Go-gRPC 实践指南

书栈(BookStack.CN)

目　录

致谢

当前文档 《Go-gRPC 实践指南》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-07-28。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：jergoo https://github.com/jergoo/go-grpc-practice-guide

文档地址：http://www.bookstack.cn/books/go-grpc

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Summary

Go-gRPC 入门实践

本项目是segmentfault上连载的 **Golang gRPC**实践 系列文章的重新整理版本，简单调整了目录结构，完善了原有的部分内容并重新整理了示例代码，计划介绍更多的gRPC实践及生态的用法和多语言支持。内容中的示例代码都放在项目 **go-grpc-example** 内，如无特殊说明，内容的源码的目录说说明将以此项目为根目录。

目录

segmentfault连载(暂未更新)

- Golang gRPC实践 连载一 gRPC介绍与安装
- Golang gRPC实践 连载二 Hello gRPC
- Golang gRPC实践 连载三 Protobuf语法
- Golang gRPC实践 连载四 gRPC认证
- Golang gRPC实践 连载五 拦截器 Interceptor
- Golang gRPC实践 连载六 内置Trace
- Golang gRPC实践 连载七 http转换

安装

环境:

- 操作系统: macOS 10.12.5 / Ubuntu 16.04
- golang 版本: 1.8.3
- protobuf 版本: 3.3.2
- grpc-go 版本: 1.4.2

准备工作

- Linux 安装 linuxbrew
- macOS 安装 homebrew

protobuf

项目地址: google/protobuf

这里直接使用 `brew` 工具安装

```
1.  $ brew install protobuf
```

`brew` 默认会安装最新版本，执行 `protoc` 命令查看当前版本:

```
1.  $ protoc --version
2.  libprotoc 3.3.2
```

grpc-go

项目地址: grpc-go

- 要求golang版本 >= 1.6

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

golang protobuf

项目地址：golang/protobuf

- 要求golang版本 > 1.4
- 使用前要求安装protobuf编译器

```
$ go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
```

安装 `protoc-gen-go` 到 `$GOPATH/bin` 。 注意：该目录必须在系统的环境变量 `$PATH` 中。

如果一路没有问题的话，到此为止，需要的环境都安装好了😀。

编译器使用

使用 `protoc` 命令编译 `.proto` 文件,不同语言支持需要指定输出参数，如：

```
$ protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --
python_out=DST_DIR --go_out=DST_DIR --ruby_out=DST_DIR --javanano_out=DST_DIR -
-objc_out=DST_DIR --csharp_out=DST_DIR path/to/file.proto
```

这里详细介绍golang的编译姿势：

- `-I` 参数：指定import路径，可以指定多个 `-I` 参数，编译时按顺序查找，不指定时默认查找当前目录
- `--go_out` ：golang编译支持，支持以下参数
  - `plugins=plugin1+plugin2` - 指定插件，目前只支持grpc，即：`plugins=grpc`
  - `M` 参数 - 指定导入的.proto文件路径编译后对应的golang包名(不指定本参数默认就是 `.proto` 文件中 `import` 语句的路径)
  - `import_prefix=xxx` - 为所有 `import` 路径添加前缀，主要用于编译子目录内的多个proto文件，这个参数按理说很有用，尤其适用替代一些情况时的 `M` 参数，但是实际使用时有个蛋疼的问题导致并不能达到我们预想的效果，自己尝试看看吧
  - `import_path=foo/bar` - 用于指定未声明 `package` 或 `go_package` 的文件的包名，最右面的斜线前的字符会被忽略
  - 末尾 `:编译文件路径 .proto文件路径(支持通配符)`

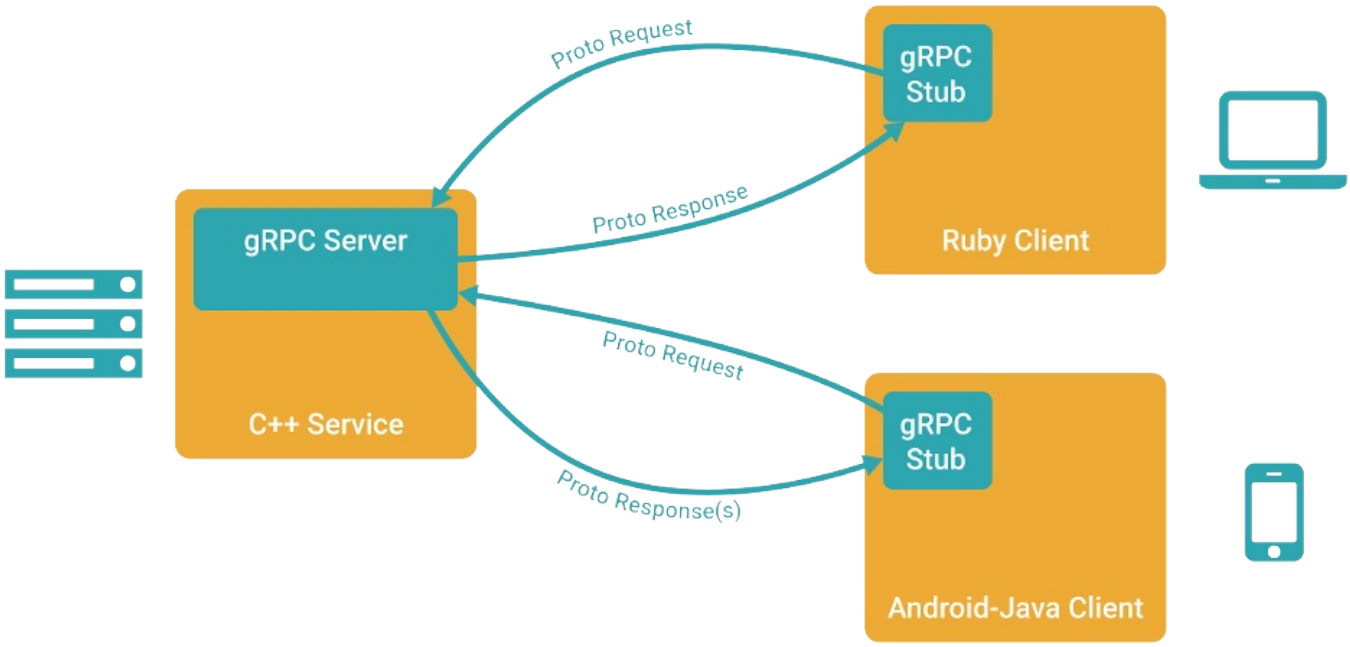
完整示例：

```
$ protoc -I . --
go_out=plugins=grpc,Mfoo/bar.proto=bar,import_prefix=foo/,import_path=foo/bar:.
./*.proto
```

gRPC简介

gRPC 是一个高性能、开源、通用的RPC框架，由Google推出，基于HTTP2协议标准设计开发，默认采用Protocol Buffers数据序列化协议，支持多种开发语言。gRPC提供了一种简单的方法来精确的定义服务，并且为客户端和服务端自动生成可靠的功能库。

在gRPC客户端可以直接调用不同服务器上的远程程序，使用姿势看起来就像调用本地程序一样，很容易去构建分布式应用和服务。和很多RPC系统一样，服务端负责实现定义好的接口并处理客户端的请求，客户端根据接口描述直接调用需要的服务。客户端和服务端可以分别使用gRPC支持的不同语言实现。



主要特性

- 强大的IDL

  gRPC使用ProtoBuf来定义服务，ProtoBuf是由Google开发的一种数据序列化协议（类似于XML、JSON、hessian）。ProtoBuf能够将数据进行序列化，并广泛应用在数据存储、通信协议等方面。

