gRPC 官方文档中文版 V1.0

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，其中：

- Guides 部分由 @sofeminer(http://my.oschina.net/altersoft) 翻译
- Tutorials 部分由 @马博文(http://my.oschina.net/ivysummer) 翻译
- 全部由 @jason0916(http://my.oschina.net/u/2350782) 校对

反馈

对《gRPC 官方文档中文版》有任何反馈，欢迎在以下网址评论：

http://www.oschina.net/news/70391/grpc-docs-cn(http://www.oschina.net/news/70391/grpc-docs-cn)

版权

指南

概览

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。

快速开始

为了直观地着手运行 gRPC，可以从你所选择的语言对应的快速开始入手，里面包含创建这个列子

的安装指导、快速上手指南等更多内容。

- C++(https://github.com/grpc/grpc/tree/master/examples/cpp)
- Java(https://github.com/grpc/grpc-java/tree/master/examples)
- Go(https://github.com/grpc/grpc-go/tree/master/examples)
- Python(https://github.com/grpc/grpc/tree/master/examples/python)
- Ruby(https://github.com/grpc/grpc/tree/master/examples/ruby)
- Node.js(https://github.com/grpc/grpc/tree/master/examples/node)
- Android Java(https://github.com/grpc/grpc-java/tree/master/examples/android)
- C#(https://github.com/grpc/grpc/tree/master/examples/csharp/helloworld)
- Objective-C(https://github.com/grpc/grpc/tree/master/examples/objective-c/helloworld)
- PHP(https://github.com/grpc/grpc/tree/master/examples/php)

你可以从这里(https://github.com/grpc/grpc)找到 gRPC 的源码库。我们大多数例子都在源码库 examples 目录下。

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。

图片地址：http://www.grpc.io/img/grpc_concept_diagram_00.png

gRPC 客户端和服务端可以在多种环境中运行和交互 - 从 google 内部的服务器到你自己的笔记本，并且可以用任何 gRPC 支持的语言(#quickstart)来编写。所以，你可以很容易地用 Java 创建一个 gRPC 服务端，用 Go、Python、Ruby 来创建客户端。此外，Google 最新 API 将有 gRPC 版本的接口，使你很容易地将 Google 的功能集成到你的应用里。

使用 protocol buffers

gRPC 默认使用 *protocol buffers*，这是 Google 开源的一套成熟的结构数据序列化机制（当然也可以使用其他数据格式如 JSON）。正如你将在下方例子里所看到的，你用 *proto files* 创建 gRPC 服务，用 protocol buffers 消息类型来定义方法参数和返回类型。你可以在 Protocol Buffers 文档(https：//developers.google.com/protocol-buffers/docs/overview)找到更多关于 Protocol Buffers 的资料。

Protocol buffers 版本

尽管 protocol buffers 对于开源用户来说已经存在了一段时间，例子内使用的却一种名叫 proto3 的新风格的 protocol buffers，它拥有轻量简化的语法、一些有用的新功能，并且支持更多新语言。当前针对 Java 和 C++ 发布了 beta 版本，针对 JavaNano（即 Android Java）发布 alpha 版本，在protocol buffers Github 源码库里(https：//github.com/google/protobuf/releases)有 Ruby 支持， 在golang/protobuf Github 源码库(https：//github.com/golang/protobuf)里还有针对 Go 语言的生成器， 对更多语言的支持正在开发中。 你可以在 proto3 语言指南(https：//developers.google.com/protocol-buffers/docs/proto3)里找到更多内容， 在与当前

默认版本的发布说明(https：//github.com/google/protobuf/releases)比较，看到两者的主要不同点。更多关于 proto3 的文档很快就会出现。虽然你*可以*使用 proto2 (当前默认的 protocol buffers 版本)， 我们通常建议你在 gRPC 里使用 proto3，因为这样你可以使用 gRPC 支持全部范围的的语言，并且能避免 proto2 客户端与 proto3 服务端交互时出现的兼容性问题，反之亦然。

你好 gRPC!

现在你已经对 gRPC 有所了解，了解其工作机制最简单的方法是看一个简单的例子。 Hello World 将带领你创建一个简单的客户端——服务端应用，向你展示：

• 通过一个 protocol buffers 模式，定义一个简单的带有 Hello World 方法的 RPC 服务。

• 用你最喜欢的语言(如果可用的话)来创建一个实现了这个接口的服务端。

• 用你最喜欢的(或者其他你愿意的)语言来访问你的服务端。

这个例子完整的代码在我们 GitHub 源码库的 examples 目录下。

我们使用 Git 版本系统来进行源码管理，但是除了如何安装和运行一些 Git 命令外，你没必要知道其他关于 Git 的任何事情。

需要注意的是，并不是所有 gRPC 支持的语言都可以编写我们例子的服务端代码，比如 PHP 和 Objective-C 仅支持创建客户端。

比起针对于特定语言的复杂教程，这更像是一个介绍性的例子。你可以在本站找到更有深度的教程，gRPC 支持的语言的参考文档很快就会全部开放。

准备

本节解释了如何在你本机上准备好例子代码的运行环境。如果你只是想读一下例子，你可以直接到下一步。

安装 Git

你可以从http://git-scm.com/download(http://git-scm.com/download)下载和安装 Git。安装好 Git 后，你应该能访问 git 命令行工具。你需要的主要命令如下：

• git clone ... ： 从远程代码库克隆一份到本机。

• git checkout ... ： 检出一个特殊分支或一个标签版本的代码来改进。

安装 gRPC

针对你选择的语言构建和安装 gRPC 插件和相关工具，可以参照快速开始(#quickstart)。 Java gRPC 除了 JDK 外不需要其他工具。

获得源码

• Java

Java 例子代码在 GitHub 源码库里。你可以运行如下命令克隆源码到本地：

git clone https：//github.com/grpc/grpc-java.git

切换当前目录到grpc-java/examples

cd grpc-java/examples

• C++

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

`

$ git clone https：//github.com/grpc/grpc.git

`

切换当前目录到 examples/cpp/helloworld

`

$ cd examples/cpp/helloworld/

`

• Python

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

`

$ git clone https://github.com/grpc/grpc.git

`

切换当前目录到 examples/python/helloworld

`

$ cd examples/python/helloworld/

`

• Go

获取例子：

```
$ go get -u github.com/grpc/grpc-go/examples/helloworld/greeter_client
$ go get -u github.com/grpc/grpc-go/examples/helloworld/greeter_server
```

切换当前目录到 examples/helloworld

• Ruby

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

`

$ git clone https：//github.com/grpc/grpc.git

`

切换当前目录到 examples/ruby ，然后使用 bundler 安装例子的包依赖：

```
$ gem install bundler # if you don't already have bundler available
$ bundle install
```

• Node.js

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

`

$ git clone https：//github.com/grpc/grpc.git
`

切换当前目录到 examples/node ， 接着安装包依赖：

```
$ cd examples/node
$ npm install
```

• C#

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：
`

$ git clone https：//github.com/grpc/grpc。git
`

从 Visual Studio (或 Linux 上的 Monodevelop) 打开 Greeter.sln。可以从 C# Quickstart(/docs/installation/csharp.html) 找到平台特定的设置步骤。

• Objective-C

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

```
$ git clone https：//github.com/grpc/grpc.git
$ cd grpc
$ git submodule update --init
```

切换当前目录到 examples/objective-c/helloworld 。

• PHP

例子代码在 GitHub 源码库的 examples 目录。你可以运行如下命令克隆源码到本地：

$ git clone https：//github.com/grpc/grpc.git

切换当前目录到 examples/php 。

虽然我们大多数例子使用同一个 .proto 文件，但 PHP 的例子有自己的 helloworld.proto 文件，这是因为它依赖 proto2 语法。 PHP暂时没有针对 proto3 的支持。

定义服务

创建我们例子的第一步是定义一个*服务*：一个 RPC 服务通过参数和返回类型来指定可以远程调用的方法。就像你在 概览(#protocol) 里所看到的， gRPC 通过 protocol buffers(https：//developers.google.com/protocol-buffers/docs/overview) 来实现。

我们使用 protocol buffers 接口定义语言来定义服务方法，用 protocol buffer 来定义参数和返回类型。客户端和服务端均使用服务定义生成的接口代码。

这里有我们服务定义的例子，在 helloworld.proto(https：//github.com/grpc/grpc-java/tree/master/examples/src/main/proto) 里用 protocol buffers IDL 定义的。Greeter 服务有一个方法 SayHello ，可以让服务端从远程客户端接收一个包含用户名的 HelloRequest 消息后，在一个 HelloReply 里发送回一个 Greeter 。这是你可以在 gRPC 里指定的最简单的 RPC - 你可以在教程里找到针对你选择的语言更多类型的例子。

```
syntax = "proto3";

option java_package = "io.grpc.examples";

package helloworld;

// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

生成 gRPC 代码

一旦定义好服务，我们可以使用 protocol buffer 编译器 protoc 来生成创建应用所需的特定客户端和服务端的代码 - 你可以生成任意 gRPC 支持的语言的代码，当然 PHP 和 Objective-C 仅支持创建客户端代码。生成的代码同时包括客户端的存根和服务端要实现的抽象接口，均包含 Greeter 所定义的方法。

(假如你没有在系统里安装 gRPC 插件和 protoc ，并且仅仅是要看一下这个例子，你可以跳过这一步，直接到下一步来查看生成的代码。)

• Java

这个例子的构建系统也是 Java gRPC 本身构建的一部分 —— 为了简单起见，我们推荐使用我们事先生成的例子代码。你可以参考 README(https：//github.com/grpc/grpc——java/blob/master/README.md) 来看一下如何从你自己的 .proto 文件生成代码。

这个例子事先生成的代码在 src/generated/main(https：//github.com/grpc/grpc——java/tree/master/examples/src/generated/main)下。

以下类包含所有我们需要创建这个例子所有的代码：

• HelloRequest.java， HelloResponse.java和其他文件包含所有 protocol buffer 用来填充、序列化和提取 HelloRequest 和 HelloReply 消息类型的代码。

• GreeterGrpc.java， 包含 (还有其他有用的代码)：

```
`Greeter` 服务端需要实现的接口
```

```
public static interface Greeter {
    public void sayHello(Helloworld.HelloRequest request,
        StreamObserver<Helloworld.HelloReply> responseObserver);
}
```

客户端用来与 Greeter 服务端进行对话的 存根 类。就像你所看到的，异步存根也实现了 Greeter 接口。

```
public static class GreeterStub extends AbstractStub<GreeterStub>
    implements Greeter {
  ...
}
```

• C++

生成客户端和服务端接口，运行：

```
$ make helloworld.grpc.pb.cc helloworld.pb.cc
```

这从内部调用 protocol buffer 编译器：

```
$ protoc -I ../../protos/ --grpc_out=. --plugin=protoc-gen-grpc=grpc_cpp_plugin
../../protos/helloworld.proto
$ protoc -I ../../protos/ --cpp_out=. ../../protos/helloworld.proto
```

生成：

helloworld.pb.h 声明了用于填充、序列化、提取 HelloRequest 和 HelloResponse 消息类型的类，并且还有它的实现 helloworld.pb.cc 。

helloworld.grpc.pb.h，声明了我们生成的服务类和它的实现 helloworld.grpc.pb.cc 。

• Python

可以用如下命令生成客户端和服务端：

```
$ ./run_codegen.sh
```

这内部调用 protocol buffer 编译器：

```
$ protoc -I ../../protos --python_out=. --grpc_out=. --plugin=protoc-gen-grpc=`which
grpc_python_plugin` ../../protos/helloworld.proto
```

这生成了 helloworld_pb2.py ，包含我们生成的客户端和服务端类，此外还有用于填充、序列化、提取 `HelloRequest` 和 `HelloResponse` 消息类型的类。

• Go

为了生成客户端和服务端接口，运行 protocol buffer 编译器：

```
protoc -I ../protos ../protos/helloworld.proto --go_out=plugins=grpc：helloworld
```

这生成了 helloworld.pb.go ，包含了我们生成的客户端和服务端类，此外还有用于填充、序列化、提取 HelloRequest 和 HelloResponse 消息类型的类。

• Ruby

为了生成客户端和服务端接口，运行 protocol buffer 编译器：

```
protoc -I ../protos --ruby_out=lib --grpc_out=lib --plugin=protoc-gen-grpc=`which
grpc_ruby_plugin` ../protos/helloworld.proto
```

在 lib 目录下生成了如下文件：

• lib/helloworld.rb 定义了一个 Helloworld 模块， 这个模块提供了用于填充、序列化、提取请求、应答消息类型的所有 protocol buffer 代码。

• lib/helloworld_services.rb 用生成的客户端和服务端代码继承了 `Helloworld` 模块。

• Node.js

Node.js库从运行时加载的 .proto 文件动态生成服务描述和客户端存根的定义，所以使用此语言时没必要生成任何特殊代码。而是在例子客户端和服务端里，我们 require gRPC 库，然后用它的 load() 方法：

```
var grpc = require('grpc');
var hello_proto = grpc.load(PROTO_PATH).helloworld;
```

• C#

• 为了生成 Windows 上的代码，我们使用来自 Google.Protobuf NuGet 包里的 protoc.exe 和来自 Grpc.Tools NuGet 包里的 Grpc.Tools ，这两个文件都在 tools 目录下。

一般你需要自己把 Grpc.Tools 包添加到解决方案，但在这个教程里，这一步已经为你做好了。你应该在 examples/csharp/helloworld 下执行以下命令：

```
  > packages\Google.Protobuf.3.0.0-alpha4\tools\protoc.exe -I../../protos --csharp_out
Greeter --grpc_out Greeter --plugin=protoc-gen-
grpc=packages\Grpc.Tools.0.7.0\tools\grpc_csharp_plugin.exe
../../protos/helloworld.proto
```

• 在 Linux 或 OS X ，我们依赖通过 Linuxbrew 或者 Homebrew 安装的 protoc 和 grpc*csharp* plugin 。请在 route_guide 目录下运行这个命令：

```
  $ protoc -I../../protos --csharp_out Greeter --grpc_out Greeter --plugin=protoc-gen-
grpc=`which grpc_csharp_plugin` ../../protos/helloworld.proto
```

根据你的 OS 运行合适的命令，在 Greeter 目录重新生成如下文件:

• Greeter/Helloworld.cs 定义了命名空间 Helloworld

它包含了所有用来填充、序列化、提取请求和应答消息类型的 protocol buffer 代码。

• Greeter/HelloworldGrpc.cs，提供了存根类和服务类，包括：

• 一个 Greeter.IGreeter 接口，可以在定义 RootGuide 服务实现的时候来继承它。

• 一个 Greeter.GreeterClient 类，可用来访问远程的 RouteGuide 实例。

• Objective-c

为了简单，我们提供了一个 Podspec 文件 (https：//github.com/grpc/grpc/blob/master/examples/objective-c/helloworld/HelloWorld.podspec)，用来使用适当的插件、输入、输出运行 protoc ，并描述如何编译生成的代码。你仅仅需要在 examples/objective-c/route_guide 下运行：

```
$ pod install
```

然后你可以打开由 Cocoapods 创建的 XCode 工作空间，看一下生成的代码。运行命令生成：

• Helloworld.pbobjc.h， 定义生成的消息类的头文件。

• Helloworld.pbobjc.m，包含消息类的实现。

• Helloworld.pbrpc.h，定义生成的服务类的头文件。

• Helloworld.pbrpc.m，包含服务类的实现。

• PHP

gRPC PHP 使用 protoc-gen-php(https：//github.com/datto/protobuf-php) 工具来从 .proto 文件生成代码。你可以在 PHP 快速开始 (https：//github.com/grpc/grpc/blob/master/src/php)里找到如何安装它。为了生成 Greeter 服务的代码，运行：

```
protoc-gen-php -i . -o . ./helloworld.proto
```

生成 helloworld.php ，包含：

• 所有用来填充、序列化、提取请求和应答消息类型的 protocol buffer 代码.

• GreeterClient 类，可以让客户端调用在 Greeter 服务里的方法。

写一个服务器

现在让我们写点代码！首先我们将创建一个服务应用来实现服务(你会记起来，我们可以是使用除了 Objective-C and PHP 外的其他所有语言来实现)。在本节，我们不打算对如何创建一个服务端进行更深入地探讨 —— 更详细的信息可以在你选择语言对应的教程里找到。

服务实现

• Java

GreeterImpl.java(https：//github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/helloworld/HelloWorldServer.java#L51) 准确地实现了 Greeter 服务所需要的行为。

正如你所见，GreeterImpl 类通过实现 sayHello 方法，实现了从 IDL(https：//github.com/grpc/grpc-java/tree/master/examples/src/main/proto) 生成的 GreeterGrpc.Greeter 接口 。

