内部资料

注意保密

Internal Document

Confidential

gRPC技术手册

**编 号： V0.8**

**版 本： 初稿**

**拟制人： 李涛**

**文件修订记录**

**版本修改记录** （A --- 增加 M --- 修改 D --- 删除）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **当前版本** | **日期** | **修改位置**  **（图、表、章节）** | **A/M/D** | **简单描述**  **（原因和修改简介）** | **修改人** | **评审人** |
| 初稿 | 2018-4-29 | ALL | A | 创建 | 李涛 |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

目录

[gRPC技术手册 1](#_Toc513542815)

[1. 前言 8](#_Toc513542816)

[1.1. 简介 8](#_Toc513542817)

[1.2. 开始 8](#_Toc513542818)

[1.3. gRPC 是什么？ 8](#_Toc513542819)

[2. 基本概念 9](#_Toc513542820)

[2.1. 概览 9](#_Toc513542821)

[2.1.1. 服务定义 9](#_Toc513542822)

[2.1.2. 使用 API 接口 11](#_Toc513542823)

[2.1.3. 同步 vs 异步 11](#_Toc513542824)

[2.2. RPC 生命周期 12](#_Toc513542825)

[2.2.1. 单项 RPC 12](#_Toc513542826)

[2.2.2. 服务端流式 RPC 13](#_Toc513542827)

[2.2.3. 客户端流式 RPC 13](#_Toc513542828)

[2.2.4. 双向流式 RPC 13](#_Toc513542829)

[2.2.5. 截止时间 14](#_Toc513542830)

[2.2.6. RPC 终止 14](#_Toc513542831)

[2.2.7. 取消 RPC 14](#_Toc513542832)

[2.2.8. 元数据集 15](#_Toc513542833)

[2.2.9. 流控制 15](#_Toc513542834)

[2.2.10. 配置 15](#_Toc513542835)

[2.2.11. 频道 15](#_Toc513542836)

[3. 安全认证 16](#_Toc513542837)

[3.1. 支持的授权机制 16](#_Toc513542838)

[3.1.1. SSL/TLS 16](#_Toc513542839)

[3.1.2. OAuth 2.0 16](#_Toc513542840)

[3.2. API 17](#_Toc513542841)

[3.2.1. 服务端认证加密使用的 SSL/TLS 17](#_Toc513542842)

[3.2.2. 通过 Google 进行认证 18](#_Toc513542843)

[3.2.3. 扩展 gRPC 支持其他的认证机制 19](#_Toc513542844)

[3.3. 例子 20](#_Toc513542845)

[3.3.1. 通过 SSL/TLS 进行服务端授权和加密(Ruby) 20](#_Toc513542846)

[3.3.2. 通过 SSL/TLS 进行服务端授权和加密 (C#) 21](#_Toc513542847)

[3.3.3. 通过 SSL/TLS 进行服务端授权和加密 (Python) 21](#_Toc513542848)

[3.3.4. 通过 Google 进行授权 (Ruby) 22](#_Toc513542849)

[3.3.5. 通过 Google 进行授权 (Node.js) 23](#_Toc513542850)

[3.3.6. 通过 Google 进行授权 (C#) 25](#_Toc513542851)

[3.3.7. 通过 Google 进行授权 (PHP) 27](#_Toc513542852)

[3.3.8. 通过 Google 进行授权 (Python) 29](#_Toc513542853)

[3.3.9. 用 OAuth2 令牌进行认证(传统途径) 29](#_Toc513542854)

[4. 通讯协议 30](#_Toc513542855)

[4.1. 大纲 30](#_Toc513542856)

[4.2. 请求 30](#_Toc513542857)

[4.3. 应答 33](#_Toc513542858)

[4.4. 例子 34](#_Toc513542859)

[4.5. 用户代理 36](#_Toc513542860)

[4.6. HTTP2 传输映射 37](#_Toc513542861)

[4.6.1. 流识别 37](#_Toc513542862)

[4.6.2. 数据帧 38](#_Toc513542863)

[4.6.3. 错误 38](#_Toc513542864)

[4.6.4. 安全 39](#_Toc513542865)

[4.6.5. 连接管理 40](#_Toc513542866)

[4.7. 附录 A - Protobuf 上的 GRPC 41](#_Toc513542867)

[5. Go教程 41](#_Toc513542868)

[5.1. 为什么使用 gRPC? 42](#_Toc513542869)

[5.2. 例子的代码和设置 42](#_Toc513542870)

[5.3. 定义服务 43](#_Toc513542871)

[5.4. 生成客户端和服务器端代码 45](#_Toc513542872)

[5.5. 创建服务器 46](#_Toc513542873)

[5.5.1. 实现RouteGuide 46](#_Toc513542874)

[5.5.2. 启动服务器 53](#_Toc513542875)

[5.6. 创建客户端 54](#_Toc513542876)

[5.6.1. 创建存根 54](#_Toc513542877)

[5.6.2. 调用服务方法 55](#_Toc513542878)

[5.7. 来试试吧！ 60](#_Toc513542879)

# 前言

## 简介

[gRPC](http://www.oschina.net/p/grpc-framework) 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。目前提供 C、Java 和 Go 语言版本，分别是：grpc, grpc-java, grpc-go. 其中 C 版本支持 C, C++, Node.js, Python, Ruby, Objective-C, PHP 和 C# 支持.

gRPC 基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特。这些特性使得其在移动设备上表现更好，更省电和节省空间占用。

《gRPC 官方文档中文版》原文出自《 [gRPC Docs](http://www.grpc.io/docs/)》，由多位网友在开源中国众包平台协作翻译完成。

## 开始

欢迎进入 gRPC 的开发文档，gRPC 一开始由 google 开发，是一款语言中立、平台中立、开源的远程过程调用(RPC)系统。 本文档通过快速概述和一个简单的 Hello World 例子来向您介绍 gRPC 。你可以在本站发现更详细的教程和参考文档——文档将会越来越丰富。

## gRPC 是什么？

在 gRPC 里*客户端*应用可以像调用本地对象一样直接调用另一台不同的机器上*服务端*应用的方法，使得您能够更容易地创建分布式应用和服务。与许多 RPC 系统类似，gRPC 也是基于以下理念：定义一个*服务*，指定其能够被远程调用的方法（包含参数和返回类型）。在服务端实现这个接口，并运行一个 gRPC 服务器来处理客户端调用。在客户端拥有一个*存根*能够像服务端一样的方法。

# 基本概念

本文档通过对于 gRPC 的架构和 RPC 生命周期的概览来介绍 gRPC 的主要概念。本文是在假设你已经读过文档部分的前提下展开的。针对具体语言细节请查看对应语言的快速开始、教程和参考文档（很快就会有完整的文档）。

## 概览

### 服务定义

正如其他 RPC 系统，gRPC 基于如下思想：定义一个服务， 指定其可以被远程调用的方法及其参数和返回类型。gRPC 默认使用 [protocol buffers](https://developers.google.com/protocol-buffers/) 作为接口定义语言，来描述服务接口和有效载荷消息结构。如果有需要的话，可以使用其他替代方案。

service HelloService {

rpc SayHello (HelloRequest) returns (HelloResponse);

}

message HelloRequest {

required string greeting = 1;

}

message HelloResponse {

required string reply = 1;