- 多语言支持

  gRPC支持多种语言，并能够基于语言自动生成客户端和服务端功能库。目前已提供了C版本

grpc、Java版本grpc-java 和 Go版本grpc-go，其它语言的版本正在积极开发中，其中，grpc支持C、C++、Node.js、Python、Ruby、Objective-C、PHP和C#等语言，grpc-java已经支持Android开发。

- HTTP2

  gRPC基于HTTP2标准设计，所以相对于其他RPC框架，gRPC带来了更多强大功能，如双向流、头部压缩、多复用请求等。这些功能给移动设备带来重大益处，如节省带宽、降低TCP链接次数、节省CPU使用和延长电池寿命等。同时，gRPC还能够提高了云端服务和Web应用的性能。gRPC既能够在客户端应用，也能够在服务器端应用，从而以透明的方式实现客户端和服务器端的通信和简化通信系统的构建。

  更多介绍请查看官方网站

Protobuf⇢Go转换

这里使用一个测试文件对照说明常用结构的protobuf到golang的转换。只说明关键部分代码，详细内容请查看完整文件。示例文件在 `proto/test` 目录下。

Package

在proto文件中使用 `package` 关键字声明包名，默认转换成go中的包名与此一致，如果需要指定不一样的包名，可以使用 `go_package` 选项：

```
1.  package test;
2.  option go_package="test";
```

Message

proto中的 `message` 对应go中的 `struct` ，全部使用驼峰命名规则。嵌套定义的 `message` ， `enum` 转换为go之后，名称变为 `Parent_Child` 结构。

示例proto:

```
1.  // Test 测试
2.  message Test {
3.      int32 age = 1;
4.      int64 count = 2;
5.      double money = 3;
6.      float score = 4;
7.      string name = 5;
8.      bool fat = 6;
9.      bytes char = 7;
10.     // Status 枚举状态
11.     enum Status {
12.         OK = 0;
13.         FAIL = 1;
14.     }
15.     Status status = 8;
16.     // Child 子结构
17.     message Child {
18.         string sex = 1;
19.     }
```

```
    Child child = 9;
    map<string, string> dict = 10;
}
```

转换结果：

```
// Status 枚举状态
type Test_Status int32

const (
    Test_OK   Test_Status = 0
    Test_FAIL Test_Status = 1
)

// Test 测试
type Test struct {
    Age    int32       `protobuf:"varint,1,opt,name=age" json:"age,omitempty"`
    Count  int64       `protobuf:"varint,2,opt,name=count"
json:"count,omitempty"`
    Money  float64     `protobuf:"fixed64,3,opt,name=money"
json:"money,omitempty"`
    Score  float32     `protobuf:"fixed32,4,opt,name=score"
json:"score,omitempty"`
    Name   string      `protobuf:"bytes,5,opt,name=name" json:"name,omitempty"`
    Fat    bool        `protobuf:"varint,6,opt,name=fat" json:"fat,omitempty"`
    Char   []byte      `protobuf:"bytes,7,opt,name=char,proto3"
json:"char,omitempty"`
    Status Test_Status
`protobuf:"varint,8,opt,name=status,enum=test.Test_Status"
json:"status,omitempty"`
    Child  *Test_Child `protobuf:"bytes,9,opt,name=child"
json:"child,omitempty"`
    Dict   map[string]string `protobuf:"bytes,10,rep,name=dict"
json:"dict,omitempty" protobuf_key:"bytes,1,opt,name=key"
protobuf_val:"bytes,2,opt,name=value"`
}

// Child 子结构
type Test_Child struct {
    Sex string `protobuf:"bytes,1,opt,name=sex" json:"sex,omitempty"`
}
```

除了会生成对应的结构外，还会有些工具方法，如字段的getter：

```go
func (m *Test) GetAge() int32 {
    if m != nil {
        return m.Age
    }
    return 0
}
```

枚举类型会生成对应名称的常量，同时会有两个map方便使用：

```go
var Test_Status_name = map[int32]string{
    0: "OK",
    1: "FAIL",
}
var Test_Status_value = map[string]int32{
    "OK":   0,
    "FAIL": 1,
}
```

Service

定义一个简单的Service， `TestService` 有一个方法 `Test` ，接收一个 `Request` 参数，返回 `Response` ：

```
// TestService 测试服务
service TestService {
    // Test 测试方法
    rpc Test(Request) returns (Response) {};
}

// Request 请求结构
message Request {
    string name = 1;
}

// Response 响应结构
message Response {
    string message = 1;
}
```

转换结果：

```
// 客户端接口
type TestServiceClient interface {
    // Test 测试方法
    Test(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Response, error)
}

// 服务端接口
type TestServiceServer interface {
    // Test 测试方法
    Test(context.Context, *Request) (*Response, error)
}
```

生成的go代码中包含该Service定义的接口，客户端接口已经自动实现了，直接供客户端使用者调用，服务端接口需要由服务提供方实现。

Protobuf语法

gRPC推荐使用proto3，这里只介绍常用语法，按照官方文档的结构翻译，英文水平有限，复杂的部分果断放弃，更多高级使用姿势请参考官方文档

> 建议初学者不要太刻意的记这里的语法，简单看一遍了解就好，使用过程有问题再回来查看。

Message定义

一个 `message` 类型定义描述了一个请求或响应的消息格式，可以包含多种类型字段。

例如定义一个搜索请求的消息格式 `SearchRequest` ，每个请求包含查询字符串、页码、每页数目。每个字段声明以分号结尾。

```
1.  syntax = "proto3";
2.
3.  message SearchRequest {
4.      string query = 1;
5.      int32  page_number = 2;
6.      int32  result_per_page = 3;
7.  }
```

> 首行要求明确声明使用的**protobuf**版本为 **`proto3`** ，如果不声明，编译器默认使用 **`proto2`** 。本声明必须在文件的首行。

一个 `.proto` 文件中可以定义多个message，一般用于同时定义多个相关的message，例如在同一个.proto文件中同时定义搜索请求和响应消息：

```
1.  syntax = "proto3";
2.
3.  message SearchRequest {
4.      string query = 1;
5.      int32  page_number = 2;
6.      int32  result_per_page = 3;
7.  }
8.
9.  message SearchResponse {
10.     ...
11. }
```

字段类型声明

所有的字段需要前置声明数据类型，上面的示例指定了两个数值类型和一个字符串类型。除了基本的标量类型还有复合类型，如枚举、map、数组、甚至其它message类型等。后面会依次说明。

分配Tags

消息的定义中，每个字段都有一个唯一的数值标签。这些标签用于标识该字段在消息中的二进制格式，使用中的类型不应该随意改动。其中，[1-15]内的标识在编码时占用一个字节，包含标识和字段类型。[16-2047]之间的标识符占用2个字节。建议为频繁出现的消息元素分配[1-15]间的标签。如果考虑到以后可能或扩展频繁元素，可以预留一些。

最小的标识符可以从1开始，最大到2^{29} - 1，或536,870,911。不可以使用[19000－19999]之间的标识符， Protobuf协议实现中预留了这些标识符。在.proto文件中使用这些预留标识号，编译时会报错。

字段规则

- 单数形态：一个message内同名单数形态的字段不能超过一个
- repeated：前置 `repeated` 关键词，声明该字段为数组类型
- `proto3` 不支持 `proto2` 中的 `required` 和 `optional` 关键字

添加注释

向 `.proto` 文件中添加注释，支持C/C++风格双斜线 `//` 单行注释。

```
1.  syntax = "proto3";              // 协议版本声明
2.
3.  // SearchRequest 搜索请求消息
4.  message SearchRequest {
5.      string query = 1;           // 查询字符串
6.      int32  page_number = 2;     // 页码
7.      int32  result_per_page = 3; // 每页条数
8.  }
```

保留字段名与Tag

可以使用 `reserved` 关键字指定保留字段名和标签。

```
1.  message Foo {
2.      reserved 2, 15, 9 to 11;
3.      reserved "foo", "bar";
```

```
4. }
```