```
@Override
public void sayHello(HelloRequest req, StreamObserver<HelloReply>
responseObserver) {
  HelloReply reply = HelloReply.newBuilder().setMessage("Hello " +
req.getName()).build();
  responseObserver.onNext(reply);
  responseObserver.onCompleted();
}
```

sayHello 有两个参数：

• HelloRequest，请求。

• StreamObserver<HelloReply>： 应答观察者，一个特殊的接口，服务器用应答来调用它。

为了返回给客户端应答并且完成调用：

1. 用我们的激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象。

2. 将 HelloReply 返回给客户端，然后表明我们已经完成了对 RPC 的处理。

• C++

greeter_server.cc(https：//github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_server.cc) 实现了 Greeter 服务所需要的行为。

正如你所见，GreeterServiceImpl 类通过实现 sayHello 方法，实现了从 proto 服务定义生成的 Greeter：：Service 接口 。

```
class GreeterServiceImpl final ： public Greeter：：Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    std：：string prefix("Hello ");
    reply->set_message(prefix + request->name());
    return Status：：OK;
  }
};
```

在此我们实现同步版本的 Greeter，它提供了默认的 gRPC 服务行为（这里也有一个异步的接口，Greeter：：AsyncService）。

sayHello 有三个参数：

• ServerContext： RPC上下文对象。

• HelloRequest ： 请求。

• HelloReply ：应答。

为了返回给客户端应答并且完成调用：

1. 用我们的激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象。

2. 将 Status::OK 返回给客户端，表明我们已经完成了对 RPC 的处理。

• Python

greeter_server.py(https：//github.com/grpc/grpc/blob/master/examples/python/helloworl

d/greeter_server.py) 实现了 Greeter 服务所需要的行为。

正如你所见，Greeter 类通过实现 sayHello 方法，实现了从 proto 服务定义生成的 helloworld_pb2.BetaGreeterServicer 接口：

```
class Greeter(helloworld_pb2.BetaGreeterServicer)：

 def SayHello(self, request, context)：
  return helloworld_pb2.HelloReply(message='Hello, %s!' % request.name)
```

为了返回给客户端应答并且完成调用：

1. 用我们的激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象。
2. 将 HelloReply 返回给客户端。

• Go

greeter_server/main.go(https：//github.com/grpc/grpc-go/blob/master/examples/helloworld/greeter_server/main.go) 实现了 Greeter 服务所需要的行为。

正如你所见，服务器有一个 server 结构。它通过实现 sayHello 方法，实现了从 proto 服务定义生成的GreeterServer 接口：

```
// server is used to implement helloworld.GreeterServer.
type server struct{}
// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply,
error) {
return &pb.HelloReply{Message： "Hello " + in.Name}, nil
}
```

为了返回给客户端应答并且完成调用：

1. 用我们的激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象。

2.将 HelloReply 返回给客户端。

• Ruby

greeter_server.rb(https：//github.com/grpc/grpc/blob/master/examples/ruby/greeter_server.rb) 实现了 Greeter 服务所需要的行为。

服务器有一个 GreeterServer 类，它通过实现 sayHello 方法，实现了从 proto 服务定义生成的 GreeterServer 接口：

```
class GreeterServer < Helloworld：：Greeter：：Service
 # say_hello implements the SayHello rpc method.
 def say_hello(hello_req, _unused_call)
  Helloworld：：HelloReply.new(message： "Hello #{hello_req.name}")
 end
```

为了返回给客户端应答并且完成调用：我们用激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象，然后返回它。

• Node.js

greeter_server.js(https：//github.com/grpc/grpc/blob/master/examples/node/greeter_server.js) 实现了 Greeter 服务所需要的行为。

服务器通过实现 SayHello 方法，实现了服务定义：

```
function sayHello(call, callback) {
  callback(null, {message： 'Hello ' + call.request.name});
}
```

为了返回给客户端应答并完成调用，我们填充了应答并将其传递给一个已提供了的回调，用 null 作为第一个参数来表示没有出现错误。

• C#

GreeterServer/Program.cs(https：//github.com/grpc/grpc/blob/master/examples/csharp/helloworld/GreeterServer/Program.cs) 实现了 Greeter 服务所需要的行为。

服务器的 GreeterImpl类，通过实现 sayHello 方法，实现了生成的IGreeter 接口：

```
class GreeterImpl ： Greeter.IGreeter
{
    public Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = "Hello " + request.Name });
    }
}
```

为了返回给客户端应答并完成以下调用：

1. 用我们的激动人心的消息构建并填充一个在我们接口定义的 HelloReply 应答对象。

2. 将 HelloReply 返回给客户端。

服务端实现

需要提供一个 gRPC 服务的另一个主要功能是让这个服务实在在网络上可用。

• Java

HelloWorldServer.java(https：//github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/helloworld/HelloWorldServer.java) 提供了以下代码作为 Java 的例子。

```
/* The port on which the server should run */
private int port = 50051;
private Server server;
private void start() throws Exception {
  server = ServerBuilder.forPort(port)
      .addService(GreeterGrpc.bindService(new GreeterImpl()))
      .build()
```

```
      .start();
  logger.info("Server started, listening on " + port);
  Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
      // Use stderr here since the logger may has been reset by its JVM shutdown hook.
      System.err.println("*** shutting down gRPC server since JVM is shutting down");
      HelloWorldServer.this.stop();
      System.err.println("*** server shut down");
    }
  });
}
```

• C++

greeter_server.cc(https：//github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_server.cc) 提供了以下代码作为 C++ 的例子。

```
void RunServer() {
  std：：string server_address("0.0.0.0：50051");
  GreeterServiceImpl service;
  ServerBuilder builder;
  builder.AddListeningPort(server_address, grpc：：InsecureServerCredentials());
  builder.RegisterService(&service);
  std：：unique_ptr<Server> server(builder.BuildAndStart());
  std：：cout << "Server listening on " << server_address << std：：endl;
  server->Wait();
}
```

• Python

greeter_server.py(https：//github.com/grpc/grpc/blob/master/examples/python/helloworld/greeter_server.py) 提供了以下代码作为 Python 的例子。

```
server = helloworld_pb2.beta_create_Greeter_server(Greeter())
server.add_insecure_port('[：：]：50051')
server.start()
try：
  while True：
    time.sleep(_ONE_DAY_IN_SECONDS)
except KeyboardInterrupt：
  server.stop()
```

• Go

greeter_server/main.go(https：//github.com/grpc/grpc-go/blob/master/examples/helloworld/greeter_server/main.go) 提供了以下代码作为 Go 的例

子。

```
const (
port = "：50051"
)
...
func main() {
lis, err ：= net.Listen("tcp", port)
if err != nil {
log.Fatalf("failed to listen： %v", err)
}
s ：= grpc.NewServer()
pb.RegisterGreeterServer(s, &server{})
s.Serve(lis)
}
```

• Ruby

greeter_server.rb(https：//github.com/grpc/grpc/blob/master/examples/ruby/greeter_server.rb) 提供了以下代码作为 Ruby 的例子。

```
def main
  s = GRPC：：RpcServer.new
  s.add_http2_port('0.0.0.0：50051')
  s.handle(GreeterServer)
  s.run
end
```

• Node.js

greeter_server.js(https：//github.com/grpc/grpc/blob/maser/examples/node/greeter_server.js) 提供了以下代码作为 Ruby 的例子。

```
function main() {
  var server = new Server({
    "helloworld.Greeter"： {
      sayHello： sayHello
    }
  });
  server.bind('0.0.0.0：50051');
  server.listen();
}
```

• C#

GreeterServer/Program.cs(https：//github.com/grpc/grpc/blob/master/examples/csharp/helloworld/GreeterServer/Program.cs) 提供了以下代码作为 C# 的例子。

```
Server server = new Server
{
    Services = { Greeter.BindService(new GreeterImpl()) },
    Ports = { new ServerPort("localhost", 50051, ServerCredentials.Insecure) }
};
server.Start();
```

在这里我们创建了合理的 gRPC 服务器，将我们实现的 Greeter 服务绑定到一个端口。然后我们启动服务器：服务器现在已准备好从 Greeter 服务客户端接收请求。我们将在具体语言对应的文档里更深入地了解这所有的工作是怎样进行的。

写一个客户端

客户端的 gRPC 非常简单。在这一步，我们将用生成的代码写一个简单的客户程序来访问我们在上一节里创建的 Greeter 服务器。

同样，我们也不打算对如何实现一个客户端程序深入更多，我们把这些内容放到教程里。

连接服务

首先我们看一下我们如何连接 Greeter 服务器。我们需要创建一个 gRPC 频道，指定我们要连接的主机名和服务器端口。然后我们用这个频道创建存根实例。

• Java

```
private final ManagedChannel channel;
private final GreeterGrpc.GreeterBlockingStub blockingStub;
public HelloWorldClient(String host, int port) {
  channel = ManagedChannelBuilder.forAddress(host, port)
      .usePlaintext(true)
      .build();
  blockingStub = GreeterGrpc.newBlockingStub(channel);
}
```

在这个例子里，我们创建了一个阻塞的存根。这意味着 RPC 调用要等待服务器应答，将会返回一个应答或抛出一个异常。 gRPC Java 还可以有其他种类的存根，可以向服务器发出非阻塞的调用，这种情况下应答是异步返回的。

• C++

```
int main(int argc, char** argv) {
  GreeterClient greeter(
      grpc：：CreateChannel("localhost：50051", grpc：：InsecureCredentials(),
                            ChannelArguments()));
...
}
...
```

```
class GreeterClient {
 public：
  GreeterClient(std：：shared_ptr<ChannelInterface> channel)
      ： stub_(Greeter：：NewStub(channel)) {}
...
 private：
  std：：unique_ptr<Greeter：：Stub> stub_;
};
```

• Python

生成的 Python 代码有一个根据频道创建存根的帮助方法。

```
channel = implementations.insecure_channel('localhost', 50051)
stub = helloworld_pb2.beta_create_Greeter_stub(channel)
...
```

• Go

```
const (
address     = "localhost：50051"
defaultName = "world"
)
func main() {
// Set up a connection to the server.
conn, err ：= grpc.Dial(address)
if err != nil {
log.Fatalf("did not connect： %v", err)
}
defer conn.Close()
c ：= pb.NewGreeterClient(conn)
...
}
```

在 gRPC Go 你是使用一个特殊的 Dial() 方法来创建频道。

• Ruby

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

在 Ruby 里，我们可以在一个方法里调用从 .proto 文件里生成的存根类。

• Node.js

```
var client = new hello_proto.Greeter('localhost：50051');
```

在 Node.js ，我们可以在一步调用 Greeter 存根构造器。

• C#

```
Channel channel = new Channel("127.0.0.1：50051", Credentials.Insecure);
var client = Greeter.NewClient(channel);
...
```

• Objective-C

在 Objective-C 里，我们可以用生成的 HLWGreeter 类指定的初始化方法完成，这个方法需要用 NSString * 类型表示的服务器和端口作为参数。

```
#import <GRPCClient/GRPCCall+Tests.h>
...
static NSString * const kHostAddress = @"localhost：50051";
...
[GRPCCall useInsecureConnectionsForHost：kHostAddress];
HLWGreeter *client = [[HLWGreeter alloc] initWithHost：kHostAddress];
```

当用给定的 host：port 对通讯的时候，注意对 useInsecureConnectionsForHost： 的调用，要通知 gRPC 库使用明文 (而不是 TLS 加密的连接)。

• PHP

```
$client = new helloworld\GreeterClient(
    new Grpc\BaseStub('localhost：50051', []));
```

在 PHP 里，我们可以使用 GreeterClient 类的构造器一步完成。

调用 RPC

现在我们可以联系服务并获得一个 greeting ：

1. 我们创建并填充一个 HelloRequest 发送给服务。

2. 我们用请求调用存根的 SayHello()，如果 RPC 成功，会得到一个填充的 HelloReply ，从其中我们可以获得 greeting。

• Java

```
HelloRequest req = HelloRequest.newBuilder().setName(name).build();
HelloReply reply = blockingStub.sayHello(req);
```

你可以在 HelloWorldClient.java(https：//github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/helloworld/HelloWorldClient.java) 里查看完整的客户端代码。

• C++

```
std：：string SayHello(const std：：string& user) {
  HelloRequest request;
  request.set_name(user);
  HelloReply reply;
```

```
  ClientContext context;
  Status status = stub_->SayHello(&context, request, &reply);
  if (status.ok()) {
    return reply.message();
  } else {
    return "Rpc failed";
  }
}
```

你可以在 greeter_client.cc(https：//github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_client.cc) 里查看完整的客户端代码。

• Python

```
response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'),
_TIMEOUT_SECONDS)
print "Greeter client received： " + response.message
```

你可以在 greeter_client.py(https：//github.com/grpc/grpc/blob/master/examples/python/helloworld/greeter_client.py) 里查看完整的客户端代码。

• Go

```
r, err ：= c.SayHello(context.Background(), &pb.HelloRequest{Name： name})
if err != nil {
log.Fatalf("could not greet： %v", err)
}
log.Printf("Greeting： %s", r.Message)
```

你可以在 greeter_client/main.go(https：//github.com/grpc/grpc-go/blob/master/examples/helloworld/greeter_client/main.go) 里查看完整的客户端代码。

• Ruby

```
message = stub.say_hello(Helloworld：：HelloRequest.new(name： user)).message
p "Greeting： #{message}"
```

你可以在 greeter_client.rb(https：//github.com/grpc/grpc/blob/master/examples/ruby/greeter_client.rb) 里查看完整的客户端代码。

• Node.js

```
client.sayHello({name： user}, function(err, response) {
  console.log('Greeting：', response.message);
});
```

你可以在

/examples/node/greeter_client.js(https：//github.com/grpc/grpc/blob/master/examples/node/greeter_client.js) 里查看完整的客户端代码。

• C#

```
var reply = client.SayHello(new HelloRequest { Name = user });
Console.WriteLine("Greeting： " + reply.Message);
```

你可以在 GreeterClient/Program.cs(https：//github.com/grpc/grpc/blob/master/examples/csharp/helloworld/GreeterClient/Program.cs) 里查看完整的客户端代码。

• Objective-C

```
HLWHelloRequest *request = [HLWHelloRequest message];
request.name = @"Objective-C";
[client sayHelloWithRequest：request handler：^(HLWHelloReply *response, NSError
*error) {
  NSLog(@"%@", response.message);
}];
```

你可以在 examples/objective-c/helloworld(https：//github.com/grpc/grpc/tree/master/examples/objective-c/helloworld) 里查看完整的客户端代码。

• PHP

```
  $request = new helloworld\HelloRequest();
  $request->setName($name);
  list($reply, $status) = $client->SayHello($request)->wait();
  $message = $reply->getMessage();
```

你可以在 greeter_client.php(https：//github.com/grpc/grpc/blob/master/examples/php/greeter_client.php) 里查看完整的客户端代码。

试一下!

你可以尝试用同一个语言在客户端和服务端构建并运行例子。或者你可以尝试 gRPC 最有用的一个功能 - 不同的语言间的互操作性，即在不同的语言运行客户端和服务端。每个服务端和客户端使用从同一过 proto 文件生成的接口代码，则意味着任何 Greeter 客户端可以与任何 Greeter 服务端对话。

• Java

首先运行服务端：

你可以从 examples 目录构建并运行服务端。首先构建客户端和服务端：

`

$ ../gradlew -PskipCodegen=true installDist

`

然后运行服务端，服务端将监听 50051 ：

```
$ ./build/install/grpc-examples/bin/hello-world-server
```

• C++

你可以从 examples/cpp/helloworld 目录下构建并运行服务端。首先构建客户端和服务端：

$ make

然后运行服务端，服务端将监听 50051 ：

$ ./greeter_server

• Python

你可以用如下命令到 examples/python/helloworld 下运行服务端：

$ ./run_server.sh

• Go

你可以用如下命令到 examples/helloworld 下运行服务端：

$ greeter_server &

• Ruby

你可以用如下命令到 examples/ruby 下运行服务端：

$ bundle exec ./greeter_server.rb &

• Node.js

你可以用如下命令到 examples/node 下运行服务端：

$ node ./greeter_server.js &

• C#

构建解决方案，然后到 examples/csharp ：