}

gRPC 允许你定义四类服务方法：

* 单项 RPC，即客户端发送一个请求给服务端，从服务端获取一个应答，就像一次普通的函数调用。

rpc SayHello(HelloRequest) returns (HelloResponse){

}

* 服务端流式 RPC，即客户端发送一个请求给服务端，可获取一个数据流用来读取一系列消息。客户端从返回的数据流里一直读取直到没有更多消息为止。

rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){

}

* 客户端流式 RPC，即客户端用提供的一个数据流写入并发送一系列消息给服务端。一旦客户端完成消息写入，就等待服务端读取这些消息并返回应答。

rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {

}

* 双向流式 RPC，即两边都可以分别通过一个读写数据流来发送一系列消息。这两个数据流操作是相互独立的，所以客户端和服务端能按其希望的任意顺序读写，例如：服务端可以在写应答前等待所有的客户端消息，或者它可以先读一个消息再写一个消息，或者是读写相结合的其他方式。每个数据流里消息的顺序会被保持。

rpc BidiHello(stream HelloRequest) returns (stream HelloResponse){

}

我们将在下面 RPC 生命周期章节里看到各类 RPC 的技术细节。

### 使用 API 接口

gRPC 提供 protocol buffer 编译插件，能够从一个服务定义的 .proto 文件生成客户端和服务端代码。通常 gRPC 用户可以在服务端实现这些API，并从客户端调用它们。

* 在服务侧，服务端实现服务接口，运行一个 gRPC 服务器来处理客户端调用。gRPC 底层架构会解码传入的请求，执行服务方法，编码服务应答。
* 在客户侧，客户端有一个*存根*实现了服务端同样的方法。客户端可以在本地存根调用这些方法，用合适的 protocol buffer 消息类型封装这些参数— gRPC 来负责发送请求给服务端并返回服务端 protocol buffer 响应。

### 同步 vs 异步

同步 RPC 调用一直会阻塞直到从服务端获得一个应答，这与 RPC 希望的抽象最为接近。另一方面网络内部是异步的，并且在许多场景下能够在不阻塞当前线程的情况下启动 RPC 是非常有用的。

在多数语言里，gRPC 编程接口同时支持同步和异步的特点。你可以从每个语言教程和参考文档里找到更多内容(很快就会有完整文档)。

## RPC 生命周期

现在让我们来仔细了解一下当 gRPC 客户端调用 gRPC 服务端的方法时到底发生了什么。我们不究其实现细节，关于实现细节的部分，你可以在我们的特定语言页面里找到更为详尽的内容。

### 单项 RPC

首先我们来了解一下最简单的 RPC 形式：客户端发出单个请求，获得单个响应。

* 一旦客户端通过桩调用一个方法，服务端会得到相关通知 ，通知包括客户端的元数据，方法名，允许的响应期限（如果可以的话）
* 服务端既可以在任何响应之前直接发送回初始的元数据，也可以等待客户端的请求信息，到底哪个先发生，取决于具体的应用。
* 一旦服务端获得客户端的请求信息，就会做所需的任何工作来创建或组装对应的响应。如果成功的话，这个响应会和包含状态码以及可选的状态信息等状态明细及可选的追踪信息返回给客户端 。
* 假如状态是 OK 的话，客户端会得到应答，这将结束客户端的调用。

### 服务端流式 RPC

服务端流式 RPC 除了在得到客户端请求信息后发送回一个应答流之外，与我们的简单例子一样。在发送完所有应答后，服务端的状态详情(状态码和可选的状态信息)和可选的跟踪元数据被发送回客户端，以此来完成服务端的工作。客户端在接收到所有服务端的应答后也完成了工作。

### 客户端流式 RPC

客户端流式 RPC 也基本与我们的简单例子一样，区别在于客户端通过发送一个请求流给服务端，取代了原先发送的单个请求。服务端通常（但并不必须）会在接收到客户端所有的请求后发送回一个应答，其中附带有它的状态详情和可选的跟踪数据。

### 双向流式 RPC

双向流式 RPC ，调用由客户端调用方法来初始化，而服务端则接收到客户端的元数据，方法名和截止时间。服务端可以选择发送回它的初始元数据或等待客户端发送请求。 下一步怎样发展取决于应用，因为客户端和服务端能在任意顺序上读写 - 这些流的操作是完全独立的。例如服务端可以一直等直到它接收到所有客户端的消息才写应答，或者服务端和客户端可以像"乒乓球"一样：服务端后得到一个请求就回送一个应答，接着客户端根据应答来发送另一个请求，以此类推。

### 截止时间

gRPC 允许客户端在调用一个远程方法前指定一个最后期限值。这个值指定了在客户端可以等待服务端多长时间来应答，超过这个时间值 RPC 将结束并返回DEADLINE\_EXCEEDED错误。在服务端可以查询这个期限值来看是否一个特定的方法已经过期，或者还剩多长时间来完成这个方法。 各语言来指定一个截止时间的方式是不同的 - 比如在 Python 里一个截止时间值总是必须的，但并不是所有语言都有一个默认的截止时间。

### RPC 终止

在 gRPC 里，客户端和服务端对调用成功的判断是独立的、本地的，他们的结论可能不一致。这意味着，比如你有一个 RPC 在服务端成功结束("我已经返回了所有应答!")，到那时在客户端可能是失败的("应答在最后期限后才来到!")。也可能在客户端把所有请求发送完前，服务端却判断调用已经完成了。

### 取消 RPC

无论客户端还是服务端均可以再任何时间取消一个 RPC 。一个取消会立即终止 RPC 这样可以避免更多操作被执行。它*不是*一个"撤销"， 在取消前已经完成的不会被回滚。当然，通过同步调用的 RPC 不能被取消，因为直到 RPC 结束前，程序控制权还没有交还给应用。

### 元数据集

元数据是一个特殊 RPC 调用对应的信息([授权详情](http://doc.oschina.net/docs/guides/auth.html)]) ，这些信息以键值对的形式存在，一般键的类型是字符串，值的类型一般也是字符串(当然也可以是二进制数据)。元数据对 gRPC 本事来说是不透明的 - 它让客户端提供调用相关的信息给服务端，反之亦然。 对于元数据的访问是语言相关的。

### 流控制

TBD

### 配置

TBD

### 频道

在创建客户端存根时，一个 gRPC 频道提供一个特定主机和端口服务端的连接。客户端可以通过指定频道参数来修改 gRPC 的默认行为，比如打开关闭消息压缩。一个频道具有状态，包含已连接和空闲 。 gRPC 如何处理关闭频道是语言相关的。有些语言可允许询问频道状态。

# 安全认证

gRPC 被设计成可以利用插件的形式支持多种授权机制。本文档对多种支持的授权机制提供了一个概览，并且用例子来论述对应API，最后就其扩展性作了讨论。 马上将会推出更多文档和例子。

## 支持的授权机制

### SSL/TLS

gRP 集成 SSL/TLS 并对服务端授权所使用的 SSL/TLS 进行了改良，对客户端和服务端交换的所有数据进行了加密。对客户端来讲提供了可选的机制提供凭证来获得共同的授权。

### OAuth 2.0

gRPC 提供通用的机制（后续进行描述）来对请求和应答附加基于元数据的凭证。当通过 gRPC 访问 Google API 时，会为一定的授权流程提供额外的获取访问令牌的支持，这将通过以下代码例子进行展示。 *警告*：Google OAuth2 凭证应该仅用于连接 Google 的服务。把 Google 对应的 OAuth2 令牌发往非 Google 的服务会导致令牌被窃取用作冒充客户端来访问 Google 的服务。