> 注意，不能在一个 **reserved** 声明中混合字段名和标签。

.proto 文件编译结果

当使用protocol buffer编译器运行 .proto 文件时，编译器将生成所选语言的代码，用于使用在 .proto 文件中定义的消息类型、服务接口约定等。不同语言生成的代码格式不同：

- C++: 每个 .proto 文件生成一个 .h 文件和一个 .cc 文件，每个消息类型对应一个类
- Java: 生成一个 .java 文件，同样每个消息对应一个类，同时还有一个特殊的 Builder 类用于创建消息接口
- Python: 姿势不太一样，每个 .proto 文件中的消息类型生成一个含有静态描述符的模块，该模块与一个元类*metaclass*在运行时创建需要的Python数据访问类
- Go: 生成一个 .pb.go 文件，每个消息类型对应一个结构体
- Ruby: 生成一个 .rb 文件的Ruby模块，包含所有消息类型
- JavaNano: 类似Java，但不包含 Builder 类
- Objective-C: 每个 .proto 文件生成一个 pbobjc.h 和一个 pbobjc.m 文件
- C#: 生成 .cs 文件包含，每个消息类型对应一个类

各种语言的更多的使用方法请参考官方API文档

基本数据类型

.proto	C++	Java	Python	Go	Ruby	
double	double	double	float	float64	Float	d
float	float	float	float	float32	Float	f
int32	int32	int	int	int32	Fixnum or Bignum	i
int64	int64	long	ing/long[3]	int64	Bignum	l
uint32	uint32	int[1]	int/long[3]	uint32	Fixnum or Bignum	u
uint64	uint64	long[1]	int/long[3]	uint64	Bignum	u
sint32	int32	int	intj	int32	Fixnum or Bignum	i
sint64	int64	long	int/long[3]	int64	Bignum	l
fixed32	uint32	int[1]	int	uint32	Fixnum or Bignum	u
fixed64	uint64	long[1]	int/long[3]	uint64	Bignum	u
sfixed32	int32	int	int	int32	Fixnum or Bignum	i
sfixed64	int64	long	int/long[3]	int64	Bignum	l
bool	bool	boolean	boolean	bool	TrueClass/FalseClass	b

string	string	String	str/unicode[4]	string	String(UTF-8)	
bytes	string	ByteString	str	[]byte	String(ASCII-8BIT)	

关于这些类型在序列化时的编码规则请参考 Protocol Buffer Encoding.

[1] java

[2] all

[3] 64

[4] Python

默认值

- 字符串类型默认为空字符串
- 字节类型默认为空字节
- 布尔类型默认false
- 数值类型默认为0值
- enums类型默认为第一个定义的枚举值，必须是0

针对不同语言的默认值的具体行为参考 generated code guide

枚举(Enum)

当定义一个message时，想要一个字段只能是一个预定义好的值列表内的一个值，就需要用到enum类型了。

示例：这里定义一个名为 `Corpus` 的enum类型值，并且指定一个字段为 `Corpus` 类型

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
  // 定义enum类型
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
```

```
    VIDEO = 6;
  }
  Corpus corpus = 4; // 使用Corpus作为字段类型
}
```

> 注意：每个**enum**定义的第一个元素值必须是**0**

还可以通过给不同的enum元素赋相同的值来定义别名，要求设置 `allow_alias` 选项的值为 `true` ，否则会报错。

```
// 正确示例
enum EnumAllowingAlias {
  option allow_alias = true; // 开启allow_alias选项
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 1;  // RUNNING和STARTED互为别名
}

// 错误示例
enum EnumNotAllowingAlias {
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 1;  // 未开启allow_alias选项，编译会报错
}
```

> enum类型值同样支持文件级定义和message内定义，引用方式与message嵌套一致

使用其它Message

可以使用其它message类型作为字段类型。

例如，在 `SearchResponse` 中包含 `Result` 类型的消息，可以在相同的 `.proto` 文件中定义 `Result` 消息类型，然后在 `SearchResponse` 中指定字段类型为 `Result` ：

```
message SearchResponse {
    repeated Result results = 1;
}

message Result {
    string url = 1;
    string title = 2;
```

```
    repeated string snippets = 3;
}
```

导入定义(import)

上面的例子中， `Result` 类型和 `SearchResponse` 类型定义在同一个 `.proto` 文件中，我们还可以使用import语句导入使用其它描述文件中声明的类型：

```
import "others.proto";
```

默认情况，只能使用直接导入的 `.proto` 文件内的定义。但是有时候需要移动 `.proto` 文件到其它位置，为了避免更新所有相关文件，可以在原位置放置一个模型.proto文件，使用 `public` 关键字，转发所有对新文件内容的引用，例如：

```
// new.proto
// 所有新的定义在这里
```

```
// old.proto
// 客户端导入的原来的proto文件
import public "new.proto";
import "other.proto";
```

```
// client.proto
import "old.proto";
// 这里可以使用old.proto和new.proto文件中的定义，但是不能使用other.proto文件中的定义。
```

protocol编译器会在编译命令中 `-I / --proto_path` 参数指定的目录中查找导入的文件，如果没有指定该参数，默认在当前目录中查找。

Message嵌套

上面已经介绍过可以嵌套使用另一个message作为字段类型，其实还可以在一个message内部定义另一个message类型，作为子级message。

示例： `Result` 类型在 `SearchResponse` 类型中定义并直接使用，作为 `results` 字段的类型

```
message SearchResponse {
    message Result {
        string url = 1;
        string title = 2;
```

```
        repeated string snippets = 3;
    }
    repeated Result results = 1;
}
```

内部声明的message类型名称只可在内部直接使用，在外部引用需要前置父级message名称，如 `Parent.Type` ：

```
message SomeOtherMessage {
    SearchResponse.Result result = 1;
}
```

支持多层嵌套：

```
message Outer {                 // Level 0
    message MiddleAA {          // Level 1
        message Inner {         // Level 2
            int64 ival = 1;
            bool  booly = 2;
        }
    }
    message MiddleBB {          // Level 1
        message Inner {         // Level 2
            int32 ival = 1;
            bool  booly = 2;
        }
    }
}
```

Map类型

proto3支持map类型声明：

```
map<key_type, value_type> map_field = N;

message Project {...}
map<string, Project> projects = 1;
```

- `key_type` 类型可以是内置的标量类型(除浮点类型和 `bytes`)
- `value_type` 可以是除map以外的任意类型

- map字段不支持 `repeated` 属性
- 不要依赖map类型的字段顺序

包(Packages)

在 `.proto` 文件中使用 `package` 声明包名，避免命名冲突。

```
syntax = "proto3";
package foo.bar;
message Open {...}
```

在其他的消息格式定义中可以使用包名+消息名的方式来使用类型，如:

```
message Foo {
    ...
    foo.bar.Open open = 1;
    ...
}
```

在不同的语言中，包名定义对编译后生成的代码的影响不同:

- C++ 中: 对应C++命名空间，例如 `Open` 会在命名空间 `foo::bar` 中
- Java 中: package会作为Java包名，除非指定了 `option jave_package` 选项
- Python 中: package被忽略
- Go 中: 默认使用package名作为包名，除非指定了 `option go_package` 选项
- JavaNano 中: 同Java
- C# 中: package会转换为驼峰式命名空间，如 `Foo.Bar` ,除非指定了 `option csharp_namespace` 选项

定义服务(Service)

如果想要将消息类型用在RPC(远程方法调用)系统中，可以在 `.proto` 文件中定义一个RPC服务接口，protocol编译器会根据所选择的不同语言生成服务接口代码。例如，想要定义一个RPC服务并具有一个方法，该方法接收 `SearchRequest` 并返回一个 `SearchResponse` ，此时可以在 `.proto` 文件中进行如下定义:

```
service SearchService {
    rpc Search (SearchRequest) returns (SearchResponse) {}
}
```

生成的接口代码作为客户端与服务端的约定，服务端必须实现定义的所有接口方法，客户端直接调用同名方法向服务端发起请求。

选项(Options)

在定义.proto文件时可以标注一系列的options。Options并不改变整个文件声明的含义，但却可以影响特定环境下处理方式。完整的可用选项可以查看 `google/protobuf/descriptor.proto` .

一些选项是文件级别的，意味着它可以作用于顶层作用域，不包含在任何消息内部、enum或服务定义中。一些选项是消息级别的，可以用在消息定义的内部。

以下是一些常用的选项:

- `java_package` (file option): 指定生成java类所在的包，如果在.proto文件中没有明确的声明java_package，会使用默认包名。不需要生成java代码时不起作用
- `java_outer_classname` (file option): 指定生成Java类的名称，如果在.proto文件中没有明确声明java_outer_classname，生成的class名称将会根据.proto文件的名称采用驼峰式的命名方式进行生成。如（foo_bar.proto生成的java类名为FooBar.java），不需要生成java代码时不起任何作用
- `objc_class_prefix` (file option): 指定Objective-C类前缀，会前置在所有类和枚举类型名之前。没有默认值，应该使用3-5个大写字母。注意所有2个字母的前缀是Apple保留的。

基本规范

- 描述文件以 `.proto` 做为文件后缀，除结构定义外的语句以分号结尾
    - 结构定义包括: message、service、enum
    - rpc方法定义结尾的分号可有可无
- Message命名采用驼峰命名方式，字段命名采用小写字母加下划线分隔方式

```
message SongServerRequest {
    required string song_name = 1;
}
```

- Enums类型名采用驼峰命名方式，字段命名采用大写字母加下划线分隔方式

```
enum Foo {
    FIRST_VALUE = 1;
    SECOND_VALUE = 2;
}
```

- Service名称与RPC方法名统一采用驼峰式命名

编译

通过定义好的 `.proto` 文件生成Java, Python, C++, Go, Ruby, JavaNano, Objective-C, or C# 代码，需要安装编译器 `protoc` 。参考Github项目google/protobuf安装编译器.

运行命令:

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --
python_out=DST_DIR --go_out=DST_DIR --ruby_out=DST_DIR --javanano_out=DST_DIR -
-objc_out=DST_DIR --csharp_out=DST_DIR path/to/file.proto
```

这里只做参考就好，具体语言的编译实例请参考详细文档。

- Go generated code reference
- C++ generated code reference
- Java generated code reference
- Python generated code reference
- Objective-C generated code reference
- C# generated code reference
- Javascript Generated Code Guide
- PHP Generated Code Guide

> 吐槽: 照着官方文档一步步操作不一定成功哦！

更多

- Any类型
- Oneof
- 自定义Options

这些用法在实践中很少使用，这里不做介绍，有需要请参考官方文档。

实践

准备工作完成了，这部分我们开始实践吧。

gRPC的基本使用非常简单，看完这部分的第一个示例就可以直接用了。但是在实际环境中，我们不会仅仅满足于能用，而是要更好的使用。一个完整的服务，包含授权认证、数据追踪、负载均衡……，我们从一个简单的项目开始，说明gRPC的基本使用姿势，然后一点点细化扩展，逐步深入完善，打造一个完整的RPC服务。

Hello gRPC

按照惯例，这里也从一个Hello项目开始，本项目定义了一个Hello Service，客户端发送包含字符串名字的请求，服务端返回Hello消息。

流程：

1. 编写 `.proto` 描述文件
2. 编译生成 `.pb.go` 文件
3. 服务端实现约定的接口并提供服务
4. 客户端按照约定调用 `.pb.go` 文件中的方法请求服务

项目结构：

```
|— hello/
    |— client/
        |— main.go   // 客户端
    |— server/
        |— main.go   // 服务端
|— proto/
    |— hello/
        |— hello.proto   // proto描述文件
        |— hello.pb.go   // proto编译后文件
```

Step1: 编写描述文件: hello.proto

```
syntax = "proto3"; // 指定proto版本
package hello;     // 指定默认包名

// 指定golang包名
option go_package = "hello";

// 定义Hello服务
service Hello {
    // 定义SayHello方法
    rpc SayHello(HelloRequest) returns (HelloResponse) {}
}

// HelloRequest 请求结构
message HelloRequest {
    string name = 1;
```

```
}

// HelloResponse 响应结构
message HelloResponse {
    string message = 1;
}
```

`hello.proto` 文件中定义了一个Hello Service，该服务包含一个 `SayHello` 方法，同时声明了 `HelloRequest` 和 `HelloResponse` 消息结构用于请求和响应。客户端使用 `HelloRequest` 参数调用 `SayHello` 方法请求服务端，服务端响应 `HelloResponse` 消息。一个最简单的服务就定义好了。

Step2: 编译生成 `.pb.go` 文件

```
$ cd proto/hello

# 编译hello.proto
$ protoc -I . --go_out=plugins=grpc:. ./hello.proto
```

在当前目录内生成的 `hello.pb.go` 文件，按照 `.proto` 文件中的说明，包含服务端接口 `HelloServer` 描述，客户端接口及实现 `HelloClient` ，及 `HelloRequest` 、 `HelloResponse` 结构体。

> 注意：不要手动编辑该文件

Step3: 实现服务端接口 `server/main.go`

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
```

```
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("Failed to listen: %v", err)
    }

    // 实例化grpc Server
    s := grpc.NewServer()

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address)
    s.Serve(listen)
}
```

服务端引入编译后的 `proto` 包，定义一个空结构用于实现约定的接口，接口描述可以查看 `hello.pb.go` 文件中的 `HelloServer` 接口描述。实例化grpc Server并注册HelloService，开始提供服务。

运行：

```
$ go run main.go
Listen on 127.0.0.1:50052  //服务端已开启并监听50052端口
```

Step4: 实现客户端调用 `client/main.go`

```go
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

func main() {
    // 连接
    conn, err := grpc.Dial(Address, grpc.WithInsecure())
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)

    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}
```

客户端初始化连接后直接调用 `hello.pb.go` 中实现的 `SayHello` 方法，即可向服务端发起请求，使用姿势就像调用本地方法一样。

运行：

```
$ go run main.go
Hello gRPC.    // 接收到服务端响应
```

如果你收到了”Hello gRPC”的回复，恭喜你已经会使用github.com/jergoo/go-grpc-example/proto/hello了。

> 建议到这里仔细看一看hello.pb.go文件中的内容，对比hello.proto文件，理解protobuf中的定义转换为golang后的结构。

Interceptor 拦截器

grpc服务端和客户端都提供了interceptor功能，功能类似middleware，很适合在这里处理验证、日志等流程。

在自定义Token认证的示例中，认证信息是由每个服务中的方法处理并认证的，如果有大量的接口方法，这种姿势就太不优雅了，每个接口实现都要先处理认证信息。这个时候interceptor就可以用来解决了这个问题，在请求被转到具体接口之前处理认证信息，一处认证，到处无忧。在客户端，我们增加一个请求日志，记录请求相关的参数和耗时等等。修改hello_token项目实现：

目录结构

```
|— hello_interceptor/
    |— client/
        |— main.go   // 客户端
    |— server/
        |— main.go   // 服务端
|— keys/             // 证书目录
    |— server.key
    |— server.pem
|— proto/
    |— hello/
        |— hello.proto   // proto描述文件
        |— hello.pb.go   // proto编译后文件
```

示例代码

Step 1. 服务端**interceptor:**

> hello_interceptor/server/main.go

```go
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

```

```
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"       // grpc 响应状态码
    "google.golang.org/grpc/credentials" // grpc认证包
    "google.golang.org/grpc/grpclog"
    "google.golang.org/grpc/metadata" // grpc metadata包
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("Failed to listen: %v", err)
    }

    var opts []grpc.ServerOption

    // TLS认证
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to generate credentials %v", err)
    }

```

```
    opts = append(opts, grpc.Creds(creds))

    // 注册interceptor
    opts = append(opts, grpc.UnaryInterceptor(interceptor))

    // 实例化grpc Server
    s := grpc.NewServer(opts...)