```
> cd GreeterServer/bin/Debug
> GreeterServer.exe
```

一旦服务器在运行，在其他的终端窗口运行客户端并确认它收到一个消息。

• Java

你可以在 examples 目录下构建并运行客户端。假如你还没有构建客户端，可以使用如下命令：

```
$ ../gradlew -PskipCodegen=true installDist
```

然后运行客户端：

```
$ ./build/install/grpc-examples/bin/hello-world-client
```

• C++

你可以在 examples/cpp/helloworld 目录下构建并运行客户端。假如你还没有构建客户端，可以使用如下命令：

$ make

然后运行客户端：

$ ./greeter_client

• Python

你可以从 examples/python/helloworld 目录下用如下命令运行客户端：

$ ./run_client.sh

• Go

你可以从 examples/helloworld 目录下用如下命令运行客户端：

$ greeter_client

• Ruby

你可以从 examples/node 目录下用如下命令运行客户端：

$ bundle exec ./greeter_client.rb

• Node.js

你可以从 examples/node 目录下用如下命令运行客户端：

$ node ./greeter_client.js

• C#

构建解决方案，然后从 examples/csharp 目录：

```
> cd GreeterClient/bin/Debug
> GreeterClient.exe
```

• Objective-C

打开由Cocoapods 创建的 XCode 工作空间，运行应用，你可以在 XCode 的控制台日志里看到结果。

• PHP

你可以从 examples/php 目录运行客户端 ：

```
$ ./run_greeter_client.sh
```

更多资料!

• 从快速开始(#quickstart)找到如何安装 gRPC 并从每个语言开始 。

• 按照你喜欢的语言对应教程来学习。

• 从 gRPC 概念(/docs/guides/concepts.html)发现更多包括 RPC 生命周期，同步、异步调用，过期时间等内容。

• 读一下 HTTP2协议上的 gRPC(/docs/guides/wire.html) 里的详细描述

• gRPC 认证支持(/docs/guides/auth.html)里则介绍了 gRPC 对认证支持的机制和例子。

gRPC概念

gRPC 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(/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

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)
```

```
// 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);
});
```

(

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)

```php
// 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

HTTP2 协议上的 gRPC

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(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"

教程

C++教程

gRPC基础：C++

本教程提供了C++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 C++ 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(/docs/index.html)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 C++ 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间通信的复杂性以及环境的不同.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。

例子代码和设置

教程的代码在这里

grpc/grpc/examples/cpp/route_guide(https://github.com/grpc/grpc/tree/{{). 要下载例子，通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
```

改变当前的目录到examples/cpp/route_guide：

```
$ cd examples/cpp/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，查看下面的设置指南 C++快速开始指南(/docs/installation/c.html)。

定义服务

我们的第一步(可以从概览(/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/{{}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，在 RouteGuide 服务中都有使用：

• 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 C++ 插件来完成。

简单起见，我们提供一个 makefile(https://github.com/grpc/grpc/blob/{{} 帮您用合适的插件，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 代码安装指南(https://github.com/grpc/grpc/blob/{{})来操作：

```
$ make route_guide.grpc.pb.cc route_guide.pb.cc
```

实际上运行的是：

```
$ protoc -I ../../protos --grpc_out=. --plugin=protoc-gen-grpc=`which
grpc_cpp_plugin` ../../protos/route_guide.proto
$ protoc -I ../../protos --cpp_out=. ../../protos/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.h， 声明生成的消息类的头文件

• route_guide.pb.cc， 包含消息类的实现

• route_guide.grpc.pb.h， 声明你生成的服务类的头文件

• route_guide.grpc.pb.cc， 包含服务类的实现

这些包括：

• 所有的填充，序列化和获取我们请求和响应消息类型的 protocol buffer 代码

• 名为 RouteGuide 的类，包含

• 为了客户端去调用定义在 RouteGuide 服务的远程接口类型(或者 *存根*)

• 让服务器去实现的两个抽象接口，同时包括定义在 RouteGuide 中的方法。

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(#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 examples/cpp/route_guide/route_guide_server.cc(https://github.com/grpc/grpc/blob/{{}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 RouteGuide::Service 接口的 RouteGuideImpl 类：

```
class RouteGuideImpl final : public RouteGuide::Service {
...
}
```

在这个场景下，我们正在实现 *同步* 版本的RouteGuide，它提供了 gRPC 服务器缺省的行为。同时，也有可能去实现一个异步的接口 RouteGuide::AsyncService，它允许你进一步定制服务器线程的行为，虽然在本教程中我们并不关注这点。

RouteGuideImpl 实现了所有的服务方法。让我们先来看看最简单的类型 GetFeature，它从客户端拿到一个 Point 然后将对应的特性返回给数据库中的 Feature。

```
  Status GetFeature(ServerContext* context, const Point* point,
                    Feature* feature) override {
    feature->set_name(GetFeatureName(*point, feature_list_));
    feature->mutable_location()——>CopyFrom(*point);
    return Status::OK;
  }
```

这个方法为 RPC 传递了一个上下文对象，包含了客户端的 Point protocol buffer 请求以及一个填充响应信息的Feature protocol buffer。在这个方法中，我们用适当的信息填充 Feature，然后返回OK的状态，告诉 gRPC 我们已经处理完 RPC，并且 Feature 可以返回给客户端。

现在让我们看看更加复杂点的情况——流式RPC。 ListFeatures 是一个服务器端的流式 RPC，因此我们需要给客户端返回多个 Feature。

```
Status ListFeatures(ServerContext* context, const Rectangle* rectangle,
                    ServerWriter<Feature>* writer) override {
  auto lo = rectangle->lo();
  auto hi = rectangle->hi();
  long left = std::min(lo.longitude(), hi.longitude());
  long right = std::max(lo.longitude(), hi.longitude());
  long top = std::max(lo.latitude(), hi.latitude());
  long bottom = std::min(lo.latitude(), hi.latitude());
  for (const Feature& f : feature_list_) {
    if (f.location().longitude() >= left &&
        f.location().longitude() <= right &&
        f.location().latitude() >= bottom &&
        f.location().latitude() <= top) {
      writer->Write(f);
    }
  }
  return Status::OK;
}
```

如你所见，这次我们拿到了一个请求对象(客户端期望在 Rectangle 中找到的 Feature)以及一个特殊的 ServerWriter 对象，而不是在我们的方法参数中获取简单的请求和响应对象。在方法中，根据返回的需要填充足够多的 Feature 对象，用 ServerWriter 的 Write() 方法写入。最后，和我们简单的 RPC 例子相同，我们返回Status::OK去告知gRPC我们已经完成了响应的写入。

如果你看过客户端流方法RecordRoute，你会发现它很类似，除了这次我们拿到的是一个ServerReader而不是请求对象和单一的响应。我们使用 ServerReader 的 Read() 方法去重复的往请求对象(在这个场景下是一个 Point)读取客户端的请求直到没有更多的消息：在每次调用后，服务器需要检查 Read() 的返回值。如果返回值为 true，流仍然存在，它就可以继续读取；如果返回值为 false，则表明消息流已经停止。

```
while (stream->Read(&point)) {
  ...//process client input
}
```

最后，让我们看看双向流RPCRouteChat()。

```
Status RouteChat(ServerContext* context,
                 ServerReaderWriter<RouteNote, RouteNote>* stream) override {
  std::vector<RouteNote> received_notes;
  RouteNote note;
  while (stream->Read(&note)) {
    for (const RouteNote& n : received_notes) {
      if (n.location().latitude() == note.location().latitude() &&
          n.location().longitude() == note.location().longitude()) {
        stream->Write(n);
```

```
      }
    }
    received_notes.push_back(note);
  }

  return Status::OK;
}
```

这次我们得到的 ServerReaderWriter 对象可以用来读 *和* 写消息。这里读写的语法和我们客户端流以及服务器流方法是一样的。虽然每一端获取对方信息的顺序和写入的顺序一致，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。

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：

```
void RunServer(const std::string& db_path) {
  std::string server_address("0.0.0.0:50051");
  RouteGuideImpl service(db_path);

  ServerBuilder builder;
  builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
  builder.RegisterService(&service);
  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << std::endl;
  server->Wait();
}
```

如你所见，我们通过使用ServerBuilder去构建和启动服务器。为了做到这点，我们需要：

1. 创建我们的服务实现类 RouteGuideImpl 的一个实例。
2. 创建工厂类 ServerBuilder 的一个实例。
3. 在生成器的 AddListeningPort() 方法中指定客户端请求时监听的地址和端口。
4. 用生成器注册我们的服务实现。
5. 调用生成器的 BuildAndStart() 方法为我们的服务创建和启动一个RPC服务器。
6. 调用服务器的 Wait() 方法实现阻塞等待，直到进程被杀死或者 Shutdown() 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C++的客户端。你可以从examples/cpp/route_guide/route_guide_client.cc(https://github.com/grpc/grpc/blob/{{}看到我们完整的客户端例子代码.

创建一个存根

为了能调用服务的方法，我们得先创建一个 *存根*。

首先需要为我们的存根创建一个gRPC *channel*，指定我们想连接的服务器地址和端口，以及channel 相关的参数——在本例中我们使用了缺省的 ChannelArguments 并且没有使用SSL：

```
grpc::CreateChannel("localhost:50051", grpc::InsecureCredentials(),
ChannelArguments());
```

现在我们可以利用channel，使用从.proto中生成的RouteGuide类提供的NewStub方法去创建存根。

```
public:
 RouteGuideClient(std::shared_ptr<ChannelInterface> channel,
                  const std::string& db)
    : stub_(RouteGuide::NewStub(channel)) {
   ...
 }
```

调用服务的方法

现在我们来看看如何调用服务的方法。注意，在本教程中调用的方法，都是 *阻塞/同步* 的版本：这意味着 RPC 调用会等待服务器响应，要么返回响应，要么引起一个异常。

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。

```
  Point point;
  Feature feature;
  point = MakePoint(409146138, -746188906);
  GetOneFeature(point, &feature);

...

  bool GetOneFeature(const Point& point, Feature* feature) {
    ClientContext context;
    Status status = stub_->GetFeature(&context, point, feature);
    ...
  }
```

如你所见，我们创建并且填充了一个请求的 protocol buffer 对象（例子中为 Point），同时为了服务器填写创建了一个响应 protocol buffer 对象。为了调用我们还创建了一个 ClientContext 对象——你可以随意的设置该对象上的配置的值，比如期限，虽然现在我们会使用缺省的设置。注意，你不能在不同的调用间重复使用这个对象。最后，我们在存根上调用这个方法，将其传给上下文，请求以及响应。如果方法的返回是OK，那么我们就可以从服务器从我们的响应对象中读取响应信

息。

```
std: : cout << "Found feature called " << feature->name()  << " at "
          << feature->location().latitude()/kCoordFactor_ << ", "
          << feature->location().longitude()/kCoordFactor_ << std: : endl;
```

流式RPC

现在来看看我们的流方法。如果你已经读过创建服务器(#server)，本节的一些内容看上去很熟悉——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。下面就是我们称作是服务器端的流方法 ListFeatures，它会返回地理的 Feature：

```
std::unique_ptr<ClientReader<Feature> > reader(
    stub_->ListFeatures(&context, rect));
while (reader->Read(&feature)) {
  std::cout << "Found feature called "
          << feature.name() << " at "
          << feature.location().latitude()/kCoordFactor_ << ", "
          << feature.location().longitude()/kCoordFactor_ << std::endl;
}
Status status = reader->Finish();
```

我们将上下文传给方法并且请求，得到 ClientReader 返回对象，而不是将上下文，请求和响应传给方法。客户端可以使用 ClientReader 去读取服务器的响应。我们使用 ClientReader 的 Read() 反复读取服务器的响应到一个响应 protocol buffer 对象(在这个例子中是一个 Feature)，直到没有更多的消息：客户端需要去检查每次调用完 Read() 方法的返回值。如果返回值为 true，流依然存在并且可以持续读取；如果是 false，说明消息流已经结束。最后，我们在流上调用 Finish() 方法结束调用并获取我们 RPC 的状态。

客户端的流方法 RecordRoute 的使用很相似，除了我们将一个上下文和响应对象传给方法，拿到一个 ClientWriter 返回。

```
std::unique_ptr<ClientWriter<Point> > writer(
    stub_->RecordRoute(&context, &stats));
for (int i = 0; i < kPoints; i++) {
  const Feature& f = feature_list_[feature_distribution(generator)];
  std::cout << "Visiting point "
          << f.location().latitude()/kCoordFactor_ << ", "
          << f.location().longitude()/kCoordFactor_ << std::endl;
  if (!writer->Write(f.location())) {
    // Broken stream.
    break;
  }
  std::this_thread::sleep_for(std::chrono::milliseconds(
      delay_distribution(generator)));
```

```
  }
  writer->WritesDone();
  Status status = writer->Finish();
  if (status.IsOk()) {
    std::cout << "Finished trip with " << stats.point_count() << " points\n"
              << "Passed " << stats.feature_count() << " features\n"
              << "Travelled " << stats.distance() << " meters\n"
              << "It took " << stats.elapsed_time() << " seconds"
              << std::endl;
  } else {
    std::cout << "RecordRoute rpc failed." << std::endl;
  }
```

一旦我们用 Write() 将客户端请求写入到流的动作完成，我们需要在流上调用 WritesDone() 通知 gRPC 我们已经完成写入，然后调用 Finish() 完成调用同时拿到 RPC 的状态。如果状态是 OK，我们最初传给 RecordRoute() 的响应对象会跟着服务器的响应被填充。

最后，让我们看看双向流式 RPC RouteChat()。在这种场景下，我们将上下文传给一个方法，拿到一个可以用来读写消息的ClientReaderWriter的返回。

```
  std::shared_ptr<ClientReaderWriter<RouteNote, RouteNote> > stream(
      stub_->RouteChat(&context));
```

这里读写的语法和我们客户端流以及服务器端流方法没有任何区别。虽然每一方都能按照写入时的顺序拿到另一方的消息，客户端和服务器端都可以以任意顺序读写——流操作起来是完全独立的。

来试试吧！

构建客户端和服务器：

```
$ make
```

运行服务器，它会监听50051端口：

```
$ ./route_guide_server
```

在另外一个终端运行客户端：

```
$ ./route_guide_client
```

教程

gRPC 基础： C#

本教程提供了 C# 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 C# 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(/docs/index.html)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 C# 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间通信的复杂性以及环境的不同。使用 protocol buffers 还能获得其他好处，包括高效的序列号，简单的 IDL 以及容易进行的接口更新。

例子代码和设置

教程的代码在这里 grpc/grpc/examples/cpp/route_guide(https://github.com/grpc/grpc/tree/{}. 要下载例子，请通过运行下面的命令去克隆grpc代码库:

```
$ git clone https://github.com/grpc/grpc.git
```

本教程的所有文件都在examples/csharp/route_guide目录下。

从 Visual Studio (或者 Linux 上的 Monodevelop)打开解决方案 examples/csharp/route_guide/RouteGuide.sln。

如果系统是Windows，除了打开解决方案文件之外，你应当不用多做任何事情。所有你需要的依赖都会在都会在构建解决方案的过程中通过 Grpc NuGet 包自动恢复。

如果系统是 Linux 或者 Mac OS X，为了生成服务器和客户端接口代码，运行例子，你首先需要安装 protobuf 和 gRPC 的 C# 原生依赖。请查看如何使用指令(https://github.com/grpc/grpc/tree/{}的文档。

定义服务

我们的第一步(可以从概览(/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/{}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，在 RouteGuide 服务中都有使用：

• 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 C# 插件来完成。

如果你想自己运行，请确保你已经安装了 protoc 和 gRPC 的 C# 插件。运行的指令因操作系统而异：

• 对于 Windows 系统来说，Grpc.Tools 和 Google.Protobuf 的 NuGet 包包含了生成代码所需要的二进制文件。

• 对于 Linux 或者 OS X，请确保你查看了如何使用指令 (https://github.com/grpc/grpc/tree/{{}的文档。

以上都完成后，你就可以生成下面的 C# 代码：

• 我们使用 Google.Protobuf NuGet 包的protoc.exe 和 Grpc.Tools NuGet 包的 grpc*csharp* plugin.exe （都在 tools 目录下）在 Windows 上生成代码。