## API

为了减少复杂性和将混乱最小化， gRPC 以一个统一的凭证对象来进行工作。 凭证可以是以下两类：

* *频道凭证*, 被附加在 频道上， 比如 SSL 凭证。
* *调用凭证*, 被附加在调用上(或者 C++ 里的 客户端上下文)。 凭证可以用组合频道凭证来进行组合。一个组合频道凭证可以将一个频道凭证和一个调用凭证关联创建一个新的频道凭证。结果在这个频道上的每次调用会发送组合的调用凭证来作为授权数据。 例如，一各频道凭证可以由一个Ssl 凭证和一个访问令牌凭证生成。结果是在这个频道上的每次调用都会发送对应的访问令牌。 调用凭证可以用 组合凭证来组装。组装后的 调用凭证应用到一个客户端上下文里，将触发发送这两个调用凭证的授权数据。

### 服务端认证加密使用的 SSL/TLS

这是个最简单的认证场景：一个客户端仅仅想认证服务器并且加密所有数据。

*// Create a default SSL ChannelCredentials object.*

auto channel\_creds = grpc::SslCredentials(grpc::SslCredentialsOptions());

*// Create a channel using the credentials created in the previous step.*

auto channel = grpc::CreateChannel(server\_name, creds);

*// Create a stub on the channel.*

std::unique\_ptr<Greeter::Stub> stub(Greeter::NewStub(channel));

*// Make actual RPC calls on the stub.*

grpc::Status s = stub->sayHello(&context, \*request, response);

对于高级的用例比如改变根 CA 或使用客户端证书，可以在发送给工厂方法的 SslCredentialsOptions 参数里的相应选项进行设置。

### 通过 Google 进行认证

gRPC应用可以使用一个简单的API来创建一个可以工作在不同部署场景下的凭证。

auto creds = grpc::GoogleDefaultCredentials();

*// Create a channel, stub and make RPC calls (same as in the previous example)*

auto channel = grpc::CreateChannel(server\_name, creds);

std::unique\_ptr<Greeter::Stub> stub(Greeter::NewStub(channel));

grpc::Status s = stub->sayHello(&context, \*request, response);

这个应用使用的频道凭证对象就像 [Google 计算引擎 (GCE)](https://cloud.google.com/compute/)里运行的应用一样使用服务账号。在前面的案例里，服务账号的密钥从环境变量 GOOGLE\_APPLICATION\_CREDENTIALS 对应的文件里加载。这些密钥被用来生成承载令牌附加在在相应频道的每次 RPC 调用里。 对于 GCE 里运行的应用，可以在虚拟机设置的时候为其配置一个默认的服务账号和相应的 OAuth2 范围。在运行时，这个凭证被用来与认证系统通讯来获取 OAuth2 访问令牌并且把令牌用作在相应的频道上的 RPC 调用。

### 扩展 gRPC 支持其他的认证机制

相应的凭证插件 API 允许开发者开发自己的凭证插件。

* MetadataCredentialsPlugin 抽象类包含需要被开发者创建的子类实现的纯虚方法 GetMetadata。
* MetadataCredentialsFromPlugin 方法可以从 MetadataCredentialsPlugin 创建一个 调用者凭证。 这类有个简单的凭证插件例子，是通过在自定义头了设置一个认证票据。

class MyCustomAuthenticator : public grpc::MetadataCredentialsPlugin {

public:

MyCustomAuthenticator(const grpc::string& ticket) : ticket\_(ticket) {}

grpc::Status GetMetadata(

grpc::string\_ref service\_url, grpc::string\_ref method\_name,

const grpc::AuthContext& channel\_auth\_context,

std::multimap<grpc::string, grpc::string>\* metadata) override {

metadata->insert(std::make\_pair("x-custom-auth-ticket", ticket\_));

return grpc::Status::OK;

}

private:

grpc::string ticket\_;

};

auto call\_creds = grpc::MetadataCredentialsFromPlugin(

std::unique\_ptr<grpc::MetadataCredentialsPlugin>(

new MyCustomAuthenticator("super-secret-ticket")));

更深层次的集成可以通过在将 gRPC 的凭证实现以插件的形式集成进核心层。gRPC 内部也允许用其他加密机制来替换 SSL/TLS 。

## 例子

这些授权机制将会在所有 gRPC 支持的语言里提供。以下的一些节里展示了上文提到的认证和授权在每种语言里如何实现：很快将会推出更多语言的支持。

### 通过 SSL/TLS 进行服务端授权和加密(Ruby)

***# Base case - No encryption***

stub = Helloworld::Greeter::Stub.new('localhost:50051', :this\_channel\_is\_insecure)

...

***# With server authentication SSL/TLS***

creds = GRPC::Core::Credentials.new(load\_certs) *# load\_certs typically loads a CA roots file*

stub = Helloworld::Greeter::Stub.new('localhost:50051', creds)

### 通过 SSL/TLS 进行服务端授权和加密 (C#)

*// Base case - No encryption/authentication*

var channel = new Channel("localhost:50051", ChannelCredentials.Insecure);

var client = new Greeter.GreeterClient(channel);

...

*// With server authentication SSL/TLS*

var channelCredentials = new SslCredentials(File.ReadAllText("roots.pem")); *// Load a custom roots file.*

var channel = new Channel("myservice.example.com", channelCredentials);

var client = new Greeter.GreeterClient(channel);

### 通过 SSL/TLS 进行服务端授权和加密 (Python)

from grpc.beta import implementations

import helloworld\_pb2

*# Base case - No encryption*

channel = implementations.insecure\_channel('localhost', 50051)

stub = helloworld\_pb2.beta\_create\_Greeter\_stub(channel)

...

*# With server authentication SSL/TLS*

creds = implementations.ssl\_channel\_credentials(open('roots.pem').read(), None, None)

channel = implementations.secure\_channel('localhost', 50051, creds)

stub = helloworld\_pb2.beta\_create\_Greeter\_stub(channel)

### 通过 Google 进行授权 (Ruby)

#### 基本案例 - 无加密/授权

stub = Helloworld::Greeter::Stub.new('localhost:50051', :this\_channel\_is\_insecure)

#### 用无限制凭证进行授权 (推荐途径)

require 'googleauth' *# from http://www.rubydoc.info/gems/googleauth/0.1.0*

...

ssl\_creds = GRPC::Core::ChannelCredentials.new(load\_certs) *# load\_certs typically loads a CA roots file*

authentication = Google::Auth.get\_application\_default()

call\_creds = GRPC::Core::CallCredentials.new(authentication.updater\_proc)

combined\_creds = ssl\_creds.compose(call\_creds)

stub = Helloworld::Greeter::Stub.new('greeter.googleapis.com', combined\_creds)

#### 用 OAuth2 令牌进行认证(传统途径)

require 'googleauth' *# from http://www.rubydoc.info/gems/googleauth/0.1.0*

...

ssl\_creds = GRPC::Core::ChannelCredentials.new(load\_certs) *# load\_certs typically loads a CA roots file*

scope = 'https://www.googleapis.com/auth/grpc-testing'

authentication = Google::Auth.get\_application\_default(scope)

call\_creds = GRPC::Core::CallCredentials.new(authentication.updater\_proc)

combined\_creds = ssl\_creds.compose(call\_creds)

stub = Helloworld::Greeter::Stub.new('greeter.googleapis.com', combined\_creds)