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address + " with TLS + Token + Interceptor")

    s.Serve(listen)
}

// auth 验证Token
func auth(ctx context.Context) error {
    md, ok := metadata.FromContext(ctx)
    if !ok {
        return grpc.Errorf(codes.Unauthenticated, "无Token认证信息")
    }

    var (
        appid  string
        appkey string
    )

    if val, ok := md["appid"]; ok {
        appid = val[0]
    }

    if val, ok := md["appkey"]; ok {
        appkey = val[0]
    }

    if appid != "101010" || appkey != "i am key" {
        return grpc.Errorf(codes.Unauthenticated, "Token认证信息无效: appid=%s,
appkey=%s", appid, appkey)
    }

    return nil
```

```
}

// interceptor 拦截器
func interceptor(ctx context.Context, req interface{}, info
*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
    err := auth(ctx)
    if err != nil {
        return nil, err
    }
    // 继续处理请求
    return handler(ctx, req)
}
```

Step 2. 实现客户端interceptor:

> hello_intercepror/client/main.go

```
package main

import (
    "time"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"

    // OpenTLS 是否开启TLS认证
    OpenTLS = true
)

// customCredential 自定义认证
type customCredential struct{}

// GetRequestMetadata 实现自定义认证接口
```

```
func (c customCredential) GetRequestMetadata(ctx context.Context, uri
...string) (map[string]string, error) {
    return map[string]string{
        "appid":  "101010",
        "appkey": "i am key",
    }, nil
}

// RequireTransportSecurity 自定义认证是否开启TLS
func (c customCredential) RequireTransportSecurity() bool {
    return OpenTLS
}

func main() {
    var err error
    var opts []grpc.DialOption

    if OpenTLS {
        // TLS连接
        creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
"server name")
        if err != nil {
            grpclog.Fatalf("Failed to create TLS credentials %v", err)
        }
        opts = append(opts, grpc.WithTransportCredentials(creds))
    } else {
        opts = append(opts, grpc.WithInsecure())
    }

    // 指定自定义认证
    opts = append(opts, grpc.WithPerRPCCredentials(new(customCredential)))
    // 指定客户端interceptor
    opts = append(opts, grpc.WithUnaryInterceptor(interceptor))

    conn, err := grpc.Dial(Address, opts...)
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

```

```
    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)
    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}

// interceptor 客户端拦截器
func interceptor(ctx context.Context, method string, req, reply interface{}, cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    start := time.Now()
    err := invoker(ctx, method, req, reply, cc, opts...)
    grpclog.Printf("method=%s req=%v rep=%v duration=%s error=%v\n", method, req, reply, time.Since(start), err)
    return err
}
```

运行结果

```
$ cd hello_inteceptor/server && go run main.go
Listen on 127.0.0.1:50052 with TLS + Token + Interceptor
```

```
$ cd hello_inteceptor/client && go run main.go
method=/hello.Hello/SayHello req=name:"gRPC"  rep=message:"Hello gRPC."
duration=33.879699ms error=<nil>

Hello gRPC.
```

项目推荐： go-grpc-middleware

这个项目对interceptor进行了封装，支持多个拦截器的链式组装，对于需要多种处理的地方使用起来会更方便些。

负载均衡（load-balancer）

流式调用

内置Trace

grpc内置了客户端和服务端的请求追踪，基于 `golang.org/x/net/trace` 包实现，默认是开启状态，可以查看事件和请求日志，对于基本的请求状态查看调试也是很有帮助的，客户端与服务端基本一致，这里以服务端开启trace server为例，修改hello项目服务端代码：

目录结构

```
|— hello_trace/
    |— client/
        |— main.go   // 客户端
    |— server/
        |— main.go   // 服务端
|— proto/
    |— hello/
        |— hello.proto   // proto描述文件
        |— hello.pb.go   // proto编译后文件
```

示例代码

```
package main

import (
    "fmt"
    "net"
    "net/http"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包

    "golang.org/x/net/context"
    "golang.org/x/net/trace"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
```

```
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("failed to listen: %v", err)
    }

    // 实例化grpc Server
    s := grpc.NewServer()

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    // 开启trace
    go startTrace()

    grpclog.Println("Listen on " + Address)
    s.Serve(listen)
}

func startTrace() {
    trace.AuthRequest = func(req *http.Request) (any, sensitive bool) {
        return true, true
    }
    go http.ListenAndServe(":50051", nil)
    grpclog.Println("Trace listen on 50051")
```

```
60. }
```

这里我们开启一个http服务监听50051端口，用来查看grpc请求的trace信息

运行：

```
1. $ go run main.go
2.
3. Listen on 127.0.0.1:50052
4. Trace listen on 50051
5.
6. # 进入client目录执行一次客户端请求
```

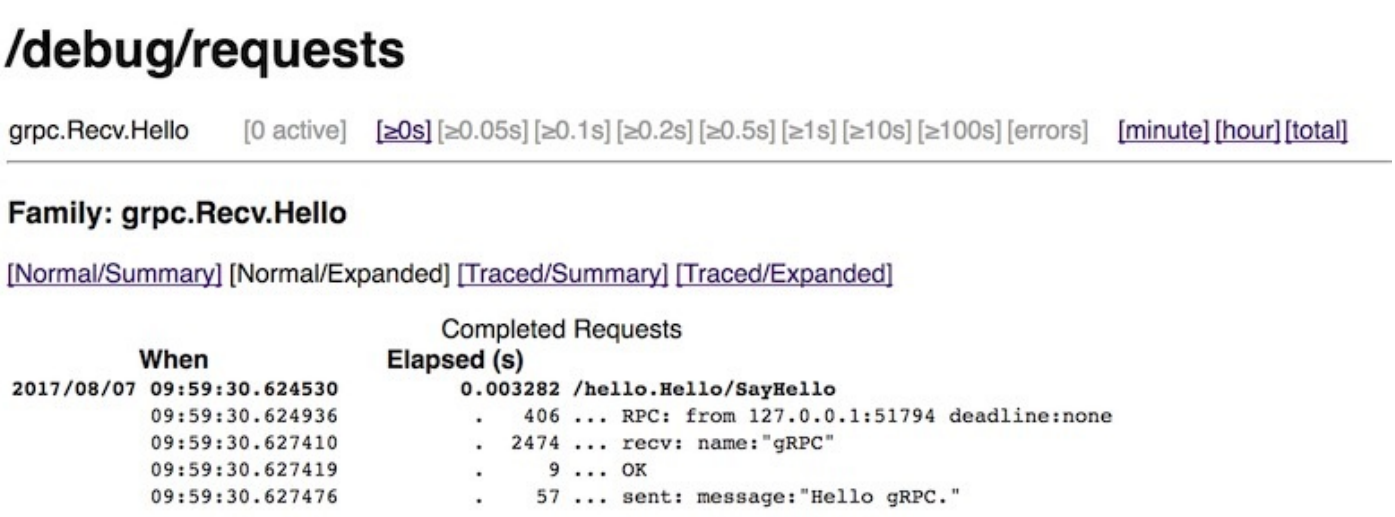
服务端事件查看

访问：localhost:50051/debug/events，结果如图：

可以看到服务端注册的服务和服务正常启动的事件信息。

请求日志信息查看

访问：localhost:50051/debug/requests，结果如图：



/debug/requests

grpc.Recv.Hello [0 active] [≥0s] [≥0.05s] [≥0.1s] [≥0.2s] [≥0.5s] [≥1s] [≥10s] [≥100s] [errors] [minute] [hour] [total]

Family: grpc.Recv.Hello

[Normal/Summary] [Normal/Expanded] [Traced/Summary] [Traced/Expanded]

Completed Requests

When	Elapsed (s)	
2017/08/07 09:59:30.624530	0.003282	/hello.Hello/SayHello
09:59:30.624936	. 406	... RPC: from 127.0.0.1:51794 deadline:none
09:59:30.627410	. 2474	... recv: name:"gRPC"
09:59:30.627419	. 9	... OK
09:59:30.627476	. 57	... sent: message:"Hello gRPC."

这里可以显示最近的请求状态，包括请求的服务、参数、耗时、响应，对于简单的状态查看还是很方便的，默认值显示最近10条记录。

认证

gRPC默认内置了两种认证方式:

- SSL/TLS认证方式
- 基于Token的认证方式

同时，gRPC提供了接口用于扩展自定义认证方式

TLS认证示例

这里直接扩展hello项目，实现TLS认证机制

首先需要准备证书，在hello目录新建keys目录用于存放证书文件。

证书制作

制作私钥 (.key)

