4.4.1 请求消息

请求消息用于初始化远程调用。在 gRPC 中，请求消息始终由客户端应用程序来触发，它包含 3 部分：请求头信息、以长度作为前缀的消息以及流结束标记（end of stream flag，以下简称 EOS 标记），如图 4-6 所示。远程调用在客户端发送请求头信息之后就会初始化，然后其中会发送以长度作为前缀的消息，最后发送 EOS 标记，通知收件方请求消息已发送。

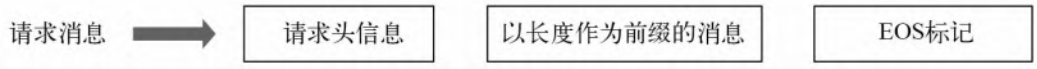

图 **4-6**：请求消息中的消息元素序列

这里可以再次使用 `ProductInfo` 服务中的 `getProduct` 方法，来理解请求消息在 HTTP/2 帧中的发送方式。当调用 `getProduct` 方法时，客户端会通过发送下面的请求头信息来初始化调用。

```
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 定义 HTTP 方法。对 gRPC 来说，`:method` 头信息始终为 `POST`。

❷ 定义 HTTP 模式。如果启用传输层安全协议（Transport Level Security，TLS），就将模式设置为 `https`，否则设置为 `http`。

❸ 定义端点路径。对 gRPC 来说，这个值的构造为 `/{ 服务名 }/{ 方法名 }`。

❹ 定义目标 URI 的虚拟主机名。

❺ 定义对不兼容代理的检测。在 gRPC 中，这个值必须为 `trailers`。

❻ 定义调用的超时时间。如果没有指定，服务器端会假定超时时间无穷大。

❼ 定义 `content-type`。对 gRPC 来说，`content-type` 应该以 `application/grpc` 开头。否则，gRPC 会给出 HTTP 状态为 415（不支持的媒体类型）的响应。

❽ 定义消息的压缩类型。可选的值是 `identity`、`gzip`、`deflate`、`snappy` 和 `{custom}`。

❾ 这是可选的元数据。`authorization` 元数据用来访问安全的端点。

在本例中，还有其他几点需要注意。

- 名称以“:”开头的头信息叫作保留头信息，HTTP/2 要求保留头信息出现在其他头信息之前。
- gRPC 通信中所传递的头信息分为两类：调用定义的头信息（call-definition header）和自定义元数据。

- 调用定义的头信息是 HTTP/2 预定义的头信息。这些头信息应该在自定义元数据之前发送。
- 自定义元数据是由应用程序层定义的任意一组键–值对。在声明自定义元数据时，需要确保不要使用以 grpc- 开头的名称。在 gRPC 核心中，这被列为保留名字。

当完成对服务器端调用的初始化之后，客户端会以 HTTP/2 数据帧的形式发送以长度作为前缀的消息。如果这条消息不适合放到一个数据帧中，那么它可以跨多个数据帧。请求消息的结束通过在最后一个 DATA 帧上添加 END_STREAM 标记来实现。当因为没有要发送的数据而需要关闭请求流时，必须发送一个带有 END_STREAM 标记的空数据帧：

```
DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

这里只是大致介绍了 gRPC 请求消息的结构，你可以通过 gRPC 官方的 GitHub 仓库了解更多细节。

与请求消息类似，响应消息也有其自身的结构。接下来看一下响应消息的结构及其关联的头信息。

4.4.2 响应消息

响应消息由服务器端生成，用来响应客户端的请求。与请求消息类似，在大多数场景中，响应消息也包含 3 个主要部分：响应头信息、以长度作为前缀的消息以及 trailer。如果没有发送以长度作为前缀的消息来响应客户端，则响应消息只会包含头信息和 trailer，如图 4-7 所示。

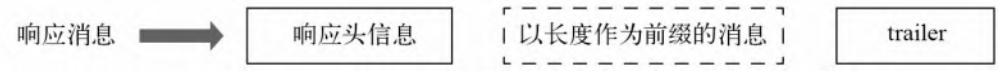

图 **4-7**：响应消息中的消息元素序列

下面通过同一个示例来介绍响应消息的 HTTP/2 帧序列。当服务器端发送响应消息至客户端时，首先会发送如下所示的响应头信息。

```
HEADERS (flags = END_HEADERS)
:status = 200 ❶
grpc-encoding = gzip ❷
```

```
content-type = application/grpc ❸
```

❶ 表明 HTTP 请求的状态。

❷ 定义消息的压缩类型。可选的值是 `identity`、`gzip`、`deflate`、`snappy` 和 `{custom}`。

❸ 定义 `content-type`。对 gRPC 来说，`content-type` 应该以 `application/grpc` 开头。

与请求头信息类似，应用程序层所定义的自定义元数据也可以按照任意键–值对集的形式在响应头信息中进行发送。

服务器端发送完响应头之后，以长度作为前缀的消息就会以 HTTP/2 数据帧的形式在调用中进行发送。与请求消息类似，如果该消息不适合放到一个数据帧中，那么它可以跨多个数据帧。如下所示，`END_STREAM` 标记并不会随数据帧一起发送，而会作为单独的头信息来发送，名为 trailer：

```
DATA
<Length-Prefixed Message>
```

最后，通过发送 trailer 来提醒客户端响应消息已发送。trailer 还会携带状态码以及请求的状态信息。

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

❶ 定义 gRPC 状态码。gRPC 会使用一组定义良好的状态码。这些状态码的定义可以在 gRPC 官方文档中找到。

❷ 定义对错误的描述。这是可选的，只有在处理请求出现错误时，才会进行设置。

trailer 会以 HTTP/2 头信息帧的形式进行投递，但会在响应消息结束时发送。响应 EOS 标记就是在 trailer 头信息中设置的

END_STREAM 标记。另外，它还会包含 grpc-status 头信息和 grpc-message 头信息。

在特定的场景中，请求调用可能会立即失败。在这些情况下，服务器端需要发回一个不包含数据帧的响应。因为服务器端只发送 trailer 作为响应，所以这些 trailer 也会以 HTTP/2 头信息帧的形式进行投递，同时会包含 END_STREAM 标记。另外，trailer 会包含下面的头信息。

- HTTP 状态：:status
- 内容类型：content-type
- 状态：grpc-status
- 状态信息：grpc-message

现在，我们已经知道了 gRPC 消息如何在 HTTP/2 连接上流动，接下来看不同通信模式的消息流。

4.4.3 理解gRPC通信模式中的消息流

第 3 章介绍了 gRPC 支持的 4 种通信模式，即一元 RPC 模式、服务器端流 RPC 模式、客户端流 RPC 模式以及双向流 RPC 模式，也讨论了这些通信模式在现实场景中的运行方式。本节将从不同的角度再来看一下这些模式，并结合本章内容，讨论每种模式在传输层中的运行方式。

01. 一元 **RPC** 模式

    在一元 RPC 模式中，gRPC 服务器端和 gRPC 客户端的通信始终只涉及一个请求和一个响应。如图 4-8 所示，请求消息包含头信息，随后是以长度作为前缀的消息，该消息可以跨一个或多个数据帧。消息最后会添加一个 EOS 标记，方便客户端半关（half-close）连接，并标记请求消息的结束。在这里，“半关”指的是客户端在自己的一侧关闭连接，这样一来，客户端无法再向服务器端发送消息，但仍能够监听来自服务器端的消息。只有在接收到完整的消息之后，服务器端才生成响应。响应消息包含一个头信息帧，随后是以长度作为前缀的消息。当服务器端发送带有状态详情的 trailer 头信息之后，通信就会关闭。

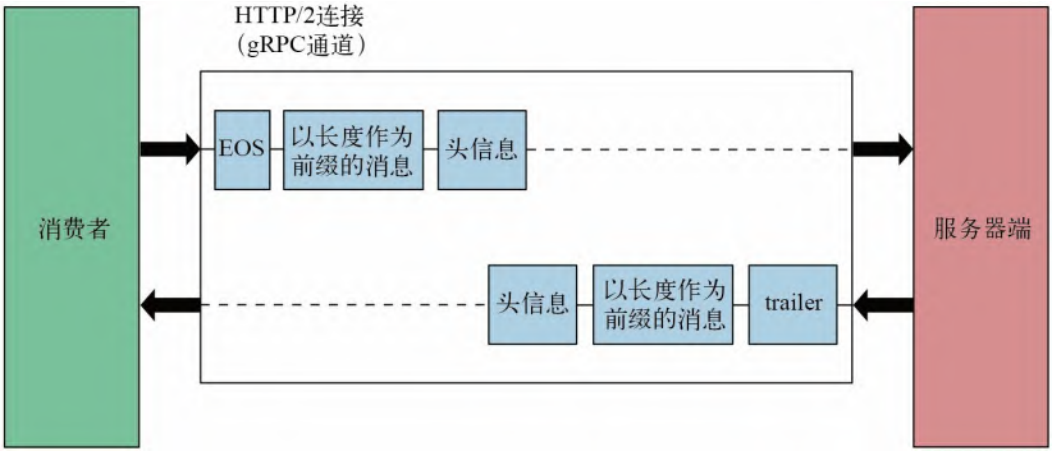

图 **4-8**：一元 **RPC** 模式：消息流

这是最简单的通信模式。接下来看一下较为复杂的服务器端流 RPC 模式。

02. 服务器端流 **RPC** 模式

从客户端的角度来说，一元 RPC 模式和服务器端流 RPC 模式具有相同的请求信息流。这两种情况都是发送一条请求消息，主要差异在于服务器端。在服务器端流 RPC 模式中，服务器端不再向客户端发送一条响应消息，而会发送多条响应消息。服务器端会持续等待，直到接收到完整的请求消息，随后它会发送响应头消息和多条以长度作为前缀的消息，如图 4-9 所示。在服务器端发送带有状态详情的 trailer 头信息之后，通信就会关闭。

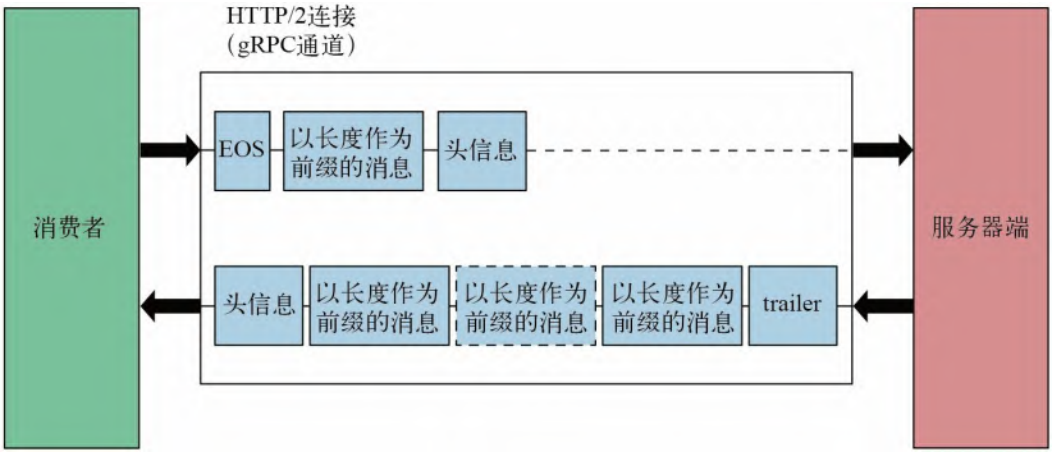

图 **4-9**：服务器端流 **RPC** 模式：消息流

下面看一下客户端流 RPC 模式，它在很大程度上与服务器端流 RPC 模式相反。

03. 客户端流 **RPC** 模式

在客户端流 RPC 模式中，客户端向服务器端发送多条消息，服务器端在响应时发送一条消息。客户端首先通过发送头信息帧来与服务器端建立连接，然后以数据帧的形式，向服务器端发送多条以长度作为前缀的消息，如图 4-10 所示。最后，通过在末尾的数据帧中发送 EOS 标记，客户端将连接设置为半关的状态。与此同时，服务器端读取所接收到的来自客户端的消息。在接收到所有的消息之后，客户端发送一条响应消息和 trailer 头信息，并关闭连接。

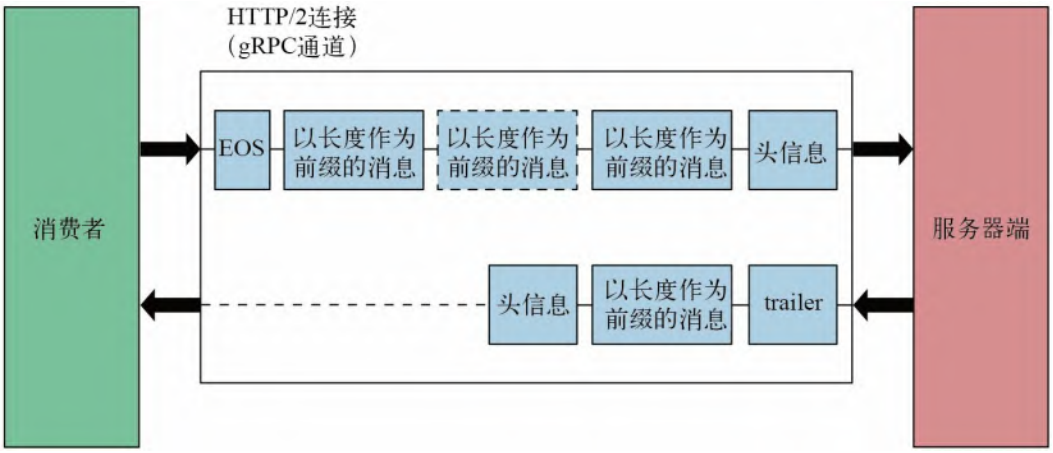

图 **4-10**：客户端流 **RPC** 模式：消息流

接下来看最后一种通信模式，也就是双向流 RPC 模式。在这种模式中，客户端和服务器端都会给对方发送多条消息，直到它们关闭连接为止。

04. 双向流 **RPC** 模式

在双向流 RPC 模式中，客户端通过发送头信息帧与服务器端建立连接。然后，它们会互发以长度作为前缀的消息，无须等待对方结

束。如图 4-11 所示，客户端和服务器端会同时发送消息。两者都可以在自己的一侧关闭连接，这意味着它们不能再发送消息了。

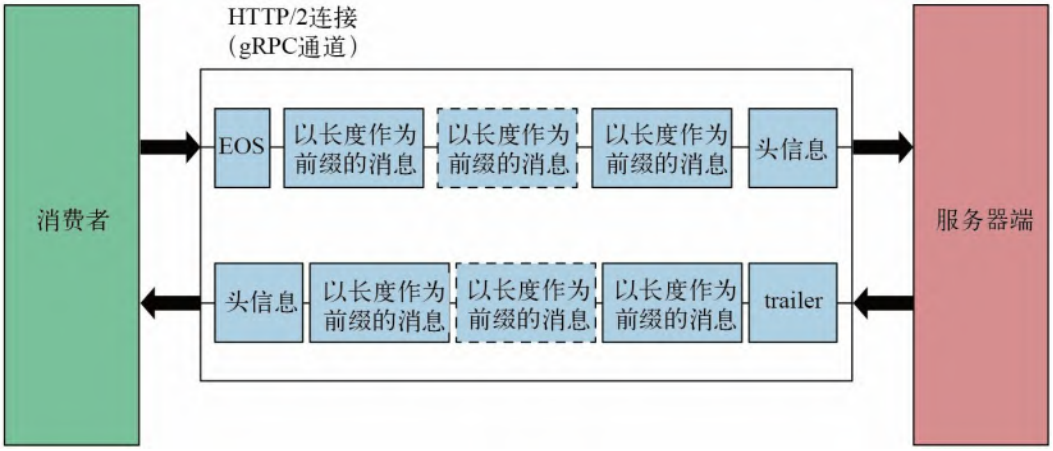

图 **4-11**：双向流 **RPC** 模式：消息流

至此，深入学习 gRPC 通信的旅程即将结束。网络以及通信中有关传输的操作通常是在 gRPC 核心层处理的，但 gRPC 应用程序开发人员无须关注这些细节。

在结束本章之前，让我们来看一下 gRPC 的实现架构和语言栈。

4.5 gRPC实现架构

如图 4-12 所示，gRPC 实现架构可以分为多层。最基础的是 gRPC 核心层，它为其上的层抽象了所有的网络操作，使得应用程序开发人员可以很容易地通过网络发送 RPC 调用。gRPC 核心层还提供了对核心功能的扩展，其中一些扩展点是认证过滤器，用来处理对安全和截止时间（deadline）过滤器的调用，从而实现调用截止时间等功能。

gRPC 原生支持 C/C++ 语言、Go 语言和 Java 语言，它还提供了很多流行语言的绑定，如 Python、Ruby、PHP 等，这些语言绑定是对低层级 C API 的包装器。

应用程序代码构建在这些语言绑定之上。应用程序层处理应用程序逻辑和数据编码逻辑，在正常情况下，开发人员会使用不同语言所提供的编译器生成数据编码逻辑的源代码。如果使用 protocol buffers 来编码数据，就可以使用 protocol buffers 编译器来生成源代码。因此，开发人员可以通过调用生成源代码的方法来编写业务逻辑。

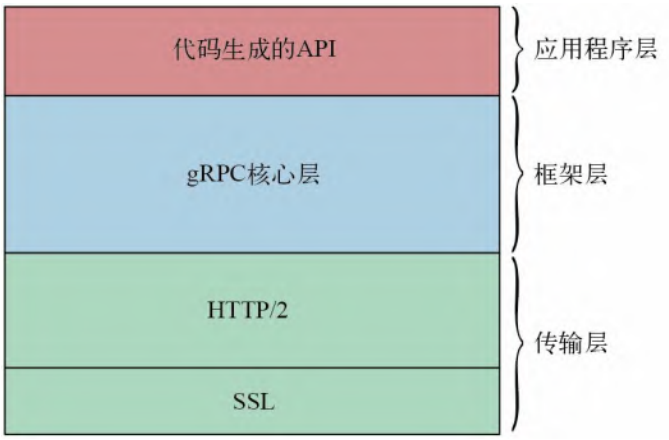

图 **4-12**：**gRPC** 原生实现架构

目前，本书已经介绍了 gRPC 应用程序的大部分底层实现和执行细节。应用程序开发人员最好对所用技术的底层细节有一定的了解。这不仅有助于设计稳健的应用程序，还能更容易地解决应用程序中的问题。

4.6 小结

gRPC 构建在两个快速、高效的协议之上，也就是 protocol buffers 和 HTTP/2。protocol buffers 是一个语言中立、平台无关的数据序列化协议，并且提供了可扩展的机制来实现结构化数据的序列化。当序列化完成之后，该协议会生成二进制载荷，这种载荷会比常见的 JSON 载荷更小，并且是强类型的。序列化之后的二进制载荷会通过名为 HTTP/2 的二进制传输协议进行发送。

HTTP/2 是互联网协议 HTTP 的第 2 个主版本。HTTP/2 是完全多路复用的，这意味着 HTTP/2 可以在 TCP 连接上并行发送多个数据请求。这样一来，使用 HTTP/2 编写的应用程序更快、更简洁、更稳健。

以上诸多因素使 gRPC 成为高性能的 RPC 框架。

本章介绍了 gRPC 通信的底层细节。对于开发 gRPC 应用程序来说，由于 gRPC 库已经处理了这些细节，因此我们无须再掌握它们，但在生产环境中使用 gRPC 时，了解底层 gRPC 消息流，对于排查 gRPC 通信的相关问题至关重要。第 5 章将讨论 gRPC 提供的一些高级功能，以满足实际需求。

第 5 章　gRPC：超越基础知识

当构建真正的 gRPC 应用程序时，可能需要增强它们的各种能力以满足需求，比如拦截 RPC 的输入和输出、弹性处理网络延迟、处理错误、在服务和消费者之间共享元数据等。

为保持一致性，本章的所有示例都使用 Go 语言来进行阐述。如果你更熟悉 Java，则可以参考源代码仓库中相同使用场景的 Java 示例。

本章将介绍一些关键的高级 gRPC 功能，包括使用 gRPC 拦截器在服务器端和客户端拦截 RPC、使用截止时间来指定等待 RPC 完成的时间、服务器端和客户端错误处理的最佳实践、使用多路复用在同一台服务器上运行多个服务、在应用程序间共享自定义的元数据、在调用其他服务的时候使用负载均衡和命名解析技术，以及压缩 RPC 调用以高效使用网络带宽。