一般来说，你需要自己添加 Grpc.Tools 包到解决方案中，但在本教程中，它已经帮你完成了。下面的命令应当从examples/csharp/route_guide 目录运行：

```
  > packages\Google.Protobuf.3.0.0-alpha4\tools\protoc.exe -I../../protos --csharp_out
RouteGuide --grpc_out RouteGuide --plugin=protoc-gen-
grpc=packages\Grpc.Tools.0.7.0\tools\grpc_csharp_plugin.exe
../../protos/route_guide.proto
```

• 在 Linux 或者 OS X 系统，我们可以用 Linuxbrew/Homebrew 安装 protoc 和 grpc*csharp* plugin 依赖。从 route_guide 目录运行下面的命令：

```
  $ protoc -I../../protos --csharp_out RouteGuide --grpc_out RouteGuide --
plugin=`which grpc_csharp_plugin` ../../protos/route_guide.proto
```

根据操作系统的不同，运行对应的命令去重新在 RouteGuide 目录生成下面的文件：

• RouteGuide/RouteGuide.cs 定义了一个命名空间 Routeguide

• 它包含了所有填充，序列化的 protocol buffer 代码以及获取我们的请求和响应的类型。

• RouteGuide/RouteGuideGrpc.cs, 提供了存根和服务类

• 定义 RouteGuide 服务实现时继承的接口 RouteGuide.IRouteGuide

• 用来访问远程 RouteGuide 实例的类 RouteGuide.RouteGuideClient

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(#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 examples/csharp/route_guide/RouteGuideServer/RouteGuideImpl.cs(https://github.com/grpc/grpc/blob/{{}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 RouteGuide.IRouteGuide 接口的 RouteGuideImpl 类：

```
// RouteGuideImpl provides an implementation of the RouteGuide service.
public class RouteGuideImpl : RouteGuide.IRouteGuide
```

简单RPC

RouteGuideImpl 实现了所有的服务方法。让我们先来看看最简单的类型 GetFeature，它从客户端拿到一个 Point 然后将对应的从数据库中取得的特征信息置于 Feature 内返回给客户端。

```
public Task<Feature> GetFeature(Point request, Grpc.Core.ServerCallContext context)
{
    return Task.FromResult(CheckFeature(request));
}
```

为了 RPC，方法被传入一个上下文（alpha 版的时候为空），指客户端的 Point protocol buffer 请求，返回一个 Feature protocol buffer。在方法中，我们用适当的信息创建 Feature 然后返回。为了允许异步的实现，方法返回 Task<Feature> 而不仅是 Feature。你可以随意的同步执行你的计算，并在完成后返回结果，就像我们在例子中做的一样。

服务器端流式 RPC

现在让我们来看看稍微复杂点的 —— 一个流式 RPC。ListFeatures 是一个服务器端的流式 RPC，所以我们需要给客户端发回多个 Feature protocol buffer。

```
// in RouteGuideImpl
public async Task ListFeatures(Rectangle request,
    Grpc.Core.IServerStreamWriter<Feature> responseStream,
    Grpc.Core.ServerCallContext context)
{
    var responses = features.FindAll( (feature) => feature.Exists() &&
request.Contains(feature.Location) );
    foreach (var response in responses)
```

```
    {
        await responseStream.WriteAsync(response);
    }
}
```

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，但是我们需要使用异步方法 WriteAsync 写入响应到 IServerStreamWriter 异步流而不是一个简单的响应。

客户端流 RPC

类似的，客户端流方法 RecordRoute 使用一个 IAsyncEnumerator(https://github.com/Reactive-Extensions/Rx.NET/blob/master/Ix.NET/Source/System.Interactive.Async/IAsyncEnumerator.cs)，使用异步方法 MoveNext 和 Current 属性去读取请求的流。

```
public async Task<RouteSummary>
RecordRoute(Grpc.Core.IAsyncStreamReader<Point> requestStream,
    Grpc.Core.ServerCallContext context)
{
    int pointCount = 0;
    int featureCount = 0;
    int distance = 0;
    Point previous = null;
    var stopwatch = new Stopwatch();
    stopwatch.Start();

    while (await requestStream.MoveNext())
    {
        var point = requestStream.Current;
        pointCount++;
        if (CheckFeature(point).Exists())
        {
            featureCount++;
        }
        if (previous != null)
        {
            distance += (int) previous.GetDistance(point);
        }
        previous = point;
    }

    stopwatch.Stop();
```

```
    return new RouteSummary
    {
        PointCount = pointCount,
        FeatureCount = featureCount,
        Distance = distance,
        ElapsedTime = (int)(stopwatch.ElapsedMilliseconds / 1000)
    };
}
```

双向流 RPC

最后，让我们来看看双向流 RPC RouteChat。

```
public async Task RouteChat(Grpc.Core.IAsyncStreamReader<RouteNote>
requestStream,
    Grpc.Core.IServerStreamWriter<RouteNote> responseStream,
    Grpc.Core.ServerCallContext context)
{
    while (await requestStream.MoveNext())
    {
        var note = requestStream.Current;
        List<RouteNote> prevNotes = AddNoteForLocation(note.Location, note);
        foreach (var prevNote in prevNotes)
        {
            await responseStream.WriteAsync(prevNote);
        }
    }
}
```

这里的方法同时接收到 requestStream 和 responseStream 参数。读取请求和客户端流方法 RecordRoute 的方式相同。写入响应和服务器端流方法 ListFeatures 的方式相同。

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：

```
var features = RouteGuideUtil.ParseFeatures(RouteGuideUtil.DefaultFeaturesFile);

Server server = new Server
{
    Services = { RouteGuide.BindService(new RouteGuideImpl(features)) },
    Ports = { new ServerPort("localhost", Port, ServerCredentials.Insecure) }
```

```
};
server.Start();

Console.WriteLine("RouteGuide server listening on port " + port);
Console.WriteLine("Press any key to stop the server...");
Console.ReadKey();

server.ShutdownAsync().Wait();
```

如你所见，我们通过使用 Grpc.Core.Server 去构建和启动服务器。为了做到这点，我们需要：

1. 创建 Grpc.Core.Server 的一个实例。
2. 创建我们的服务实现类 RouteGuideImpl 的一个实例。
3. 通过在 Services 集合中添加服务的定义（我们从生成的 RouteGuide.BindService 方法中获得服务定义）注册我们的服务实现。
4. 指定想要接受客户端请求的地址和监听的端口。通过往 Ports 集合中添加 ServerPort 即可完成。
5. 在服务器实例上调用 Start 为我们的服务启动一个 RPC 服务器。

创建客户端

在这一部分，我们将会学习用RouteGuide创建一个 C# 客户端。可以在 examples/csharp/route_guide/RouteGuideClient/Program.cs(https://github.com/grpc/grpc/blob/{{}查看完整的客户端代码。

创建一个存根

为了能调用服务的方法，我们得先创建一个 *存根*。

首先需要为我们的存根创建一个可以连接到 gRPC 服务器的 gRPC channel。然后我们使用 .proto 生成的RouteGuide类的RouteGuide.NewClient方法。

```
Channel channel = new Channel("127.0.0.1:50052", ChannelCredentials.Insecure)
var client = RouteGuide.NewClient(channel);

// YOUR CODE GOES HERE

channel.ShutdownAsync().Wait();
```

调用服务的方法

现在我们来看看如何调用服务的方法。gRPC C# 的每个支持的方法类型都提供了异步的版本。为了方便起见，gPRC C# 也提供了同步方法存根，不过只能用于简单的（单个请求/单个响应）RPC。

简单 RPC

以同步的方式调用简单 RPC GetFeature 几乎是和调用一个本地方法一样直观。

```
Point request = new Point { Latitude = 409146138, Longitude = -746188906 };
Feature feature = client.GetFeature(request);
```

如你所见，我们创建并且填充了一个请求的 protocol buffer 对象（例子中为 Point），在客户端对象上调用期望的方法，并传入请求。如果 RPC 成功结束，则返回响应的 protocol buffer（在例子中是Feature）。否则抛出 RpcException 类型的异常，指出问题的状态码。

或者，如果你在异步的上下文环境中，你可以调用这个方法的异步版本并且使用 await 关键字来等待结果:

```
Point request = new Point { Latitude = 409146138, Longitude = -746188906 };
Feature feature = await client.GetFeatureAsync(request);
```

流式 RPC

现在来看看我们的流方法。如果你已经读过创建服务器(#server)，本节的一些内容看上去很熟悉——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。和简单的调用不同的地方在于客户端方法返回了调用对象的实例。它提供了使用请求/响应流和（或者）异步的结果，取决于你使用的流类型。

下面就是我们称作是服务器端的流方法 ListFeatures，它有IAsyncEnumerator<Feature>类型的属性ReponseStream：

```
using (var call = client.ListFeatures(request))
{
    while (await call.ResponseStream.MoveNext())
    {
        Feature feature = call.ResponseStream.Current;
        Console.WriteLine("Received " + feature.ToString());
    }
}
```

客户端的流方法 RecordRoute 的使用和它很相似，除了我们通过 WriteAsync 使用 RequestStream 属性挨个写入请求，最后使用 CompleteAsync 去通知不再需要发送更多的请求。可以通过 ResponseAsync 获取方法的结果。

```
using (var call = client.RecordRoute())
{
    foreach (var point in points)
    {
        await call.RequestStream.WriteAsync(point);
    }
    await call.RequestStream.CompleteAsync();

    RouteSummary summary = await call.ResponseAsync;
```

```
}
```

最后，让我们看看双向流式 RPC RouteChat()。在这种场景下，我们将请求写入 RequestStream 并且从 ResponseStream 接受到响应。从例子可以看出，流之间是互相独立的。

```
using (var call = client.RouteChat())
{
    var responseReaderTask = Task.Run(async () =>
    {
        while (await call.ResponseStream.MoveNext())
        {
            var note = call.ResponseStream.Current;
            Console.WriteLine("Received " + note);
        }
    });

    foreach (RouteNote request in requests)
    {
        await call.RequestStream.WriteAsync(request);
    }
    await call.RequestStream.CompleteAsync();
    await responseReaderTask;
}
```

来试试吧！

构建客户端和服务器：

• 用 Visual Studio (或者Linux上的Monodevelop) 打开解决方案 examples/csharp/route_guide/RouteGuide.sln并选择 **Build**。

• 运行服务器，它会监听50052端口：

```
> cd RouteGuideServer/bin/Debug
> RouteGuideServer.exe
```

• 在另一个终端运行客户端：

```
> cd RouteGuideClient/bin/Debug
> RouteGuideClient.exe
```

你也可以直接从 Visual Studio 里直接运行服务器和客户端。

在Linux 或者 Mac系统中， 使用 mono RouteGuideServer.exe 和 mono RouteGuideClient.exe 命令去运行服务器和客户端。

Go 教程

gRPC 基础: Go

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(/docs/installation/go.html)。

定义服务

我们的第一步(可以从概览(/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/{{}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(#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(#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
```

Java 教程

gRPC 基础： Java

本教程提供了 Java 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 Java 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 Java 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-java/examples/src/main/java/io/grpc/examples(https://github.com/grpc/grpc-java/tree/master/examples/src/main/java/io/grpc/examples)。 要下载例子，通过运行下面的命令去克隆grpc-java代码库：

```
$ git clone https://github.com/grpc/grpc-java.git
```

然后改变当前的目录到 grpc-java/examples：

```
$ cd grpc-java/examples
```

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 Java快速开始指南(/docs/installation/java.html)。

定义服务

我们的第一步(可以从概览(/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grpc-java/examples/src/main/proto/route_guide.proto(https://github.com/grpc/grpc-java/blob/master/examples/src/main/proto/route_guide.proto)看到完整的 .proto 文件。

在生成例子中的 Java 代码的时候，在 .proto 文件中我们指定了一个 java_package 文件的选项：

```
option java_package = "io.grpc.examples";
```

这个指定的包是为我们生成 Java 类使用的。如果在 .proto 文件中没有显示的 java_package 参数，

那么就会使用缺省的 proto 包（通过 "package" 关键字指定）。但是，因为 proto 包一般不是以域名

翻转的格式命名，所以它不是好的 Java 包。 如果我们用其它语言通过 .proto 文件生成代码，java_package 是不起任何作用的。

要定义一个服务，你必须在你的 .proto 文件中指定 service：

```
service RouteGuide {
```

```
  ...
}
```

然后在我们的服务中定义 rpc 方法，指定它们的请求的和响应类型。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 Java 插件来完成。为了生成 gRPC 服务，你必须使用proto3(https://github.com/google/protobuf/releases)编译器（同时支持 proto2 和 proto3 语法）。

这个例子使用的构建系统也是 Java gRPC 本身构建的一部分——为了简单起见，我们推荐使用为这个例子

提前生成的代码。你可以参考README(https://github.com/grpc/grpc-java/blob/master/README.md)学习如何从你的 .proto 文件中生成代码。

从这里src/generated/main(https://github.com/grpc/grpc-java/tree/master/examples/src/generated/main)可以看到为了例子预生成的代码。

下面的类都是从我们的服务定义中生成：

• 包含了所有填充，序列化以及获取请求和应答的消息类型的Feature.java，Point.java，Rectangle.java以及其它类文件。

• RouteGuideGrpc.java 文件包含（以及其它一些有用的代码）：

• RouteGuide 服务器要实现的一个接口 RouteGuideGrpc.RouteGuide，其中所有的方法都定义在RouteGuide服务中。

• 客户端可以用来和RouteGuide服务器交互的 *存根* 类。

```
异步的存根也实现了 `RouteGuide` 接口。
```

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(#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-java/examples/src/main/java/io/grpc/examples/RouteGuideServer.java](https://github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/routeguide/RouteGuideServer.java)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

如你所见，我们的服务器有一个实现了生成的 RouteGuideGrpc.Service 接口的 RouteGuideService类：

```
private static class RouteGuideService implements RouteGuideGrpc.RouteGuide {
...
}
```

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。

```
    @Override
    public void getFeature(Point request, StreamObserver<Feature> responseObserver)
{
      responseObserver.onNext(checkFeature(request));
      responseObserver.onCompleted();
    }

...

    private Feature checkFeature(Point location) {
      for (Feature feature : features) {
        if (feature.getLocation().getLatitude() == location.getLatitude()
            && feature.getLocation().getLongitude() == location.getLongitude()) {
          return feature;
        }
      }

      // No feature was found, return an unnamed feature.
      return Feature.newBuilder().setName("").setLocation(location).build();
    }
```

getFeature() 接收两个参数：

- Point： 请求
- StreamObserver<Feature>： 一个应答的观察者，实际上是服务器调用它应答的一个特殊接口

。

要将应答返回给客户端，并完成调用：

1. 如在我们的服务定义中指定的那样，我们组织并填充一个 Feature 应答对象返回给客户端。在这个

例子中，我们通过一个单独的私有方法checkFeature()来实现。

1. 我们使用应答观察者的 onNext() 方法返回 Feature。

2. 我们使用应答观察者的 onCompleted() 方法来指出我们已经完成了和 RPC的交互。

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。

```
private final Collection<Feature> features;

...