### 通过 Google 进行授权 (Node.js)

#### 基本案例 - 无加密/授权

var stub = new helloworld.Greeter('localhost:50051', grpc.credentials.createInsecure());

#### 用无限制凭证进行授权 (推荐途径)

*// Authenticating with Google*

var GoogleAuth = require('google-auth-library'); *// from https://www.npmjs.com/package/google-auth-library*

...

var ssl\_creds = grpc.credentials.createSsl(root\_certs);

(new GoogleAuth()).getApplicationDefault(function(err, auth) {

var call\_creds = grpc.credentials.createFromGoogleCredential(auth);

var combined\_creds = grpc.credentials.combineChannelCredentials(ssl\_creds, call\_creds);

var stub = new helloworld.Greeter('greeter.googleapis.com', combined\_credentials);

});

#### 用 OAuth2 令牌进行认证(传统途径)

var GoogleAuth = require('google-auth-library'); *// from https://www.npmjs.com/package/google-auth-library*

...

var ssl\_creds = grpc.Credentials.createSsl(root\_certs); *// load\_certs typically loads a CA roots file*

var scope = 'https://www.googleapis.com/auth/grpc-testing';

(new GoogleAuth()).getApplicationDefault(function(err, auth) {

if (auth.createScopeRequired()) {

auth = auth.createScoped(scope);

}

var call\_creds = grpc.credentials.createFromGoogleCredential(auth);

var combined\_creds = grpc.credentials.combineChannelCredentials(ssl\_creds, call\_creds);

var stub = new helloworld.Greeter('greeter.googleapis.com', combined\_credentials);

});

### 通过 Google 进行授权 (C#)

#### 基本案例 - 无加密/授权

var channel = new Channel("localhost:50051", ChannelCredentials.Insecure);

var client = new Greeter.GreeterClient(channel);

...

#### 用无限制凭证进行授权 (推荐途径)

using Grpc.Auth; *// from Grpc.Auth NuGet package*

...

*// Loads Google Application Default Credentials with publicly trusted roots.*

var channelCredentials = await GoogleGrpcCredentials.GetApplicationDefaultAsync();

var channel = new Channel("greeter.googleapis.com", channelCredentials);

var client = new Greeter.GreeterClient(channel);

...

#### 用 OAuth2 令牌进行认证(传统途径)

using Grpc.Auth; *// from Grpc.Auth NuGet package*

...

string scope = "https://www.googleapis.com/auth/grpc-testing";

var googleCredential = await GoogleCredential.GetApplicationDefaultAsync();

if (googleCredential.IsCreateScopedRequired)

{

googleCredential = googleCredential.CreateScoped(new[] { scope });

}

var channel = new Channel("greeter.googleapis.com", googleCredential.ToChannelCredentials());

var client = new Greeter.GreeterClient(channel);

...

#### 授权一个 gRPC 调用

var channel = new Channel("greeter.googleapis.com", new SslCredentials()); *// Use publicly trusted roots.*

var client = new Greeter.GreeterClient(channel);

...

var googleCredential = await GoogleCredential.GetApplicationDefaultAsync();

var result = client.SayHello(request, new CallOptions(credentials: googleCredential.ToCallCredentials()));

...

### 通过 Google 进行授权 (PHP)

#### 基本案例 - 无加密/授权

$client = new helloworld\GreeterClient('localhost:50051', [

'credentials' => Grpc\ChannelCredentials::createInsecure(),

]);

...

#### 用无限制凭证进行授权 (推荐途径)

#### Authenticate using scopeless credentials (recommended approach)

function updateAuthMetadataCallback($context)

{

$auth\_credentials = ApplicationDefaultCredentials::getCredentials();

return $auth\_credentials->updateMetadata($metadata = [], $context->service\_url);

}

$channel\_credentials = Grpc\ChannelCredentials::createComposite(

Grpc\ChannelCredentials::createSsl(file\_get\_contents('roots.pem')),

Grpc\CallCredentials::createFromPlugin('updateAuthMetadataCallback')

);

$opts = [

'credentials' => $channel\_credentials

];

$client = new helloworld\GreeterClient('greeter.googleapis.com', $opts);

`

#### 用 OAuth2 令牌进行认证(传统途径)

*// the environment variable "GOOGLE\_APPLICATION\_CREDENTIALS" needs to be set*

$scope = "https://www.googleapis.com/auth/grpc-testing";

$auth = Google\Auth\ApplicationDefaultCredentials::getCredentials($scope);

$opts = [

'credentials' => Grpc\Credentials::createSsl(file\_get\_contents('roots.pem'));

'update\_metadata' => $auth->getUpdateMetadataFunc(),

];

$client = new helloworld\GreeterClient('greeter.googleapis.com', $opts);

### 通过 Google 进行授权 (Python)

#### 基本案例 - 无加密/授权

channel = implementations.insecure\_channel('localhost', 50051)

stub = helloworld\_pb2.beta\_create\_Greeter\_stub(channel)

...

### 用 OAuth2 令牌进行认证(传统途径)

transport\_creds = implementations.ssl\_channel\_credentials(open('roots.pem').read(), None, None)

def oauth2token\_credentials(context, callback):

try:

credentials = oauth2client.client.GoogleCredentials.get\_application\_default()

scoped\_credentials = credentials.create\_scoped([scope])

except Exception as error:

callback([], error)

return

callback([('authorization', 'Bearer %s' % scoped\_credentials.get\_access\_token().access\_token)], None)

auth\_creds = implementations.metadata\_plugin\_credentials(oauth2token\_credentials)

channel\_creds = implementations.composite\_channel\_credentials(transport\_creds, auth\_creds)

channel = implementations.secure\_channel('localhost', 50051, channel\_creds)

stub = helloworld\_pb2.beta\_create\_Greeter\_stub(channel)

# 通讯协议

本文档作为 gRPC 在 HTTP2 草案17框架上的实现的详细描述，假设你已经熟悉 HTTP2 的规范。产品规则采用的是[ABNF 语法](http://tools.ietf.org/html/rfc5234)

## 大纲

以下是 gRPC 请求和应答消息流中一般的消息顺序：

* 请求 → 请求报头 \*有定界符的消息 EOS
* 应答 → 应答报头 \*有定界符的消息 EOS
* 应答 → (应答报头 \*有定界符的消息 跟踪信息) / 仅仅跟踪时

## 请求

* 请求 → 请求报头 \*界定的消息 EOS 请求报头是通过报头+联系帧方式以 HTTP2 报头来发送的。
* **请求报头** → 调用定义 \*自定义元数据
* **调用定义** → 方法模式路径TE [授权] [超时] [内容类型] [消息类型] [消息编码] [接受消息类型] [用户代理]
* **方法** → “:method POST”
* **模式** → “:scheme ” (“http” / “https”)
* **路径** → “:path” {*开放的 API 对应的方法路径*}
* **Authority** → “:authority” {*授权的对应的虚拟主机域名*}
* **TE** → “te” “trailers” # 用来检测不兼容的代理
* **超时** → “grpc-timeout” 超时时间值 超时时间单位
* **超时时间值** → {*至少8位数字正整数的 ASCII 码字符串*}
* **超时时间单位** → 时 / 分 / 秒 / 毫秒 / 微秒 / 纳秒
* **时** → “H”
* **分** → “M”
* **秒** → “S”
* **毫秒** → “m”
* **微秒** → “u”
* **纳秒** → “n”
* **内容类型** → “content-type” “application/grpc” [(“+proto” / “+json” / {*自定义*})]
* **内容编码** → “gzip” / “deflate” / “snappy” / {*自定义*}
* **消息编码** → “grpc-encoding” Content-Coding
* **接受消息编码** → “grpc-accept-encoding” Content-Coding \*("," Content-Coding)
* **用户代理** → “user-agent” {*结构化的用户代理字符串*}
* **消息类型** → “grpc-message-type” {*消息模式的类型名*}
* **自定义数据** → 二进制报头 / ASCII 码报头
* **二进制报头** → {以“-bin”结尾小写的报头名称的 ASCII 码 } {*以 base64 进行编码的值*}
* **ASCII 码报头** → {小写报头名称的 ASCII 码} {*值*}