下面先来看一下拦截器的相关概念。

5.1 拦截器

在构建 gRPC 应用程序时，无论是客户端应用程序，还是服务器端应用程序，在远程方法执行之前或之后，都可能需要执行一些通用逻辑。在 gRPC 中，可以拦截 RPC 的执行，来满足特定的需求，如日志、认证、性能度量指标等，这会使用一种名为拦截器的扩展机制。gRPC 提供了简单的 API，用来在客户端和服务器端的 gRPC 应用程序中实现并安装拦截器。它是 gRPC 核心扩展机制之一，在一些使用场景中非常有用，比如日志、身份验证、授权、性能度量指标、跟踪以及其他一些自定义需求。

支持 gRPC 的所有语言并非都支持拦截器功能，而且每种语言的拦截器实现可能会有所差异。本书只涉及 Go 语言和 Java 语言。

根据所拦截的 RPC 调用的类型，gRPC 拦截器可以分为两类。对于一元 RPC，可以使用一元拦截器（unary interceptor）；对于流 RPC，则可以使用流拦截器（streaming interceptor）。这些拦截器既可以用在 gRPC 服务器端，也可以用在 gRPC 客户端。接下来看如何在服务器端使用拦截器。

5.1.1 服务器端拦截器

当客户端调用 gRPC 服务的远程方法时，通过使用服务器端拦截器，可以在执行远程方法之前，执行一个通用的逻辑。当需要在调用远程方法之前应用认证等特性时，这会非常有帮助。如图 5-1 所示，在所开发的任意 gRPC 服务器端，都可以插入一个或多个拦截器。如果希望向 `OrderManagement` gRPC 服务中插入新服务器端拦截器，则可以实现该拦截器并在创建 gRPC 服务器端时将其注册进来。

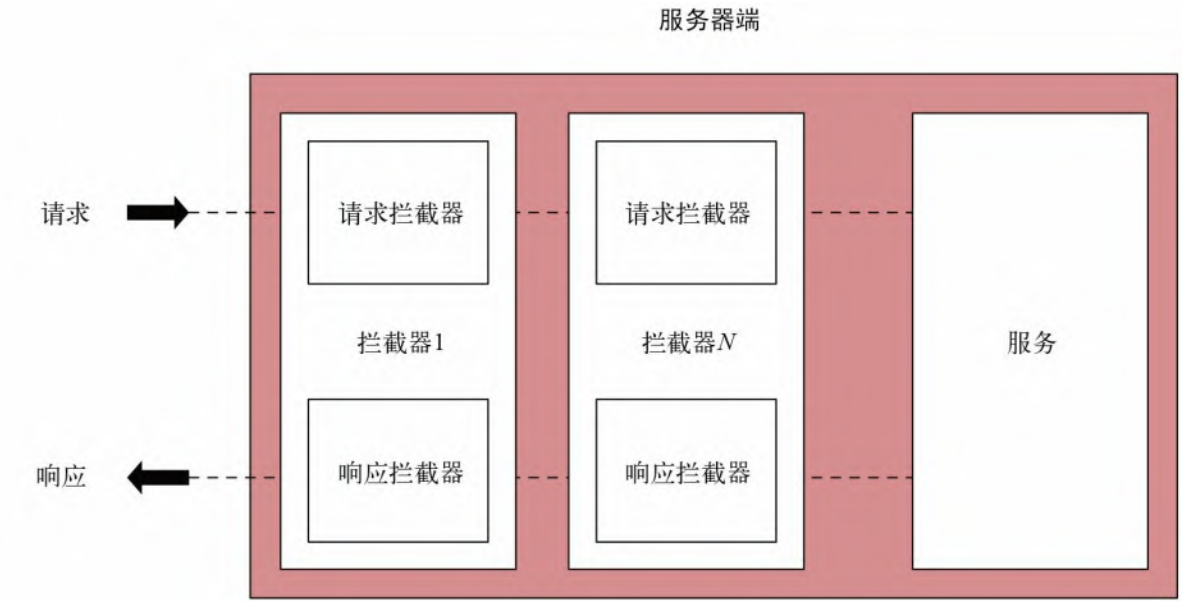

图 **5-1**：服务器端拦截器

在服务器端，一元拦截器拦截一元 RPC，流拦截器则拦截流 RPC。下面来看一下服务器端一元拦截器。

01. 一元拦截器

如果想在服务器端拦截 gRPC 服务的一元 RPC，需要为 gRPC 服务器端实现一元拦截器。如代码清单 5-1 所示，要实现这一点，需要先实现 `UnaryServerInterceptor` 类型的函数，并在创建 gRPC 服务器端时将函数注册进来。`UnaryServerInterceptor` 是用于服务器端一元拦截器的类型，它具有以下签名：

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

在这个函数中，我们能够完全控制传入 gRPC 服务器端的所有一元 RPC。

代码清单 **5-1**　gRPC 服务器端一元拦截器

```
// 服务器端一元拦截器
func orderUnaryServerInterceptor(ctx context.Context, req interface{},
                              info *grpc.UnaryServerInfo, handler grpc.
                              (interface{}, error) {
```

```
        // 前置处理逻辑
        // 通过检查传入的参数，获取关于当前RPC的信息
        log.Println("======= [Server Interceptor] ", info.FullMethod)

        // 调用handler完成一元RPC的正常执行
        m, err := handler(ctx, req) ❷

        // 后置处理逻辑
        log.Printf(" Post Proc Message : %s", m) ❸
        return m, err ❹
}

// ...

func main() {

...
     // 在服务器端注册拦截器
        s := grpc.NewServer(
                grpc.UnaryInterceptor(orderUnaryServerInterceptor)) ❺
...
```

❶ 前置处理阶段：可以在调用对应的 RPC 之前拦截消息。

❷ 通过 UnaryHandler 调用 RPC 方法。

❸ 后置处理阶段：可以在这里处理 RPC 响应。

❹ 将 RPC 响应发送回去。

❺ 使用 gRPC 服务器端注册一元拦截器。

服务器端一元拦截器的实现通常可以分为 3 个部分：前置处理、调用 RPC 方法以及后置处理。顾名思义，前置处理阶段是在调用预期的 RPC 远程方法之前执行。在前置处理阶段，用户可以通过检查传入的参数来获取关于当前 RPC 的信息，比如 RPC 上下文、RPC 请求和服务器端信息。因此，我们甚至可以在预处理阶段修改 RPC。

随后，在调用阶段，需要调用 gRPC UnaryHandler 来触发 RPC 方法。在调用 RPC 之后，就进入后置处理阶段。这意味着，RPC 响应要流经后置处理阶段。在这个阶段中，可以按需处理返回的响应

和错误。当后置处理阶段完成之后，需要以拦截器函数返回参数的形式将消息和错误返回。如果不需要后置处理器，那么可以直接返回 handler 调用（handler(ctx, req)）。接下来看一下流拦截器。

02. 流拦截器

服务器端流拦截器会拦截 gRPC 服务器所处理的所有流 RPC。流拦截器包括前置处理阶段和流操作拦截阶段。

如代码清单 5-2 所示，假设希望拦截对 OrderManagement 服务的流 RPC。StreamServerInterceptor 是服务器端的流拦截器类型。orderServerStreamInterceptor 是具有如下签名的 StreamServerInterceptor 类型的拦截器函数：

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

与一元拦截器类似，在前置处理阶段，可以在流 RPC 进入服务实现之前对其进行拦截。在前置处理阶段之后，则可以调用 StreamHandler 来完成远程方法的 RPC 执行，而且通过已实现 grpc.ServerStream 接口的包装器流接口，可以拦截流 RPC 的消息。在通过 handler(srv, newWrappedStream(ss)) 方法调用 grpc.StreamHandler 时，可以将这个包装器结构传递进来。grpc.ServerStream 的包装器可以拦截 gRPC 服务发送或接收到的数据。它实现了 SendMsg 函数和 RecvMsg 函数，这两个函数分别会在服务发送和接收 RPC 流消息的时候被调用。

代码清单 **5-2**　gRPC 服务器端流拦截器

```
// 服务器端流拦截器
// wrappedStream包装嵌入的grpc.ServerStream
// 并拦截对RecvMsg函数和SendMsg函数的调用

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
}

...
// 注册拦截器
s := grpc.NewServer(
                grpc.StreamInterceptor(orderServerStreamInterceptor))
...
```

❶ grpc.ServerStream 的包装器流。

❷ 实现包装器的 RecvMsg 函数，来处理流 RPC 所接收到的消息。

❸ 实现包装器的 SendMsg 函数，来处理流 RPC 所发送的消息。

❹ 创建新包装器流的实例。

❺ 流拦截器的实现。

❻ 前置处理阶段。

❼ 使用包装器流调用流 RPC。

❽ 注册拦截器。

下面的 gRPC 服务器端日志输出，可以帮助你理解服务器端流拦截器的行为。根据每条日志的打印顺序，可以看出流拦截器的行为。这里调用的流远程方法是 `searchOrders`，它是一个服务器端流 RPC：

```
[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...
```

客户端拦截器的术语与服务器端拦截器的非常相似，只不过在接口和函数签名方面有细微的差异。下面来看关于客户端拦截器的介绍。

5.1.2 客户端拦截器

当客户端发起 RPC 来触发 gRPC 服务的远程方法时，可以在客户端拦截这些 RPC。如图 5-2 所示，借助客户端拦截器，可以拦截一元 RPC 和流 RPC。

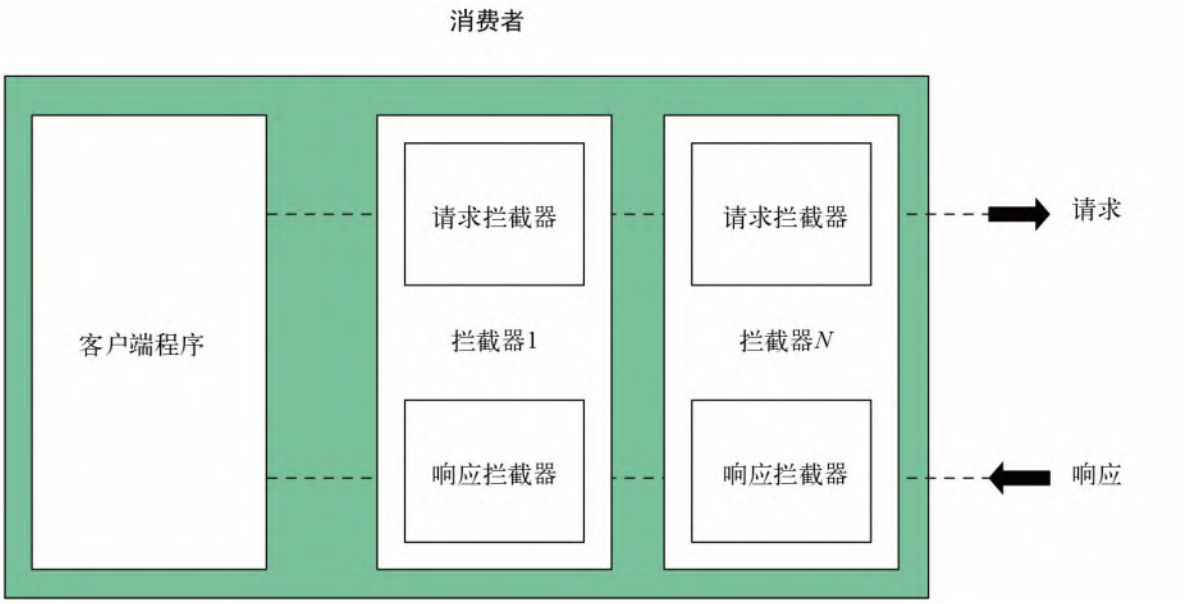

图 **5-2**：客户端拦截器

当需要实现一些特定的可重用特性时，这会非常有用，比如在客户端应用程序代码之外实现对 gRPC 服务调用的安全保护。

01. 一元拦截器

客户端一元拦截器用于拦截一元 RPC 客户端的调用。`UnaryClientInterceptor` 是客户端一元拦截器的类型，函数签名如下：

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

与前面介绍的服务器端一元拦截器一样，客户端一元拦截器也有不同的阶段。代码清单 5-3 展示了客户端一元拦截器的基本 Go 语言实现。在前置处理阶段，可以在调用远程方法之前拦截 RPC。这里可以通过检查传入的参数来访问关于当前 RPC 的信息，比如 RPC 的上下文、方法字符串、要发送的请求以及 `CallOption` 配置。这样一来，我们甚至可以在原始的 RPC 发送至服务器端应用程序之前，对其进行修改。随后，借助 `UnaryInvoker` 参数，可以调用实际的一元 RPC。在后置处理阶段，可以访问 RPC 的响应结果或错误结果。

代码清单 **5-3**　gRPC 客户端一元拦截器

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

        // 调用远程方法
        err := invoker(ctx, method, req, reply, cc, opts...) ❷

        // 后置处理阶段
        log.Println(reply) ❸

        return err ❹
}
...

func main() {
        // 建立到服务器端的连接
        conn, err := grpc.Dial(address, grpc.WithInsecure(),
                grpc.WithUnaryInterceptor(orderUnaryClientInterceptor)
...
```

❶ 前置处理阶段能够在 RPC 请求发送至服务器端之前访问它。

❷ 通过 `UnaryInvoker` 调用 RPC 方法。

❸ 后置处理阶段，可以在这里处理响应结果或错误结果。

❹ 向 gRPC 客户端应用程序返回错误，同时包含作为参数传递进来的答复。

❺ 通过传入一元拦截器作为 `grpc.Dial` 的选项，建立到服务器端的连接。

注册拦截器函数通过使用 `grpc.WithUnaryInterceptor`，来在 `grpc.Dial` 操作中实现。

02. 流拦截器

客户端流拦截器会拦截 gRPC 客户端所处理的所有流 RPC。客户端流拦截器的实现与服务器端流拦截器的实现非常相似。`StreamClientInterceptor` 是客户端流拦截器的类型，其函数类型签名如下所示：

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

如代码清单 5-4 所示，客户端流拦截器实现包括前置处理和流操作拦截。

代码清单 **5-4**　gRPC 客户端流拦截器

```
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() {
        // 建立到服务器端的连接
        conn, err := grpc.Dial(address, grpc.WithInsecure(),
        grpc.WithStreamInterceptor(clientStreamInterceptor)) ❼
...
```

❶ 前置处理阶段能够在将 RPC 请求发送至服务器端之前访问它。

❷ 调用传入的 `streamer` 来获取 `ClientStream`。

❸ 包装 `ClientStream`，使用拦截逻辑重载其方法并返回给客户端应用程序。

❹ `grpc.ClientStream` 的包装器流。

❺ 拦截流 RPC 所接收消息的函数。

❻ 拦截流 RPC 所发送消息的函数。

❼ 注册流拦截器。

流操作拦截是通过流的包装器实现完成的，该实现中必须实现包装 `grpc.ClientStream` 的新结构。这里实现了两个包装流的函数，即 `RecvMsg` 函数和 `SendMsg` 函数，分别用来拦截客户端接收及发送的流消息。拦截器的注册和一元拦截器是一样的，都是通过 `grpc.Dial` 操作完成的。

接下来看一下在客户端应用程序中调用 gRPC 服务时经常需要的另一项功能，那就是截止时间。

5.2 截止时间

在分布式计算中，截止时间（deadline）和超时时间（timeout）是两个常用的模式。超时时间可以指定客户端应用程序等待 RPC 完成的时间（之后会以错误结束），它通常会以持续时长的方式来指定，并且在每个客户端本地进行应用。例如，一个请求可能会由多个下游 RPC 组成，它们会将多个服务链接在一起。因此，可以在每个服务调用上，针对每个 RPC 都指定超时时间。这意味着超时时间不能直接应用于请求的整个生命周期，这时需要使用截止时间。

截止时间以请求开始的绝对时间来表示（即使 API 将它们表示为持续时间偏移），并且应用于多个服务调用。发起请求的应用程序设置截止时间，整个请求链需要在截止时间之前进行响应。gRPC API 支持为 RPC 使用截止时间，出于多种原因，在 gRPC 应用程序中使用截止时间始终是一种最佳实践。由于 gRPC 通信是在网络上发生的，因此在 RPC 和响应之间会有延迟。另外，在一些特定的场景中，gRPC 服务本身可能要花费更多的时间来响应，这取决于服务的业务逻辑。如果客户端应用程序在开发时没有指定截止时间，那么它们会无限期地等待自己所发起的 RPC 请求的响应，而资源都会被正在处理的请求所占用。这会让服务和客户端都面临资源耗尽的风险，增加服务的延迟，甚至可能导致整个 gRPC 服务崩溃。

在图 5-3 中，gRPC 客户端应用程序调用商品管理服务，而商品管理服务又调用库存服务。

客户端应用程序的截止时间设置为 50 毫秒（截止时间 = 当前时间 + 偏移量）。客户端和 `ProductMgt` 服务之间的网络延迟为 0 毫秒，`ProductMgt` 服务的处理延迟为 20 毫秒。商品管理服务（`ProductMgt` 服务）必须将截止时间的偏移量设置为 30 毫秒。因为库存服务（`Inventory` 服务）需要 30 毫秒来响应，所以截止时间的事件会在两个客户端上发生（`ProductMgt` 调用 `Inventory` 服务和客户端应用程序）。

`ProductMgt` 服务的业务逻辑将延迟时间增加了 20 毫秒。随后，`ProductMgt` 服务的调用逻辑触发了超出截止时间的场景，并且传播回客户端应用程序。因此，在使用截止时间时，要明确它们适用于所

有服务场景。

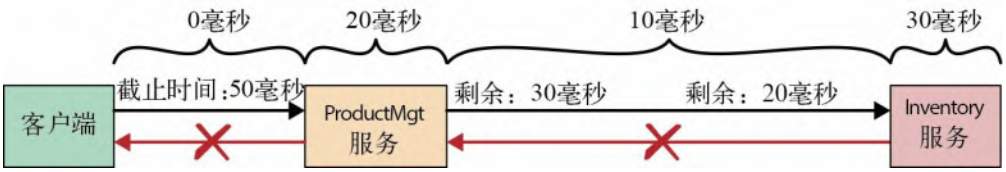

图 **5-3**：在调用服务时使用截止时间

客户端应用程序在初始化与 gRPC 的连接时，可以设置截止时间。当 RPC 发送之后，客户端应用程序会在截止时间所声明的时间范围内等待，如果在该时间内 RPC 没有返回，那么该 RPC 会以 `DEADLINE_EXCEEDED` 错误的形式终止。

接下来看在调用 gRPC 服务时使用截止时间的示例。在相同的 `OrderManagement` 服务使用场景中，假设 `AddOrder` RPC 要耗费较长的时间才能完成（通过在 `OrderManagement` gRPC 服务的 `AddOrder` 方法中引入延迟来模拟）。但是，客户端只会等待一定的时间，如果超过该时间，响应对它就没有用处了。假设 `AddOrder` 响应所占用的持续时间是 5 秒，但是客户端只等待 2 秒来获取响应。为了实现这一点（见代码清单 5-5 所示的 Go 代码片段），客户端应用程序可以通过 `context.WithDeadline` 操作设置 2 秒的超时时间。这里使用了 status 包来确定错误码，5.4 节会对其进行详细讨论。

代码清单 **5-5**　客户端应用程序的 gRPC 截止时间

```
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()

// 添加订单
```

```
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 在当前上下文中设置 2 秒的截止时间。

❷ 调用 `AddOrder` 远程方法并将可能出现的错误捕获到 `addErr` 中。

❸ 使用 status 包以确定错误码。

❹ 如果调用超出了指定的截止时间，它应该返回 `DEADLINE_EXCEEDED` 类型的错误。

该如何确定理想的截止时间值呢？这个问题并没有固定答案，但是在做出决策之前，需要考虑几个因素，主要包括所调用的每个服务的端到端延迟、支持串行模式的 RPC、支持并行模式的 RPC、底层网络的延迟以及下游服务的截止时间。在确定了最初的截止时间值之后，再根据 gRPC 应用程序的运行情况进行微调。

在 Go 语言中，设置 gRPC 截止时间是通过其中的 context 包实现的，其中 `WithDeadline` 是一个内置函数。Go 语言中的 context 包通常用来向下传递通用的数据，使其能够在整个下游操作中使用。当 gRPC 客户端应用程序发起调用时，客户端的 gRPC 库就会创建所需的 gRPC 头信息，用来表述客户端应用程序和服务器端应用程序之间的截止时间。在 Java 语言中，这略微有所差异，其实现直接来源于 io.grpc.stub.* 包的存根实现。可以使用 `blockingStub.withDeadlineAfter(long, java.util.concurrent.TimeUnit)` 设置 gRPC 的截止时间。也可以参考 Java 实现的源代码仓库了解更多信息。