    @Override
    public void listFeatures(Rectangle request, StreamObserver<Feature>
responseObserver) {
      int left = min(request.getLo().getLongitude(), request.getHi().getLongitude());
      int right = max(request.getLo().getLongitude(), request.getHi().getLongitude());
      int top = max(request.getLo().getLatitude(), request.getHi().getLatitude());
      int bottom = min(request.getLo().getLatitude(), request.getHi().getLatitude());

      for (Feature feature : features) {
        if (!RouteGuideUtil.exists(feature)) {
          continue;
        }

        int lat = feature.getLocation().getLatitude();
        int lon = feature.getLocation().getLongitude();
        if (lon >= left && lon <= right && lat >= bottom && lat <= top) {
          responseObserver.onNext(feature);
        }
      }
      responseObserver.onCompleted();
    }
```

和简单 RPC 类似，这个方法拿到了一个请求对象（客户端期望从 Rectangle 找到 Feature）和一个应答观察者 StreamObserver。

这次我们得到了需要返回给客户端的足够多的 Feature 对象（在这个场景下，我们根据他们是否在我们的 Rectangle 请求中，从服务的特性集合中选择他们），并且使用 onNext() 方法轮流往响应

观察者写入。最后，和简单 RPC 的例子一样，我们使用响应观察者的 onCompleted() 方法去告诉 gRPC 写入应答已完成。

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，并且返回一个包括它们路径的信息 RouteSummary。

```
 @Override
   public StreamObserver<Point> recordRoute(final
StreamObserver<RouteSummary> responseObserver) {
     return new StreamObserver<Point>() {
       int pointCount;
       int featureCount;
       int distance;
       Point previous;
       long startTime = System.nanoTime();

       @Override
       public void onNext(Point point) {
         pointCount++;
         if (RouteGuideUtil.exists(checkFeature(point))) {
           featureCount++;
         }
         // For each point after the first, add the incremental distance from the previous
point
         // to the total distance value.
         if (previous != null) {
           distance += calcDistance(previous, point);
         }
         previous = point;
       }

       @Override
       public void onError(Throwable t) {
         logger.log(Level.WARNING, "Encountered error in recordRoute", t);
       }

       @Override
       public void onCompleted() {
         long seconds = NANOSECONDS.toSeconds(System.nanoTime() - startTime);

responseObserver.onNext(RouteSummary.newBuilder().setPointCount(pointCount)
```

```
            .setFeatureCount(featureCount).setDistance(distance)
            .setElapsedTime((int) seconds).build());
        responseObserver.onCompleted();
      }
    };
  }
```

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。

如你所见，我们的方法和前面的方法类型相似，拿到一个 StreamObserver 应答观察者参数，但是这次它返回一个 StreamObserver 以便客户端写入它的 Point。

在这个方法体中，我们返回了一个匿名 StreamObserver 实例，其中我们：

- 覆写了 onNext() 方法，每次客户端写入一个 Point 到消息流时，拿到特性和其它信息。
- 覆写了 onCompleted() 方法（在 *客户端* 结束写入消息时调用），用来填充和构建我们的 RouteSummary。然后我们用 RouteSummary 调用方法自己的的响应观察者的 onNext()，之后调用它的 onCompleted() 方法，结束服务器端的调用。

双向流式 RPC

最后，让我们看看双向流式 RPC RouteChat()。

```
  @Override
  public StreamObserver<RouteNote> routeChat(final StreamObserver<RouteNote>
responseObserver) {
    return new StreamObserver<RouteNote>() {
      @Override
      public void onNext(RouteNote note) {
        List<RouteNote> notes = getOrCreateNotes(note.getLocation());

        // Respond with all previous notes at this location.
        for (RouteNote prevNote : notes.toArray(new RouteNote[0])) {
          responseObserver.onNext(prevNote);
        }

        // Now add the new note to the list
        notes.add(note);
      }

      @Override
      public void onError(Throwable t) {
        logger.log(Level.WARNING, "Encountered error in routeChat", t);
```

```
    }

    @Override
    public void onCompleted() {
      responseObserver.onCompleted();
    }
  };
}
```

和我们的客户端流的例子一样，我们拿到和返回一个 StreamObserver 应答观察者，除了这次我们在客户端仍然写入消息到 *它们的* 消息流时通过我们方法的应答观察者返回值。这里读写的语法和客户端流以及服务器流方法一样。虽然每一端都会按照它们写入的顺序拿到另一端的消息，客户端和服务器都可以任意顺序读写——流的操作是互不依赖的。

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：

```
public void start() {
  gRpcServer = NettyServerBuilder.forPort(port)
      .addService(RouteGuideGrpc.bindService(new RouteGuideService(features)))
      .build().start();
  logger.info("Server started, listening on " + port);
  ...
}
```

如你所见，我们用一个 NettyServerBuilder 构建和启动服务器。这个服务器的生成器基于 Netty(http://netty.io/) 传输框架。

为了做到这个，我们需要：

1. 创建我们服务实现类 RouteGuideService 的一个实例并且将其传给生成的 RouteGuideGrpc 类的静态方法 bindService() 去获得服务定义。

2. 使用生成器的 forPort() 方法指定地址以及期望客户端请求监听的端口。

3. 通过传入将 bindService() 返回的服务定义，用生成器注册我们的服务实现到生成器的 addService() 方法。

4. 调用生成器上的 build() 和 start() 方法为我们的服务创建和启动一个 RPC 服务器。

创建客户端

在这部分，我们将尝试为 RouteGuide 服务创建一个 Java 的客户端。你可以从grpc-java/examples/src/main/java/io/grpc/examples/RouteGuideClient.java(https://github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/routeguide/RouteGuideClient.java)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，我们需要首先创建一个 *存根*，或者两个存根：

• 一个 *阻塞/同步* 存根：这意味着 RPC 调用等待服务器响应，并且要么返回应答，要么造成异常。

• 一个 *非阻塞/异步* 存根可以向服务器发起非阻塞调用，应答会异步返回。你可以使用异步存根去发起特定类型的流式调用。

我们首先为存根创建一个 gRPC *channel*，指明服务器地址和我们想连接的端口号：

```
channel = NettyChannelBuilder.forAddress(host, port)
      .negotiationType(NegotiationType.PLAINTEXT)
      .build();
```

如你所见，我们用一个 NettyServerBuilder 构建和启动服务器。这个服务器的生成器基于 Netty(http://netty.io/) 传输框架。

我们使用 Netty(http://netty.io/) 传输框架，所以我们用一个 NettyServerBuilder 启动服务器。

现在我们可以通过从 .proto 中生成的 RouteGuideGrpc 类的 newStub 和 newBlockingStub 方法，使用频道去创建我们的存根。

```
   blockingStub = RouteGuideGrpc.newBlockingStub(channel);
   asyncStub = RouteGuideGrpc.newStub(channel);
```

调用服务方法

现在让我们看看如何调用服务方法。

简单 RPC

在阻塞存根上调用简单 RPC GetFeature 几乎是和调用一个本地方法一样直观。

```
    Point request = Point.newBuilder().setLatitude(lat).setLongitude(lon).build();
    Feature feature = blockingStub.getFeature(request);
```

我们创建和填充了一个请求 protocol buffer 对象（在这个场景下是 Point），在我们的阻塞存根上将其传给 getFeature() 方法，拿回一个 Feature。

服务器端流式 RPC

接下来，让我们看一个对于 ListFeatures 的服务器端流式调用，这个调用会返回一个地理性的 Feature 流：

```
    Rectangle request =
      Rectangle.newBuilder()
        .setLo(Point.newBuilder().setLatitude(lowLat).setLongitude(lowLon).build())
        .setHi(Point.newBuilder().setLatitude(hiLat).setLongitude(hiLon).build()).build();
    Iterator<Feature> features = blockingStub.listFeatures(request);
```

如你所见，这和我们刚看过的简单 RPC 很相似，除了方法返回客户端用来读取所有返回的 Feature 的 一个 Iterator，而不是单个的 Feature。

客户端流式 RPC

现在看看稍微复杂点的东西：我们在客户端流方法 RecordRoute 中发送了一个 Point 流给服务器并且拿到一个 RouteSummary。为了这个方法，我们需要使用异步存根。如果你已经阅读了创建服务器(#server)，一些部分看起来很相近——异步流式 RPC 是在两端通过相似的方式实现的。

```
  public void recordRoute(List<Feature> features, int numPoints) throws Exception {
    info("*** RecordRoute");
    final SettableFuture<Void> finishFuture = SettableFuture.create();
    StreamObserver<RouteSummary> responseObserver = new
StreamObserver<RouteSummary>() {
      @Override
      public void onNext(RouteSummary summary) {
        info("Finished trip with {0} points. Passed {1} features. "
            + "Travelled {2} meters. It took {3} seconds.", summary.getPointCount(),
            summary.getFeatureCount(), summary.getDistance(),
summary.getElapsedTime());
      }

      @Override
      public void onError(Throwable t) {
        finishFuture.setException(t);
      }

      @Override
      public void onCompleted() {
        finishFuture.set(null);
      }
    };

    StreamObserver<Point> requestObserver =
asyncStub.recordRoute(responseObserver);
    try {
      // Send numPoints points randomly selected from the features list.
      StringBuilder numMsg = new StringBuilder();
      Random rand = new Random();
      for (int i = 0; i < numPoints; ++i) {
        int index = rand.nextInt(features.size());
        Point point = features.get(index).getLocation();
```

```
      info("Visiting point {0}, {1}", RouteGuideUtil.getLatitude(point),
          RouteGuideUtil.getLongitude(point));
      requestObserver.onNext(point);
      // Sleep for a bit before sending the next one.
      Thread.sleep(rand.nextInt(1000) + 500);
      if (finishFuture.isDone()) {
        break;
      }
    }
    info(numMsg.toString());
    requestObserver.onCompleted();

    finishFuture.get();
    info("Finished RecordRoute");
  } catch (Exception e) {
    requestObserver.onError(e);
    logger.log(Level.WARNING, "RecordRoute Failed", e);
    throw e;
  }
}
```

如你所见，为了调用这个方法我们需要创建一个 StreamObserver，它为了服务器用它的 RouteSummary 应答实现了一个特殊的接口。在 StreamObserver 中，我们：

- 覆写了 onNext() 方法，在服务器把 RouteSummary 写入到消息流时，打印出返回的信息。
- 覆写了 onCompleted() 方法（在 *服务器* 完成自己的调用时调用）去设置 SettableFuture，这样我们可以检查服务器是不是完成写入。

之后，我们将 StreamObserver 传给异步存根的 recordRoute() 方法，拿到我们自己的 StreamObserver 请求观察者将 Point 发给服务器。一旦完成点的写入，我们使用请求观察者的 onCompleted() 方法告诉 gRPC 我们已经完成了客户端的写入。一旦完成，我们就检查 SettableFuture 验证服务器是否已经完成写入。

双向流式 RPC

最后，让我们看看双向流式 RPC RouteChat()。

```
public void routeChat() throws Exception {
  info("*** RoutChat");
  final SettableFuture<Void> finishFuture = SettableFuture.create();
  StreamObserver<RouteNote> requestObserver =
      asyncStub.routeChat(new StreamObserver<RouteNote>() {
        @Override
        public void onNext(RouteNote note) {
          info("Got message \"{0}\" at {1}, {2}", note.getMessage(), note.getLocation()
```

```
            .getLatitude(), note.getLocation().getLongitude());
        }

        @Override
        public void onError(Throwable t) {
          finishFuture.setException(t);
        }

        @Override
        public void onCompleted() {
          finishFuture.set(null);
        }
      });

    try {
      RouteNote[] requests =
          {newNote("First message", 0, 0), newNote("Second message", 0, 1),
              newNote("Third message", 1, 0), newNote("Fourth message", 1, 1)};

      for (RouteNote request : requests) {
        info("Sending message \"{0}\" at {1}, {2}", request.getMessage(),
request.getLocation()
            .getLatitude(), request.getLocation().getLongitude());
        requestObserver.onNext(request);
      }
      requestObserver.onCompleted();

      finishFuture.get();
      info("Finished RouteChat");
    } catch (Exception t) {
      requestObserver.onError(t);
      logger.log(Level.WARNING, "RouteChat Failed", t);
      throw t;
    }
  }
```

和我们的客户端流的例子一样，我们拿到和返回一个 StreamObserver 应答观察者，除了这次我们在客户端仍然写入消息到 *它们的* 消息流时通过我们方法的应答观察者返回值。这里读写的语法和客户端流以及服务器流方法一样。虽然每一端都会按照它们写入的顺序拿到另一端的消息，客户端和服务器都可以任意顺序读写——流的操作是互不依赖的。

来试试吧！

根据example目录下的README(https://github.com/grpc/grpc-java/blob/master/examples/README.md)的指导去构建和运行客户端及服务器。

Node 教程

gRPC 基础： Node.js

本教程提供了 Node.js 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 Node.js 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 Node.js 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/examples/node/route_guide(https://github.com/grpc/grpc/tree/{{)。 要下载例子，通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
```

然后改变当前的目录到 examples/node/route_guide：

```
$ cd examples/node/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 Node.js 快速开始指南(/docs/installation/node.html)。

定义服务

我们的第一步(可以从概览(/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grpc-java/examples/src/main/proto/route_guide.proto(https://github.com/grpc/grpc/blob/{{}}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，在 RouteGuide 服务中都有使用：

• 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;
}
```

从 proto 文件加载服务描述符

Node.js 的类库在运行时加载 .proto 中的客户端存根并动态生成服务描述符。

要加载一个 .proto 文件，只需要 require gRPC 类库，然后使用它的 load() 方法：

```
var grpc = require('grpc');
var protoDescriptor = grpc.load(__dirname + '/route_guide.proto');
// The protoDescriptor object has the full package hierarchy
var example = protoDescriptor.examples;
```

一旦你完成这个，存根构造函数是在 examples 命名空间（protoDescriptor.examples.RouteGuide）中而服务描述符（用来创建服务器）是存根（protoDescriptor.examples.RouteGuide.service）的一个属性。

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(#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 examples/node/route_guide/route_guide_server.js(https://github.com/grpc/grpc/blob/{{}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

可以看出，我们的服务器有一个从 RouteGuide.service 描述符对象生成的 Server 构造函数：

```
var Server = grpc.buildServer([examples.RouteGuide.service]);
```

在这个场景下，我们实现了 *异步* 版本的 RouteGuide，它提供了 gRPC 缺省的行为。

route*guide*server.js 中的函数实现了所有的服务方法。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，然后返回包含从数据库拿到的feature信息的 Feature。

```
function checkFeature(point) {
  var feature;
  // Check if there is already a feature object for the given point
  for (var i = 0; i < feature_list.length; i++) {
    feature = feature_list[i];
    if (feature.location.latitude === point.latitude &&
        feature.location.longitude === point.longitude) {
      return feature;
    }
  }
  var name = '';
  feature = {
    name: name,
    location: point
  };
  return feature;
}
function getFeature(call, callback) {
  callback(null, checkFeature(call.request));
}
```

该方法传入了 RPC 的把 Point 参数作为属性的调用对象，以及一个可以传入我们返回的 Feature 的回调函数。在方法中我们根据给出的点去对应的填充 Feature，并将其传给回调函数，其中第一个参数为 null，表示没有错误。

现在让我们看看稍微复杂点的东西 —— 流式 RPC。 listFeatures 是一个服务器端流式 RPC，所以我们需要发回多个 Feature 给客户端。

```
function listFeatures(call) {
  var lo = call.request.lo;
  var hi = call.request.hi;
  var left = _.min([lo.longitude, hi.longitude]);
  var right = _.max([lo.longitude, hi.longitude]);
  var top = _.max([lo.latitude, hi.latitude]);
  var bottom = _.min([lo.latitude, hi.latitude]);
  // For each feature, check if it is in the given bounding box
  _.each(feature_list, function(feature) {
```

```
    if (feature.name === '') {
      return;
    }
    if (feature.location.longitude >= left &&
        feature.location.longitude <= right &&
        feature.location.latitude >= bottom &&
        feature.location.latitude <= top) {
      call.write(feature);
    }
  });
  call.end();
}
```

如你所见，这次我们拿到了一个实现了 Writable 接口的 call 对象，而不是调用对象和方法参数中的回调函数。

在方法中，我们根据返回的需要填充足够多的 Feature 对象，用它的 write() 方法写入到 call。最后，我们调用 call.end() 表示我们已经完成了所有消息的发送。

如果你看过客户端流方法 RecordRoute，你会发现它很类似，除了这次 call 参数实现了 Reader 的接口。 每次有新数据的时候，call 的 'data' 事件被触发，每次数据读取完成时， 'end' 事件被触发。和一元的场景一样，我们通过调用回调函数来应答：

```
call.on('data', function(point) {
  // Process user data
});
call.on('end', function() {
  callback(null, result);
});
```

最后，让我们来看看双向流式 RPC RouteChat()`。

```
function routeChat(call) {
  call.on('data', function(note) {
    var key = pointKey(note.location);
    /* For each note sent, respond with all previous notes that correspond to
     * the same point */
    if (route_notes.hasOwnProperty(key)) {
      _.each(route_notes[key], function(note) {
        call.write(note);
      });
    } else {
      route_notes[key] = [];
    }
    // Then add the new note to the list
```

```
    route_notes[key].push(JSON.parse(JSON.stringify(note)));
  });
  call.on('end', function() {
    call.end();
  });
}
```

这次我们得到的是一个实现了 Duplex 的 call 对象，可以用来读 *和* 写消息。这里读写的语法和我们客户端流以及服务器流方法是一样的。虽然每一端获取对方信息的顺序和写入的顺序一致，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。

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：

```
function getServer() {
  var server = new grpc.Server();
  server.addProtoService(routeguide.RouteGuide.service, {
    getFeature: getFeature,
    listFeatures: listFeatures,
    recordRoute: recordRoute,
    routeChat: routeChat
  });
  return server;
}
var routeServer = getServer();
routeServer.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());
routeServer.listen();
```