HTTP2 需要一个在其他报头之前以“:”开始的保留报头。额外的实现应该在保留报头后面马上传送**超时信息**，并且应该在发送**自定义元数据**前发送**调用定义**报头。 如果**超时信息**被遗漏，服务端会认为是无限时长的超时。客户端实现可以根据发布需要自由地发送一个默认最小超时时间。 **自定义元数据**是应用层定义的任意的键值对集合。除了 HTTP2 报头部总长度的传输限制外，唯一的约束就是以“grpc-”开始的报头名称是为将来使用保留的。

注意 HTTP2 并不允许随意使用字节序列来作为报头值，所以二进制的报头值必须使用 Base64 来编码，参见<https://tools.ietf.org/html/rfc4648#section-4>。 实现必须接受填充的和非填充的值，并且发出非填充的值。应用以“-bin”结尾的名称来定义二进制报头。运行时库在报头被发送和接收时，用这个后缀来检测二进制报头并且正确地在报头被发送和接收时进行 Base64 编码和解码。

**界定的消息**的重复序列通过数据帧来进行传输。

* **界定的消息** → 压缩标志 消息长度 消息
* **压缩标志** → 0 / 1 # 编码为 1 byte 的无符号整数
* **消息长度** → {*消息长度*} # 编码为 4 byte 的无符号整数
* **消息** → \*{二进制字节}

**压缩标志** 值为1 表示**消息**的二进制序列通过**消息编码**报头声明的机制进行压缩，为0表示消息的字节码没有进行编码。压缩上下文不在消息编辑间维护，声明必须为流中的每个消息创建一个新的上下文。假如 **压缩标志** 被遗漏了，那么**压缩标志** 必须为0。

对请求来讲，**EOS** (end-of-stream)以最后接收到的数据帧出现 END\_STREAM 标志为准。 在**请求流**需要关闭但是没有数据继续发送的情况下，代码必须发送包含这个标志的空数据帧。

## 应答

* **应答** → (应答报头 界定的消息 跟踪信息) / 仅仅跟踪
* **应答报头** → HTTP 状态 [消息编码] [消息接受编码] 内容类型 \*自定义元数据
* **仅仅跟踪** → HTTP 状态 内容类型 跟踪消息
* **跟踪消息** → 状态 [状态消息] \*自定义元数据
* **HTTP状态** → “:status 200”
* **状态** → “grpc-status” <状态码的 ASCII 字符串>
* **状态消息** → “grpc-message” <状态描述文本对应的 ASCII 字符串>

**应答报头** 和 **仅仅跟踪** 分别在一个HTTP2报头帧块里发送。大多数应答期望既有报头又有跟踪消息，但是调用允许**仅仅跟踪**生成一个立即的错误。假如状态码是 OK 的话，则必须在**跟踪消息**里发送状态。 对于应答来讲，通过在最后一个接收的包含跟踪信息的报头帧里提供一个 END\_STREAM 标志来表明流结束。

实现应当会让中断的部署在应答里发送一个非200的HTTP状态码和一系列非GRPC内容类型并且省略**状态**和**状态消息**。 当发生这种情况时实现应当合成**状态**和**状态消息**来扩散到应用层。

## 例子

单项调用HTTP2帧序列例子

**请求**

HEADERS (flags = END\_HEADERS)

:method = POST

:scheme = http

:path = /google.pubsub.v2.PublisherService/CreateTopic

:authority = pubsub.googleapis.com

grpc-timeout = 1S

content-type = application/grpc+proto

grpc-encoding = gzip

authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v

DATA (flags = END\_STREAM)

<Delimited Message>

**应答**

HEADERS (flags = END\_HEADERS)

:status = 200

grpc-encoding = gzip

DATA

<Delimited Message>

HEADERS (flags = END\_STREAM, END\_HEADERS)

grpc-status = 0 *# OK*

trace-proto-bin = jher831yy13JHy3hc

## 用户代理

当协议不需要一个用户代理时，建议客户端提供一个结构化的用户代理字符串来对要调用的库、版本和平台提供一个基本的描述来帮助在异质的环境里进行问题诊断。库开发者建议使用以下结构:

User-Agent → “grpc-” Language ?(“-” Variant) “/” Version ?( “ (“ \*(AdditionalProperty “;”) “)” )

例如

grpc-java/1.2.3

grpc-ruby/1.2.3

grpc-ruby-jruby/1.3.4

grpc-java-android/0.9.1 (gingerbread/1.2.4; nexus5; tmobile)

## HTTP2 传输映射

### 流识别

所有的 GRPC 调用需要定义指定一个内部 ID。我们将在这个模式里使用 HTTP2 流 ID 来作为调用标识。注意：这些 ID 在一个打开的 HTTP2 会话里是前后关联的，在一个处理多个 HTTP2 会话的进程里不是唯一的，也不能被用作 GUID。

### 数据帧

数据帧边界与**界定消息**的边界无关，实现时不应假定它们有一致性。

### 错误

当应用错误或运行时错误在 PRC 调用过程中出现时，**状态**和**状态消息**应当通过**跟踪消息**发送。

在有些情况下可能消息流的帧已经中断，RPC 运行时会选择使用 **RST\_STREAM** 帧来给对方表示这种状态。RPC 运行时声明应当将 RST\_STREAM 解释为流的完全关闭，并且将错误传播到应用层。 以下为从 RST\_STREAM 错误码到 GRPC 的错误码的映射：

| **HTTP2 编码** | **GRPC 编码** |
| --- | --- |
| NO\_ERROR(0) | INTERNAL -一个显式的GRPC OK状态应当被发出，但是这个也许在某些场景里会被侵略性地使用 |
| PROTOCOL\_ERROR(1) | INTERNAL |
| INTERNAL\_ERROR(2) | INTERNAL |
| FLOW\_CONTROL\_ERROR(3) | INTERNAL |
| SETTINGS\_TIMEOUT(4) | INTERNAL |
| STREAM\_CLOSED | 无映射，因为没有打开的流来传播。实现应记录。 |
| FRAME\_SIZE\_ERROR | INTERNAL |
| REFUSED\_STREAM | UNAVAILABLE-表示请求未作处理且可以重试，可能在他处重试。 |
| CANCEL(8) | 当是由客户端发出时映射为调用取消，当是由服务端发出时映射为 CANCELLED。注意服务端在需要取消调用时应仅仅使用这个机制，但是有效荷载字节顺序是不完整的 |
| COMPRESSION\_ERROR | INTERNAL |
| CONNECT\_ERROR | INTERNAL |
| ENHANCE\_YOUR\_CALM | RESOURCE\_EXHAUSTED...并且运行时提供有额外的错误详情，表示耗尽资源是带宽 |
| INADEQUATE\_SECURITY | PERMISSION\_DENIED... 并且有额外的信息表明许可被拒绝，因为对调用来说协议不够安全 |