在 gRPC 的截止时间方面，客户端和服务器端都可以对 RPC 是否成功做出自己的判断，这意味着它们的结论可能会不一致。例如，在前面的示例中，当客户端满足 `DEADLINE_EXCEEDED` 条件的时候，服务器端可能依然会试图做出响应。因此，服务器端应用程序需要判断当前 RPC 是否依然有效。在服务器端，还可以探测客户端何时达到调用 RPC 时所指定的截止时间。在 `AddOrder` 操作中，可以通过 `ctx.Err() == context.DeadlineExceeded` 来判断客户端是否已经满足超出截止时间的状态，随后就可以在服务器端废弃该 RPC 并返回一个错误。在 Go 语言中，这通常会通过非阻塞的 `select` 构造来实现。

与截止时间类似，在某些特定的情况下，客户端应用程序或服务器端应用程序可能要终止正在进行中的 gRPC 通信，这就要用到 gRPC 的取消功能了。

5.3 取消

在客户端应用程序和服务器端应用程序之间的 gRPC 连接中，客户端和服务器端都能够对调用是否成功在本地做出独立判断。例如，可以让同一个 RPC 在服务器端成功完成，但在客户端让其失败。类似地，在不同的情况下，客户端和服务器端可能会对同一个 RPC 得出不同的结论。但是，无论是客户端应用程序，还是服务器端应用程序，当希望终止 RPC 时，都可以通过取消该 RPC 来实现。一旦取消 RPC，就不能再进行与之相关的消息传递了，并且一方已经取消 RPC 的事实会传递到另一方。

在 Go 语言中，与截止时间类似，取消功能也是由 context 包实现的，其中 `WithCancel` 是一个内置函数。当 gRPC 应用程序调用该函数后，客户端的 gRPC 库会创建所需的 gRPC 头信息，表示客户端应用程序和服务器端应用程序之间的 gRPC 已终止。

以客户端应用程序和服务器端应用程序的双向流为例。在代码清单 5-6 所示的 Go 代码示例中，可以通过 `context.WithTimeout` 获取 `cancel` 函数。在得到 `cancel` 的引用之后，就可以在任何想终止 RPC 的地方调用它。

代码清单 **5-6**　gRPC 取消

```
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)

// 取消RPC
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", errProcOr
...
}
```

❶ 获取对 `cancel` 的引用。

❷ 调用流 RPC。

❸ 通过流发送消息给服务。

❹ 在客户端，取消 RPC（终止 RPC）。

❺ 当前上下文的状态。

❻ 当试图从已取消的上下文中接收消息时，会返回上下文已取消的错误。

当某一方取消 RPC 之后，另一方可以通过检查 `context` 来确定这一点。在本例中，服务器端应用程序可以通过使用 `stream.Context().Err() == context.Canceled`，来检查当前的上下文是否已经取消。

正如以上例子所示，在 RPC 中处理错误是一个非常常见的需求。5.4 节将详细介绍 gRPC 的错误处理技术。

5.4 错误处理

当发起 gRPC 调用时，客户端会接收成功状态的响应或者带有对应错误状态的错误。在编写客户端应用程序时，需要处理所有潜在的错误和错误条件。编写服务器端应用程序也需要处理错误，并生成适当的错误状态码。

当发生错误时，gRPC 返回一个错误状态码，并附带一条可选的错误消息，该消息提供错误条件的更多细节。状态对象由一个整型码和一条字符串消息组成，适用于不同语言的所有 gRPC 实现。

gRPC 使用一组定义良好的专用状态码，举例如下。

`OK`

成功状态，非错误。

`CANCELLED`

操作被取消，通常由调用者发起。

`DEADLINE_EXCEEDED`

在操作完成前，就已超过了截止时间。

`INVALID_ARGUMENT`

客户端指定了非法参数。

表 5-1 展示了可用的 gRPC 错误码以及每个错误码的描述。完整的错误码列表可以在 gRPC 官方文档或与 Go 和 Java 相关的文档中查看。

表**5-1：gRPC**错误码

错误码	数字	描述

OK	0	成功状态
CANCELLED	1	操作已被（调用者）取消
UNKNOWN	2	未知错误
INVALID_ARGUMENT	3	客户端指定了非法参数
DEADLINE_EXCEEDED	4	在操作完成前，就已超过了截止时间
NOT_FOUND	5	某些请求实体没有找到
ALREADY_EXISTS	6	客户端试图创建的实体已存在
PERMISSION_DENIED	7	调用者没有权限执行特定的操作
RESOURCE_EXHAUSTED	8	某些资源已被耗尽
FAILED_PRECONDITION	9	操作被拒绝，系统没有处于执行操作所需的状态
ABORTED	10	操作被中止
OUT_OF_RANGE	11	尝试进行的操作超出了合法的范围
UNIMPLEMENTED	12	在该服务中，未实现或不支持（未启用）本操作
INTERNAL	13	内部错误
UNAVAILABLE	14	该服务当前不可用
DATA_LOSS	15	不可恢复的数据丢失或损坏

UNAUTHENTICATED	16	客户端没有进行操作的合法认证凭证

gRPC 所提供的“开箱即用”的错误模型非常有限，并且与底层的 gRPC 数据格式无关，其中最常见的格式就是 protocol buffers。如果使用 protocol buffers 作为数据格式，那么可以利用 google.rpc 包所提供的更丰富的错误模型。但是，只有 C++、Go、Java、Python 和 Ruby 的库可以支持该错误，如果使用其他语言，则需要注意这一点。

下面来看在真实的 gRPC 错误处理场景中，对于这些理念的具体运用方式。在订单管理场景中，假设需要在 `AddOrder` 远程方法中处理非法订单 ID 请求。如代码清单 5-7 所示，假设给定的订单 ID 是 `-1`，然后需要生成一个错误并将其返回给消费者。

代码清单 **5-7**　服务器端的错误创建和传播

```
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 非法请求，需要生成一个错误并将其返回给客户端。

❷ 创建一个错误码为 InvalidArgument 的新错误状态。

❸ 包含错误类型 BadRequest_FieldViolation 的所有错误详情。可以在 GoDoc 网站上搜索 errdetails，了解更多关于 BadRequest_FieldViolation 的信息。

❹ 返回生成的错误。

通过 status 包，可以很容易地基于所需的错误码和详情创建错误状态。本例使用了 status.New(codes.InvalidArgument, "Invalid information received")，只需借助 return nil, errorStatus.Err() 将该错误发送回客户端即可。但是，为了包含更丰富的错误模型，可以使用 Google API 的 google.rpc 包。本例根据特定的错误类型设置了错误详情，该错误类型同样可以通过在 GoDoc 网站上搜索 errdetails 进行浏览。

至于客户端的错误处理，只需处理 RPC 返回的错误即可。例如，在代码清单 5-8 中，可以看到该订单管理场景中客户端应用程序的 Go 实现。这里调用了 AddOrder 方法，并将返回的错误赋值给 addOrderError 变量。因此，下一步就是探查 addOrderError 的结果并处理错误。为了实现这一点，可以获取在服务器端设置的错误码和特定的错误类型。

代码清单 **5-8**　客户端的错误处理

```
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", inf
                        default:
```

```
                                log.Printf("Unexpected error type: %s", inf
                        }
                }
        } else {
                log.Printf("Unhandled error : %s ", errorCode)
        }
} else {
        log.Print("AddOrder Response -> ", res.Value)
}
```

❶ 这是一个非法订单。

❷ 调用 `AddOrder` 远程方法并将错误赋值给 `addOrderError`。

❸ 使用 status 包获取错误码。

❹ 检查 `InvalidArgument` 错误码。

❺ 从错误中获取错误状态。

❻ 检查 `BadRequest_FieldViolation` 错误类型。

在 gRPC 应用程序中，尽可能使用适当的 gRPC 错误码和丰富的错误模型，这始终是一个最佳实践。gRPC 错误状态和详情通常会通过 trailer 头信息在传输层发送。

下面来看一下多路复用，这是针对同一个 gRPC 服务器端运行时的服务托管机制。

5.5 多路复用

关于 gRPC 服务和客户端应用程序，目前已经介绍了在给定的 gRPC 服务器端上注册唯一的 gRPC 服务，并且由单个客户端存根使用 gRPC 客户端进行连接。但是，gRPC 还允许在同一个 gRPC 服务器端上运行多个 gRPC 服务（见图 5-4），也允许多个客户端存根使用同一个 gRPC 客户端连接，这种功能叫作多路复用（multiplexing）。

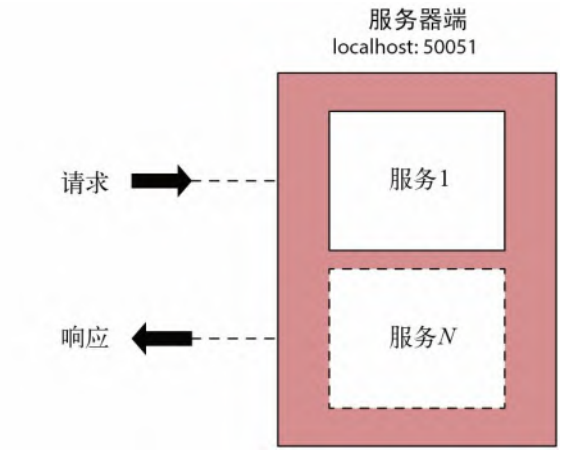

图 **5-4**：在同一台服务器上多路复用多个 **gRPC** 服务

例如，在 `OrderManagement` 服务示例中，假设为了满足订单管理需求，希望在同一个 gRPC 服务器端运行另一个服务，这样客户端就能重用同一个连接，从而按需调用这两个服务。通过对应的服务器端注册函数，也就是 `ordermgt_pb.RegisterOrderManagementServer` 和 `hello_pb.RegisterGreeterServer`，可以在同一个服务器端注册这两个服务，如代码清单 5-9 所示。

代码清单 **5-9**　两个 gRPC 服务共享同一个服务器端（`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

        // 在gRPC orderMgtServer上注册订单管理服务
        ordermgt_pb.RegisterOrderManagementServer(grpcServer, &orderMgtServ

        // 在gRPC orderMgtServer上注册问候服务
        hello_pb.RegisterGreeterServer(grpcServer, &helloServer{}) ❸

    ...
}
```

❶ 创建 gRPC 服务器端。

❷ 在 gRPC 服务器端注册 `OrderManagement` 服务。

❸ 在同一个 gRPC 服务器端注册 `Hello` 服务。

同理，通过客户端，可以在两个 gRPC 客户端存根间共享相同的 gRPC 连接。

如代码清单 5-10 所示，因为两个 gRPC 服务在同一个 gRPC 服务器端运行，所以可以创建一个 gRPC 连接，并在为两个服务创建 gRPC 客户端实例时使用该连接。

代码清单 **5-10**　两个 gRPC 客户端存根共享同一个连接（`grpc.ClientConn`）

```
// 建立到服务器端的连接
conn, err := grpc.Dial(address, grpc.WithInsecure()) ❶
...

orderManagementClient := pb.NewOrderManagementClient(conn) ❷

...

// 添加订单的RPC
        ...
res, addErr := orderManagementClient.AddOrder(ctx, &order1)

...


helloClient := hwpb.NewGreeterClient(conn) ❸
```

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

❶ 创建 gRPC 连接。

❷ 使用创建的 gRPC 连接来建立 `OrderManagement` 客户端。

❸ 使用相同的 gRPC 连接来建立 `Hello` 客户端。

对于多个服务或者多个存根使用相同的连接，这只涉及设计形式，与 gRPC 理念无关。在微服务等大多数日常使用场景中，通常并不会在两个服务间共享同一个 gRPC 服务器端。

在微服务架构中，gRPC 多路复用的一个强大的用途就是在同一个服务器端进程中托管同一个服务的多个主版本。这样做能够保证 API 在发生破坏性变更之后，依然能够适应遗留的客户端。一旦服务契约的旧版本不再有效，就可以在服务器端将其移除了。

5.6 节将讨论在客户端和服务器端通信的过程中，如何交换非 RPC 参数和非 RPC 响应的数据。

5.6 元数据

gRPC 应用程序通常会通过 gRPC 服务和消费者之间的 RPC 来共享信息。在大多数场景中，与服务业务逻辑和消费者直接相关的信息会作为远程方法调用参数的一部分，但在某些场景中，因为预期共享的关于 RPC 的信息可能与 RPC 业务上下文并没有关联，所以它们不应该作为 RPC 参数的一部分。在这样的场景中，可以使用 gRPC 元数据（gRPC metadata），元数据可以在 gRPC 服务或 gRPC 客户端发送和接收。如图 5-5 所示，在客户端或服务器端创建的元数据，可以通过 gRPC 头信息在客户端应用程序和服务器端应用程序之间进行交换。元数据的构造遵循键（字符串）–值对的形式。

元数据最常见的一个用途就是在 gRPC 应用程序之间交换安全头信息。与之类似，可以使用这种方式在 gRPC 应用程序之间交换任意类似信息。拦截器一般会大量使用 gRPC 元数据 API。下面将探讨 gRPC 如何支持在客户端和服务器端之间发送元数据。

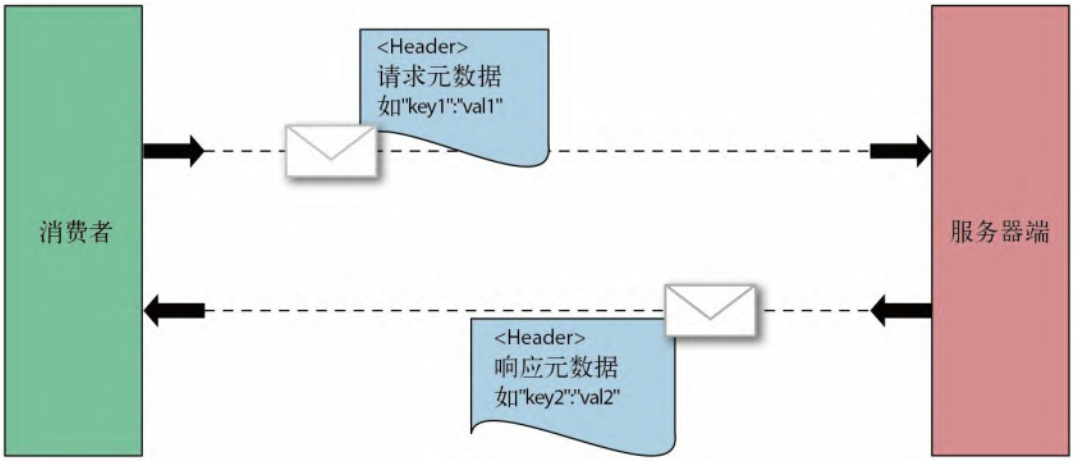

图 **5-5**：在客户端应用程序和服务器端应用程序之间交换元数据

5.6.1 创建和检索元数据

在 gRPC 应用程序中，创建元数据非常简单直接。在如下的 Go 代码片段中，可以发现两种创建元数据的方式。在 Go 语言中，元数据以正常的 `map` 形式来表述，可以通过

metadata.New(map[string]string{"key1": "val1", "key2": "val2"}) 格式进行创建。另外，还可以通过 metadata.Pairs 来创建元数据对，具有相同键的元数据会被合并为一个列表：

```
// 元数据创建：方案1
md := metadata.New(map[string]string{"key1": "val1", "key2": "val2"})

// 元数据创建：方案2
md := metadata.Pairs(
    "key1", "val1",
    "key1", "val1-2", // "key1"的map值为[]string{"val1", "val1-2"}
    "key2", "val2",
)
```

二进制数据也可以设置为元数据值。以元数据值形式所设置的二进制数据在发送之前会进行 base64 编码，在传输之后，则会进行解码。

在客户端或服务器端读取元数据，则可以通过传入的 RPC 上下文以 metadata.FromIncomingContext(ctx) 函数来实现，它会返回 Go 语言的元数据 map：

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

md, metadataAvailable := metadata.FromIncomingContext(ctx)
// 从"md"元数据map中读取所需的元数据
```

对于发送和接收元数据，下面分别来看客户端和服务器端针对不同的一元 RPC 和流 RPC 所用的方式。

5.6.2 发送和接收元数据：客户端

在客户端，要发送元数据到 gRPC 服务，可以创建元数据并将其设置到 RPC 上下文中。在 Go 实现中，可以采用两种方式来实现这一点。如代码清单 5-11 所示，可以使用 NewOutgoingContext 创建带有新元数据的上下文，或者使用 AppendToOutgoingContext 将元数据附加到已有的上下文中，但当使用 NewOutgoingContext 时会替换掉上下文中所有已有的元数据。在创建完带有元数据的上下文后，它就可以用于一元 RPC 或流 RPC 了。如第 4 章所述，在上下文中所设置的元数据会转换

成线路层的 gRPC 头信息（位于 HTTP/2 上）或 trailer。这样一来，在客户端发送这些头信息后，收件方会以头信息的形式接收它们。

代码清单 **5-11**　在 gRPC 客户端发送元数据

```
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

// 发送一元RPC
response, err := client.SomeRPC(ctxA, someRequest) ❹

// 也可以发送流RPC
stream, err := client.SomeStreamingRPC(ctxA) ❺
```

❶ 创建元数据。

❷ 基于新的元数据创建新的上下文。

❸ 在现有的上下文中附加更多的元数据。

❹ 一元 RPC 使用带有元数据的新上下文。

❺ 相同的上下文也可用于流 RPC。

因此，在客户端接收元数据的时候，需要将它们视为头信息或 trailer。在代码清单 5-12 中，可以看到为一元 RPC 和流 RPC 接收元数据的 Go 代码示例。

代码清单 **5-12**　在 gRPC 客户端接收元数据

```
var header, trailer metadata.MD ❶

// *****一元RPC*****

r, err := client.SomeRPC( ❷
    ctx,
```

```
    someRequest,
    grpc.Header(&header),
    grpc.Trailer(&trailer),
)

// 在这里处理头信息和trailer map

// *****流RPC*****

stream, err := client.SomeStreamingRPC(ctx)

// 检索头信息
header, err := stream.Header() ❸

// 检索trailer
trailer := stream.Trailer() ❹

// 在这里处理头信息和trailer map
```

❶ 用来存储 RPC 所返回的头信息和 trailer 的变量。

❷ 传递头信息和 trailer 引用来存储一元 RPC 所返回的值。

❸ 从流中获取头信息。

❹ 从流中获取 trailer，用于发送状态码和状态消息。

从对应的 RPC 操作获取值之后，就可以像一般的 `map` 那样对它们进行处理，进而处理所需的元数据。

接下来看一下如何在服务器端发送和接收元数据。

5.6.3 发送和接收元数据：服务器端

在服务器端接收元数据非常简单直接。使用 Go 语言，只需在远程方法调用中使用 `metadata.FromIncomingContext(ctx)`，即可读取元数据（见代码清单 5-13）。

代码清单 **5-13** 在 gRPC 服务器端读取元数据

```
func (s *server) SomeRPC(ctx context.Context,
    in *pb.someRequest) (*pb.someResponse, error) { ❶
```

```
    md, ok := metadata.FromIncomingContext(ctx) ❷
    // 使用元数据执行某些操作
}

func (s *server) SomeStreamingRPC(
    stream pb.Service_SomeStreamingRPCServer) error { ❸
    md, ok := metadata.FromIncomingContext(stream.Context()) ❹
    // 使用元数据执行某些操作
}
```

❶ 一元 RPC。

❷ 从远程方法传入的上下文中读取元数据 map。

❸ 流 RPC。

❹ 从流中获取上下文并从中读取元数据。

要从服务器端发送元数据，可以根据元数据发送头信息或者设置 trailer。创建元数据的方法与前文讨论的相同。在代码清单 5-14 中，可以看到在服务器端自一元 RPC 和流 RPC 远程方法中发送元数据的 Go 代码示例。

代码清单 **5-14**　在 gRPC 服务器端发送元数据