如你所见，我们通过下面的步骤去构建和启动服务器：

1. 通过 RouteGuide 服务描述符创建一个 Server 构造函数。
2. 实现服务的方法。
3. 通过调用 Server 的构造函数以及方法实现去创建一个服务器的实例。
4. 用实例的 bind() 方法指定地址以及我们期望客户端请求监听的端口。
5. 调用实例的 listen() 方法启动一个RPC服务器。

创建客户端

在这部分，我们将尝试为 RouteGuide 服务创建一个 Node.js 的客户端。你可以从 examples/node/route_guide/route_guide_client.js(https://github.com/grpc/grpc/blob/{{}看到我们完整的客户端例子代码。

创建一个存根

为了能调用服务的方法，我们得先创建一个 *存根*。要做到这点，我们只需要调用 RouteGuide 的存根构造函数，指定服务器地址和端口。

```
new example.RouteGuide('localhost:50051', grpc.Credentials.createInsecure());
```

调用服务的方法

现在我们来看看如何调用服务的方法。注意这些方法都是异步的：他们使用事件或者回调函数去获得结果。

简单 RPC

调用简单 RPC GetFeature 几乎是和调用一个本地的异步方法一样直观。

```
var point = {latitude: 409146138, longitude: -746188906};
stub.getFeature(point, function(err, feature) {
  if (err) {
    // process error
  } else {
    // process feature
  }
});
```

如你所见，我们创建并且填充了一个请求对象。最后我们调用了存根上的方法，传入请求和回调函数。如果没有错误，就可以从我们的服务器从应答对象读取应答信息。

```
      console.log('Found feature called "' + feature.name + '" at ' +
          feature.location.latitude/COORD_FACTOR + ', ' +
          feature.location.longitude/COORD_FACTOR);
```

流式 RPC

现在来看看我们的流方法。如果你已经读过创建服务器(#server)，本节的一些内容看上去很熟悉——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。下面就是我们称作是服务器端的流方法 ListFeatures，它会返回地理的 Feature：

```
var call = client.listFeatures(rectangle);
  call.on('data', function(feature) {
      console.log('Found feature called "' + feature.name + '" at ' +
          feature.location.latitude/COORD_FACTOR + ', ' +
          feature.location.longitude/COORD_FACTOR);
  });
  call.on('end', function() {
```

```
  // The server has finished sending
});
call.on('status', function(status) {
  // process status
});
```

我们传给它一个请求并拿回一个 Readable 流对象，而不是给方法传入请求和回调函数。客户端可以使用 Readable 的 'data' 事件去读取服务器的应答。这个事件由每个 Feature 消息对象触发，知道没有更多的消息：'end' 事件揭示调用已经结束。最后，当服务器发送状态时，触发状态事件。

客户端的流方法 RecordRoute 的使用很相似，除了我们将一个回调函数传给方法，拿到一个 Writable 返回。

```
var call = client.recordRoute(function(error, stats) {
  if (error) {
    callback(error);
  }
  console.log('Finished trip with', stats.point_count, 'points');
  console.log('Passed', stats.feature_count, 'features');
  console.log('Travelled', stats.distance, 'meters');
  console.log('It took', stats.elapsed_time, 'seconds');
});
function pointSender(lat, lng) {
  return function(callback) {
    console.log('Visiting point ' + lat/COORD_FACTOR + ', ' +
        lng/COORD_FACTOR);
    call.write({
      latitude: lat,
      longitude: lng
    });
    _.delay(callback, _.random(500, 1500));
  };
}
var point_senders = [];
for (var i = 0; i < num_points; i++) {
  var rand_point = feature_list[_.random(0, feature_list.length - 1)];
  point_senders[i] = pointSender(rand_point.location.latitude,
                                 rand_point.location.longitude);
}
async.series(point_senders, function() {
  call.end();
});
```

一旦我们用 write() 将客户端请求写入到流的动作完成，我们需要在流上调用 end() 通知 gRPC 我

们已经完成写。如果状态是 OK，stats 对象会跟着服务器的响应被填充。

最后，让我们看看双向流式 RPC routeChat()。在这种场景下，我们将上下文传给一个方法，拿到一个可以用来读写消息的 Duplex 流对象的返回。

```
var call = client.routeChat();
```

这里读写的语法和我们客户端流以及服务器端流方法没有任何区别。虽然每一方都能按照写入时的顺序拿到另一方的消息，客户端和服务器端都可以以任意顺序读写——流操作起来是完全独立的。

来试试吧！

构建客户端和服务器：

```
$ npm install
```

运行服务器，它会监听50051端口：

```
$ node ./route_guide_server.js
```

在另一个终端运行客户端：

```
$ node ./route_guide_client.js
```

php 教程

gRPC 基础: PHP

本教程提供了 PHP 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 PHP 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 proto2 版本。

同时注意目前你只能用 PHP 创建 gRPC 服务的客户端——你可以从我们的其他教程中，如 Node.js(/docs/tutorials/basic/node.html)，找到如何创建 gRPC 服务器的例子。

这算不上是一个在 PHP 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?

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/examples/php/route_guide(https://github.com/grpc/grpc/tree/{{)。 要下载例子，通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
```

然后改变当前的目录到 examples/php/route_guide:

```
$ cd examples/php/route_guide
```

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。

你还需要安装生成客户端的接口代码的相关工具（以及一个用其他语言实现的服务器，出于测试的目的）——如果你还没有安装的话，请查看下面的设置指南这些设置指南 (https://github.com/grpc/homebrew-grpc)。

来试试吧！

为了使用例子应用，我们需要本地运行一个 gRPC 的服务器。让我们来编译运行，比如这个代码库中的 Node.js 服务器：

```
$ cd ../../node
$ npm install
$ cd route_guide
$ nodejs ./route_guide_server.js --db_path=route_guide_db.json
```

在一个不同的命令窗口运行 PHP 客户端：

```
$ ./run_route_guide_client.sh
```

下面的部分会指导你一步步的理解 proto 服务如何定义，如何从中生成一个客户端类库，以及如何使用类库创建一个应用。

定义服务

首先来看看我们使用的服务是如何定义的。gRPC 的 *service* 和它的方法 *request* 以及 *response* 类型使用了protocol buffers(https://developers.google.com/protocol-buffers/docs/overview)。你可以在 examples/protos/route_guide.proto(https://github.com/grpc/grpc/blob/{{}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，在 RouteGuide 服务中都有使用：

• 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，就像平常的远程过程调用调用一样。

```
// Obtains the feature at a given position.
rpc GetFeature(Point) returns (Feature) {}
```

• 一个 *应答流式 RPC* ， 客户端发送请求到服务器，拿到返回的应答消息流。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 *请求流式 RPC* ， 客户端发送一个消息序列到服务器。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;
}
```

生成客户端代码

可以用protoc-gen-php(https://github.com/datto/protobuf-php)工具从 proto 文件中生成 PHP 客户端存根实现。要安装这个工具，运行：

```
$ cd examples/php
$ php composer.phar install
$ cd vendor/datto/protobuf-php
$ gem install rake ronn
$ rake pear:package version=1.0
$ sudo pear install Protobuf-1.0.tgz
```

从 .php 文件 中生成客户端存根实现：

```
$ cd php/route_guide
$ protoc-gen-php -i . -o . ./route_guide.proto
```

php/route_guide 目录下会生成一个 `route_guide.php` 文件。你不需要修改这个文件。

要加载生成的客户端存根文件，只需要在你的 PHP 应用中 require 它：

```
require dirname(__FILE__) . '/route_guide.php';
```

文件包括：

• 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

• 一个名为 examplesRouteGuideClient 的类，可以让客户端调用定义在 RouteGuide 服务中的方法。

创建客户端

在这个部分，我们会使用 RouteGuide 服务去创建一个 PHP 客户端。在 examples/php/route_guide/route_guide_client.php(https://github.com/grpc/grpc/blob/{{} 可以看到我们完整的客户端例子代码。

构造一个客户端对象

要调用一个服务方法，我们首先需要创建一个客户端对象，生成的 RouteGuideClient 类的一个实例。该类的构造函数接受一个我们想连接的服务器地址和端口：

```
$client = new examples\RouteGuideClient('localhost:50051', []);
```

调用服务方法

现在让我们来看看如何调用服务方法。

简单 RPC

调用简单 RPC GetFeature 几乎是和调用本地的异步方法一样直观。

```
$point = new examples\Point();
$point->setLatitude(409146138);
$point->setLongitude(-746188906);
list($feature, $status) = $client->GetFeature($point)->wait();
```

如你所见，我们创建并且填充了一个请求对象，如一个 examplesPoint。然后，我们调用了存根上的方法，传入请求对象。如果没有错误，那么我们就可以从服务器从应答对象，如一个 examplesFeature，中读取应答信息。

```
print sprintf("Found %s \n  at %f, %f\n", $feature->getName(),
            $feature->getLocation()->getLatitude() / COORD_FACTOR ,
            $feature->getLocation()->getLongitude() / COORD_FACTOR);
```

流式 RPC

现在让我们看看流式方法。下面我们调用了服务器端流方法 ListFeatures，它会返回一个地理的 Feature 流：

```
$lo_point = new examples\Point();
$hi_point = new examples\Point();

$lo_point->setLatitude(400000000);
$lo_point->setLongitude(-750000000);
$hi_point->setLatitude(420000000);
$hi_point->setLongitude(-730000000);

$rectangle = new examples\Rectangle();
$rectangle->setLo($lo_point);
$rectangle->setHi($hi_point);

$call = $client->ListFeatures($rectangle);
// an iterator over the server streaming responses
$features = $call->responses();
foreach ($features as $feature) {
  // process each feature
} // the loop will end when the server indicates there is no more responses to be sent.
```

$call->responses() 方法调用返回一个迭代器。当服务器发送应答时，foreach 循环中会返回一个 $feature 对象，直到服务器表示没有更多的应答发送。

客户端流方法 RecordRoute 的使用很类似，除了我们为每个从客户端写入的每个点调用 $call->write($point) ，并拿到一个 examplesRouteSummary 返回。

```
$call = $client->RecordRoute();

for ($i = 0; $i < $num_points; $i++) {
  $point = new examples\Point();
  $point->setLatitude($lat);
  $point->setLongitude($long);
  $call->write($point);
}

list($route_summary， $status) = $call->wait();
```

最后，让我们看看双向流式 RPC routeChat()。在这个场景下，我们给方法传入一个上下文，拿到一个 BidiStreamingCall 流对象的返回，我们可以用这个流对象读写消息。

```
$call = $client->RouteChat();
```

从客户端写入消息：

```
foreach ($notes as $n) {
  $route_note = new examples\RouteNote();
  $call->write($route_note);
}
$call->writesDone();
```

从服务器读取消息：

```
while ($route_note_reply = $call->read()) {
  // process $route_note_reply
}
```

客户端和服务器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。

python 教程

gRPC 基础: Python

本教程提供了 Python 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 Python 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(/docs/index.html)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 Python 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。

例子代码和设置

教程的代码在这里 grpc/grpc/examples/python/route_guide(https://github.com/grpc/grpc/tree/{{)。 要下载例子，请通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
```

改变当前的目录到 examples/python/route_guide：

```
$ cd examples/python/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，查看下面的设置指南 Python快速开始指南(/docs/installation/python.html)。

定义服务

你的第一步(可以从概览(/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/{{)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 {
  // (Method definitions not shown)
```

```
}
```

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，在 RouteGuide 服务中都有使用：

• 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 *应答流式 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 *请求流式 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) {}
```

你的 .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

接下来你需要从 .proto 的服务定义中生成 gRPC 客户端和服务器端的接口。你可以通过 protocol buffer 的编译器 protoc 以及一个特殊的 gRPC Python 插件来完成。确保你已经安装了 protoc 并且按照 gRPC Python 插件installation instructions(https://github.com/grpc/grpc/blob/{{}操作。

安装了 protoc 和 gRPC Python 插件后，使用下面的命令来生成 Python 代码：

```
$ protoc -I ../../protos --python_out=. --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_python_plugin` ../../protos/route_guide.proto
```

注意我们在例子代码库中已经提供一个版本的生成代码，运行这个命令会重新生成对应的文件而不是创建一个全新的版本。生成的代码文件叫做 route*guide*pb2.py 并且包括：

- 定义在 route_guide.proto 中的消息类
- 定义在 route_guide.proto 中的服务的抽象类
- BetaRouteGuideServicer， 定义了 RouteGuide 服务实现的接口
- BetaRouteGuideStub，可以被客户端用来激活 RouteGuide RPC
- 应用使用的函数
- beta*create*RouteGuide_server，根据已有的 `BetaRouteGuideServicer` 对象创建一个 gRPC 服务器
- beta*create*RouteGuide_stub，客户端可以用来创建一个存根对象

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(#client) (当然你也可能发现它也很有意思)。

创建和运行 RouteGuide 服务可以分为两个部分：

- 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”的函数。
- 运行一个 gRPC 服务器，监听来自客户端的请求并传输服务的响应。

你可以从 examples/python/route_guide/route_guide_server.py(https://github.com/grpc/grpc/blob/{{}看到我们的 RouteGuide 服务器的例子。

实现RouteGuide

route*guide*server.py 有一个实现了生成的 route*guide*pb2.BetaRouteGuideServicer 接口的 RouteGuideServicer 类：

```
# RouteGuideServicer provides an implementation of the methods of the RouteGuide service.
class RouteGuideServicer(route_guide_pb2.BetaRouteGuideServicer):
```

RouteGuideServicer 实现了 RouteGuide 所有的服务方法：

简单 RPC

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。

```
def GetFeature(self, request, context):
  feature = get_feature(self.db, request)
  if feature is None:
    return route_guide_pb2.Feature(name="", location=request)
  else:
    return feature
```

方法传入了一个 route*guide*pb2.Point 的 RPC 请求，以及一个提供了 RPC-specific 信息，如超时限制，的 ServicerContext 对象。

应答流式 RPC

现在让我们看看下一个方法。ListFeatures 是一个应答流 RPC，它会发送多个 Feature 给客户端。

```
def ListFeatures(self, request, context):
  left = min(request.lo.longitude, request.hi.longitude)
  right = max(request.lo.longitude, request.hi.longitude)
  top = max(request.lo.latitude, request.hi.latitude)
  bottom = min(request.lo.latitude, request.hi.latitude)
  for feature in self.db:
    if (feature.location.longitude >= left and
        feature.location.longitude <= right and
        feature.location.latitude >= bottom and
        feature.location.latitude <= top):
      yield feature
```

这里的请求信息是 route*guide*pb2.Rectangle，客户端想从这里找到 Feature。该方法会产生0个或者更多的应答而不是单个的应答。

请求流式 RPC

请求流方法 RecordRoute 使用了一个请求值的 迭代器 (https://docs.python.org/2/library/stdtypes.html#iterator-types) 并返回了单个的应答值。

```
  def RecordRoute(self, request_iterator, context):
    point_count = 0
    feature_count = 0
    distance = 0.0
    prev_point = None

    start_time = time.time()
    for point in request_iterator:
      point_count += 1
      if get_feature(self.db, point):
        feature_count += 1
      if prev_point:
        distance += get_distance(prev_point, point)
      prev_point = point

    elapsed_time = time.time() - start_time
    return route_guide_pb2.RouteSummary(point_count=point_count,
                                        feature_count=feature_count,
                                        distance=int(distance),
                                        elapsed_time=int(elapsed_time))
```

双向流式 RPC

最后让我们来看看双向流方法 RouteChat。

```
  def RouteChat(self, request_iterator, context):
    prev_notes = []
    for new_note in request_iterator:
      for prev_note in prev_notes:
        if prev_note.location == new_note.location:
          yield prev_note
      prev_notes.append(new_note)
```

方法的语义是请求流方法和应答流方法的结合。它传入请求值的迭代器并且它本身也是应答值的迭代器。

启动服务器

一旦我们实现了所有的 RouteGuide 方法，下一步就是启动一个gRPC服务器，这样客户端才可以使用服务：

```
def serve():
  server = route_guide_pb2.beta_create_RouteGuide_server(RouteGuideServicer())
```

```
server.add_insecure_port('[::]:50051')
server.start()
```

因为 start() 不会阻塞，如果运行时你的代码没有其它的事情可做，你可能需要循环等待。

创建客户端

你可以在 examples/python/route_guide/route_guide_client.py(https://github.com/grpc/grpc/blob/{{)看到完整的例子代码。