```
# Key considerations for algorithm "RSA" ≥ 2048-bit
$ openssl genrsa -out server.key 2048

# Key considerations for algorithm "ECDSA" ≥ secp384r1
# List ECDSA the supported curves (openssl ecparam -list_curves)
$ openssl ecparam -genkey -name secp384r1 -out server.key
```

自签名公钥(x509) (PEM-encodings `.pem` | `.crt`)

```
$ openssl req -new -x509 -sha256 -key server.key -out server.pem -days 3650
```

自定义信息

```
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:XxXx
Locality Name (eg, city) []:XxXx
Organization Name (eg, company) [Internet Widgits Pty Ltd]:XX Co. Ltd
Organizational Unit Name (eg, section) []:Dev
Common Name (e.g. server FQDN or YOUR name) []:server name
```

```
Email Address []:xxx@xxx.com
```

目录结构

```
|— hello-tls/
    |— client/
        |— main.go   // 客户端
    |— server/
        |— main.go   // 服务端
|— keys/                 // 证书目录
    |— server.key
    |— server.pem
|— proto/
    |— hello/
        |— hello.proto   // proto描述文件
        |— hello.pb.go   // proto编译后文件
```

示例代码

`proto/helloworld.proto` 及 `proto/hello.pb.go` 文件不需要改动

修改服务端代码：server/main.go

```go
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)
```

```
19.
20. // 定义helloService并实现约定的接口
21. type helloService struct{}
22.
23. // HelloService Hello服务
24. var HelloService = helloService{}
25.
26. // SayHello 实现Hello服务接口
    func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest)
27. (*pb.HelloResponse, error) {
28.     resp := new(pb.HelloResponse)
29.     resp.Message = fmt.Sprintf("Hello %s.", in.Name)
30.
31.     return resp, nil
32. }
33.
34. func main() {
35.     listen, err := net.Listen("tcp", Address)
36.     if err != nil {
37.         grpclog.Fatalf("Failed to listen: %v", err)
38.     }
39.
40.     // TLS认证
        creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
41. "../../keys/server.key")
42.     if err != nil {
43.         grpclog.Fatalf("Failed to generate credentials %v", err)
44.     }
45.
46.     // 实例化grpc Server, 并开启TLS认证
47.     s := grpc.NewServer(grpc.Creds(creds))
48.
49.     // 注册HelloService
50.     pb.RegisterHelloServer(s, HelloService)
51.
52.     grpclog.Println("Listen on " + Address + " with TLS")
53.
54.     s.Serve(listen)
55. }
```

运行:

```
$ go run main.go

Listen on 127.0.0.1:50052 with TLS
```

服务端在实例化grpc Server时，可配置多种选项，TLS认证是其中之一

客户端添加TLS认证：client/main.go

```
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

func main() {
    // TLS连接
    creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
"server name")
    if err != nil {
        grpclog.Fatalf("Failed to create TLS credentials %v", err)
    }

    conn, err := grpc.Dial(Address, grpc.WithTransportCredentials(creds))
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
```

```
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)
    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}
```

运行:

```
$ go run main.go

Hello gRPC
```

客户端添加TLS认证的方式和服务端类似，在创建连接 `Dial` 时，同样可以配置多种选项，后面的示例中会看到更多的选项。

Token认证示例

再进一步，继续扩展hello-tls项目，实现TLS + Token认证机制

目录结构

```
|— hello_token/
    |— client/
        |— main.go   // 客户端
    |— server/
        |— main.go   // 服务端
|— keys/             // 证书目录
    |— server.key
    |— server.pem
|— proto/
    |— hello/
        |— hello.proto   // proto描述文件
        |— hello.pb.go   // proto编译后文件
```

示例代码

先修改客户端实现: client/main.go

```go
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"

    // OpenTLS 是否开启TLS认证
    OpenTLS = true
)

// customCredential 自定义认证
type customCredential struct{}

// GetRequestMetadata 实现自定义认证接口
func (c customCredential) GetRequestMetadata(ctx context.Context, uri
...string) (map[string]string, error) {
    return map[string]string{
        "appid":  "101010",
        "appkey": "i am key",
    }, nil
}

// RequireTransportSecurity 自定义认证是否开启TLS
func (c customCredential) RequireTransportSecurity() bool {
    return OpenTLS
}

func main() {
    var err error
    var opts []grpc.DialOption

    if OpenTLS {
        // TLS连接
```

```
            creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
"server name")
            if err != nil {
                grpclog.Fatalf("Failed to create TLS credentials %v", err)
            }
            opts = append(opts, grpc.WithTransportCredentials(creds))
        } else {
            opts = append(opts, grpc.WithInsecure())
        }

        // 使用自定义认证
        opts = append(opts, grpc.WithPerRPCCredentials(new(customCredential)))

        conn, err := grpc.Dial(Address, opts...)

        if err != nil {
            grpclog.Fatalln(err)
        }

        defer conn.Close()

        // 初始化客户端
        c := pb.NewHelloClient(conn)

        // 调用方法
        req := &pb.HelloRequest{Name: "gRPC"}
        res, err := c.SayHello(context.Background(), req)
        if err != nil {
            grpclog.Fatalln(err)
        }

        grpclog.Println(res.Message)
}
```

这里我们定义了一个 `customCredential` 结构，并实现了两个方法 `GetRequestMetadata` 和 `RequireTransportSecurity` 。这是gRPC提供的自定义认证方式，每次RPC调用都会传输认证信息。 `customCredential` 其实是实现了 `grpc/credential` 包内的 `PerRPCCredentials` 接口。每次调用，token信息会通过请求的metadata传输到服务端。下面具体看一下服务端如何获取metadata中的信息。

修改server/main.go中的SayHello方法:

```go
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
    "google.golang.org/grpc/metadata" // 引入grpc meta包
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService ...
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest)
(*pb.HelloResponse, error) {
    // 解析metada中的信息并验证
    md, ok := metadata.FromContext(ctx)
    if !ok {
        return nil, grpc.Errorf(codes.Unauthenticated, "无Token认证信息")
    }

    var (
        appid  string
        appkey string
    )

    if val, ok := md["appid"]; ok {
```

```
42.         appid = val[0]
43.     }
44.
45.     if val, ok := md["appkey"]; ok {
46.         appkey = val[0]
47.     }
48.
49.     if appid != "101010" || appkey != "i am key" {
            return nil, grpc.Errorf(codes.Unauthenticated, "Token认证信息无效:
50. appid=%s, appkey=%s", appid, appkey)
51.     }
52.
53.     resp := new(pb.HelloResponse)
        resp.Message = fmt.Sprintf("Hello %s.\nToken info: appid=%s,appkey=%s",
54. in.Name, appid, appkey)
55.
56.     return resp, nil
57. }
58.
59. func main() {
60.     listen, err := net.Listen("tcp", Address)
61.     if err != nil {
62.         grpclog.Fatalf("failed to listen: %v", err)
63.     }
64.
65.     // TLS认证
        creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
66. "../../keys/server.key")
67.     if err != nil {
68.         grpclog.Fatalf("Failed to generate credentials %v", err)
69.     }
70.
71.     // 实例化grpc Server, 并开启TLS认证
72.     s := grpc.NewServer(grpc.Creds(creds))
73.
74.     // 注册HelloService
75.     pb.RegisterHelloServer(s, HelloService)
76.
77.     grpclog.Println("Listen on " + Address + " with TLS + Token")
78.
79.     s.Serve(listen)
80. }
```

服务端可以从 `context` 中获取每次请求的metadata，从中读取客户端发送的token信息并验证有效性。

运行：

```
$ go run main.go

Listen on 50052 with TLS + Token
```

运行客户端程序 client/main.go：

```
$ go run main.go

// 认证成功结果
Hello gRPC
Token info: appid=101010,appkey=i am key

// 修改key验证认证失败结果：
rpc error: code = 16 desc = Token认证信息无效: appID=101010, appKey=i am not key
```

`google.golang.org/grpc/credentials/oauth` 包已实现了用于Google API的oauth和jwt验证的方法，使用方法可以参考官方文档。在实际应用中，我们可以根据自己的业务需求实现合适的验证方式。

连接管理

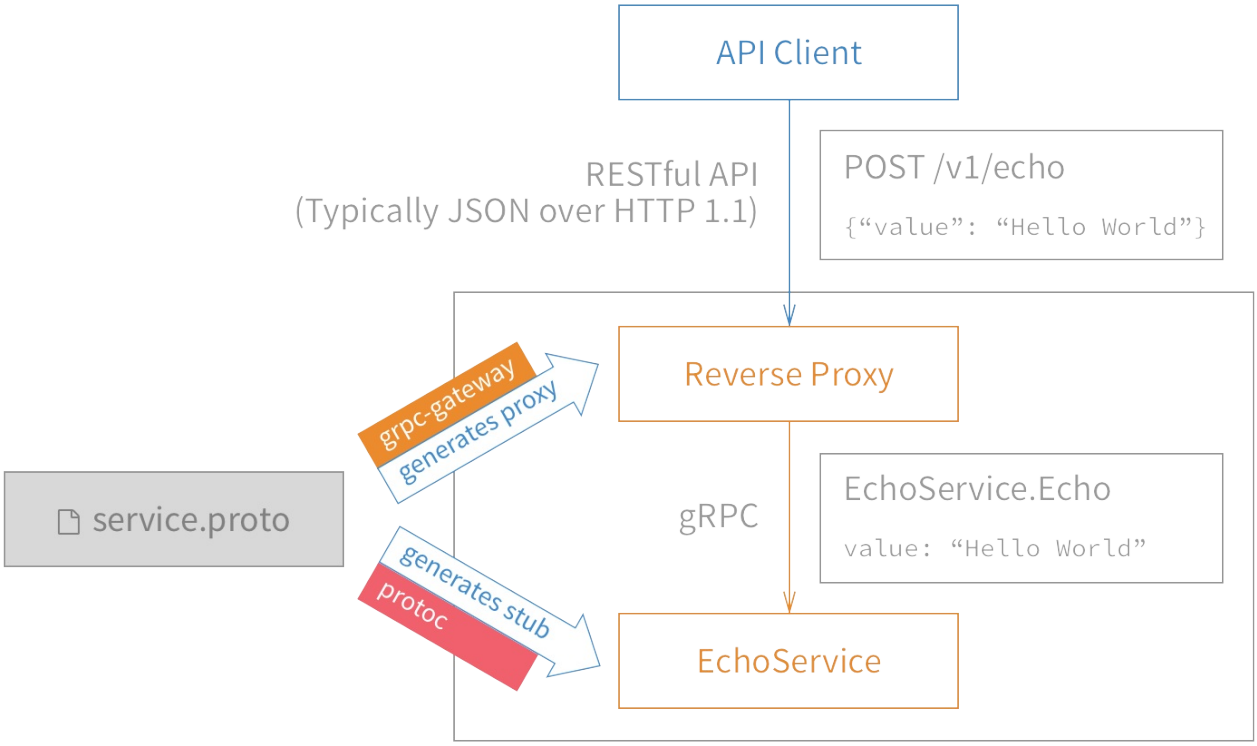
gRPC生态

HTTP网关

源自coreos的一篇博客 Take a REST with HTTP/2, Protobufs, and Swagger。

etcd3 API全面升级为gRPC后，同时要提供REST API服务，维护两个版本的服务显然不太合理，所以grpc-gateway诞生了。通过protobuf的自定义option实现了一个网关，服务端同时开启gRPC和HTTP服务，HTTP服务接收客户端请求后转换为grpc请求数据，获取响应后转为json数据返回给客户端。

结构如图:



安装grpc-gateway

```
$ go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
```

目录结构

```
|— hello_http/
    |— client/
```

```
        |— main.go   // 客户端
    |— server/
        |— main.go   // GRPC服务端
    |— server_http/
        |— main.go   // HTTP服务端
|— proto/
    |— google       // googleApi http-proto定义
        |— api
            |— annotations.proto
            |— annotations.pb.go
            |— http.proto
            |— http.pb.go
    |— hello_http/
        |— hello_http.proto   // proto描述文件
        |— hello_http.pb.go   // proto编译后文件
        |— hello_http_pb.gw.go // gateway编译后文件
```

这里用到了google官方Api中的两个proto描述文件，直接拷贝不要做修改，里面定义了protocol buffer扩展的HTTP option，为grpc的http转换提供支持。

示例代码

Step 1. 编写proto描述文件: proto/hello_http.proto

```
syntax = "proto3";

package hello_http;
option go_package = "hello_http";

import "google/api/annotations.proto";

// 定义Hello服务
service HelloHTTP {
    // 定义SayHello方法
    rpc SayHello(HelloHTTPRequest) returns (HelloHTTPResponse) {
        // http option
        option (google.api.http) = {
            post: "/example/echo"
            body: "*"
        };
    }
```

```
}

// HelloRequest 请求结构
message HelloHTTPRequest {
    string name = 1;
}

// HelloResponse 响应结构
message HelloHTTPResponse {
    string message = 1;
}
```

这里在原来的 `SayHello` 方法定义中增加了http option，POST方式，路由为”/example/echo”。

Step 2. 编译proto

```
$ cd proto

# 编译google.api
$ protoc -I . --go_out=plugins=grpc,Mgoogle/protobuf/descriptor.proto=github.com/go/descriptor:. google/api/*.proto

# 编译hello_http.proto
$ protoc -I . --go_out=plugins=grpc,Mgoogle/api/annotations.proto=github.com/jergexample/proto/google/api:. hello_http/*.proto

# 编译hello_http.proto gateway
$ protoc --grpc-gateway_out=logtostderr=true:. hello_http/hello_http.proto
```

注意这里需要编译google/api中的两个proto文件，同时在编译hello_http.proto时使用 `M` 参数指定引入包名，最后使用grpc-gateway编译生成 `hello_http_pb.gw.go` 文件，这个文件就是用来做协议转换的，查看文件可以看到里面生成的http handler，处理proto文件中定义的路由”example/echo”接收POST参数，调用HelloHTTP服务的客户端请求grpc服务并响应结果。

Step 3: 实现服务端和客户端

server/main.go和client/main.go的实现与hello项目一致，这里不再说明。

> server_http/main.go

```
package main
```

```
2.
3.  import (
4.      "net/http"
5.
6.      "github.com/grpc-ecosystem/grpc-gateway/runtime"
7.      "golang.org/x/net/context"
8.      "google.golang.org/grpc"
9.      "google.golang.org/grpc/grpclog"
10.
11.     gw "github.com/jergoo/go-grpc-example/proto/hello_http"
12. )
13.
14. func main() {
15.     ctx := context.Background()
16.     ctx, cancel := context.WithCancel(ctx)
17.     defer cancel()
18.
19.     // grpc服务地址
20.     endpoint := "127.0.0.1:50052"
21.     mux := runtime.NewServeMux()
22.     opts := []grpc.DialOption{grpc.WithInsecure()}
23.
24.     // HTTP转grpc
25.     err := gw.RegisterHelloHTTPHandlerFromEndpoint(ctx, mux, endpoint, opts)
26.     if err != nil {
27.         grpclog.Fatalf("Register handler err:%v\n", err)
28.     }
29.
30.     grpclog.Println("HTTP Listen on 8080")
31.     http.ListenAndServe(":8080", mux)
32. }
```

就是这么简单。开启了一个http server，收到请求后根据路由转发请求到对应的RPC接口获得结果。grpc-gateway做的事情就是帮我们自动生成了转换过程的实现。

运行结果

依次开启gRPC服务和HTTP服务端：

```
1.  $ cd hello_http/server && go run main.go
2.  Listen on 127.0.0.1:50052
```

```
$ cd hello_http/server_http && go run main.go
HTTP Listen on 8080
```

调用grpc客户端:

```
$ cd hello_http/client && go run main.go
Hello gRPC.

# HTTP 请求
$ curl -X POST -k http://localhost:8080/example/echo -d '{"name": "gRPC-HTTP is working!"}'
{"message":"Hello gRPC-HTTP is working!."}
```

升级版服务端

上面的使用方式已经实现了我们最初的需求，grpc-gateway项目中提供的示例也是这种使用方式，这样后台需要开启两个服务两个端口。其实我们也可以只开启一个服务，同时提供http和gRPC调用方式。

新建一个项目 `hello_http_2` ，基于 `hello_tls` 项目改造。客户端只要修改调用的proto包地址就可以了，这里我们看服务端的实现:

> hello_http_2/server/main.go

```
package main

import (
    "crypto/tls"
    "io/ioutil"
    "net"
    "net/http"
    "strings"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    pb "github.com/jergoo/go-grpc-example/proto/hello_http"
    "golang.org/x/net/context"
    "golang.org/x/net/http2"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/grpclog"
)
```

```
18.
19. // 定义helloHTTPService并实现约定的接口
20. type helloHTTPService struct{}
21.
22. // HelloHTTPService Hello HTTP服务
23. var HelloHTTPService = helloHTTPService{}
24.
25. // SayHello 实现Hello服务接口
    func (h helloHTTPService) SayHello(ctx context.Context, in
26. *pb.HelloHTTPRequest) (*pb.HelloHTTPResponse, error) {
27.     resp := new(pb.HelloHTTPResponse)
28.     resp.Message = "Hello " + in.Name + "."
29.
30.     return resp, nil
31. }
32.
33. func main() {
34.     endpoint := "127.0.0.1:50052"
35.     conn, err := net.Listen("tcp", endpoint)
36.     if err != nil {
37.         grpclog.Fatalf("TCP Listen err:%v\n", err)
38.     }
39.
40.     // grpc tls server
        creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
41. "../../keys/server.key")
42.     if err != nil {
43.         grpclog.Fatalf("Failed to create server TLS credentials %v", err)
44.     }
45.     grpcServer := grpc.NewServer(grpc.Creds(creds))
46.     pb.RegisterHelloHTTPServer(grpcServer, HelloHTTPService)
47.
48.     // gw server
49.     ctx := context.Background()
        dcreds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
50. "server name")
51.     if err != nil {
52.         grpclog.Fatalf("Failed to create client TLS credentials %v", err)
53.     }
54.     dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}
55.     gwmux := runtime.NewServeMux()
        if err = pb.RegisterHelloHTTPHandlerFromEndpoint(ctx, gwmux, endpoint,
56. dopts); err != nil {
```

```
57.         grpclog.Fatalf("Failed to register gw server: %v\n", err)
58.     }
59.
60.     // http服务
61.     mux := http.NewServeMux()
62.     mux.Handle("/", gwmux)
63.
64.     srv := &http.Server{
65.         Addr:      endpoint,
66.         Handler:   grpcHandlerFunc(grpcServer, mux),
67.         TLSConfig: getTLSConfig(),
68.     }
69.
70.     grpclog.Infof("gRPC and https listen on: %s\n", endpoint)
71.
72.     if err = srv.Serve(tls.NewListener(conn, srv.TLSConfig)); err != nil {
73.         grpclog.Fatal("ListenAndServe: ", err)
74.     }
75.
76.     return
77. }
78.
79. func getTLSConfig() *tls.Config {
80.     cert, _ := ioutil.ReadFile("../../keys/server.pem")
81.     key, _ := ioutil.ReadFile("../../keys/server.key")
82.     var demoKeyPair *tls.Certificate
83.     pair, err := tls.X509KeyPair(cert, key)
84.     if err != nil {
85.         grpclog.Fatalf("TLS KeyPair err: %v\n", err)
86.     }
87.     demoKeyPair = &pair
88.     return &tls.Config{
89.         Certificates: []tls.Certificate{*demoKeyPair},
90.         NextProtos:   []string{http2.NextProtoTLS}, // HTTP2 TLS支持
91.     }
92. }
93.
    // grpcHandlerFunc returns an http.Handler that delegates to grpcServer on
94. incoming gRPC
95. // connections or otherHandler otherwise. Copied from cockroachdb.
    func grpcHandlerFunc(grpcServer *grpc.Server, otherHandler http.Handler)
96. http.Handler {
97.     if otherHandler == nil {
```

```
98.         return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
99.             grpcServer.ServeHTTP(w, r)
100.         })
101.     }
102.     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"),
103. "application/grpc") {
104.             grpcServer.ServeHTTP(w, r)
105.         } else {
106.             otherHandler.ServeHTTP(w, r)
107.         }
108.     })
109. }
```

gRPC服务端接口的实现没有区别，重点在于HTTP服务的实现。gRPC是基于http2实现的， `net/http` 包也实现了http2，所以我们可以开启一个HTTP服务同时服务两个版本的协议，在注册http handler的时候，在方法 `grpcHandlerFunc` 中检测请求头信息，决定是直接调用gRPC服务，还是使用gateway的HTTP服务。 `net/http` 中对http2的支持要求开启https，所以这里要求使用https服务。

步骤

- 注册开启TLS的grpc服务
- 注册开启TLS的gateway服务，地址指向grpc服务
- 开启HTTP server

运行结果

```
1. $ cd hello_http_2/server && go run main.go
2. gRPC and https listen on: 127.0.0.1:50052
```

```
1. $ cd hello_http_2/client && go run main.go
2. Hello gRPC.
3.
4. # HTTP 请求
   $ curl -X POST -k https://localhost:50052/example/echo -d '{"name": "gRPC-HTTP
5. is working!"}'
6. {"message":"Hello gRPC-HTTP is working!."}
```

中间件

多语言支持

- Java
- Node.js
- PHP
- Python

Java

node.js

PHP

Python

参考

相关资料:

- grpc官网
- grpc中文文档
- protobuf官方文档

相关项目

- grpc/grpc
- grpc-go
- grpc-java
- google/protobuf
- golang/protobuf
- grpc-middleware
- grpc-gateway