```
func (s *server) SomeRPC(ctx context.Context,
    in *pb.someRequest) (*pb.someResponse, error) {
    // 创建并发送头信息
    header := metadata.Pairs("header-key", "val")
    grpc.SendHeader(ctx, header) ❶
    // 创建并设置trailer
    trailer := metadata.Pairs("trailer-key", "val")
    grpc.SetTrailer(ctx, trailer) ❷
}

func (s *server) SomeStreamingRPC(stream pb.Service_SomeStreamingRPCServer)
    // 创建并发送头信息
    header := metadata.Pairs("header-key", "val")
    stream.SendHeader(header) ❸
    // 创建并设置trailer
    trailer := metadata.Pairs("trailer-key", "val") stream.SetTrailer(trail
}
```

❶ 以头信息的形式发送元数据。

❷ 和 trailer 一起发送元数据。

❸ 在流中，以头信息的形式发送元数据。

❹ 和流的 trailer 一起发送元数据。

在一元 RPC 和流 RPC 这两种场景中，都可以通过 `grpc.SendHeader` 来发送元数据。如果想将元数据作为 trailer 的一部分发送，则需要通过 `grpc.SetTrailer` 或对应流的 `SetTrailer` 方法，将元数据设置为上下文 trailer 中的一部分。

接下来看一下调用 gRPC 应用程序所涉及的另外一项常用的技术：命名解析。

5.6.4 命名解析器

命名解析器（name resolver）接受一个服务的名称并返回后端 IP 的列表。代码清单 5-15 所使用的解析器会将 `lb.example.grpc.io` 解析为 `localhost:50051` 和 `localhost:50052`。

代码清单 **5-15** gRPC 命名解析器的 Go 语言实现

```
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 命名解析器构建器。

❷ 创建解析 `lb.example.grpc.io` 的示例解析器。

❸ 将 `lb.example.grpc.io` 解析为 `localhost:50051` 和 `localhost:50052`。

❹ 为 `example` 模式创建的解析器。

❺ 命名解析器的结构。

基于这个命名解析器实现，可以为所选的任意服务注册中心实现解析器，如 Consul、etcd 和 Zookeeper。gRPC 负载均衡的需求可能非常依赖所使用的部署模式或使用场景。随着容器编排平台（如 Kubernetes）和更高层次抽象（如服务网格）越来越普及，在客户端实现负载均衡逻辑的需求变得越来越小。第 7 章将探索一些在本地容器和 Kubernetes 上部署 gRPC 应用程序的最佳实践。

下面先来看一下 gRPC 应用程序最常见的需求之一，也就是负载均衡，在其中某些特定的情况下，可以使用命名解析器。

5.7 负载均衡

在开发生产级 gRPC 应用程序时，通常需要确保该应用程序能够满足高可用性和高扩展性的需求。因此，在生产环境中，始终需要多个 gRPC 服务器端。在这些服务之间分发 RPC 需要由某个实体来处理，这就需要使用负载均衡器了。gRPC 通常使用两种主要的负载均衡机制：负载均衡器代理和客户端负载均衡。这里先从负载均衡器代理开始讨论。

5.7.1 负载均衡器代理

如图 5-6 所示，在代理负载均衡场景中，客户端向负载均衡器代理发起 RPC。随后，负载均衡器代理将 RPC 分发给一台可用的后端 gRPC 服务器，该后端 gRPC 服务器实现了满足服务调用的逻辑。负载均衡器代理会跟踪每台后端服务器的负载，并为后端服务分配负载提供不同的负载均衡算法。

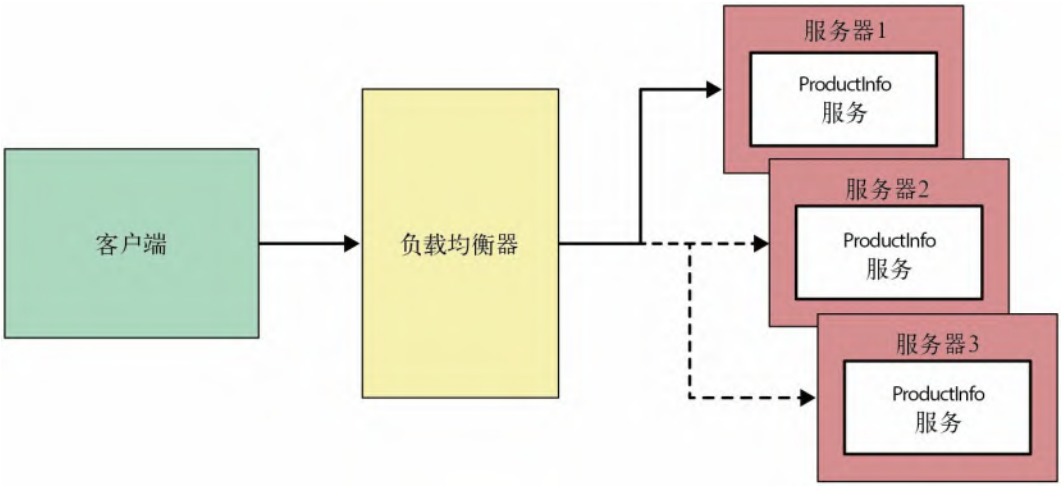

图 **5-6**：客户端应用程序调用面向多个 **gRPC** 服务的负载均衡器

后端服务的拓扑结构对 gRPC 客户端是不透明的，它们只知道负载均衡器的端点就可以了。因此，为了满足负载均衡的使用场景，除了使用负载均衡器作为 gRPC 连接的目的地外，在客户端无须任何变更。后端服务可以将负载情况报告给负载均衡器，这样它就能使用该信息确定负载均衡的逻辑。

在理论上，可以选择任意支持 HTTP/2 的负载均衡器作为 gRPC 应用程序的负载均衡器代理。但是，它必须完全支持 HTTP/2。因此，选择明确提供 gRPC 支持的负载均衡器是很明智的做法。例如，可以使用 Nginx 代理、Envoy 代理等作为 gRPC 应用程序的负载均衡器代理。

如果不使用 gRPC 负载均衡器，那么可以在编写的客户端应用程序中实现负载均衡逻辑。下面来更详细地了解客户端负载均衡。

5.7.2 客户端负载均衡

这个方案不再借助负载均衡的中间代理层，而是在 gRPC 客户端层实现负载均衡的逻辑。在这种方法中，客户端要知道多台后端 gRPC 服务器，并为每个 RPC 选择一台后端 gRPC 服务器。如图 5-7 所示，负载均衡逻辑可以完全作为客户端应用程序（也被称为厚客户端）的一部分来进行开发，也可以实现为一个专用的服务器端，叫作后备负载均衡器。客户端可以查询它，从而选择最优的 gRPC 服务器来进行连接。客户端直接连接到选定的 gRPC 服务器，其地址从后备负载均衡器获取。

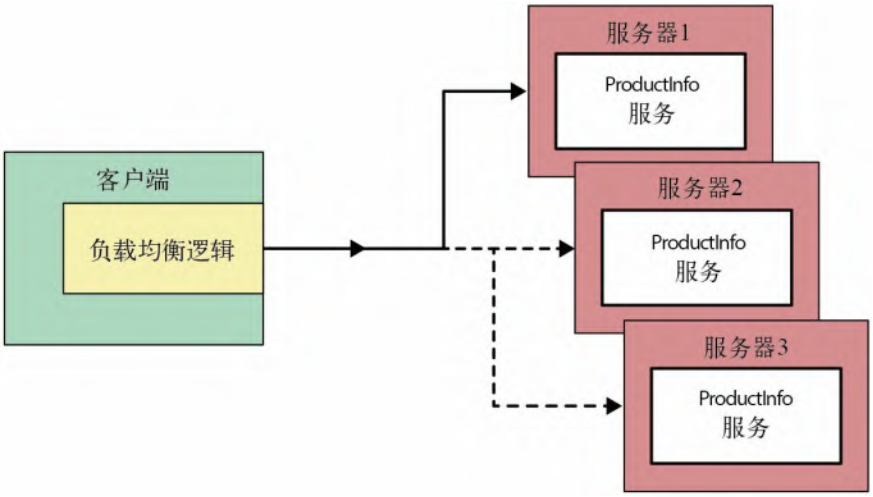

图 **5-7**：客户端负载均衡

为了理解客户端负载均衡的实现方式，这里看一个使用 Go 语言的厚客户端实现。在该使用场景中，假设有两个后端 gRPC 服务，它们分别在 :50051 和 :50052 上运行 echo 服务器端。这些 gRPC 服务在 RPC 响应中会包含提供服务的服务器地址。因此，可以将这两个服务作为一个 echo gRPC 服务集群的两个成员。现在，假设希望构建一个 gRPC 客户端应用程序，该应用程序在选择 gRPC 服务器端点的时候使用轮询调度算法

（round-robin algorithm，轮流执行每个端点），而另一个客户端始终选择第一个服务器端点。代码清单 5-16 展示了厚客户端负载均衡实现。可以看到，客户端访问了 example:///lb.example.grpc.io，这里使用 `example` 模式名和 `lb.example.grpc.io` 作为服务器名称。基于该模式，它会查找命名解析器来发现后端服务地址的绝对值。根据命名解析器所返回的值列表，gRPC 针对这些服务器端运行不同的负载均衡算法。该行为是通过 `grpc.WithBalancerName("round_robin")` 配置的。

代码清单 **5-16**　使用厚客户端的客户端负载均衡

```
pickfirstConn, err := grpc.Dial(
                fmt.Sprintf("%s:///%s",
        // exampleScheme       = "example"
        //     exampleServiceName = "lb.example.grpc.io"
        exampleScheme, exampleServiceName), ❶
        // pick_first是默认选项 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)

// 使用round_robin策略生成另一个ClientConn
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 使用模式和服务名创建 gRPC 连接。模式是通过模式解析器解析的，它是客户端应用程序的一部分。

❷ 指定负载均衡算法，该算法会使用服务器端点列表中的第一个服务器端。

❸ 使用轮询调度算法。

gRPC 有两个默认支持的负载均衡算法：`pick_first` 和 `round_robin`。`pick_first` 会尝试连接第一个地址，如果能够连接成功，就会将该地址用于所有的 RPC；如果失败，则会尝试下一个地址。`round_robin` 会连接所有地址，并会按顺序每次向后端发送一个 RPC。

在代码清单 5-16 所示的客户端负载均衡场景中，有一个解析 `example` 模式的模式解析器，它包含发现端点 URL 实际值的逻辑。下面讨论关于压缩的话题。对于通过 RPC 发送大量数据的场景来说，这是 gRPC 另一个常用的特性。

5.7.3 压缩

为了高效利用网络带宽，在执行客户端和服务之间的 RPC 时，可以使用压缩技术。如果要在客户端使用压缩技术，那么可以通过在发送 RPC 时设置一个压缩器来实现。例如，在 Go 语言中，借助 `client.AddOrder(ctx, &order1, grpc.UseCompressor(gzip.Name))` 便可以很容易地实现。可以在 GoDoc 网站上搜索 encoding/gzip 来了解更多信息。

在服务器端，已注册的压缩器会自动解码请求消息，并编码响应消息。在 Go 语言中，注册压缩器只需在 gRPC 服务器端应用程序中导入 GoDoc 网站上的 gzip 包即可（获取方式同上）。服务器端始终会使用客户端所指定的压缩方法。如果对应的压缩器没有注册，则会向客户端返回一个 `Unimplemented` 状态。

5.8 小结

在构建生产级 gRPC 应用程序时，除了定义服务接口、生成服务器端代码和客户端代码并实现业务逻辑外，通常还需要提供各种额外的特性。如本章所述，gRPC 提供了构建 gRPC 应用程序所需的各种功能，包括拦截器、截止时间、取消和错误处理。

但是，本书目前还没有谈到如何保护 gRPC 应用程序以及如何消费它们，这些便是第 6 章要介绍的内容。

第 6 章　安全的 gRPC

基于 gRPC 的应用程序会通过网络彼此进行远程通信，这需要每个 gRPC 应用程序向其他需要与之通信的应用程序暴露其入口点。从安全的角度来看，这并不是一件好事。拥有的入口点越多，攻击面就越广，受到攻击的风险也就越高。因此，保护通信和保护入口点的安全对于任何真实的使用场景都至关重要。每个 gRPC 应用程序都必须能够处理加密的消息，加密所有节点间的通信，并对所有消息进行认证和签名等。

本章将介绍一组安全基础措施和模式，以应对我们在启用应用级安全性时所面临的挑战。简而言之，我们将探索如何保护微服务之间的通信通道，并对用户进行认证和访问控制。让我们从保护通信通道开始讨论吧。

6.1 使用TLS认证gRPC通道

传输层安全协议（transport layer security，TLS）旨在为两个应用程序之间的通信提供隐私性和数据完整性。在这里，它可用于在 gRPC 客户端应用程序和服务器端应用程序之间提供安全连接。根据传输层安全协议规范，如果客户端和服务器端之间的连接是安全的，那么它应该具备以下一项或两项特性。

连接是私密的

使用对称加密的方式进行数据加密。在这种类型的加密中，只用一个密钥加密和解密。对于每个连接，这些密钥都是唯一的，它们是基于会话开始时所协商的共享密钥生成的。

连接是可靠的

这之所以能够实现，是因为每条消息都包含消息完整性检查，以防止在传输期间出现未被检测到的数据丢失或数据修改。

可见，通过安全的连接发送数据非常重要。借助 TLS 保护 gRPC 连接并不难，因为这种认证机制内置在了 gRPC 库中，它还促使我们使用 TLS 对数据交换进行认证和加密。

我们该如何在 gRPC 连接中启用 TLS 呢？客户端和服务器端之间的安全数据传输可以采用单向或双向（也称为相互 TLS 或 mTLS）的方式来实现。下面讨论如何按照每种方式启用 TLS。

6.1.1 启用单向安全连接

在单向连接中，只有客户端会校验服务器端，以确保它所接收的数据来自预期的服务器。在建立连接时，服务器端会与客户端共享其公开证书，客户端则会校验接收到的证书。这是通过证书授权中心（certificate authority，CA）完成的，也就是 CA 签署的证书。证书校验之后，客户端会发送使用密钥加密的数据。

CA 是一个受信任的实体，它管理和发布用于公共网络中安全通信的安

全证书和公钥。由该受信任实体所签署或颁发的证书称为 CA 签名的证书。

要启用 TLS，首先需要创建以下证书和密钥。

`server.key`

RSA 私钥，用于签名和认证公钥。

`server.pem/server.crt`

用于分发的自签名 X.509 公钥。

RSA 是其三位发明者的首字母组成的缩写：Rivest、Shamir 和 Adleman。RSA 是最流行的公钥密码系统之一，广泛应用于安全数据传输。在 RSA 中有一个每个人都可以知道的公钥来加密数据，一个私钥来解密数据。其理念是，使用公钥加密的消息只能在合理的时间内通过私钥解密。

为了生成密钥，我们可以使用 OpenSSL，这是一个适用于 TLS 和安全套接字层（secure socket layer，SSL）协议的开源工具集。它能够生成具有不同长度和密码的私钥以及公开证书等。另外，还有其他一些工具，如 mkcert 和 certstrap，它们也能很容易地生成密钥和证书。

这里不会详细描述如何生成自签名证书的密钥，因为生成这些密钥和证书的详细步骤在源代码仓库的 README 文件中进行了描述。

假设我们已经创建了私钥和公共证书，接下来将它们用于第 1 章和第 2 章所讨论的在线商品管理系统，并保护 gRPC 服务器端和客户端之间的通信。

01. 在 **gRPC** 服务器端启用单向安全连接

    这是加密客户端和服务器端通信的最简单的方式。在这里，服务器端需要使用一个公钥–私钥对进行初始化。我们将阐述如何使用 gRPC Go 服务器实现这一点。

    为了启用安全的 Go 服务器，需要更新服务器实现的主函数，如代

码清单 6-1 所示。

代码清单 **6-1**　用于托管 `ProductInfo` 服务的安全 gRPC 服务器实现

```
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 读取和解析公钥–私钥对，并创建启用 TLS 的证书。

❷ 添加证书作为 TLS 服务器凭证，从而为所有传入的连接启用 TLS。

❸ 通过传入 TLS 服务器凭证来创建新的 gRPC 服务器实例。

❹ 通过调用生成的 API，将服务实现注册到新创建的 gRPC 服务器上。

❺ 在端口 50051 上创建 TCP 监听器。

❻ 绑定 gRPC 服务器到监听器，并开始监听端口 50051 上传入的消息。

现在已经修改了服务器，使其能够接收来自客户端的请求，客户端可以验证服务器的证书。再修改一下客户端代码，使其能够与服务器“交流”。

02. 在 **gRPC** 客户端启用单向安全连接

为了与服务器连接，客户端需要服务器端的自认证公钥。我们可以修改 Go 的客户端代码以连接服务器，如代码清单 6-2 所示。

代码清单 **6-2**　安全的 gRPC 客户端应用程序

```
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
  ... // 省略了RPC方法调用
}
```

❶ 读取并解析公开证书，创建启用 TLS 的证书。

❷ 以 `DialOption` 的形式添加传输凭证。

❸ 通过传入 `dial` 选项，建立到服务器的安全连接。

❹ 传入连接并创建存根。该存根实例包含了调用服务器的所有远程方法。

❺ 所有事情完成后关闭连接。

这是一个非常简单直接的过程。只需添加 3 行代码并修改原始代码中的一行代码就可以了。首先，根据服务器端的公钥文件创建凭据对象，然后将传输凭证传递到 gRPC `dialer` 中，这样客户端每次建立与服务器之间的连接时就会启用 TLS 握手。

在单向 TLS 中，我们只认证服务器的身份。下一节会对双方（客户端和服务器端）都进行认证。

6.1.2 启用mTLS保护的连接

客户端和服务器端采用 mTLS 连接的主要目的是，控制能够连接服务器

端的客户端。与单向安全连接不同，这种方式会将服务器配置为仅接受来自一组范围有限、已验证的客户端的连接。在这种方式中，双方彼此共享公开证书，并校验对方的身份。连接的基本流程如下所示。

01. 客户端发送一个请求，试图访问服务器端受保护的信息。
02. 服务器端发送它的 X.509 证书给客户端。
03. 客户端通过 CA 对接收到的证书进行校验，判断是否为 CA 签名的证书。
04. 如果校验成功，则客户端发送其自身的证书到服务器端。
05. 服务器端也通过 CA 验证客户端证书。
06. 验证成功之后，服务器端就允许客户端访问受保护的数据了。

为了在示例中启用 mTLS，我们要了解如何处理客户端和服务器端的证书问题。首先需要创建一个具有自签名证书的 CA，还需为客户端和服务器端创建证书签名请求，并且需要使用我们的 CA 对它们进行签名。与单向安全连接的示例一样，可以使用 OpenSSL 工具生成密钥和证书。

假设我们已经具有了启用客户端–服务器端 mTLS 通信所需的全部证书。如果正确生成了它们，那么在工作空间中会创建以下密钥和证书。

`server.key`

服务器端的 RSA 私钥。

`server.crt`

服务器端的公开证书。

`client.key`

客户端的 RSA 私钥。

`client.crt`

客户端的公开证书。

`ca.crt`

CA 的公开证书，用来签名所有的公开证书。

我们首先修改示例中的服务器端代码，以便于直接创建 X.509 密钥对，并基于 CA 公钥创建证书池。

01. 在 **gRPC** 服务器端启用 **mTLS**

要为 Go 服务器启用 mTLS，需要更新服务器实现的主函数，如代码清单 6-3 所示。

代码清单 **6-3**　用 Go 语言编写的用于托管 `ProductInfo` 服务的安全 gRPC 服务器实现

```
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{
    // 为所有传入的连接启用TLS
    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 通过服务器端的证书和密钥直接创建 X.509 密钥对。

❷ 通过 CA 创建证书池。

❸ 将来自 CA 的客户端证书附加到证书池中。

❹ 通过创建 TLS 凭证为所有传入的连接启用 TLS。

❺ 通过传入的 TLS 服务器凭证创建新的 gRPC 服务器实例。

❻ 通过调用生成的 API 将 gRPC 服务注册到新创建的 gRPC 服务器上。

❼ 在端口 50051 上创建 TCP 监听器。

❽ 绑定 gRPC 服务器到监听器，并开始在端口 50051 上监听传入的消息。

我们已经修改了服务器端，让它只接受已验证客户端的请求。接下来修改客户端代码，使其能够和服务器“交流”。

02. 在 **gRPC** 客户端启用 **mTLS**

为了让客户端能够进行连接，客户端代码需要遵循和服务器端代码类似的步骤。可以修改 Go 客户端代码，如代码清单 6-4 所示。

代码清单 **6-4**　用 Go 语言编写的安全的 gRPC 客户端应用程序

```
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, // 注意，这是必需的!
        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