创建一个存根

为了能调用服务的方法，我们得先创建一个 *存根*。

我们使用 .proto 中生成的 route*guide*pb2 模块的函数beta*create*RouteGuide_stub。

```
channel = implementations.insecure_channel('localhost', 50051)
stub = beta_create_RouteGuide_stub(channel)
```

返回的对象实现了定义在 BetaRouteGuideStub 接口中的所有对象。

调用服务方法

对于返回单个应答的 RPC 方法（"response-unary" 方法），gRPC Python 同时支持同步（阻塞）和异步（非阻塞）的控制流语义。对于应答流式 RPC 方法，调用会立即返回一个应答值的迭代器。调用迭代器的 next() 方法会阻塞，直到从迭代器产生的应答变得可用。

简单 RPC

同步调用简单 RPC GetFeature 几乎是和调用一个本地方法一样直观。RPC 调用等待服务器应答，它要么返回应答，要么引起异常：

```
feature = stub.GetFeature(point, timeout_in_seconds)
```

GetFeature 的异步调用很类似，但和在一个线程池里异步调用一个本地方法很像：

```
feature_future = stub.GetFeature.future(point, timeout_in_seconds)
feature = feature_future.result()
```

应答流 RPC

调用应答流 ListFeatures 和使用序列类型类似：

```
for feature in stub.ListFeatures(rectangle, timeout_in_seconds):
```

请求流 RPC

调用请求流 RecordRoute 和给一个本地方法传入序列类似。和前面的简单 RPC 一样，它也会返回单个应答，可以被同步或者异步调用：

```
route_summary = stub.RecordRoute(point_sequence, timeout_in_seconds)
```

```
route_summary_future = stub.RecordRoute.future(point_sequence,
timeout_in_seconds)
route_summary = route_summary_future.result()
```

双向流 RPC

调用双向流 RouteChat 是请求流和应答流语义的结合（这个场景是在服务器端）：

```
for received_route_note in stub.RouteChat(sent_routes, timeout_in_seconds):
```

来试试吧！

运行服务器，它会监听50051端口：

```
$ python route_guide_server.py
```

在另一个终端运行客户端：

```
$ python route_guide_client.py
```

ruby 教程

gRPC 基础: Ruby

本教程提供了 Python 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 Ruby 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(/docs/index.html)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 Ruby 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。

例子代码和设置

教程的代码在这里 grpc/grpc/examples/python/route_guide(https://github.com/grpc/grpc/tree/{{)。 要下载例子，通过运行下面的命令去克隆grpc代码库：

教程的代码在这里 grpc/grpc/examples/ruby/route_guide(https://github.com/grpc/grpc/tree/{{)。要下载例子，通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
```

改变当前的目录到 examples/ruby/route_guide:

```
$ cd examples/ruby/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，查看下面的设置指南 Ruby快速开始指南(/docs/installation/python.html)。

定义服务

我们的第一步(可以从概览(/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/{{ site.data.config.grpc*release*branch }}/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，在 RouteGuide 服务中都有使用：

• 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 Ruby 插件来完成。

如果你想自己运行，确保你已经安装了 protoc 并且先遵照 gRPC Ruby 插件installation instructions(https://github.com/grpc/grpc/blob/{{)。

一旦这些完成，就可以用下面的命令来生成 ruby 代码。

```
$ protoc -I ../../protos --ruby_out=lib --grpc_out=lib --plugin=protoc-gen-grpc=`which grpc_ruby_plugin` ../../protos/route_guide.proto
```

运行下面的命令可以在 lib 目录下重新生成下面的文件：

- lib/route_guide.pb 定义了一个模块 `Examples::RouteGuide`
- 包含了所有的填充，序列化和获取我们请求和响应消息类型的 protocol buffer 代码
- lib/route*guide*services.pb，继承了 Examples::RouteGuide 以及存根和服务类
- 在定义 RouteGuide 服务实现时用作基类的 Service 类
- 用来访问远程 RouteGuide的类 Stub

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(#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 examples/ruby/route_guide/route_guide_server.rb(https://github.com/grpc/grpc/blob/{{)看到 RouteGuide 服务器的例子。 现在让我们近距离瞧瞧它是如何工作的。

实现 RouteGuide

如你所见，我们的服务器有一个继承生成的 RouteGuide::Service 的 ServerImpl 类：

```
# ServerImpl provides an implementation of the RouteGuide service.
class ServerImpl < RouteGuide::Service
```

ServerImpl 实现了所有的服务方法。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，然后返回包含从数据库拿到的feature信息的 Feature。

```
  def get_feature(point, _call)
    name = @feature_db[{
      'longitude' => point.longitude,
      'latitude' => point.latitude }] || ''
    Feature.new(location: point, name: name)
  end
```

方法被传入一个 RPC 调用，也就是客户端的 Point protocol buffer 请求，并且返回一个 Feature protocol buffer。在方法中我们用适当的信息创建了 Feature，然后 return。

现在看看稍微复杂点的东西 —— 一个流式 RPC。ListFeatures 是一个服务器端流式 RPC，所以我们需要发回多个 Feature 给客户端。

```
# in ServerImpl

  def list_features(rectangle, _call)
    RectangleEnum.new(@feature_db, rectangle).each
  end
```

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，但是我们需要返回一个产生应答的Enumerator(http://ruby-doc.org//core-2.2.0/Enumerator.html)而不是一个简单应答。在方法中，我们使用帮助类 RectangleEnum 作为一个 Enumerator 的实现。

类似的，客户端流方法 record_route 使用一个Enumerable(http://ruby-doc.org//core-2.2.0/Enumerable.html)，但是这里是从调用对象获得，我们在先前的例子中略过了这点。`call.each*remote*read` 会依次产生由客户端发送的消息。

```
  call.each_remote_read do |point|
    ...
  end
```

最后，让我们来看看双向流式 RPC route_chat。

```
  def route_chat(notes)
    q = EnumeratorQueue.new(self)
    t = Thread.new do
      begin
        notes.each do |n|
...
      end
end
    q = EnumeratorQueue.new(self)
  ...
    return q.each_item
  end
```

这里方法接收一个Enumerable(http://ruby-doc.org//core-2.2.0/Enumerable.html)，但是也会返回一个产生应答的Enumerator(http://ruby-doc.org//core-2.2.0/Enumerator.html)。实现展示了如何设置它们，而后请求和应答才能并行处理。虽然每一端都会按照它们写入的顺序拿到另一端的消息，客户端和服务器都可以任意顺序读写——流的操作是互不依赖的。

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：

```
  s = GRPC::RpcServer.new
```

```
  s.add_http2_port(port, :this_port_is_insecure)
  logger.info("... running insecurely on #{port}")
  s.handle(ServerImpl.new(feature_db))
  s.run_till_terminated
```

如你所见，我们用 GRPC::RpcServer 构建和启动服务器。要做到这点，我们：

1. 用服务的实现类 ServerImpl 创建一个实例。
2. 使用生成器的 add*http2*port 方法指定地址以及期望客户端请求监听的端口。
3. 用 GRPC::RpcServer 注册我们的服务实现。
4. 调用 GRPC::RpcServer 的 run 去为我们的服务创建和启动 RPC 服务。

创建客户端

在这部分，我们将尝试为 RouteGuide 服务创建一个 Ruby 的客户端。你可以从 examples/ruby/route_guide/route_guide_client.rb(https://github.com/grpc/grpc/blob/{{}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，我们需要首先创建一个 *存根*：

我们使用从 .proto 生成的模块 RouteGuide 的 Stub 类。

```
  stub = RouteGuide::Stub.new('localhost:50051')
```

调用服务方法

现在让我们看看如何调用服务方法。注意， gRPC Ruby 只提供了 *阻塞/同步* 版本的方法：这意味着 RPC 调用需要等待服务器响应，并且要么返回应答，要么抛出异常。

简单 RPC

调用简单 RPC GetFeature 几乎和调用一个本地方法一样直接。

```
GET_FEATURE_POINTS = [
  Point.new(latitude:  409_146_138, longitude: -746_188_906),
  Point.new(latitude:  0, longitude: 0)
]
..
  GET_FEATURE_POINTS.each do |pt|
    resp = stub.get_feature(pt)
...
    p "- found '#{resp.name}' at #{pt.inspect}"
  end
```

我们创建和填充了一个请求 protocol buffer 对象（在这个场景下是 Point），并且创建了一个应答 protocol buffer 对象让服务器去填充。最后，我们调用存根上的方法，传入上下文，请求以及应答。如果方法返回 OK，那么我们就可以从服务器给我们的应答对象中读取应答信息。

流式 RPC

现在让我们看看流方法。如果你已经阅读了Creating the server(#server)部分，这些可能看上去很相似 —— 流方法在两端的实现很类似。这里我们调用了服务器端的流方法 list_features，它会返回 `Features` 的 `Enumerable`。

```
resps = stub.list_features(LIST_FEATURES_RECT)
resps.each do |r|
  p "- found '#{r.name}' at #{r.location.inspect}"
end
```

客户端流方法 record_route 也很类似，除了我们给服务器传入一个 `Enumerable`。

```
...
reqs = RandomRoute.new(features, points_on_route)
resp = stub.record_route(reqs.each, deadline)
...
```

最后，让我们看看双向流 RPC route_chat。在这个场景下，我们传入一个 `Enumerable`，拿到一个 `Enumerable` 返回。

```
resps = stub.route_chat(ROUTE_CHAT_NOTES)
resps.each { |r| p "received #{r.inspect}" }
```

虽然这个例子展现还不够好，每个 enumerable 之间都是互不依赖的 —— 客户端和服务器都可以以任意顺序读写 —— 流的操作是独立的。

来试试吧！

构建客户端和服务器：

```
$ # from examples/ruby
$ gem install bundler && bundle install
```

运行服务器，它会监听50051端口：

```
$ # from examples/ruby
$ bundle exec route_guide/route_guide_server.rb
../node/route_guide/route_guide_db.json &
```

在另一个终端运行客户端：

```
$ # from examples/ruby
```

```
$ bundle exec route_guide/route_guide_client.rb
../node/route_guide/route_guide_db.json &
```

objective-c 教程

gRPC 基础： Objective-C

本教程提供了 Objective-C 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 编译器生成客户端代码。
- 使用 gRPC 的 Objective-C API 为你的服务实现一个简单的客户端。

假设你已经熟悉了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 Objective-C 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?

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。

gRPC 和 proto3 特别适合移动客户端：gRPC 基于 HTTP/2 实现，相比 HTTP/1.1 更加节省网络带宽。序列化和解析 proto 的二进制格式效率高于 JSON，节省了 CPU 和 电池消耗。proto3 使用的运行时在 Google 以及被优化了多年，代码量极小。这对于 Objective-C 非常重要，因为语言的动态天性，编译器在优化不使用的代码时受到了限制。

例子的代码和设置

教程的代码在这里 grpc/grpc/examples/objective-c/route_guide(https://github.com/grpc/grpc/tree/{{)。 要下载例子，通过运行下面的命令去克隆grpc代码库：

```
$ git clone https://github.com/grpc/grpc.git
$ cd grpc
$ git submodule update --init
```

然后改变当前的目录到 examples/objective-c/route_guide:

```
$ cd examples/objective-c/route_guide
```

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。

你还需要安装 Cocoapods(https://cocoapods.org/#install) 以及相关的生成客户端类库的工具（以及一个用其他语言实现的服务器，出于测试的目的）。你可以根据这些设置指南(https://github.com/grpc/homebrew-grpc)来得到后面的内容。

来试试吧！

为了使用例子应用，我们需要本地运行一个 gRPC 的服务器。让我们来编译运行，比如这个代码库中的 C++ 服务器：

```
$ pushd ../../cpp/route_guide
$ make
$ ./route_guide_server &
$ popd
```

现在让 Cocoapods 为我们的 .proto 文件生成和安装客户端类库：

```
$ pod install
```

（这也许需要编译 OpenSSL， 如果电脑上没有 Cocoapods 的缓存，大概需要15分钟能够完成）。

最后，打开 Cocoapods 生成的 Xcode workspace，运行应用。你可以在 ViewControllers.m 中检查调用的代码，并且从 XCode 的日志窗口看到结果。

下面的部分会指导你一步步的理解 proto 服务如何定义，如何从中生成一个客户端类库，以及如何使用类库创建一个应用。

定义服务

首先来看看我们使用的服务是如何定义的。gRPC 的 *service* 和它的方法 *request* 以及 *response* 类型使用了protocol buffers(https://developers.google.com/protocol-buffers/docs/overview)。你可以在 examples/protos/route_guide.proto(https://github.com/grpc/grpc/blob/{{}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，在 RouteGuide 服务中都有使用：

• 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 *应答流式 RPC* ， 客户端发送请求到服务器，拿到返回的应答消息流。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 *请求流式 RPC* ， 客户端发送一个消息序列到服务器。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;
}
```

通过在文件开始处添加 objc*class*prefix 选项，你可以为生成的类指定一个前缀。比如：

```
option objc_class_prefix = "RTG";
```

生成客户端代码

接下来我们需要从 .proto 的服务定义中生成 gRPC 客户端接口。我们通过 protocol buffer 的编译

器 protoc 以及一个特殊的 gRPC Objective-C 插件来完成。

简单起见，我们提供一个 Podspec 文件(https://github.com/grpc/grpc/blob/{{} 帮你用合适的插件，输入，输出以及描述如何编译生成的文件去运行 protoc。你只需要在(examples/objective-c/route_guide)目录下运行：

```
$ pod install
```

这样会在这个例子的 XCode 项目中安装类库之前，运行：

```
$ protoc -I ../../protos --objc_out=Pods/RouteGuide --objcgrpc_out=Pods/RouteGuide
../../protos/route_guide.proto
```

运行这个命令会在 Pods/RouteGuide/ 目录下生成下面的文件：

- RouteGuide.pbobjc.h，声明生成的消息类的头文件。
- RouteGuide.pbobjc.m，包含你的消息类的实现。
- RouteGuide.pbrpc.h，声明生成的服务类的头文件。
- RouteGuide.pbrpc.m，包含了你的服务类的实现。

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
- 一个名为 RTGRouteGuide 的类，可以让客户端调用定义在 RouteGuide 服务中的方法。

你也可以使用提供的 Podspec 文件从任意其它的 proto 服务生成客户端代码；只需要替换名字（匹配文件名），版本以及其它metadata。

创建客户端应用

在这个部分，我们会使用 RouteGuide 服务去创建一个 Objective-C 客户端。在 examples/objective-c/route_guide/ViewControllers.m(https://github.com/grpc/grpc/blob/{{}可以看到我们完整的客户端例子代码。（注意：在你的应用中，出于维护和可读的原因，你不应该将所有的view controller放在一个文件中；这里这么做只是为了简化学习过程）。

构造一个服务对象

要调用一个服务方法，我们首先需要创建一个服务对象，生成的 RTGRouteGuide 类的一个实例。该类的初始化期望一个带有服务器地址以及我们期望连接端口的 NSString * ：

```
#import <GRPCClient/GRPCCall+Tests.h>
#import <RouteGuide/RouteGuide.pbrpc.h>

static NSString * const kHostAddress = @"localhost:50051";

...