### 安全

HTTP2 规范当使用 TLS 时强制使用 TLS 1.2 及以上的版本，并且在部署上对允许的密码施加一些额外的限制以避免已知的比如需要 SNI 支持的问题。并且期待 HTTP2 与专有的传输安全机制相结合，这些传输机制的规格说明不能提供有意义的建议。

### 连接管理

#### GOAWAY 帧

服务端发出这种帧给客户端表示服务端在相关的连接上不再接受任何新流。这种帧包含服务端最后成功接受的流的ID。客户端应该认为任何在最后成功的流后面初始化的任意流为 UNAVAILABLE，并且在别处重试这些调用。客户端可以自由地在已经接受的流上继续工作直到它们完成或者连接中断。 服务端应该在终止连接前发送 GOAWAY 帧，以可靠地通知客户端哪些工作已经被服务端接受并执行。

#### PING 帧

客户端和服务端均可以发送一个 PING 帧，对方必须精确回显它们所接收到的信息。这可以被用来确认连接仍然是活动的，并且能够提供估计端对端延迟估计的方法。假如服务端初始的 PING 在最后期限仍然没有收到运行时所期待的应答的话，所有未完成的调用将会被以取消状态关闭。一个客户端期满的初始的PING则会导致所有的调用被以用不可用状态关闭。注意PING的频率高度依赖于网络环境，实现可以根据网络和应用需要，自由地调整PING频率。

#### 连接失败

假如客户端检测到连接失败，所有的调用都会被以不可用状态关闭。而服务端侧则所有已经打开的调用都会被以取消状态关闭。

## 附录 A - Protobuf 上的 GRPC

用 protobuf 定义的服务接口可以通过 protoc 的代码生成扩展简单地映射成 GRPC ，以下定义了所用的映射：

* **路径** → / 服务名 / {*方法名*}
* **服务名** → ?( {*proto 包名*} "." ) {*服务名*}
* **消息类型** → {*全路径 proto 消息名*}
* **内容类型** → "application/grpc+proto"

# Go教程

本教程提供了 Go 程序员如何使用 gRPC 的指南。

通过学习教程中例子，你可以学会如何：

* 在一个 .proto 文件内定义服务。
* 用 protocol buffer 编译器生成服务器和客户端代码。
* 使用 gRPC 的 Go API 为你的服务实现一个简单的客户端和服务器。

假设你已经阅读了[概览](http://grpc.mydoc.io/?v=10467&t=58008) 并且熟悉[protocol buffers](https://developers.google.com/protocol-buffers/docs/overview)。 注意，教程中的例子使用的是 protocol buffers 语言的 proto3 版本，它目前只是 alpha 版：可以在 [proto3 语言指南](https://developers.google.com/protocol-buffers/docs/proto3)和 protocol buffers 的 Github 仓库的[版本注释](https://github.com/google/protobuf/releases)发现更多关于新版本的内容。

这算不上是一个在 Go 中使用 gRPC 的综合指南：以后会有更多的参考文档.

## 为什么使用 gRPC?

我们的例子是一个简单的路由映射的应用，它允许客户端获取路由特性的信息，生成路由的总结，以及交互路由信息，如服务器和其他客户端的流量更新。

有了 gRPC， 我们可以一次性的在一个 .proto 文件中定义服务并使用任何支持它的语言去实现客户端和服务器，反过来，它们可以在各种环境中，从Google的服务器到你自己的平板电脑—— gRPC 帮你解决了不同语言及环境间通信的复杂性.使用 protocol buffers 还能获得其他好处，包括高效的序列号，简单的 IDL 以及容易进行接口更新。

## 例子的代码和设置

教程的代码在这里 [grpc/grpc-go/examples/cpp/route\_guide](https://github.com/grpc/grpc-go/tree/master/examples/route_guide)。 要下载例子，通过运行下面的命令去克隆grpc-go代码库：

$ go get google.golang.org/grpc

然后改变当前的目录到 grpc-go/examples/route\_guide:

$ cd $GOPATH/src/google.golang.org/grpc/examples/route\_guide

你还需要安装生成服务器和客户端的接口代码相关工具-如果你还没有安装的话，请查看下面的设置指南 [Go快速开始指南](http://doc.oschina.net/docs/installation/go.html)。

## 定义服务

我们的第一步(可以从[概览](http://doc.oschina.net/docs/index.html)中得知)是使用 [protocol buffers](https://developers.google.com/protocol-buffers/docs/overview)去定义 gRPC *service* 和方法 *request* 以及 *response* 的类型。你可以在[examples/protos/route\_guide.proto](https://github.com/grpc/grpc/blob/%7B%7B%20site.data.config.grpc_release_branch%20%7D%7D/examples/protos/route_guide.proto)看到完整的 .proto 文件。

要定义一个服务，你必须在你的 .proto 文件中指定 service：

service RouteGuide {

...

}

然后在你的服务中定义 rpc 方法，指定请求的和响应类型。gRPC 允许你定义4种类型的 service 方法，这些都在 RouteGuide 服务中使用：

* 一个 *简单 RPC* ， 客户端使用存根发送请求到服务器并等待响应返回，就像平常的函数调用一样。

*// Obtains the feature at a given position.*

rpc GetFeature(Point) returns (Feature) {}

* 一个 *服务器端流式 RPC* ， 客户端发送请求到服务器，拿到一个流去读取返回的消息序列。 客户端读取返回的流，直到里面没有任何消息。从例子中可以看出，通过在 *响应* 类型前插入 stream 关键字，可以指定一个服务器端的流方法。

*// Obtains the Features available within the given Rectangle. Results are*

*// streamed rather than returned at once (e.g. in a response message with a*

*// repeated field), as the rectangle may cover a large area and contain a*

*// huge number of features.*

rpc ListFeatures(Rectangle) returns (stream Feature) {}

* 一个 *客户端流式 RPC* ， 客户端写入一个消息序列并将其发送到服务器，同样也是使用流。一旦客户端完成写入消息，它等待服务器完成读取返回它的响应。通过在 *请求* 类型前指定 stream 关键字来指定一个客户端的流方法。

// Accepts a stream of Points on a route being traversed, returning a

// RouteSummary when traversal is completed.

rpc RecordRoute(stream Point) returns (RouteSummary) {}

* 一个 *双向流式 RPC* 是双方使用读写流去发送一个消息序列。两个流独立操作，因此客户端和服务器可以以任意喜欢的顺序读写：比如， 服务器可以在写入响应前等待接收所有的客户端消息，或者可以交替的读取和写入消息，或者其他读写的组合。 每个流中的消息顺序被预留。你可以通过在请求和响应前加 stream 关键字去制定方法的类型。

// Accepts a stream of RouteNotes sent while a route is being traversed,

// while receiving other RouteNotes (e.g. from other users).

rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}

我们的 .proto 文件也包含了所有请求的 protocol buffer 消息类型定义以及在服务方法中使用的响 应类型——比如，下面的Point消息类型：

// Points are represented as latitude-longitude pairs in the E7 representation

// (degrees multiplied by 10\*\*7 and rounded to the nearest integer).