  ... // 省略了RPC方法调用
}
```

❶ 通过服务器端的证书和密钥直接创建 X.509 密钥对。

❷ 通过 CA 创建证书池。

❸ 将来自 CA 的客户端证书附加到证书池中。

❹ 添加传输凭证作为连接选项。这里，ServerName 必须与证书中的 Common Name 一致。

❺ 传入连接选项，搭建到服务器的安全连接。

❻ 传入连接并创建存根。该存根实例包含调用服务器的所有远程方法。

❼ 所有事情完成后关闭连接。

现在，我们使用单向 TLS 和 mTLS 搭建了 gRPC 应用程序客户端和服务器端的安全通信通道。下一步是在每次调用的时候启用认证，这意味着凭证信息要附加到调用上。每次客户端调用都带有认证凭证，服务器端检查调用的凭证，并决定是允许还是拒绝客户端的调用。

6.2 对gRPC调用进行认证

gRPC 使用严格的认证机制。前文介绍了如何使用 TLS 实现客户端和服务器端的加密数据交换。下面将讨论如何验证调用者的身份，并使用不同的调用凭证技术（如基于令牌的认证等）实现访问控制功能。

为了方便对调用者进行验证，gRPC 为客户端提供了在每次调用中插入凭证（如用户名和密码）的功能。gRPC 服务器端能够拦截来自客户端的请求，并检查每一个传入调用的凭证。

下面将先介绍一个简单的认证场景，从而阐释对每个客户端调用进行认证的方式。

6.2.1 使用basic认证

basic 认证是最简单的认证机制。在这种机制中，客户端发送的请求带有 `Authorization` 头信息，该头信息的值以单词 `Basic` 开头，随后是一个空格和 base64 编码的字符串 `<` 用户名 `>:<` 密码 `>`。如果用户名和密码均为 `admin`，那么头信息将如下所示：

```
Authorization: Basic YWRtaW46YWRtaW4=
```

总体而言，gRPC 并不提倡使用用户名/密码来对服务进行认证。这是因为，相对于 JSON Web Token（JWT）和 OAuth2 Access Token 等其他令牌，用户名/密码没有时间方面的限制。这意味着当生成一个令牌时，我们可以指定它的有效时间，但对于用户名/密码，则不能指定它的有效期。在我们更改密码之前，它始终是有效的。如果需要在应用程序中启用 basic 认证，建议在客户端和服务器端之间的安全连接中共享基本凭证。我们选择 basic 认证，是为了能更方便地阐述 gRPC 中的认证原理。

我们先讨论如何将用户凭证以 basic 认证的方式注入调用之中。因为在 gRPC 中没有内置的 basic 认证支持，所以需要将其以自定义凭证的形式添加到客户端上下文中。在 Go 语言中，可以很容易地实现这一点，只需要定义一个凭证结构体并实现 `PerRPCCredentials` 接口，如代码清

单 6-5 所示。

代码清单 **6-5**　实现 PerRPCCredentials 接口以传递自定义凭证

```
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 定义结构体来存放要注入 RPC 的字段集合（在我们的场景中，也就是用户的凭证，如用户名和密码）。

❷ 实现 GetRequestMetadata 方法，并将用户凭证转换成请求元数据。在我们的场景中，键是 Authorization，值则由 Basic 和加上 < 用户名 >:< 密码 > 的 base64 算法计算结果所组成。

❸ 声明在传递凭证时是否需要启用通道安全性。如前所述，建议启用。

实现完凭证对象后，需要使用合法的凭证对其进行初始化，并在建立连接时将其传递进去，如代码清单 6-6 所示。

代码清单 **6-6**　使用 basic 认证的安全 gRPC 客户端应用程序

```
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)

  ... // 省略了RPC方法调用
}
```

❶ 使用有效的用户凭证（用户名和密码）初始化 auth 变量。auth 变量存放了我们要使用的值。

❷ 传递 auth 变量给 grpc.WithPerRPCCredentials 函数。该函数接受一个接口作为参数。因为我们定义的认证结构符合该接口，所以可以传递变量。

现在，客户端在调用服务器端的时候加入了额外的元数据，但服务器端还没有注意到这一点。因此，我们需要告诉服务器端检查元数据。接下

来更新服务器端，使其读取元数据，如代码清单 6-7 所示。

代码清单 **6-7**　支持 basic 认证校验的安全 gRPC 服务器端

```
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 metada
  errInvalidToken    = status.Errorf(codes.Unauthenticated, "invalid creden
)

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{
    // 为所有传入的连接启用TLS
    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
  }
  // 在确保令牌合法之后，继续执行handler
  return handler(ctx, req)
}
```

❶ 通过 TLS 服务器证书添加新的服务器选项（grpc.ServerOption）。grpc.UnaryInterceptor 是一个函数，我们在其中添加拦截器来拦截所有来自客户端的请求。我们向该函数传递一个引用（ensureValidBasicCredentials），拦截器会将所有的客户端请求传递给该函数。

❷ 定义名为 ensureValidBasicCredentials 的函数来校验调用者的身份。在这里，context.Context 对象包含所需的元数据，在请求的

生命周期内，该元数据会一直存在。

❸ 从上下文中抽取元数据，获取 `authentication` 的值并校验凭证。由于 `metadata.MD` 中的键会被标准化为小写字母，因此需要检查键的值。

现在，服务器端已经能够校验每个调用中的客户端身份了。这是一个非常简单的示例。在服务器端拦截器中，可以包含非常复杂的认证逻辑以校验客户端身份。

我们基本了解了如何为每个请求进行客户端认证，接下来讨论常用且推荐使用的 OAuth 2.0，它是基于令牌的认证机制。

6.2.2 使用OAuth 2.0

OAuth 2.0 是一个用于访问委托的框架。它允许用户以自己的名义授予服务有限的访问权限，而不会像用户名和密码方式那样给予服务全部访问权限。在这里，我们不会详细讨论什么是 OAuth 2.0。如果你掌握 OAuth 2.0 的基础知识，那么更容易理解如何在应用程序中启用该功能。

在 OAuth 2.0 的流程中，有 4 个主要的角色：客户端、授权服务器、资源服务器和资源所有者。客户端要访问资源服务器上的资源。为了访问资源，客户端需要获取一个来自授权服务器的令牌（这是任意的一个字符串）。这个令牌必须具备恰当的长度，并且应该是不可预知的。客户端接收到该令牌之后，就可以使用它向资源服务器发送请求了。随后，资源服务器会与对应的授权服务器通信，并校验该令牌。如果该资源所有者校验了它，那么客户端就可以访问该资源。

gRPC 提供了在应用程序中启用 OAuth 2.0 的内置支持。我们先讨论如何将令牌注入调用中。因为在示例中，并没有授权服务器，所以我们硬编码任意的一个字符串来作为令牌的值。代码清单 6-8 展示了如何将 OAuth 令牌添加到客户端请求中。

代码清单 **6-8** 用 Go 语言编写的使用 OAuth 令牌的安全 gRPC 客户端应用程序

```
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)

  ... // 省略了RPC方法调用
}

func fetchToken() *oauth2.Token {
  return &oauth2.Token{
    AccessToken: "some-secret-token",
  }
}
```

❶ 设置连接的凭证，需要提供 OAuth 令牌值来创建凭证。这里使用一个硬编码的字符串值作为令牌的值。

❷ 配置 gRPC `DialOption`，为同一个连接的所有 RPC 使用同一个令牌。如果想为每个调用使用专门的 OAuth 令牌，那么需要使用 `CallOption` 配置 gRPC 调用。

需要注意，我们还启用了通道安全性，这是因为 OAuth 需要底层传输安全。在 gRPC 内部，所提供的令牌会以令牌类型作为前缀，并以 `authorization` 作为键附加到元数据上。

在服务器端，我们添加类似的拦截器，来检查和校验请求所带来的客户端令牌，如代码清单 6-9 所示。

代码清单 **6-9**　使用 OAuth 用户令牌校验的安全 gRPC 服务器端

```
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"
)

// 用来实现ecommerce/product_info的服务器
type server struct {
  productMap map[string]*pb.Product
}

var (
  port = ":50051"
  crtFile = "server.crt"
  keyFile = "server.key"
  errMissingMetadata = status.Errorf(codes.InvalidArgument, "missing metada
```

```
  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.Unar
     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 添加新的服务器选项（`grpc.ServerOption`）以及 TLS 服务器证

书。借助 grpc.UnaryInterceptor 函数，添加拦截器以拦截所有来自客户端的请求。

❷ 定义名为 ensureValidToken 的函数来校验令牌。如果令牌丢失或不合法，则拦截器会阻止执行并提示错误；否则，拦截器调用传递上下文和接口的下一个 handler。

可以使用拦截器为所有 RPC 配置令牌校验。根据服务的类型，服务器端可能会配置 grpc.UnaryInterceptor 或 grpc.StreamInterceptor。

与 OAuth 2.0 认证类似，gRPC 还支持基于 JWT 的认证。下面讨论启用 JWT 认证所需要做的变更。

6.2.3 使用JWT

JWT 定义了一个在客户端和服务器端传输身份信息的容器。签名的 JWT 可用作自包含的访问令牌，这意味着资源服务器无须与授权服务器通信来验证客户端的令牌，它可以通过验证签名来校验令牌。客户端请求访问授权服务器，授权服务器校验客户端的凭证，创建 JWT 并将其发送给客户端。带有 JWT 的客户端应用程序就允许访问资源了。

gRPC 内置了对 JWT 的支持。如果具有来自授权服务器的 JWT 文件，则需要传递该文件并创建 JWT 凭证。代码清单 6-10 说明了如何从 JWT 令牌文件（token.json）创建 JWT 凭证，并在 Go 客户端应用程序中将它们作为 DialOption 进行传递。

代码清单 **6-10** 在 Go 客户端应用程序中使用 JWT 建立连接

```
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),
  // 传输凭证
  grpc.WithTransportCredentials(creds), ❷
}

// 建立到服务器的连接
conn, err := grpc.Dial(address, opts...)
if err != nil {
  log.Fatalf("did not connect: %v", err)
}
  ... // 省略了存根生成和RPC方法调用
```

❶ 调用 `oauth.NewJWTAccessFromFile` 初始化 `credentials.PerRPCCredentials`，需要提供一个有效的令牌文件来创建凭证。

❷ 使用 `DialOption WithPerRPCCredentials` 配置 gRPC dial，为相同连接的所有 RPC 使用同一个 JWT 令牌。

除了这些认证技术之外，还可以在客户端扩展 RPC 凭证，并在服务器端添加新的拦截器，从而添加任意的认证机制。gRPC 还为部署在 Google Cloud 上的 gRPC 服务提供了特殊的内置支持。下面讨论如何调用这些服务。

6.2.4 使用基于令牌的谷歌认证

识别用户，并决定是否允许他们访问部署在谷歌云平台上的服务，该平台是由可扩展服务代理（extensible service proxy，ESP）控制的。ESP 支持多种认证方法，包括 Firebase、Auth0 和 Google ID Token。不管使用哪种方法，客户端都需要在它们的请求中包含一个有效的 JWT。为了生成认证 JWT，我们必须为每个部署的服务创建一个服务账号。

获取到服务的 JWT 令牌之后，就可以通过和请求一起发送令牌来调用服务方法了。我们可以在创建通道时将凭证传递进来，如代码清单 6-11 所示。

代码清单 **6-11** 在 Go 客户端应用程序中使用谷歌端点建立连接

```
perRPC, err := oauth.NewServiceAccountFromFile("service-account.json", scop
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)
}
... // 省略了存根生成和RPC方法调用
```

❶ 调用 `oauth.NewServiceAccountFromFile` 来初始化 `credentials.PerRPCCredentials`。需要提供一个有效的令牌文件来创建凭证。

❷ 与之前讨论的认证机制类似，我们使用 `DialOption` `WithPerRPCCredentials` 配置 gRPC dial，从而将认证令牌作为元数据应用于相同连接的所有 RPC。

6.3 小结

生产级 gRPC 应用程序必须至少满足最低的安全要求，以确保客户端和服务器端之间的安全通信。gRPC 库旨在支持不同类型的认证机制，并能够通过添加自定义的认证机制来进行扩展。这样一来，gRPC 便能够很容易地与其他系统安全地进行交互。

gRPC 提供了两种凭证支持：通道和调用。通道凭证是附加到 TLS 这样的通道上的。调用凭证是附加到调用上的，如 OAuth 2.0 令牌、basic 认证等。我们甚至可以将这两种凭证类型都用到一个 gRPC 应用程序中。例如，可以让 TLS 启用客户端和服务器端之间的连接，同时在连接的每个 RPC 上附加凭证信息。

本章介绍了如何在 gRPC 应用程序中启用两种凭证类型。第 7 章将扩展前文介绍的理念和技术，以便于构建在生产环境中运行的 gRPC 应用程序；还将讨论如何为服务和客户端应用程序编写测试用例，如何在 Docker 和 Kubernetes 上部署应用程序，以及如何在生产环境中观察系统的运行情况。

第 7 章　在生产环境中运行 gRPC

前面的章节涵盖了设计和开发 gRPC 应用程序的各个方面。现在该深入研究在生产环境中运行 gRPC 的细节了。本章将讨论如何为 gRPC 服务和客户端开发单元测试和集成测试，以及如何将它们与持续集成工具集成到一起。随后，将转向 gRPC 应用程序的持续部署，这里会探讨一些在虚拟机（VM）、Docker 和 Kubernetes 上的部署模式。最后，为了在生产环境中运行 gRPC 应用程序，需要有一个坚实的可观察性平台。在此方面，我们会讨论面向 gRPC 应用程序的多种可观察性工具，还会探讨 gRPC 应用程序的问题排查和调试技术。下面从测试这些应用程序开始进行讨论。

7.1 测试gRPC应用程序

开发任何软件应用程序（包括 gRPC 应用程序）都要有和应用程序相关联的单元测试。gRPC 应用程序始终会与网络交互，测试应该涵盖服务器端和客户端 gRPC 应用程序的网络方面。我们首先测试 gRPC 服务器端。

7.1.1 测试gRPC服务器端

gRPC 服务的测试通常使用 gRPC 客户端应用程序来完成，该客户端应用程序是测试用例的一部分。服务器端的测试包括使用所需的服务启动 gRPC 服务器，并使用实现测试用例的客户端应用程序连接到服务器。我们看一个使用 Go 语言编写的测试用例，它对 `ProductInfo` 服务进行了测试。在 Go 语言中，gRPC 测试用例应该是使用 testing 包的 Go 通用测试用例来实现的（见代码清单 7-1）。

代码清单 **7-1** 使用 Go 语言编写的 gRPC 服务器端测试

```
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, launc
        February 2019"
        price := float32(700.0)
        ctx, cancel := context.WithTimeout(context.Background(), time.Secon
        defer cancel()
        r, err := c.AddProduct(ctx, &pb.Product{Name: name,
                                        Description: description, Price: pr
        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 常规测试，启动 gRPC 服务器和客户端以使用 RPC 测试服务。

❷ 在 HTTP/2 之上启动常规的 gRPC 服务器。

❸ 连接服务器端应用程序。

❹ 向 `AddProduct` 方法发送 RPC。

❺ 校验响应消息。

因为 gRPC 测试用例是基于语言的标准测试用例，所以执行它们的方式与标准测试用例没什么不同。服务器端测试用例有一个特殊的地方，那就是它们需要服务器端应用程序开启一个供客户端应用程序连接的端口。如果你不喜欢这样做，或者测试环境不允许这样做，那么可以使用某个库来避免在真正的端口上启用服务。在 Go 语言中，可以使用 bufconn 包，它提供了 `net.Conn`，这是通过缓冲区和相关的 `dial` 与监听功能实现的。你可以在本章的源代码仓库中找到完整的代码示例。如果使用的是 Java，那么可以使用像 JUnit 这样的测试框架，并遵循完全相同的过程来编写服务器端的 gRPC 测试。但是，如果想在编写测试用例时不启动 gRPC 服务器实例，那么可以使用 Java 实现的 gRPC 进程内服务器端。

在本书的源代码仓库中，可以找到完整的 Java 代码示例。

我们还可以编写远程函数业务逻辑的单元测试，避免涉及 RPC 网络层。可以通过调用函数来直接对它们进行测试，而无须使用 gRPC 客户端。

到此为止，我们已经学习了如何为 gRPC 服务编写测试。接下来看一下如何测试 gRPC 客户端。

7.1.2 测试gRPC客户端

当为 gRPC 客户端开发测试时，有种可行的测试方式就是启动一台 gRPC 服务器并实现 mock 服务。但是，这并不是一个简单的任务，因为这会有打开端口和连接服务器端的开销。因此，想要测试客户端的逻辑却不想要连接真正的服务器端所带来的开销，可以使用 mock 框架。对 gRPC 服务器端进行 mock，能够让开发人员在客户端编写轻量级单元测试，来对功能进行检查，避免对服务器进行 RPC。

如果使用 Go 语言开发 gRPC 客户端应用程序，则可以（借助生成的代码）使用 Gomock 来模拟客户端接口，并通过编码的方式设置方法以接收和返回预先确定的值。在使用 Gomock 时，可以通过如下命令为 gRPC 客户端应用程序生成 mock 接口：

```
mockgen github.com/grpc-up-and-running/samples/ch07/grpc-docker/go/proto-ge
ProductInfoClient > mock_prodinfo/prodinfo_mock.go
```

这里指定 `ProductInfoClient` 是要模拟的接口。然后，所编写的测试代码可以导入 `mockgen` 生成的包以及 gomock 包，从而为客户端逻辑编写单元测试。如代码清单 7-2 所示，可以创建一个 `mock` 对象，预期对它的方法进行调用并返回一个响应。

代码清单 **7-2** 使用 Gomock 进行 gRPC 客户端测试

```
func TestAddProduct(t *testing.T) {
        ctrl := gomock.NewController(t)
        defer ctrl.Finish()
        mocklProdInfoClient := NewMockProductInfoClient(ctrl) ❶
    ...
        req := &pb.Product{Name: name, Description: description, Price: pri
        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.Secon
        defer cancel()
        ...
        r, err := client.AddProduct(ctx, &pb.Product{Name: name,
```

```
    Description: description, Price: price})

        // 测试并校验响应
}
```

❶ 创建 mock 对象，预期对远程方法进行调用。

❷ 对 mock 对象进行编码。

❸ 预期调用 AddProduct 方法。

❹ 返回商品 ID 的 mock 值。

❺ 调用实际的测试方法，它会调用客户端存根的远程方法。

如果你使用的是 Java，则可以使用 Mockito 和针对 Java gRPC 的进程内服务器实现来测试客户端应用程序。你可以参考源代码仓库来了解示例详情。服务器端和客户端所需的测试准备好后，就可以将它们集成到所使用的持续集成工具中了。

需要记住的要点是，mock gRPC 服务器不会带来与真实的 gRPC 服务器端完全相同的行为。因此，特定的功能可能无法通过测试来校验，除非重新实现 gRPC 服务器端可能出现的所有错误逻辑。在实践中，可以通过 mock 校验一组选定的功能，而其他的功能则需要通过真正的 gRPC 服务器实现来验证。下面来看一下如何对 gRPC 应用程序进行负载测试和基准测量。

7.1.3 负载测试

使用常规的工具很难对 gRPC 应用程序进行负载测试和基准测量，这是因为这些应用程序都或多或少是与特定协议（如 HTTP）绑定的。对于 gRPC 来说，我们需要定制的负载测试工具，这些工具能够生成对服务器端的虚拟 RPC 负载，从而实现对 gRPC 服务器端的负载测试。

ghz 就是这样的负载测试工具，它是使用 Go 语言实现的命令行工具。它能够在本地对服务进行测试和调试，也能用在自动化持续集成环境中，实现性能回归测试。例如，可以通过如下命令利用 ghz 执行负载测试：

```
ghz --insecure \
 --proto ./greeter.proto \
 --call helloworld.Greeter.SayHello \
 -d '{"name":"Joe"}'\
 -n 2000 \
 -c 20 \

 0.0.0.0:50051
```

这里以非安全的方式调用 `Greeter` 服务的 `SayHello` 远程方法。可以设置总的请求数（`-n 2000`）和并发数（20 个线程）。测试结果能够以各种输出格式生成。

服务器端和客户端所需的测试准备就绪之后，就可以将它们集成到所使用的持续集成工具中了。

7.1.4 持续集成

如果你刚接触持续集成（continuous integration，CI），那么可以将其描述为一种需要开发人员频繁地将代码集成到一个共享仓库的开发实践。每次提交的代码都会通过自动构建进行验证，这样能够让团队尽早地发现问题。就 gRPC 应用程序来讲，通常服务器端应用程序和客户端应用程序相互独立，它们可能是使用完全不同的技术构建的。因此，作为 CI 过程的一部分，必须使用 7.1.3 节介绍的单元测试和集成测试技术，来验证 gRPC 客户端和服务器端的代码。然后，基于所使用的语言来构建 gRPC 应用程序。可以用所选择的 CI 工具来集成这些应用程序的测试（如 Go testing 或 Java JUnit）。例如，你使用 Go 编写测试，那么就可以很容易地将 Go 测试与 Jenkins、TravisCI 和 Spinnaker 这样的工具集成。