[GRPCCall useInsecureConnectionsForHost:kHostAddress];
```

```
RTGRouteGuide *service = [[RTGRouteGuide alloc] initWithHost:kHostAddress];
```

注意，在构造我们的服务对象前，我们被告知 gRPC 类库在使用不安全的连接到 host:port。这是因为用来测试我们客户端的服务器没有使用 TLS(http://en.wikipedia.org/wiki/Transport_Layer_Security)。这么做没什么关系因为服务器只在本地开发环境运行。虽然最常见的场景是通过互联网连接支持 TLS 的 gRPC 服务器。对于那种场景，就不需要 useInsecureConnectionsForHost: 调用，如果没有指明，端口缺省为443。

调用服务方法

现在让我们来看看如何调用服务方法。如你所见，所有的这些方法都是异步的，所以你可以在应用的主线程中调用他们，不用担心 UI 被冻结或者 OS 杀掉你的应用。

简单 RPC

调用简单 RPC GetFeature 几乎是和调用 Cocoa 的任何异步方法一样直观。

```
RTGPoint *point = [RTGPoint message];
point.latitude = 40E7;
point.longitude = -74E7;

[service getFeatureWithRequest:point handler:^(RTGFeature *response, NSError *error)
{
  if (response) {
    // Successful response received
  } else {
    // RPC error
  }
}];
```

如你所见，我们创建并且填充了一个请求的 protocol buffer 对象（例子中为 RTGPoint ）。然后，我们调用了服务对象的方法，传入请求，处理应答（或者任何 RPC 错误）的块。如果 RPC 顺利完成，处理程序块和一个 nil 错误参数被调用，我们可以从服务器从应答参数中读取应答信息。如果，相反的，发生了 RPC 错误，处理程序块和一个 nil 错误参数被调用，我们可以从错误参数中读取到问题的细节。

```
NSLog(@"Found feature called %@ at %@.", response.name, response.location);
```

流式 RPC

现在让我们看看流式方法。下面是我们调用的应答流方法 ListFeatures，我们的客户端应用之后会收到一个地理位置的 RTGFeature 流：

```
[service listFeaturesWithRequest:rectangle handler:^(BOOL done, RTGFeature
```

```
*response, NSError *error) {
  if (response) {
    // Element of the stream of responses received
  } else if (error) {
    // RPC error; the stream is over.
  }
  if (done) {
    // The stream is over (all the responses were received, or an error occured). Do any
cleanup.
  }
}];
```

注意处理程序块的签名现在包括了一个 BOOL done 的参数。处理程序块可以被随意调用；只有在最后一次调用后 done 参数会被设置为 YES。一旦有错误发生，RPC 结束，处理程序块和参数 (YES, nil, error) 一起被调用。

请求流方法 RecordRoute 期望从客户端发来的 RTGPoint 流。这个流以遵循 GRXWriter 协议的对象形式被传入方法中。创建流的最简单的办法就是从 NSArray 对象中初始化一个：

```
#import <gRPC/GRXWriter+Immediate.h>

...

RTGPoint *point1 = [RTGPoint message];
point.latitude = 40E7;
point.longitude = -74E7;

RTGPoint *point2 = [RTGPoint message];
point.latitude = 40E7;
point.longitude = -74E7;

GRXWriter *locationsWriter = [GRXWriter writerWithContainer:@[point1, point2]];

[service recordRouteWithRequestsWriter:locationsWriter handler:^(RTGRouteSummary
*response, NSError *error) {
  if (response) {
    NSLog(@"Finished trip with %i points", response.pointCount);
    NSLog(@"Passed %i features", response.featureCount);
    NSLog(@"Travelled %i meters", response.distance);
    NSLog(@"It took %i seconds", response.elapsedTime);
  } else {
    NSLog(@"RPC error: %@", error);
  }
```

```
}];
```

GRXWriter 足够通用，可以允许异步流，feature 值流，甚至无限流。

最后，让我们看看双向流式 RPC RouteChat()。调用一个双向流式 RPC 的方式仅是如何调用请求流 RPC 和应答流 RPC 的组合。

```
[service routeChatWithRequestsWriter:notesWriter handler:^(BOOL done,
RTGRouteNote *note, NSError *error) {
  if (note) {
    NSLog(@"Got message %@ at %@", note.message, note.location);
  } else if (error) {
    NSLog(@"RPC error: %@", error);
  }
  if (done) {
    NSLog(@"Chat ended.");
  }
}];
```

处理程序块的语义以及这里的 GRXWriter 参数和我们的请求流和应答流方法一致。虽然客户端和服务器获取对方信息的顺序和信息被写入的顺序一致，读写流的操作是完全独立的。

异步基础: C++

异步基础: C++

本教程介绍如何使用 C++ 的 gRPC 异步/非阻塞 API 去实现简单的服务器和客户端。假设你已经熟悉实现同步 gRPC 代码，如gRPC 基础: C++(/docs/tutorials/basic/c.html)所描述的。本教程中的例子基本来自我们在overview(/docs/index.html)中使用的Greeter 例子(https://github.com/grpc/grpc/tree/{{}。你可以在grpc/examples/cpp/helloworld(https://github.com/grpc/grpc/tree/{{}找到安装指南。

概览

gRPC 的异步操作使用CompletionQueue(http://www.grpc.io/grpc/cpp/classgrpc_1_1_completion_queue.html)。基本工作流如下：

- 在 RPC 调用上绑定一个 CompletionQueue
- 做一些事情如读取或者写入，以唯一的 voide* 标签展示
- 调用 CompletionQueue::Next 去等待操作结束。如果标签出现，表示对应的操作已经完成。

异步客户端

要使用一个异步的客户端调用远程方法，你首先得创建一个频道和存根，如你在同步客户端

(https://github.com/grpc/grpc/blob/{{}中所作的那样。一旦有了存根，你就可以通过下面的方式来做异步调用：

• 初始化 RPC 并为之创建句柄。将 RPC 绑定到一个 CompletionQueue。

```
CompletionQueue cq;
std::unique_ptr<ClientAsyncResponseReader<HelloReply> > rpc(
    stub_->AsyncSayHello(&context, request, &cq));
```

• 用一个唯一的标签，寻求回答和最终的状态

```
Status status;
rpc->Finish(&reply, &status, (void*)1);
```

• 等待完成队列返回下一个标签。当标签被传入对应的 Finish() 调用时，回答和状态就可以被返回了。

```
void* got_tag;
bool ok = false;
cq.Next(&got_tag, &ok);
if (ok && got_tag == (void*)1) {
  // check reply and status
}
```

你可以在这里greeter_async_client.cc(https://github.com/grpc/grpc/blob/{{}看到完整的客户端例子。

异步服务器

服务器实现请求一个带有标签的 RPC 调用，然后等待完成队列返回标签。异步处理 RPC 的基本工作流如下：

• 构建一个服务器导出异步服务

```
helloworld::Greeter::AsyncService service;
ServerBuilder builder;
builder.AddListeningPort("0.0.0.0:50051", InsecureServerCredentials());
builder.RegisterAsyncService(&service);
auto cq = builder.AddCompletionQueue();
auto server = builder.BuildAndStart();
```

• 请求一个 RPC 提供唯一的标签

```
ServerContext context;
HelloRequest request;
ServerAsyncResponseWriter<HelloReply> responder;
service.RequestSayHello(&context, &request, &responder, &cq, &cq, (void*)1);
```

• 等待完成队列返回标签。当取到标签时，上下文，请求和应答器都已经准备就绪。

```
HelloReply reply;
Status status;
void* got_tag;
bool ok = false;
cq.Next(&got_tag, &ok);
if (ok && got_tag == (void*)1) {
  // set reply and status
  responder.Finish(reply, status, (void*)2);
}
```

• 等待完成队列返回标签。标签返回时 RPC 结束。

```
void* got_tag;
bool ok = false;
cq.Next(&got_tag, &ok);
if (ok && got_tag == (void*)2) {
  // clean up
}
```

然而，这个基本的工作流没有考虑服务器并发处理多个请求。要解决这个问题，我们的完成异步服务器例子使用了 CallData 对象去维护每个 RPC 的状态，并且使用这个对象的地址作为调用的唯一标签。

```
class CallData {
 public:
  // Take in the "service" instance (in this case representing an asynchronous
  // server) and the completion queue "cq" used for asynchronous communication
  // with the gRPC runtime.
  CallData(Greeter::AsyncService* service, ServerCompletionQueue* cq)
      : service_(service), cq_(cq), responder_(&ctx_), status_(CREATE) {
    // Invoke the serving logic right away.
    Proceed();
  }

  void Proceed() {
    if (status_ == CREATE) {
      // As part of the initial CREATE state, we *request* that the system
      // start processing SayHello requests. In this request, "this" acts are
      // the tag uniquely identifying the request (so that different CallData
      // instances can serve different requests concurrently), in this case
      // the memory address of this CallData instance.
      service_->RequestSayHello(&ctx_, &request_, &responder_, cq_, cq_,
                                this);
```

```
      // Make this instance progress to the PROCESS state.
      status_ = PROCESS;
    } else if (status_ == PROCESS) {
      // Spawn a new CallData instance to serve new clients while we process
      // the one for this CallData. The instance will deallocate itself as
      // part of its FINISH state.
      new CallData(service_, cq_);

      // The actual processing.
      std::string prefix("Hello ");
      reply_.set_message(prefix + request_.name());

      // And we are done! Let the gRPC runtime know we've finished, using the
      // memory address of this instance as the uniquely identifying tag for
      // the event.
      responder_.Finish(reply_, Status::OK, this);
      status_ = FINISH;
    } else {
      GPR_ASSERT(status_ == FINISH);
      // Once in the FINISH state, deallocate ourselves (CallData).
      delete this;
    }
  }
```

简单起见，服务器对于所有的事件只使用了一个完成队列，并且在 HandleRpcs 中运行了一个主循环去查询队列：

```
  void HandleRpcs() {
    // Spawn a new CallData instance to serve new clients.
    new CallData(&service_, cq_.get());
    void* tag;  // uniquely identifies a request.
    bool ok;
    while (true) {
      // Block waiting to read the next event from the completion queue. The
      // event is uniquely identified by its tag, which in this case is the
      // memory address of a CallData instance.
      cq_->Next(&tag, &ok);
      GPR_ASSERT(ok);
      static_cast<CallData*>(tag)->Proceed();
    }
  }
```

你可以在greeter_async_server.cc(https://github.com/grpc/grpc/blob/{{}看到完整

的服务器例子。

在 gRPC 上使用 OAuth2: Objective-C

在 gRPC 上使用 OAuth2: Objective-C

这个例子展示了如何在 gRPC 上使用 OAuth2 代表用户发起身份验证 API 调用。通过它你还会学到如何

使用 Objective-C gRPC API 去：

- 在 RPC 启动前初始化和配置一个远程调用对象。
- 在一个调用上设置请求的元数据元素，语义上等同于 HTTP 的请求头部。
- 从调用上读取应答的元数据，等同于 HTTP 应答的头和尾。

假设你知道如何使用 Objective-C 的客户端类库去发起 gRPC 调用，如在gRPC 基础: Objective-C(/docs/tutorials/basic/objective-c.html)和概览(/docs/index.html)中介绍的那样，以及熟悉 OAuth2 的概念如 *access token*。

例子代码和设置

我们教程的例子代码在这里gprc/examples/objective-c/auth_sample(https://github.com/grpc/grpc/tree/master/examples/objective-c/auth_sample)。要下载这个例子，通过运行下面的命令克隆代码库：

```
$ git clone https://github.com/grpc/grpc.git
$ cd grpc
$ git submodule update --init
```

然后切换目录到 examples/objective-c/auth_sample：

```
$ cd examples/objective-c/auth_sample
```

我们的例子是一个有两个视图的简单应用。第一个视图让用户使用 Google 的iOS 登陆类库(https://developers.google.com/identity/sign-in/ios/)的 OAuth2 工作流去登陆和登出。（例子中用到了 Google 的类库，是因为我们要调用的测试 gRPC 服务需要 Google 账号身份，但是 gRPC 和 Objective-C 客户端类库都没有绑定任何特定的 OAuth2 提供商）。第二个视图使用第一个视图获得的 access token 向测试服务器发起 gRPC 请求。

注意：OAuth2 类库需要应用注册并且从身份提供者获得一个 ID（在例子应用中是 Google）。应用的 XCode 项目配置使用那个 ID，所以你不应该拷贝这个工程”当做是“自己的应用：这会导致你的应用程序作为 "gRPC-AuthSample" 在同意界面被确定，并且不能访问真正的 Google 服务。相反，根据指南(https://developers.google.com/identity/sign-in/ios/)去配置你自己的 XCode 工程。

在使用其它的 Objective-C 例子时，你应该已经安装了 Cocoapods(https://cocoapods.org/#install)，还有相关的生成客户端类库代码的工具。你也可以按照这些设置指南(https://github.com/grpc/homebrew-grpc)得到后者。

来试试吧！

要试试例子应用呢，首先为我们的 .proto 文件使用 Cocoapods 生成和安装客户端类库：

```
$ pod install
```

（这也许需要编译 OpenSSL，如果你的电脑上没有 Cocoapods 的缓存，这也许要花上 15 分钟左右）。

最后，打开 Cocoapods 创建的 XCode workspace，运行应用。

第一个视图 SelectUserViewController.h/m，要求你用 Google 账户登录，授于 "gRPC-AuthSample" 应用如下的权限：

- 查看你的邮件地址。
- 查看你基本的档案信息。
- "访问 Zoo 服务的测试范围"。

最后一个权限，对应的范围是 https://www.googleapis.com/auth/xapi.zoo，并没有给予任何正式的能力：这只是用来测试。你随时可以登出。

第二个视图 MakeRPCViewController.h/m，向位于 https://grpc-test.sandbox.google.com 的测试服务器发起 gRPC 请求，包括 access token 。测试服务器只是检验了 token，而后将它所属的用户以及授予访问的范围写入到应答中。（客户端应用已经知道这两个值；这是验证所有事情都按照我们期望方式进行的一种方法）。

下一个部分的指南会一步步指导你如何实行 MakeRPCViewController 中的 gRPC 调用。你可以在 MakeRPCViewController.m(https://github.com/grpc/grpc/blob/master/examples/objective-c/auth_sample/MakeRPCViewController.m)看到完整例子的代码。

创建一个 RPC 对象

另一个基本的教程展示如何通过调用生成的客户端对象中的异步方法来激活一个 RPC。但是，发起身份验证的调用需要你去初始化一个代表 RPC 的对象，在发起网络请求 *前* 配置好它。首先让我们看看如何创建 RPC 对象。

假设你的 proto 服务定义如下：

```
option objc_class_prefix = "AUTH";

service TestService {
  rpc UnaryCall(Request) returns (Response);
}
```

为了 AUTHTestService 类生成的一个 unaryCallWithRequest:handler: 方法，你应该已经很熟悉了：

```
[client unaryCallWithRequest:request handler:^(AUTHResponse *response, NSError
*error) {
  ...
}];
```

此外，一个 RPCToUnaryCallWithRequest:handler: 被生成，它会返回一个还没有开始的 RPC 对象：

```
#import <ProtoRPC/ProtoRPC.h>

ProtoRPC *call =
    [client RPCToUnaryCallWithRequest:request handler:^(AUTHResponse *response,
NSError *error) {
      ...
    }];
```

你可以像这样在任何以后的时间开始这个对象代表的 RPC：

```
[call start];
```

设置请求元数据：：有一个 access token 的身份验证头

现在让我们看看如何配置 RPC 对象上的一些设置。ProtoRPC 有个 requestHeaders 属性（从 GRPCCall 继承）定义如下：

```
@property(atomic, readonly) id<GRPCRequestHeaders> requestHeaders
```

你可以把 GRPCRequestHeaders 协议等同于 NSMutableDictionary 类。设置元数据键值的词典的元素意味着这个元数据在调用开始后将会被发送。gRPC 元数据是关于客户端发往服务器调用的信息片（反之亦然）。它们以键值对的形式存在，并且对于 gRPC 本身基本不透明。

方便起见，属性通过空的 NSMutableDictionary 初始化，以便请求元数据元素可以像下面一样设置：

```
call.requestHeaders[@"My-Header"] = @"Value for this header";
call.requestHeaders[@"Another-Header"] = @"Its value";
```

元数据的典型使用是验证细节，像我们例子中一样。如果你已经有了 access token，OAuth2 指定它以下面的格式发送：

```
call.requestHeaders[@"Authorization"] = [@"Bearer "
stringByAppendingString:accessToken];
```

拿到应答元数据：验证挑战头

ProtoRPC 类也继承了一对属性，responseHeaders 和 responseTrailers， 类似于我们刚刚看的请求元数据，不过却是由服务器向客户端发回的。它们的定义如下：

```
@property(atomic, readonly) NSDictionary *responseHeaders;
@property(atomic, readonly) NSDictionary *responseTrailers;
```

在 OAuth2 中，如果验证出错，服务器会返回一个挑战头。它通过 RPC 的应答头返回。要访问这

个，如我们例子中的错误处理的代码，你可以这么写：

```
call.responseHeaders[@"www-authenticate"]
```

注意，gRPC 的元数据元素可以映射到 HTTP/2 的头（或者尾），应答元数据的键永远是小写的 ASCII 码字符串。

许多应答元数据的使用场景都涉及如何拿到关于一个 RPC 错误的更多细节。简单起见，当 RPC 的处理块传入一个 NSError 实例时，应答头和尾词典也可以这种方式访问：

```
error.userInfo[kGRPCHeadersKey] == call.responseHeaders
error.userInfo[kGRPCTrailersKey] == call.responseTrailers
```