// Latitudes should be in the range +/- 90 degrees and longitude should be in

// the range +/- 180 degrees (inclusive).

message Point {

int32 latitude = 1;

int32 longitude = 2;

}

## 生成客户端和服务器端代码

接下来我们需要从 .proto 的服务定义中生成 gRPC 客户端和服务器端的接口。我们通过 protocol buffer 的编译器 protoc 以及一个特殊的 gRPC Go 插件来完成。

简单起见，我们提供一个 [bash 脚本](https://github.com/grpc/grpc-go/blob/master/codegen.sh) 帮你用合适的插件，输入，输出去运行 protoc(如果你想自己去运行，确保你已经安装了 protoc，并且请遵循下面的 gRPC-Go [安装指南](https://github.com/grpc/grpc-go/blob/master/README.md))来操作：

$ codegen.sh route\_guide.proto

实际上运行的是：

$ protoc --go\_out=plugins=grpc:. route\_guide.proto

运行这个命令可以在当前目录中生成下面的文件：

* route\_guide.pb.go

这些包括：

* 所有用于填充，序列化和获取我们请求和响应消息类型的 protocol buffer 代码
* 一个为客户端调用定义在RouteGuide服务的方法的接口类型（或者 *存根* ）
* 一个为服务器使用定义在RouteGuide服务的方法去实现的接口类型（或者 *存根* ）

## 创建服务器

首先来看看我们如何创建一个 RouteGuide 服务器。如果你只对创建 gRPC 客户端感兴趣，你可以跳 过这个部分，直接到[创建客户端](http://doc.oschina.net/grpc?t=60133#client) (当然你也可能发现它也很有意思)。

让 RouteGuide 服务工作有两个部分：

* 实现我们服务定义的生成的服务接口：做我们的服务的实际的“工作”。
* 运行一个 gRPC 服务器，监听来自客户端的请求并返回服务的响应。

你可以从[grpc-go/examples/route\_guide/server/server.go](https://github.com/grpc/grpc-go/tree/master/examples/route_guide/server/server.go)看到我们的 RouteGuide 服务器的实现代码。现在让我们近距离研究它是如何工作的。

### 实现RouteGuide

我们可以看出，服务器有一个实现了生成的 RouteGuideServer 接口的 routeGuideServer 结构类型：

type routeGuideServer struct {

...

}

...

func (s \*routeGuideServer) GetFeature(ctx context.Context, point \*pb.Point) (\*pb.Feature, error) {

...

}

...

func (s \*routeGuideServer) ListFeatures(rect \*pb.Rectangle, stream pb.RouteGuide\_ListFeaturesServer) error {

...

}

...

func (s \*routeGuideServer) RecordRoute(stream pb.RouteGuide\_RecordRouteServer) error {

...

}

...

func (s \*routeGuideServer) RouteChat(stream pb.RouteGuide\_RouteChatServer) error {

...

}

...

#### 简单 RPC

routeGuideServer 实现了我们所有的服务方法。首先让我们看看最简单的类型 GetFeature，它从客户端拿到一个 Point 对象，然后从返回包含从数据库拿到的feature信息的 Feature.

func (s \*routeGuideServer) GetFeature(ctx context.Context, point \*pb.Point) (\*pb.Feature, error) {

for \_, feature := range s.savedFeatures {

if proto.Equal(feature.Location, point) {

return feature, nil

}

}

*// No feature was found, return an unnamed feature*

return &pb.Feature{"", point}, nil

}

该方法传入了 RPC 的上下文对象，以及客户端的 Point protocol buffer请求。它返回了一个包含响应信息和error 的 Feature protocol buffer对象。在方法中我们用适当的信息填充 Feature，然后将其和一个nil错误一起返回，告诉 gRPC 我们完成了对 RPC 的处理，并且 Feature 可以返回给客户端。

#### 服务器端流式 RPC

现在让我们来看看我们的一种流式 RPC。 ListFeatures 是一个服务器端的流式 RPC，所以我们需要将多个 Feature 发回给客户端。

func (s \*routeGuideServer) ListFeatures(rect \*pb.Rectangle, stream pb.RouteGuide\_ListFeaturesServer) error {

for \_, feature := range s.savedFeatures {

if inRange(feature.Location, rect) {

if err := stream.Send(feature); err != nil {

return err

}

}

}

return nil

}

如你所见，这里的请求对象是一个 Rectangle，客户端期望从中找到 Feature，这次我们得到了一个请求对象和一个特殊的RouteGuide\_ListFeaturesServer来写入我们的响应，而不是得到方法参数中的简单请求和响应对象。

在这个方法中，我们填充了尽可能多的 Feature 对象去返回，用它们的 Send() 方法把它们写入 RouteGuide\_ListFeaturesServer。最后，在我们的简单 RPC中，我们返回了一个 nil 错误告诉 gRPC 响应的写入已经完成。如果在调用过程中发生任何错误，我们会返回一个非 nil 的错误；gRPC 层会将其转化为合适的 RPC 状态通过线路发送。

#### 客户端流式 RPC

现在让我们看看稍微复杂点的东西：客户端流方法 RecordRoute，我们通过它可以从客户端拿到一个 Point 的流，其中包括它们路径的信息。如你所见，这次这个方法没有请求参数。相反的，它拿到了一个 RouteGuide\_RecordRouteServer 流，服务器可以用它来同时读 *和* 写消息——它可以用自己的 Recv() 方法接收客户端消息并且用 SendAndClose() 方法返回它的单个响应。

func (s \*routeGuideServer) RecordRoute(stream pb.RouteGuide\_RecordRouteServer) error {

var pointCount, featureCount, distance int32

var lastPoint \*pb.Point

startTime := time.Now()

for {

point, err := stream.Recv()

if err == io.EOF {

endTime := time.Now()

return stream.SendAndClose(&pb.RouteSummary{

PointCount: pointCount,

FeatureCount: featureCount,

Distance: distance,

ElapsedTime: int32(endTime.Sub(startTime).Seconds()),

})

}

if err != nil {

return err

}

pointCount++

for \_, feature := range s.savedFeatures {

if proto.Equal(feature.Location, point) {

featureCount++

}

}

if lastPoint != nil {

distance += calcDistance(lastPoint, point)

}

lastPoint = point

}

}

在方法体中，我们使用 RouteGuide\_RecordRouteServer 的 Recv() 方法去反复读取客户端的请求到一个请求对象（在这个场景下是 Point），直到没有更多的消息：服务器需要在每次调用后检查 Read() 返回的错误。如果返回值为 nil，流依然完好，可以继续读取；如果返回值为 io.EOF，消息流结束，服务器可以返回它的 RouteSummary。如果它还有其它值，我们原样返回错误，gRPC 层会把它转换为 RPC 状态。

#### 双向流式 RPC

最后，让我们看看双向流式 RPC RouteChat()。

func (s \*routeGuideServer) RouteChat(stream pb.RouteGuide\_RouteChatServer) error {

for {

in, err := stream.Recv()

if err == io.EOF {

return nil

}

if err != nil {

return err

}

key := serialize(in.Location)

... *// look for notes to be sent to client*

for \_, note := range s.routeNotes[key] {

if err := stream.Send(note); err != nil {

return err

}

}

}

}

这次我们得到了一个 RouteGuide\_RouteChatServer 流，和我们的客户端流的例子一样，它可以用来读写消息。但是，这次当客户端还在往 *它们* 的消息流中写入消息时，我们通过方法的流返回值。

这里读写的语法和客户端流方法相似，除了服务器会使用流的 Send() 方法而不是 SendAndClose()，因为它需要写多个响应。虽然客户端和服务器端总是会拿到对方写入时顺序的消息，它们可以以任意顺序读写——流的操作是完全独立的。

### 启动服务器

一旦我们实现了所有的方法，我们还需要启动一个gRPC服务器，这样客户端才可以使用服务。下面这段代码展示了在我们RouteGuide服务中实现的过程：

flag.Parse()

lis, err := net.Listen("tcp", fmt.Sprintf(":%d", \*port))

if err != nil {

log.Fatalf("failed to listen: %v", err)

}

grpcServer := grpc.NewServer()

pb.RegisterRouteGuideServer(grpcServer, &routeGuideServer{})

... *// determine whether to use TLS*

grpcServer.Serve(lis)

为了构建和启动服务器，我们需要：

1. 使用 lis, err := net.Listen("tcp", fmt.Sprintf(":%d", \*port)) 指定我们期望客户端请求的监听端口。
2. 使用grpc.NewServer()创建 gRPC 服务器的一个实例。
3. 在 gRPC 服务器注册我们的服务实现。
4. 用服务器 Serve() 方法以及我们的端口信息区实现阻塞等待，直到进程被杀死或者 Stop() 被调用。

## 创建客户端

在这部分，我们将尝试为 RouteGuide 服务创建一个 Go 的客户端。你可以从[grpc-go/examples/route\_guide/client/client.go](https://github.com/grpc/grpc-go/tree/master/examples/route_guide/client/client.go)看到我们完整的客户端例子代码.

### 创建存根

为了调用服务方法，我们首先创建一个 gRPC *channel* 和服务器交互。我们通过给 grpc.Dial() 传入服务器地址和端口号做到这点，如下：

conn, err := grpc.Dial(\*serverAddr)

if err != nil {

...

}

defer conn.Close()

你可以使用 DialOptions 在 grpc.Dial 中设置授权认证（如， TLS，GCE认证，JWT认证），如果服务有这样的要求的话 —— 但是对于 RouteGuide 服务，我们不用这么做。

一旦 gRPC *channel* 建立起来，我们需要一个客户端 *存根* 去执行 RPC。我们通过 .proto 生成的 pb 包提供的 NewRouteGuideClient 方法来完成。

client := pb.NewRouteGuideClient(conn)

### 调用服务方法

现在让我们看看如何调用服务方法。注意，在 gRPC-Go 中，RPC以阻塞/同步模式操作，这意味着 RPC 调用等待服务器响应，同时要么返回响应，要么返回错误。

#### 简单 RPC

调用简单 RPC GetFeature 几乎是和调用一个本地方法一样直观。

feature, err := client.GetFeature(context.Background(), &pb.Point{409146138, -746188906})

if err != nil {

...

}

如你所见，我们调用了前面创建的存根上的方法。在我们的方法参数中，我们创建并且填充了一个请求的 protocol buffer 对象（例子中为 Point）。我们同时传入了一个 context.Context ，在有需要时可以让我们改变 RPC 的行为，比如超时/取消一个正在运行的 RPC。 如果调用没有返回错误，那么我们就可以从服务器返回的第一个返回值中读到响应信息。

log.Println(feature)

#### 服务器端流式 RPC

ListFeatures 就是我们说的服务器端流方法，它会返回地理的Feature 流。 如果你已经读过[创建服务器](http://doc.oschina.net/grpc?t=60133#server)，本节的一些内容也许看上去会很熟悉——流式 RPC 是在客户端和服务器两端以一种类似的方式实现的。

rect := &pb.Rectangle{ ... } *// initialize a pb.Rectangle*

stream, err := client.ListFeatures(context.Background(), rect)

if err != nil {

...

}

for {

feature, err := stream.Recv()

if err == io.EOF {

break

}

if err != nil {

log.Fatalf("%v.ListFeatures(\_) = \_, %v", client, err)

}

log.Println(feature)

}

在简单 RPC 的例子中，我们给方法传入一个上下文和请求。然而，我们得到返回的是一个 RouteGuide\_ListFeaturesClient 实例，而不是一个应答对象。客户端可以使用 RouteGuide\_ListFeaturesClient 流去读取服务器的响应。

我们使用 RouteGuide\_ListFeaturesClient 的 Recv() 方法去反复读取服务器的响应到一个响应 protocol buffer 对象（在这个场景下是Feature）直到消息读取完毕：每次调用完成时，客户端都要检查从 Recv() 返回的错误 err。如果返回为 nil，流依然完好并且可以继续读取；如果返回为 io.EOF，则说明消息流已经结束；否则就一定是一个通过 err 传过来的 RPC 错误。

#### 客户端流式 RPC

除了我们需要给方法传入一个上下文而后返回 RouteGuide\_RecordRouteClient 流以外，客户端流方法 RecordRoute 和服务器端方法类似，它可以用来读 *和* 写消息。

*// Create a random number of random points*

r := rand.New(rand.NewSource(time.Now().UnixNano()))

pointCount := int(r.Int31n(100)) + 2 *// Traverse at least two points*

var points []\*pb.Point

for i := 0; i < pointCount; i++ {

points = append(points, randomPoint(r))

}

log.Printf("Traversing %d points.", len(points))

stream, err := client.RecordRoute(context.Background())

if err != nil {

log.Fatalf("%v.RecordRoute(\_) = \_, %v", client, err)

}

for \_, point := range points {

if err := stream.Send(point); err != nil {

log.Fatalf("%v.Send(%v) = %v", stream, point, err)

}

}

reply, err := stream.CloseAndRecv()

if err != nil {

log.Fatalf("%v.CloseAndRecv() got error %v, want %v", stream, err, nil)

}

log.Printf("Route summary: %v", reply)

RouteGuide\_RecordRouteClient 有一个 Send() 方法，我们可以用它来给服务器发送请求。一旦我们完成使用 Send() 方法将客户端请求写入流，就需要调用流的 CloseAndRecv()方法，让 gRPC 知道我们已经完成了写入同时期待返回应答。我们从 CloseAndRecv() 返回的 err 中获得 RPC 的状态。如果状态为nil，那么CloseAndRecv()的第一个返回值将会是合法的服务器应答。

#### 双向流式 RPC

最后，让我们看看双向流式 RPC RouteChat()。 和 RecordRoute 的场景类似，我们只给函数传 入一个上下文对象，拿到可以用来读写的流。但是，当服务器依然在往 *他们* 的消息流写入消息时，我们 通过方法流返回值。

stream, err := client.RouteChat(context.Background())

waitc := make(chan struct{})

go func() {

for {

in, err := stream.Recv()

if err == io.EOF {

*// read done.*

close(waitc)

return

}

if err != nil {

log.Fatalf("Failed to receive a note : %v", err)

}

log.Printf("Got message %s at point(%d, %d)", in.Message, in.Location.Latitude, in.Location.Longitude)

}

}()

for \_, note := range notes {

if err := stream.Send(note); err != nil {

log.Fatalf("Failed to send a note: %v", err)

}

}

stream.CloseSend()

<-waitc

这里读写的语法和我们的客户端流方法很像，除了在完成调用时，我们会使用流的 CloseSend() 方法。 虽然每一端获取对方信息的顺序和信息被写入的顺序一致，客户端和服务器都可以以任意顺序读写——流的操作是完全独立的。

## 来试试吧！

假设你在 $GOPATH/src/google.golang.org/grpc/examples/route\_guide 目录，要编译和运行服务器，只需要运行：

$ go run server/server.go

同样的，运行客户端:

$ go run client/client.go