为 gRPC 应用程序完成搭建测试和 CI 过程后，接下来要学习的就是 gRPC 应用程序的部署。

7.2 部署

现在，我们看一下 gRPC 应用程序的多种部署方法。如果你想在本地或 VM 中运行 gRPC 服务器端和客户端应用程序，部署仅仅依赖于你为 gRPC 应用程序的编程语言所生成的二进制文件。对于本地和基于 VM 的部署来讲，gRPC 服务器端应用程序的扩展和高可用性通常是通过标准的部署实践来实现的，比如使用支持 gRPC 协议的负载均衡器。

大多数现代应用程序会部署为容器。因此，学习如何将 gRPC 应用程序部署在容器中是非常有用的。Docker 是基于容器进行应用程序部署的标准平台。

7.2.1 部署到Docker上

Docker 是一个开发、发布和运行应用程序的开放平台。借助 Docker，可以将应用程序与基础设施分离开来。它能够在隔离环境（容器）中打包和运行应用程序，这样就可以在同一台主机上运行多个容器了。容器要比常规的 VM 轻得多，可以直接在宿主机的内核上运行。

下面看一些将 gRPC 应用程序部署为 Docker 容器的示例。

关于 Docker 的基础知识已经超出了本书的范围。因此，如果你不熟悉 Docker，请参考 Docker 文档和其他资源。

在开发了 gRPC 服务器端应用程序后，就可以为其创建一个 Docker 容器。代码清单 7-3 展示了一个基于 Go 语言的 gRPC 服务器端的 Dockerfile。在 Dockerfile 中，有 gRPC 特有的很多构造。本例使用了多阶段 Docker 构建：第一阶段构建了应用程序，第二阶段以一个非常轻量级的运行时来运行该应用程序。在构建应用程序之前，生成的服务器端代码也添加到了容器中。

代码清单 **7-3** 基于 Go 语言的 gRPC 服务器端的 Dockerfile

```
# 多阶段构建
```

```
# 构建阶段1: ❶
FROM golang AS build
ENV location /go/src/github.com/grpc-up-and-running/samples/ch07/grpc-docke
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

# 构建阶段2: ❺
FROM scratch
COPY --from=build /bin/grpc-productinfo-server /bin/grpc-productinfo-server

ENTRYPOINT ["/bin/grpc-productinfo-server"]
EXPOSE 50051
```

❶ 构建程序只需要 Go 语言和 Alpine Linux。

❷ 下载所有的依赖项。

❸ 安装所有的包。

❹ 构建服务器端应用程序。

❺ Go 二进制文件是自包含的可执行文件。

❻ 将我们在上一阶段构建的二进制文件复制到新的位置。

创建完 Dockerfile 之后，就可以使用如下命令构建 Docker 镜像了：

```
docker image build -t grpc-productinfo-server -f server/Dockerfile
```

gRPC 客户端应用程序可以按照相同的方式进行创建。这里有个特殊情况，因为是在 Docker 中运行服务器端应用程序，所以客户端应用程序连接 gRPC 的主机名和端口会有所不同。

当服务器端应用程序和客户端应用程序在 Docker 中运行时，它们需要通过主机相互通信并与外部通信。因此，这里必须涉及一个网络层。

Docker 支持不同类型的网络，每种类型都适用于特定的使用场景。当我们运行服务器端和客户端 Docker 容器时，可以指定一个通用的网络，这样客户端应用程序就能基于域名发现服务器端应用程序的位置。这意味着，客户端应用程序的代码必须进行修改才能连接到服务器的主机名上。例如，我们的 Go gRPC 应用程序必须修改成调用服务器主机名，而不是 localhost：

```
conn, err := grpc.Dial("productinfo:50051", grpc.WithInsecure())
```

我们可以从环境中读取主机名，避免在客户端应用程序中硬编码。客户端应用程序的修改完成之后，需要重新构建 Docker 镜像，并按照如下方式运行服务器端和客户端镜像：

```
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 借助主机名 `productinfo` 和端口 50051 在 Docker 网络 my-net 上运行 gRPC 服务器。

❷ 在 Docker 网络 my-net 上运行 gRPC 客户端。

在启动 Docker 容器时，可以指定给定容器在哪个 Docker 网络上运行。如果服务共享同一个网络，那么客户端应用程序可以通过 `docker run` 命令所提供的主机名发现托管服务的实际地址。

如果所运行的容器比较少，且它们之间的交互相对简单，则我们可以把解决方案完全构建在 Docker 上。但是，大多数真实场景需要管理多个容器及其之间的交互。仅仅基于 Docker 来构建这样的解决方案就非常枯燥了，而这正是容器编排平台的用武之地。

7.2.2 部署到Kubernetes上

Kubernetes 是一个自动部署、可扩展和管理容器化应用程序的开源平台。在使用 Docker 运行容器化的 gRPC 应用程序时，并没有方便的扩展性和高可用性保障，需要在 Docker 容器之外构建这些功能。

Kubernetes 提供了范围广泛的此类功能，以便将大多数容器管理和编排的任务交给底层的 Kubernetes 平台。

Kubernetes 提供了一个可靠的和可扩展的平台，来运行容器化的工作负载。Kubernetes 负责扩展需求、故障转移、服务发现、配置管理、安全性、部署模式等。

Kubernetes 的基础知识超出了本书的范围。因此，推荐参考 Kubernetes 文档和其他资源来学习它的更多知识。

接下来看一下如何将 gRPC 服务器端应用程序部署到 Kubernetes 上。

01. **gRPC** 服务器的 **Kubernetes** `Deployment` 资源

为了在 Kubernetes 上进行部署，需要做的第一件事情就是为 gRPC 服务器端应用程序创建 Docker 容器。上一节做过完全一样的事情，在这里可以使用同一个容器。可以将容器镜像推送到一个容器注册中心，比如 Docker Hub。

对于本例，我们已将 gRPC 服务器端的 Docker 镜像推送到了 Docker Hub 中名为 `kasunindrasiri/grpc-productinfo-server` 的 tag 下。Kubernetes 平台不会直接管理容器，而是使用一个名为 pod 的抽象。pod 是逻辑单元，可以包含一个或多个容器，Kubernetes 以 pod 作为单位实现复制功能。如果需要 gRPC 服务器端应用程序的多个实例，Kubernetes 就会创建更多的 pod。在给定 pod 中运行的多个容器会共享相同的资源和本地网络。但是，在我们的场景中，只需要在 pod 中运行一个 gRPC 服务器端容器。因此，这是一个具有单个容器的 pod。Kubernetes 没有直接管理 pod，而是通过另一个名为 deployment 的抽象来管理。`Deployment` 指定了同时要运行的 pod 的数量。当新的 `Deployment` 创建时，Kubernetes 会根据 `Deployment` 的设定，生成指定数量的 pod。

要在 Kubernetes 中部署 gRPC 服务器端应用程序，需要使用 YAML 描述符创建 Kubernetes `Deployment`，如代码清单 7-4 所示。

代码清单 **7-4**　基于 Go gRPC 服务器端应用程序的 Kubernetes

Deployment 描述符

```
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 声明 Kubernetes Deployment 对象。

❷ Deployment 的名称。

❸ 要同时运行的 gRPC 服务器端 pod 的数量。

❹ 相关联的 gRPC 服务器端容器的名称。

❺ gRPC 服务器端容器的镜像名称和 tag。

当在 Kubernetes 中通过命令 kubectl apply -f server/grpc-prodinfo-server.yaml 应用该描述符时，Kubernetes 集群会运行由一个 gRPC 服务器端 pod 所组成的 Kubernetes Deployment。但是，如果 gRPC 客户端应用程序要访问在同一个 Kubernetes 集群中运行的 gRPC 服务器端 pod，它就需要确定 pod 的准确 IP 地址和端口并发送 RPC。不过，在 pod 重启时，IP 地址可能会发生变化，

而且在运行多个副本时，还需要处理每个副本所带来的多个 IP 地址。为了克服这种局限性，Kubernetes 提供了名为 service 的抽象。

02. gRPC 服务器的 Kubernetes Service 资源

可以创建 Kubernetes Service 并将其与匹配的 pod（在本例中，也就是 gRPC 服务器端 pod）关联，这样会得到一个 DNS 名，它会自动将流量路由到任意匹配的 pod 上。因此，可以将 Service 视为一个 Web 代理或者负载均衡器，它能够将请求转发到底层的 pod 上。代码清单 7-5 展示了 gRPC 服务器端应用程序的 Kubernetes Service 描述符。

代码清单 **7-5**　基于 Go gRPC 服务器端应用程序的 Kubernetes Service 描述符

```
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 声明 Service 描述符。

❷ Service 的名称。客户端应用程序在连接 Service 的时候，会用到这个名称。

❸ 这将告诉 Service，将请求路由至匹配 grpc-productinfo-server label 的 pod。

❹ 服务在端口 50051 上运行并将请求转发至目标端口 50051。

创建完 Deployment 描述符和 Service 描述符之后，就可以通过

`kubectl apply -f server/grpc-prodinfo-server.yaml` 命令将该应用程序部署到 Kubernetes 中了（可以将这两个描述符放到同一个 YAML 文件中）。这些对象部署成功后，我们将得到运行中的 gRPC 服务器

端 pod、gRPC 服务器端 Kubernetes `Service` 以及 `Deployment`。

下一步是将 gRPC 客户端部署到 Kubernetes 中。

03. 运行 **gRPC** 客户端的 **Kubernetes `Job`**

gRPC 服务器端在 Kubernetes 集群中启动和运行之后，就可以在同一个集群中运行 gRPC 客户端应用程序了。客户端可以通过我们在上一步创建的 gRPC service `productinfo` 来访问 gRPC 服务器端。因此，在客户端的代码中，我们应该使用 Kubernetes `Service` 的名称作为主机名，并使用 `Service` 的端口作为 gRPC 服务器端的端口名。因此，在 Go 语言的客户端实现中，客户端要使用 `grpc.Dial("productinfo:50051", grpc.WithInsecure())` 来连接至服务器端。假设客户端应用程序需要运行指定的次数（只需要调用 gRPC 服务、用日志记录响应并退出），那么我们可以使用 Kubernetes `Job`，而非 Kubernetes `Deployment`。Kubernetes `Job` 旨在让一个 pod 运行指定的次数。

可以按照与 gRPC 服务器端相同的方式来创建客户端应用程序容器。在将容器推送至 Docker 注册中心后，就可以按照代码清单 7-6 所示创建 Kubernetes `Job` 的描述符了。

代码清单 **7-6**　以 Kubernetes `Job` 形式运行的 gRPC 客户端应用程序

```
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 声明 Kubernetes `Job`。

❷ `Job` 的名称。

❸ 在 `Job` 完成之前，pod 需要成功运行的次数。

❹ 要有多少个 pod 并行运行。

❺ 相关 gRPC 客户端容器的名称。

❻ 该 `Job` 相关联的容器镜像。

接下来，就可以通过 `kubectl apply –f client/grpc-prodinfo-client-job.yaml` 部署 gRPC 客户端应用程序的 job 并检查 pod 的状态了。

job 执行成功时会发送一个添加商品的 RPC 到 `ProductInfo` gRPC 服务。因此，你可以观察服务器端和客户端 pod 的日志，从而判断是否得到了预期信息。

随后可以使用 `Ingress` 资源将 gRPC 服务暴露到 Kubernetes 集群之外。

04. 通过 **Kubernetes `Ingress`** 对外暴露 **gRPC** 服务

到目前为止，我们已完成了将 gRPC 服务器端部署在 Kubernetes 上，并且让它能够被同一个集群中的其他 pod（在这里，以 `Job` 的方式运行）访问。如果我们想将 gRPC 服务暴露给 Kubernetes 集群外部的应用程序，该怎么办呢？我们知道，Kubernetes `Service` 只能暴露指定的 Kubernetes pod 给集群中运行的其他 pod。因此，Kubernetes `Service` 不能为 Kubernetes 之外的应用程序所访问。为了实现这一目的，Kubernetes 提供了名为 ingress 的抽象。

我们可以将 Ingress 视为 Kubernetes Service 和外部服务之间的一个负载均衡器。Ingress 将外部的流量路由至 Service，随后 Service 在匹配的 pod 之间路由内部的流量。Ingress 控制器管理给定 Kubernetes 集群中的 Ingress 资源。Ingress 控制器的类型和行为可能会根据你所使用的集群有所变化。同时，在将 gRPC 服务暴露给外部应用程序时，有一个强制的需求就是在 Ingress 层面要支持 gRPC 路由。因此，我们要选择一个支持 gRPC 的控制器。

对于本例，我们将使用 Nginx Ingress 控制器，它是基于 Nginx 负载均衡器的。（根据你所使用的 Kubernetes 集群，可以选择支持 gRPC 的最合适的 Ingress 控制器。）Nginx Ingress 支持 gRPC 将外部流量路由至内部服务。

为了将 ProductInfo gRPC 服务器端应用程序暴露到外部世界（Kubernetes 集群之外），可以创建代码清单 7-7 所示的 Ingress 资源。

代码清单 **7-7**　基于 Go gRPC 服务器端应用程序的 Kubernetes Ingress 资源

```
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 声明 Ingress 资源。

❷ Nginx Ingress 控制器相关的注解，指定 gRPC 作为后端协议。

❸ `Ingress` 资源的名称。

❹ 暴露给外部的主机名。

❺ 关联的 Kubernetes `Service` 的名称。

❻ 在 Kubernetes `Service` 中所声明的服务端口名称。

在部署上述的 `Ingress` 资源之前，需要安装 Nginx Ingress 控制器。在 Kubernetes 的 Ingress- Nginx 仓库中，有更多安装和使用 Nginx Ingress 实现 gRPC 的详情供参考。部署完 `Ingress` 资源后，任意的外部应用程序都可以通过主机名（`productinfo`）和默认端口（80）来调用 gRPC 服务器端。

到此为止，我们已经学习了在 Kubernetes 上部署生产级 gRPC 应用程序的所有基础知识。可以看到，借助 Kubernetes 和 Docker 所提供的功能，我们无须再担心大多数非功能性需求，如扩展性、高可用性、负载均衡、故障转移等，因为 Kubernetes 作为底层平台的一部分，已经提供了这些功能。如果你在 Kubernetes 上运行 gRPC 应用程序，那么第 5 章所介绍的某些概念，比如 gRPC 代码级别的负载均衡、命名解析等，就没有什么用处了。

基于 gRPC 的应用程序运行起来之后，需要确保它在生产环境中平稳运行。为了实现该目标，需要不间断地观察 gRPC 应用程序，并在需要时采取必要的行动。接下来深入研究 gRPC 应用程序在可观察性方面的细节。

7.3 可观察性

如前所述，gRPC 应用程序正常部署和运行在容器化的环境中，其中会有多个这样的容器在运行，并通过网络进行彼此交流。这就带来了一个问题：该如何跟踪每个容器并确保它们真正在运行？这就是可观察性能够发挥作用的地方了。

按照维基百科的定义，“可观察性是一种度量指标，衡量系统的内部状态是否能够由其对外输出的知识推断出来”。简单地说，系统的可观察性是为了回答这样一个问题：“现在系统中有问题吗？”如果答案是肯定的，则我们应该能够回答后续的一系列问题，比如“发生了什么问题”以及“为什么会发生这种问题”。如果我们在任意时间针对系统的任何组成部分都能回答这些问题，那么就可以说该系统是可观察的。

需要注意的要点是，可观察性是系统的一个属性，与效率、可用性和可靠性同等重要。因此，在构建 gRPC 应用程序之初就必须考虑到它。

在讨论可观察性时，我们通常会涉及其三个主要方面：度量指标、日志和跟踪。这是实现系统可观察性的主要技术。以下几个小节将分别介绍它们。

7.3.1 度量指标

度量指标是一段时间内测量数据的数字形式表示。在讨论度量指标时，我们会收集两种类型的数据：一种是系统级的指标，如 CPU 使用情况、内存使用情况等；另一种是应用级的指标，如入站请求率、请求错误率等。

系统级的指标通常是在应用程序运行期间捕获的。现在，有很多工具可以捕获这些指标，它们通常都是由 DevOps 团队去捕获的。但是，应用级的度量指标因应用程序不同而有所不同。因此，在设计一个新的应用程序时，应用程序开发人员的任务是决定要捕获哪些应用级的度量指标才能了解系统的行为。本节将关注如何在应用程序中启用应用级的度量指标。

01. 在 **gRPC** 中使用 **OpenCensus**

OpenCensus 库为 gRPC 应用程序提供了标准的度量指标。通过在客户端应用程序和服务器端应用程序中添加 `handler`，可以很容易地启用它们。我们还可以添加自己的度量指标收集器（见代码清单 7-8）。

OpenCensus 是一组开源库，用来实现应用程序度量指标的收集和分布式跟踪，它支持各种语言。它会从目标应用程序收集度量指标，并将数据实时转移到所选择的后端。目前所支持的后端包括 Azure Monitor、Datadog、Instana、Jaeger、SignalFX、Stackdriver 和 Zipkin 等。我们还可以为其他后端编写自己的导出器（exporter）。

代码清单 **7-8**　为 Go gRPC 服务器端启用 OpenCensus 监控

```
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"
)

// 用来实现ecommerce/product_info的服务器
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{
 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 为了启用监控，指明需要添加的外部库。gRPC OpenCensus 提供了一组预先定义好的 `handler` 以支持 OpenCensus 监控。这里会使用这些 `handler`。

❷ 注册统计导出器以导出收集的数据。这里添加了 `PrintExporter`，它会将导出数据以日志的形式打印到控制台上。这只是为了展示功能，正常情况下，不推荐日志记录所有的生产环境负载。

❸ 注册视图以收集服务器请求的数量。这些是预定义的默认服务视图，会收集每个 RPC 所接收的字节、每个 RPC 发送的字节、每个 RPC 的延迟以及完成的 RPC。我们可以编写自己的视图来收集数据。

❹ 使用数据统计 `handler` 来创建 gRPC 服务器端。

❺ 注册 ProductInfo 服务到服务器端上。

❻ 开始在端口（50051）上监听传入的消息。

❼ 启动一台 z-Pages 服务器。在端口 8081 的 /debug 上下文中启动一个 HTTP 端点，实现度量指标的可视化。

与 gRPC 服务器端类似，可以使用客户端的 handler 在 gRPC 客户端启用 OpenCensus 监控。代码清单 7-9 提供了使用 Go 语言添加度量指标 handler 到 gRPC 客户端的代码片段。

代码清单 **7-9**　为 Go gRPC 客户端启用 OpenCensus 监控

```
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 {
       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

  ... // 省略了RPC方法的调用
}
```

❶ 声明为了启用监控需要添加的外部库。

❷ 注册统计数据和跟踪的导出器，以导出收集的数据。这里添加了 `PrintExporter`，它会将导出数据以日志的形式打印到控制台上。这只是为了展示功能，正常情况下，不推荐日志记录所有的生产环境负载。

❸ 注册视图以收集服务器请求的数量。这些是预定义的默认服务视图，会收集每个 RPC 所接收到的字节、每个 RPC 发送的字节、每个 RPC 的延迟以及完成的 RPC。我们可以编写自己的视图来收集数据。

❹ 使用客户端统计数据的 `handler` 建立到服务器端的连接。

❺ 使用服务器端连接创建客户端存根。

❻ 在所有的事情完成后关闭连接。

运行服务器端和客户端之后，我们可以通过创建的 HTTP 端点访问服务器端和客户端的度量指标。

如前所述，我们可以使用预先定义的导出器将数据发布到支持的后端，也可以编写我们自己的导出器，将跟踪数据和度量指标发送给任意能够消费它们的后端。

下一小节将讨论另外一项流行的技术——Prometheus，它经常用来为 gRPC 应用程序启用度量指标功能。

02. 在 **gRPC** 中使用 **Prometheus**

Prometheus 是一个用于系统监控和警告的开源工具集。可以通过

gRPC Prometheus 库为 gRPC 应用程序实现基于 Prometheus 的度量指标功能。通过为客户端应用程序和服务器端应用程序添加拦截器，可以很容易地实现这一点，并且还可以添加自己的收集器。

Prometheus 通过调用一个“`/metrics`”上下文开头的 HTTP 端点来收集目标应用程序的度量指标。它会存储所有收集到的数据，并且基于这些数据运行规则，要么基于已有的数据进行聚合并记录新的时序数据，要么生成警告。可以使用像 Grafana 这样的工具对聚合结果进行可视化。

代码清单 7-10 展示了如何为 Go 语言编写的商品管理服务器端添加度量指标拦截器和自定义指标收集器。

代码清单 **7-10**　为 Go gRPC 服务器端启用 Prometheus 监控

```
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.CounterOp
       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

  // 为Prometheus启动HTTP服务器
  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 声明要启用监控功能所需的外部库。gRPC 提供了预定义的一组拦截器以支持 Prometheus 监控。在这里将使用这些拦截器。

❷ 创建度量指标的注册中心。它会持有系统中所有注册的数据收集器。如需添加新的收集器，就要在这个注册中心中对其进行注册。

❸ 创建标准的服务器端度量指标。这是在库中预先定义好的度量指标。

❹ 创建名为 `product_mgt_server_handle_count` 的自定义度量指标计数器。

❺ 将标准的服务器度量指标和自定义的度量指标收集器注册到第 2 步所创建的注册中心里。

❻ 为 Prometheus 创建 HTTP 服务器。在端口 9092 上以上下文 `/metrics` 开头的 HTTP 端点用来进行度量指标收集。

❼ 使用度量指标拦截器创建 gRPC 服务器。这里使用了 grpcMetrics.UnaryServerInterceptor，因为我们具有一元服务。还有另一个适用于流服务的拦截器，名为 grpcMetrics.StreamServerInterceptor。

❽ 初始化所有的标准度量指标。

借助在第 4 步添加的自定义度量指标计数器，能够为监控添加更多的度量指标。假设我们想收集相同名称的商品向商品管理系统中添加的次数，如代码清单 7-11 所示，可以在 AddProduct 方法中添加名为 customMetricCounter 的新度量指标。

代码清单 **7-11**　添加新的度量指标到自定义的度量指标收集器

```
// AddProduct实现了ecommerce.AddProduct
func (s *server) AddProduct(ctx context.Context,
     in *pb.Product) (*wrapper.StringValue, error) {
     customMetricCounter.WithLabelValues(in.Name).Inc()
  ...
}
```

与 gRPC 服务器端类似，可以通过客户端的拦截器为 gRPC 客户端启用 Prometheus 监控功能。代码清单 7-12 提供了在 Go 语言中为 gRPC 客户端添加度量指标拦截器的代码片段。

代码清单 **7-12**　为 Go gRPC 客户端启用 Prometheus 监控

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

   // 为Prometheus创建HTTP服务器端
   httpServer := &http.Server{
        Handler: promhttp.HandlerFor(reg, promhttp.HandlerOpts{}),
          Addr: fmt.Sprintf("0.0.0.0:%d", 9094)} ❻

   // 启动Prometheus的HTTP服务器端
   go func() {
       if err := httpServer.ListenAndServe(); err != nil {
           log.Fatal("Unable to start a http server.")
       }
   }()

  c := pb.NewProductInfoClient(conn)
  ...
}
```

❶ 声明要启用监控功能所需要的外部库。

❷ 创建度量指标的注册中心。与服务器端代码类似，它会持有系统中所有注册的数据收集器。如果要添加新的收集器，就需要在这个注册中心中对其进行注册。

❸ 创建标准的客户端度量指标，这是在库中预定义好的度量指标。

❹ 注册标准的客户端度量指标到第 2 步所创建的注册中心里。

❺ 使用度量指标拦截器创建到 gRPC 服务器端的连接。这里使用了 grpcMetrics.UnaryClient- Interceptor，因为具有一元客户端。还有另一个适用于流客户端的拦截器，叫作 grpcMetrics.StreamClientInterceptor。

❻ 为 Prometheus 创建 HTTP 服务器。在 9094 端口上以上下文 `/metrics` 开头的 HTTP 端点用来进行度量指标收集。

运行服务器端和客户端之后，我们就可以通过 HTTP 端点访问服务器端和客户端的度量指标了。例如，服务器端度量指标在 http://localhost:9092/metrics 上，客户端度量指标在 http://localhost:9094/metrics 上。

如前所述，Prometheus 能够通过访问上述的 URL 收集度量指标。Prometheus 将所有的度量指标数据存储在本地，并使用一组规则聚合和创建新的记录。另外，使用 Prometheus 作为数据源，我们还可以使用像 Grafana 这样的工具在一个仪表盘中可视化度量指标。

Grafana 是一个开源的度量指标仪表盘和图编辑器，适用于 Graphite、Elasticsearch 和 Prometheus。它能够让我们查询、可视化和理解度量指标数据。

在系统中使用基于度量指标的监控有一项显著的优势，那就是处理度量指标数据的成本并不会随着系统的活动而增加。例如，应用程序流量的增加不会增加磁盘利用率、处理复杂性、可视化速度、运维等方面的处理成本，它具有固定的开销。同时，收集完度量数据之后，我们可以进行大量的数学和统计转换，并得出关于系统状况的有价值的结论。

可观察性的另一个重要方面是日志，下一节会讨论。

7.3.2 日志

日志是不可变的、带时间戳的记录，描述了一段时间内所发生的离散事件。作为应用程序的开发人员，我们通常会将数据转储到日志中，以判断在给定的时间点上系统的位置和内部状态。日志的好处在于，它们很容易生成，而且比度量指标更加细粒度。可以为其附加特定的操作或一些上下文信息，比如唯一 ID、我们要做什么，以及栈跟踪信息等。日志的不足之处在于，它们代价高昂，因为需要存储它们并为其建立索引，这样才能更容易地搜索和使用它们。

在 gRPC 应用程序中，可以使用拦截器来启用日志功能。如第 5 章所

述，可以在客户端和服务器端添加新的日志拦截器，并记录每个远程调用的请求和响应消息。

gRPC 生态系统为 Go 应用程序提供了一组预定义的日志拦截器，包括 `grpc_ctxtags`、`grpc_zap` 和 `grpc_logrus`。其中，`grpc_ctxtags` 库会添加一个 Tag map 到上下文中，其数据来源于请求体；`grpc_zap` 将 zap 日志库集成到了 gRPC `handler` 中；`grpc_logrus` 则将 logrus 日志库集成到了 gRPC `handler` 中。关于这些拦截器的更多信息，请参阅 gRPC Go 的中间件仓库。

在将日志功能添加到 gRPC 应用程序中之后，它们就会被打印到控制台上或日志文件中，这取决于如何对日志进行配置。日志如何配置则依赖于所使用的日志框架。

我们已经讨论了可观察性的两个重要方面：度量指标和日志。它们对于理解单个系统的性能和行为已经足够了，但对于理解跨多个系统的请求生命周期还不够。分布式跟踪技术使跨多个系统的请求生命周期可见。

7.3.3 跟踪

trace 是对一系列相关事件的描述，这些事件组成了分布式系统中端到端的请求流。如 3.5 节所述，在真实场景中，我们会有多个微服务，分别用来实现不同的业务功能。因此，在将响应返回给客户端之前，客户端所发起的请求要经历多个服务和不同的系统。所有的这些中间事件都是请求流的一部分。借助跟踪技术，我们能够看见请求所遍历的路径以及请求的结构。

在跟踪技术中，trace 是 span 所组成的一棵树，在分布式跟踪中，span 是最基础的构造。span 包含与任务相关的元数据、延迟（完成该任务所耗费的时间）以及该任务的其他属性。trace 有自己的 ID，叫作 traceID，这是独一无二的字节序列。traceID 会对 span 进行分组和区分。接下来在 gRPC 应用程序中启用跟踪功能。

与度量指标类似，OpenCensus 库提供了在 gRPC 应用程序中启用跟踪功能的支持。在商品管理应用程序中，会使用 OpenCensus 来启用跟踪。如前所述，可以插入任意的导出器将跟踪数据导出到不同的后端。这里

将使用 Jaeger 进行分布式跟踪的采样。

在默认情况下，gRPC Go 就是启用跟踪功能的。因此，只需要注册导出器，从而借助 gRPC Go 集成功能收集跟踪数据就可以了。接下来，将 Jaeger 导出器应用到客户端和服务器端的应用程序中。代码清单 7-13 阐述了如何使用 Jaeger 库初始化 OpenCensus Jaeger 导出器。

代码清单 **7-13**　初始化 OpenCensus Jaeger 导出器

```
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 导入 OpenTracing 和 Jaeger 库。

❷ 使用收集器端点、服务名和代理端点创建 Jaeger 导出器。

❸ 使用 OpenCensus tracer 注册导出器。

在服务器端注册完导出器后，我们就可以对服务器端安装（instrument）跟踪功能了。代码清单 7-14 展示了如何在服务方法中安

装跟踪功能。

代码清单 **7-14**　为 gRPC 服务方法启用跟踪功能

```
// GetProduct实现了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.V
}
```

❶ 使用 span 名称和上下文启动新的 span。

❷ 当所有的事情完成后，停止该 span。

与 gRPC 服务器端类似，我们可以为客户端安装跟踪功能，如代码清单 7-15 所示。

代码清单 **7-15**　为 gRPC 客户端启用跟踪功能

```
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 导入 OpenTracing 和 Jaeger 库。

❷ 调用 `initTracing` 函数，初始化 Jaeger 导出器实例，并使用 trace 进行注册。

❸ 使用 span 名称和上下文启动新的 span。

❹ 当所有的事情完成后，停止该 span。

❺ 通过传递新的商品详情来调用 `AddProduct` 远程方法。

❻ 通过传递 `ProductID` 来调用 `GetProduct` 远程方法。

运行服务器端和客户端之后，trace span 就会发送到 Jaeger 代理上，会有一个守护进程作为缓冲，它将批处理和路由从客户端抽象了出来。Jaeger 代理接收到来自客户端的 trace 日志之后，它就会将日志转发到收集器。收集器处理日志并将它们存储起来。在 Jaeger 服务器端，我们就可以可视化跟踪了。

到此为止，我们完成了对可观察性的讨论。日志、度量指标和跟踪都有其特定的用途，在你的系统中，最好全部启用这三项功能，以便于获取内部状态的最大可见性。

基于 gRPC 的可观察应用程序运行在生产环境中后，就可以持续观察它的状态，并且能够很容易地随时发现问题或系统不可用的状况。当诊断系统中的问题时，很重要的一点就是要尽快找到解决方案、对方案进行测试并部署到生产环境中。为了实现这个目标，就要有好的调试和问题排查机制。接下来详细了解 gRPC 应用程序的这些机制。

7.4 调试和问题排查

调试和问题排查是找到应用程序中发生问题的根本原因并解决问题的过程。为了对问题进行调试和排查，首先需要在等级较低的环境（开发环境或测试环境）中重现问题。因此，需要有一组工具来生成和生产环境相似的请求负载。

相对 HTTP 服务，gRPC 服务的这个过程更困难一些，因为这些工具需要支持基于服务定义的消息编码和解码，并且还要能够支持 HTTP/2。用于测试 HTTP 服务的常见工具，如 curl 和 Postman，并不能用来测试 gRPC 服务。

但是，有许多有趣的工具可用来调试和测试 gRPC 服务。在 awesome gRPC 仓库中，可以找到这些工具的列表。该仓库包含了 gRPC 可用资源的一个大集合。调试 gRPC 应用程序的一个通用方式就是使用额外日志（extra logging）。

启用额外日志

可以启用额外日志和跟踪功能来诊断 gRPC 应用程序的问题。在 gRPC Go 应用程序中，可以通过设置如下环境变量来启用额外日志：

```
GRPC_GO_LOG_VERBOSITY_LEVEL=99 ❶
GRPC_GO_LOG_SEVERITY_LEVEL=info ❷
```

❶ `VERBOSITY` 意味着每 5 分钟单条的 `info` 消息能打印多少次。在默认情况下，`VERBOSITY` 会被设置为 `0`。

❷ 设置日志的严重级别（`SEVERITY`）为 `info`。所有信息级别的消息都会打印出来。

在 gRPC Java 应用程序中，没有控制日志级别的环境变量。可以通过提供 logging.properties 文件来启用额外日志，该文件包含日志级别的变化。假设我们想在应用程序中排查传输级别的帧，那么可以在应用程序中创建一个 logging.properties 文件，并为特定的 Java 包（netty 传输

包）设置较低的日志等级，如下所示：

```
handlers=java.util.logging.ConsoleHandler
io.grpc.netty.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatte
```

然后，使用 JVM 标记来启动 Java 二进制文件：

```
-Djava.util.logging.config.file=logging.properties
```

在为应用程序设置了较低的日志级别之后，等于或高于所配置日志级别的所有日志都会打印到控制台上或日志文件中。通过读取日志，就能获取应用程序内部状态的有价值的信息了。

到此为止，我们已经了解了在生产环境中运行 gRPC 应用程序的大部分知识了。

7.5 小结

让 gRPC 应用程序在生产环境中可用，需要我们关注与应用程序开发相关的许多方面。首先设计服务契约，并生成服务或客户端的代码，然后实现服务的业务逻辑。实现完服务之后，需要关注测试和部署等方面，以确保 gRPC 应用程序在生产环境中可用。gRPC 服务器端和客户端应用程序的测试是至关重要的。

gRPC 应用程序的部署遵循标准的应用程序开发方法论。对于本地部署和 VM 部署来说，只需使用客户端和服务器端生成的二进制文件即可。可以将 gRPC 应用程序运行为 Docker 容器，并且能够找到在 Docker 上部署 Go 应用程序和 Java 应用程序的标准 Dockerfile 示例。在 Kubernetes 上运行 gRPC 应用程序类似于标准的 Kubernetes `Deployment`。在 Kubernetes 上运行 gRPC 应用程序时，会用到一些底层特性，如负载均衡、高可用性、Ingress 控制器等。对于生产环境来讲，让 gRPC 具备可观察性是至关重要的，在生产环境中运行 gRPC 应用程序时，通常会用到 gRPC 应用级的度量指标。

在 gRPC 支持的最流行的一个度量指标实现中，即 gRPC Prometheus 库，我们在服务器端和客户端使用拦截器收集度量指标，同时 gRPC 中的日志也是通过拦截器实现的。对于生产环境中的 gRPC 应用程序来说，可能还需要通过启用额外日志来调试或排查相关问题。第 8 章将探讨在构建 gRPC 应用程序时非常有用的一些 gRPC 生态系统组件。

第 8 章　gRPC 的生态系统

本章将讨论一些并非 gRPC 核心实现的项目，但它们对于构建和运行真正的 gRPC 应用程序非常有帮助。这些项目是 gRPC 生态系统父项目的一部分，对于运行 gRPC 应用程序来说，这里提到的技术都不是强制要求的。如果你的需求与这些项目提供的功能类似，请探索并评估这些技术。

我们首先讨论 gRPC 网关。

8.1 gRPC网关

gRPC 网关插件能够让 protocol buffers 编译器读取 gRPC 服务定义，并生成反向代理服务器端，该服务器端能够将 RESTful JSON API 翻译为 gRPC。这是专门为 Go 编写的，为了同时支持从 gRPC 和 HTTP 客户端应用程序调用 gRPC 服务。图 8-1 展示了如何以 gRPC 方式和 RESTful 方式调用 gRPC 服务。

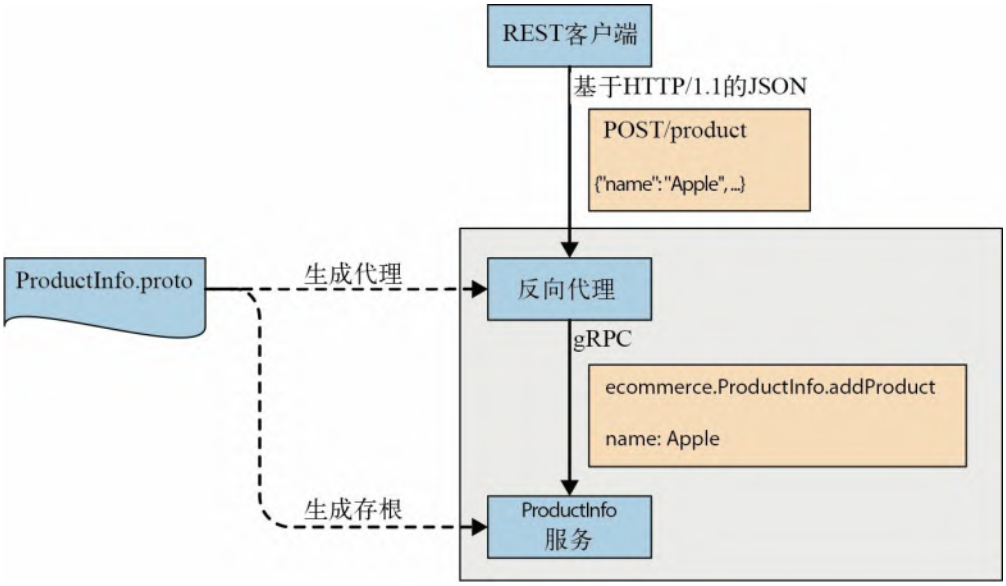

图 **8-1**：**gRPC** 网关

如图 8-1 所示，有一个 `ProductInfo` 服务契约，使用该契约构建了名为 `ProductInfo` 服务的 gRPC 服务。在此之前，我们曾构建了 gRPC 客户端来与该 gRPC 服务进行交互，但在这里，我们没有构建 gRPC 客户端，而是构建了一个反向代理服务。该服务为 gRPC 服务中的每个远程方法暴露了 RESTful API，并且接收了来自 REST 客户端的 HTTP 请求，之后，它会将请求翻译成 gRPC 消息，并调用后端服务的远程方法。来自后端服务器的响应消息会再次转换成 HTTP 响应，并发送给客户端。

要为服务定义生成反向代理服务，首先需要更新服务定义，从而将 gRPC 方法映射为 HTTP 资源。我们以已经创建好的同一个 `ProductInfo` 服务为例，为其添加映射条目。代码清单 8-1 展示了更新

后的 protocol buffers 定义。

代码清单 **8-1**　更新 `ProductInfo` 服务的 protocol buffers 定义

```
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 导入 proto 文件（google/api/annotations.proto）以添加对协议定义的注解支持。

❷ 为 `addProduct` 方法添加 gRPC/HTTP 映射。声明 URL 路径模板（`/v1/product`）、HTTP 方法（`post`）以及消息体的样子。在这里，消息体映射使用了“*”，表示没有在路径模板绑定的所有字段都应该映射到请求体中。

❸ 为 `getProduct` 方法添加 gRPC/HTTP 映射。这里是一个 GET 方法，URL 路径模板是 `/ v1/product/{value}`，传入的 `ProductID` 作为路径参数。

在将 gRPC 方法映射为 HTTP 资源时，还有另一些需要知道的规则。下面列出几个重要的规则。你可以参考 Google API 文档了解 HTTP 和 gRPC 映射的更多详细信息。

- 每个映射都需要指定一个 URL 路径模板和一个 HTTP 方法。
- 路径模板可以包含一个或多个 gRPC 请求消息中的字段。但是，这些字段应该是 nonrepeated 的原始类型字段。
- 如果没有 HTTP 请求体，那么出现在请求消息中但没有出现在路径模板中的字段，将自动成为 HTTP 查询参数。
- 映射为 URL 查询参数的字段应该是原始类型、repeated 原始类型或 nonrepeated 消息类型。
- 对于查询参数的 repeated 字段，参数可以在 URL 中重复，形式为 ...?param=A&param=B。
- 对于查询参数中的消息类型，消息的每个字段都会映射为单独的参数，比如 ...?foo.a=A&foo.b=B&foo.c=C。

在编写完服务定义后，需要使用 protocol buffers 编译器对其进行编译，并生成反向代理服务的源代码。接下来讨论一下如何通过 Go 语言生成代码并实现服务器端。

在编译服务定义之前，需要获取几个依赖包。通过以下命令下载这些包：

```
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
```

在下载完这些包之后，执行以下命令编译服务定义（product_info.proto）并生成存根：

```
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
```

执行完命令后，会在相同的位置生成一个存根（product_info.pb.go）。除了生成的存根之外，我们还需要创建一个反向代理服务，以支持 RESTful 客户端的调用。这个反向代理服务可

以通过 Go 编译器支持的网关插件来生成。

gRPC 网关只支持 Go 语言，这意味着我们无法为其他语言编译和生成 gRPC 网关的反向代理服务。

可以通过执行以下命令根据服务定义生成反向代理服务：

```
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
```

执行完命令后，它会在相同的位置生成一个反向代理服务（`product_info.pb.gw.go`）。

接下来为 HTTP 服务器创建监听器端点，并运行刚刚创建的反向代理服务。代码清单 8-2 展示了如何创建新的服务器实例并注册服务，以监听传入的 HTTP 请求。

代码清单 **8-2**　以 Go 语言编写的 HTTP 反向代理

```
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 导入生成的反向代理代码所在的包。

❷ 声明 gRPC 服务器端点 URL，确保后端 gRPC 服务器在所述的端点上正常运行，这里使用与第 2 章相同的 gRPC 服务。

❸ 使用代理 `handler` 注册 gRPC 服务器端点。在运行时，请求多路转换器（multiplexer）将 HTTP 请求匹配为模式，并调用对应的 `handler`。

❹ 开始在端口 8081 上监听 HTTP 请求。

构建完 HTTP 反向代理服务器后，就可以通过同时运行 gRPC 服务器和 HTTP 服务器来测试它了。在本例中，gRPC 服务器监听端口 50051，而 HTTP 服务器监听端口 8081。

我们通过 curl 发送几个 HTTP 请求并观察它的行为。

01. 添加新商品到 `ProductInfo` 服务：

```
$ curl -X POST http://localhost:8081/v1/product
-d '{"name": "Apple", "description": "iphone7", "price": 699}'
"38e13578-d91e-11e9"
```

02. 使用 `ProductID` 获取已有的商品：

```
$ curl http://localhost:8081/v1/product/38e13578-d91e-11e9

{"id":"38e13578-d91e-11e9","name":"Apple","description":"iphone7",
"price":699}
```

03. 添加反向代理服务后，gRPC 网关还支持生成反向代理服务的 swagger 定义，这可以通过执行以下命令实现：

```
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
```

04. 执行命令后，它会在相同的位置为反向代理服务生成 swagger 定义（product_info.swagger.json）。对于 ProductInfo 服务来说，生成的 swagger 定义如下所示：

```
{
  "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"
        }
      }
    }
  }
}
```

到现在为止，我们使用 gRPC 网关为 gRPC 服务实现了 HTTP 反向代理服务。通过这种方式，可以将 gRPC 服务器端暴露给 HTTP 客户端应用程序使用。通过 gRPC 网关的仓库，可以了解更多关于网关实现的信息。

如前所述，gRPC 网关只支持 Go，同样的概念也被称为 HTTP/JSON 转码。8.2 节将讨论 HTTP/JSON 转码的更多内容。

8.2 gRPC的HTTP/JSON转码

转码（transcode）指的是将 HTTP JSON 调用转换成 RPC，并将它们传递给 gRPC 服务的过程。当客户端应用程序不支持 gRPC，并且需要通过基于 HTTP 的 JSON 方式提供对 gRPC 服务的访问时，转码是非常有用的。有一个使用 C++ 语言编写的库支持 HTTP/JSON 转码，名为 grpc-httpjson-transcoding，目前它用到了 Istio 和 Google Cloud 端点。

Envoy 代理也通过提供一个到 gRPC 服务的 HTTP/JSON 接口来支持转码。与 gRPC 网关类似，我们需要为 gRPC 服务提供带有 HTTP 映射的服务定义。它使用与 Google API 文档中所声明的相同的映射规则。因此，代码清单 8-1 所修改的服务定义也能用到 HTTP/JSON 转码中。

例如，`ProductInfo` 服务的 `getProduct` 方法定义在 .proto 文件中，其请求和响应类型如下所示：

```
rpc getProduct(google.protobuf.StringValue) returns (Product) {
     option (google.api.http) = {
          get:"/v1/product/{value}"
     };
}
```

如果客户端通过 GET 方法对 URL http://localhost:8081/v1/product/2 调用该方法，那么代理服务器会创建一个值为 2 的 `google.protobuf.StringValue`，并使用它来调用 gRPC 方法 `getProduct()`。接下来，gRPC 后端会返回所请求的 ID 为 2 的 `Product`，代理服务器会将其转换成 JSON 格式并返回给客户端。

我们讨论完了 HTTP/JSON 转码。8.3 节将讨论另一个重要的概念，叫作 gRPC 服务器端反射。

8.3 gRPC服务器端反射协议

服务器端反射（server reflection）是在 gRPC 服务器端定义的一个服务，它能提供该服务器端上可公开访问的 gRPC 服务的信息。简而言之，服务器端反射向客户端应用程序提供了服务器端所注册的服务的定义信息。因此，客户端不需要预编译服务定义就能与服务进行通信了。

如第 2 章所述，客户端应用程序要连接 gRPC 服务并与其通信，它必须要有服务的服务定义信息。首先需要编译服务定义，并生成对应的客户端存根。随后，需要创建客户端应用程序，调用存根的方法。借助 gRPC 服务器端反射，我们无须预编译服务定义就能与服务通信。

当我们构建命令行界面（CLI）工具来调试 gRPC 服务器时，服务器端反射是非常有用的。无须为工具提供服务定义，而只需提供要调用的方法和文本载荷。它会发送二进制载荷到服务器端，并以易于人类阅读的格式将响应返回给用户。为了使用服务器端反射，首先要在服务器端启用它。代码清单 8-3 展示了如何启用服务器端反射功能。

代码清单 **8-3** 在 Go gRPC 服务器端启用服务器端反射

```
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 导入反射包以访问反射 API。

❷ 在 gRPC 服务器上注册反射服务。

服务器端应用程序在启用服务器端反射功能之后，就可以通过 gRPC CLI 工具来检查服务器端了。

gRPC CLI 工具来源于 gRPC 仓库。它支持很多功能，比如列出服务器端的服务和方法，以及通过元数据发送和接收 RPC。在编写本书时，需要通过源代码构建该工具。关于如何构建和使用该工具的详细信息，请参考 gRPC CLI 工具仓库。

通过源代码构建完 gRPC CLI 工具后，就可以使用它来检查服务了。我们通过第 2 章构建的商品管理服务来理解一下它的功能。启动商品管理服务的 gRPC 服务器后，就可以运行 CLI 工具来检索服务信息了。

以下是可以借助 CLI 工具执行的操作。

列出服务

执行以下命令将列出 `localhost: 50051` 端点所有公开的服务。

```
$ ./grpc_cli ls localhost:50051

Output:
ecommerce.ProductInfo
grpc.reflection.v1alpha.ServerReflection
```

列出服务详情

通过给定服务的全名（按照 <package>.<service> 的格式），执行以下命令能够探查服务详情。

```
$ ./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) {}
}
```

列出方法详情

通过给定方法的全名（按照 <package>.<service>. 的格式），执行以下命令能够获取方法详情。

```
$ ./grpc_cli ls localhost:50051 ecommerce.ProductInfo.addProduct -l

Output:
rpc addProduct(ecommerce.Product) returns
(google.protobuf.StringValue) {}
```

探查消息类型

通过给定消息类型的全名（按照 <package>.<type> 的格式），执行以下命令能够探查消息类型的详情。

```
$ ./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"];
}
```

调用远程方法

执行以下命令将发送到服务器端的远程调用并获取响应。

01. 调用 ProductInfo 服务中的 addProduct 方法。

```
$ ./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
```

02. 调用 ProductInfo 服务中的 getProduct 方法。

```
$ ./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
```

现在，可以在 Go gRPC 服务器端启用服务器端反射，并使用 CLI 工具对其进行测试，还可以在 Java gRPC 服务器端启用服务器端反射。如果你更熟悉 Java，那么可以参考源代码仓库中的 Java 示例。

接下来，我们讨论另外一个很有意思的概念，叫作 gRPC 中间件。

8.4 gRPC中间件

简单地说，在分布式系统中，中间件（middleware）是一个软件组件，用来连接不同的软件组件，从而将客户端生成的请求路由至后端服务器。在 gRPC 中间件中，我们讨论的也是在 gRPC 服务器端或客户端应用程序之前或之后运行代码。

实际上，gRPC 中间件基于第 5 章所介绍的拦截器的概念。它是基于 Go 语言的一组拦截器、辅助器（helper）和工具的集合，在构建基于 gRPC 的应用程序时，我们会用到它们。它允许在客户端或服务器端以拦截器链的形式应用多个拦截器。同时，因为拦截器经常用来实现通用的模式，如认证、日志、消息、校验、重试或监控，所以 gRPC 中间件项目是 Go 语言中实现这些可重用功能的首选方案。代码清单 8-4 展示了 gRPC 中间件包常见的用法。在这里，一元消息和流消息都使用了多个拦截器。

代码清单 **8-4**　在服务器端使用 Go gRPC 中间件实现的拦截器链

```
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 为服务器端添加一元拦截器链。

❷ 为服务器端添加流拦截器链。

这些拦截器会按照 Go gRPC 中间件注册的顺序进行调用。该项目还为通用的模式提供了一些可重用的拦截器。以下是一些通用的模式和拦截器实现。

认证

`grpc_auth`

可自定义的（通过 `AuthFunc`）认证中间件。

日志

`grpc_ctxtags`

添加 Tag map 到上下文的库，数据是通过请求体来填充的。

`grpc_zap`

将 zap 日志库集成到 gRPC `handler` 中。

`grpc_logrus`

将 logrus 日志库集成到 gRPC `handler` 中。

监控

`grpc_prometheus`

Prometheus 客户端和服务器端的监控中间件。

`grpc_opentracing`

OpenTracing 客户端和服务器端的拦截器，支持流和 `handler` 返回的标签。

客户端

`grpc_retry`

通用的 gRPC 响应码重试机制，客户端中间件。

服务器端

`grpc_validator`

根据 .proto 选项生成入站消息校验。

`grpc_recovery`

将 panic 转换成 gRPC 错误。

`ratelimit`

通过自己的限制器对 gRPC 进行速度限制。

在客户端，Go gRPC 中间件的用法是完全相同的。代码清单 8-5 展示了使用 Go gRPC 中间件实现客户端拦截器链的代码片段。

代码清单 **8-5**　在客户端使用 Go gRPC 中间件实现的拦截器链

```
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 客户端一元拦截器链。

❷ 客户端流拦截器链。

与服务器端类似，拦截器会按照它们在客户端注册的顺序执行。

接下来将讨论如何暴露 gRPC 服务器的健康状态。在一个高可用的系统中，具有一个检查服务器健康状态的方法是至关重要的，这样能定期检查并采取措施来减少损害。

8.5 健康检查协议

gRPC 定义了一个健康检查协议（Health Checking API），它允许 gRPC 服务暴露服务器的状态，这样消费者就能探查服务器的健康信息。服务器的健康情况是由服务器是否响应非健康状态来确定的，当服务器还没有准备好处理 RPC 或者根本没有响应健康探针的请求时，就会发生这种情况。如果响应表明服务器处于非健康状态或者在规定的时间窗口内没有收到响应，客户端就可以采取相应的行动了。

gRPC 健康检查协议基于 gRPC 定义了 API。gRPC 服务作为健康检查机制，适用于普通客户端–服务器端场景以及其他的控制系统（如负载均衡）。代码清单 8-6 展示了 gRPC 健康检查 API 的标准服务定义。

代码清单 **8-6** 健康检查 API 的 gRPC 服务定义

```
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 健康检查请求的消息结构。

❷ 带有服务状态的健康检查响应。

❸ 客户端可以通过调用 Check 方法查询服务器的健康状态。

❹ 客户端可以通过调用 Watch 方法执行流式健康检查。

健康检查服务的实现与常规的 gRPC 服务非常类似。通常会使用多路复用（第 5 章讨论过），将健康检查服务和相关的 gRPC 业务服务放到同一个 gRPC 服务器实例中。因为它是一个 gRPC 服务，所以进行健康检查与执行正常的 RPC 是一样的。它还提供了一个细粒度的服务健康语义，其中包含了每个服务健康状态的详情。同时，它还能够重用服务器上所有的已有信息，并对其进行全面控制。

基于代码清单 8-6 所示的服务接口，客户端可以调用 Check 方法（带有一个可选参数的服务名称）来检查特定服务或服务器的健康状态。

此外，客户端还可以调用 Watch 方法来执行流式健康检查。这样会采用服务器流消息模式，这意味着客户端调用方法后，服务器端会发送表示当前状态的消息，并且每当状态发生变化时，都会发送后续的新消息。

在 gRPC 健康检查协议中，有几个关键的地方需要注意。

- 为了反映服务器端注册的每个服务的状态，应该手动注册所有服务及其在服务器端的状态。还需要以空的服务名来设置服务器的整体状态。
- 客户端的每个健康检查请求都应该设置一个截止时间，如果 RPC 没有在截止时间内完成，客户端就可以确定服务器处于非健康状态。
- 对于每个健康检查请求，客户端可以设置一个服务名，也可以将其留空。如果请求带有服务名，并且服务可以在服务器注册中心找到，则响应必须是 HTTP OK 状态，并且 HealthCheckResponse 消息的状态字段要设置为特定服务的状态（SERVING 或 NOT_SERVING）。如果在服务器注册中心中无法找到该服务，则服务器应该响应 NOT_FOUND 状态。
- 如果客户端需要查询服务器的整体状态，而不是特定服务的状态，那么客户端在发送请求时，可以带一个空的字符串值，这样服务器就会以整体健康状态作为响应。

- 如果服务器端没有健康检查 API，那么客户端需要自行处理。

健康检查服务是被其他 gRPC 消费者或中间子系统（如负载均衡器或代理）所使用的。相对于从头实现客户端，我们更倾向于使用现有的健康检查客户端实现，如 `grpc_health_probe`。

8.6 gRPC健康探针

gRPC 健康探针 grpc_health_probe 是社区提供的一个工具，用来检查服务器的健康状态，服务器通过 gRPC 健康检查协议将它的状态暴露为服务。这是一个通用的客户端，能够与 gRPC 标准的健康检查服务通信。可以以 CLI 工具的方式来使用 grpc_health_probe，如下所示：

```
$ 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 向本地端口 50051 运行的 gRPC 服务器发送健康检查请求。

❷ 带有连接性相关参数的健康检查请求。

如前面的 CLI 输出所示，grpc_health_probe 发送了一个对 /grpc.health.v1.Health/Check 的 RPC 请求。如果它以 SERVING 状态作为响应，那么 grpc_health_probe 会成功退出；否则，它退出时会给出一个非零的退出码。

如果是在 Kubernetes 上运行 gRPC 应用程序，那么就可以运行 grpc_health_probe，作为 Kubernetes 中 gRPC 服务器端 pod 的存活性和就绪性状态检查。

为了实现这一点，可以将 gRPC 健康探针一起打包到 Docker 镜像中，如以下 Dockerfile 片段所示：

```
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
```

在 Kubernetes `Deployment` 的 pod 定义中，我们可以按照以下方式定义 `livenessProbe` 和 `readinessProbe`：

```
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 声明 `grpc_health_probe` 作为就绪性探针。

❷ 声明 `grpc_health_probe` 作为存活性探针。

在将存活性探针和就绪性探针设置为 gRPC 健康探针之后，Kubernetes 就可以基于 gRPC 服务器的状态做决策了。

8.7 其他生态系统项目

在构建基于 gRPC 的应用程序时，还有一些其他很有用的生态系统项目。定制化 protoc 插件是一个类似的生态系统需求，像 protoc-gen-star（PG*）这样的库正在得到越来越多的关注。另外，protoc-gen-validate（PGV）库提供了 protoc 插件，以生成多语言的消息验证器。随着生态系统的不断增长，不断有新项目出现，以满足 gRPC 应用程序开发中的各种需求。

到此为止，我们结束对 gRPC 生态系统组件的讨论。需要记住的是，这些生态系统项目不是 gRPC 项目的一部分。在生产环境中使用它们之前，要对其进行恰当的评估。同时，它们可能会不断变化：有些项目会过时，有些可能会成为主流，而在 gRPC 生态系统中，可能还会涌现其他全新的项目。

8.8 小结

可以看到，尽管 gRPC 生态系统项目不是核心 gRPC 实现的一部分，但在构建和运行真正的 gRPC 应用程序时，它们可能会非常有用。这些项目是围绕 gRPC 构建的，用来克服使用 gRPC 构建生产系统时所遇到的问题或限制。例如，当从 RESTful 服务迁移至 gRPC 服务时，需要考虑如何使用 RESTful 方式调用服务的现有客户端。为解决这种问题，引入了 HTTP/JSON 转码和 gRPC 网关的概念，这样现有的 RESTful 客户端和新的 gRPC 客户端都可以调用相同的服务。同样，为了解决使用 CLI 工具测试 gRPC 服务所面临的问题，又引入了服务器端反射。

因为 gRPC 在云原生领域非常流行，所以开发人员正在从 REST 服务逐渐转移至 gRPC，在未来我们会看到更多类似的基于 gRPC 的项目。

恭喜你！你现在已经读完了本书！你几乎游历了构建 gRPC 应用程序的整个生命周期，学习了大量基于 Go 语言和 Java 语言的代码示例。希望本书能够为你在应用程序和微服务中将 gRPC 作为进程间通信技术来使用打下坚实的基础。本书的内容会帮助你快速构建 gRPC 应用程序，理解它们如何与其他技术共存，并在生产环境中运行它们。

是时候进一步探索 gRPC 了。请尝试使用在本书中学到的技术来构建真实的应用程序。由于 gRPC 有大量的特性依赖于开发 gRPC 应用程序时所使用的编程语言，因此你必须学习所使用语言的某些特定技术。此外，gRPC 生态系统正在呈指数级增长，时刻了解支撑 gRPC 的最新技术和框架大有裨益。请继续探索吧！

关于作者

卡山 • 因德拉西里（**Kasun Indrasiri**）是一位作家和架构师，他在微服务、云原生和企业集成架构方面拥有丰富的经验，是 WSO2 的集成架构总监和 WSO2 Enterprise Integrator 的产品经理。他撰写了 *Microservices for the Enterprise* 一书，并在多个会议上发表演讲，包括 2019 年在美国圣何塞举行的 O'Reilly 软件架构师会议、2019 年在美国芝加哥举行的 GOTO Con 大会，以及 WSO2 的各种会议。卡山现居圣何塞，是“硅谷微服务、API 和集成”聚会的创始人，这是旧金山湾区最大的微服务聚会之一。

丹尼什 • 库鲁普（**Danesh Kuruppu**）是 WSO2 的技术副主管，在企业集成和微服务技术方面有 5 年多的经验。丹尼什是为开源、云原生编程语言 Ballerina 添加 gRPC 支持的主要设计者和开发者。他是 gRPC 社区的成员，也是 WSO2 Java 微服务框架和 WSO2 治理注册中心的主要贡献者。

关于封面

本书封面上的动物是大美洲斑背潜鸭（学名 Aythya marila）。春夏季节，这种鸭子在环极地苔原中繁殖，然后在冬季迁徙到北美、欧洲和亚洲的沿海地区。雄性斑背潜鸭眼睛黄色，喙亮蓝色，头部亮黑色且带有明显的深绿色，侧面白色，背上带有精细的灰色和白色羽毛。在筑巢时，为了保护自己，雌性斑背潜鸭身上的颜色很微妙，喙是淡蓝色的，脸上有一小块白色斑点，头和身体是棕色的。这些鸭子平均长约 50 厘米，翼展超过 76 厘米，平均体重约 1 千克。

大美洲斑背潜鸭在春天交配，一只雌鸭平均会在地上的巢里产下 8 枚蛋，巢里铺着从它自己身上掉下来的羽毛。小鸭子孵化后立即离巢，它们从一出生就能自己觅食。雏鸭需要 40 多天的时间才能长出羽毛，虽然有母亲的保护，但这时它们很容易受到猛禽和陆地食肉动物（如狐狸）的攻击。

斑背潜鸭属于潜水鸭的一种，但它们主要在陆地或水面上觅食，也会潜入水下寻找食物。和其他潜水鸭一样，斑背潜鸭的腿在其紧凑的身体后部，以帮助它们在水下前进。因为它们的生理机能经过演化，所以在潜水时能够使用更少的氧气。大美洲斑背潜鸭可以潜到 6 米的深度，能够憋气大约一分钟，这使它们能够比在其他潜水鸭更深的地方觅食。

尽管在过去的四十年里，这些鸭子的数量一直在减少，但它们目前被世界自然保护联盟濒危物种红色名录列为“最不值得关注的”。尽管如此，O'Reilly 图书封面上的许多动物濒临灭绝，但它们对世界很重要。

封面上的彩色插图由 Karen Montgomery 创作，以 *British Birds* 的黑白版画为基础。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks
