

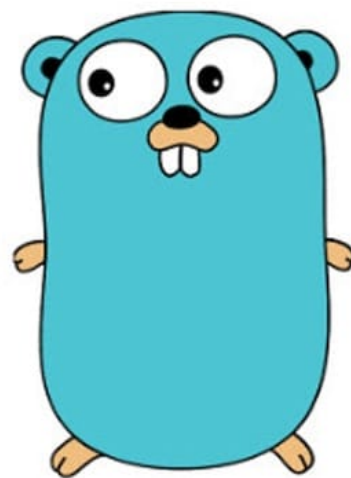
Go Protobuf 简明教程

Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)



protobuf
Protocol Buffers



1 Protocol Buffers 简介

protobuf 即 Protocol Buffers，是一种轻便高效的结构化数据存储格式，与语言、平台无关，可扩展可序列化。protobuf 性能和效率大幅度优于 JSON、XML 等其他结构化数据格式。protobuf 是以二进制方式存储的，占用空间小，但也带来了可读性差的缺点。protobuf 在通信协议和数据存储等领域应用广泛。例如著名的分布式缓存工具 [Memcached](#) 的 Go 语言版本 [groupcache](#) 就使用了 protobuf 作为其 RPC 数据格式。

Protobuf 在 .proto 定义需要处理的结构化数据，可以通过 protoc 工具，将 .proto 文件转换为 C、C++、Golang、Java、Python 等多种语言的代码，兼容性好，易于使用。

2 安装

2.1 protoc

从 [Protobuf Releases](#) 下载最先版本的发布包安装。如果是 Ubuntu，可以按照如下步骤操作（以3.11.2为例）。

```
1  # 下载安装包
2  $ wget https://github.com/protocolbuffers/protobuf/releases/download/v3.11.2/protoc-3.11
3  # 解压到 /usr/local 目录下
4  $ sudo 7z x protoc-3.11.2-linux-x86_64.zip -o/usr/local
```

如果不想安装在 /usr/local 目录下，可以解压到其他的路径，并把解压路径下的 bin 目录 加入到环境变量即可。

如果能正常显示版本，则表示安装成功。

```
1  $ protoc --version
2  libprotoc 3.11.2
```

2.2 protoc-gen-go

我们需要在 Golang 中使用 protobuf，还需要安装 protoc-gen-go，这个工具用来将 .proto 文件转换为 Golang 代码。

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

protoc-gen-go 将自动安装到 \$GOPATH/bin 目录下，也需要将这个目录加入到环境变量中。

3 定义消息类型

接下来，我们创建一个非常简单的示例， student.proto

```
1  syntax = "proto3";
2  package main;
3
4  // this is a comment
5  message Student {
6      string name = 1;
7      bool male = 2;
8      repeated int32 scores = 3;
9  }
```

在当前目录下执行：

```
1  $ protoc --go_out=. *.proto
2  $ ls
3  student.pb.go  student.proto
```

即是，将该目录下的所有的 .proto 文件转换为 Go 代码，我们可以看到该目录下多出了一个 Go 文件 student.pb.go。这个文件内部定义了一个结构体 Student，以及相关的方法：

```
1  type Student struct {
2      Name string `protobuf:"bytes,1,opt,name=name,proto3" json:"name,omitempty"`
3      Male bool `protobuf:"varint,2,opt,name=male,proto3" json:"male,omitempty"`
4      Scores []int32 `protobuf:"varint,3,rep,packed,name=scores,proto3" json:"scores,omite
5      ...
6  }
```

逐行解读 student.proto

- protobuf 有2个版本，默认版本是 proto2，如果需要 proto3，则需要在非空非注释第一行使用 `syntax = "proto3"` 标明版本。
- `package`，即包名声明符是可选的，用来防止不同的消息类型有命名冲突。
- 消息类型使用 `message` 关键字定义，Student 是类型名，name, male, scores 是该类型的 3 个字段，类型分别为 string, bool 和 []int32。字段可以是标量类型，也可以是合成类型。
- 每个字段的修饰符默认是 singular，一般省略不写，repeated 表示字段可重复，即用来表示 Go 语言中的数组类型。
- 每个字符 = 后面的数字称为标识符，每个字段都需要提供一个唯一的标识符。标识符用来在消息的二进制格式中识别各个字段，一旦使用就不能够再改变，标识符的取值范围为 $[1, 2^{29} - 1]$ 。
- .proto 文件可以写注释，单行注释 `//`，多行注释 `/* ... */`
- 一个 .proto 文件中可以写多个消息类型，即对应多个结构体(struct)。

接下来，就可以在项目代码中直接使用了，以下是一个非常简单的例子，即证明被序列化的和反序列化后的实例，包含相同的数据。

```
1  package main
2
3  import (
4      "log"
5
6      "github.com/golang/protobuf/proto"
7  )
8
```

```
9 func main() {
10     test := &Student{
11         Name: "geektutu",
12         Male: true,
13         Scores: []int32{98, 85, 88},
14     }
15     data, err := proto.Marshal(test)
16     if err != nil {
17         log.Fatal("marshaling error: ", err)
18     }
19     newTest := &Student{}
20     err = proto.Unmarshal(data, newTest)
21     if err != nil {
22         log.Fatal("unmarshaling error: ", err)
23     }
24     // Now test and newTest contain the same data.
25     if test.GetName() != newTest.GetName() {
26         log.Fatalf("data mismatch %q != %q", test.GetName(), newTest.GetName())
27     }
28 }
```

■ 保留字段(Reserved Field)

更新消息类型时，可能会将某些字段/标识符删除。这些被删掉的字段/标识符可能被重新使用，如果加载老版本的数据时，可能会造成数据冲突，在升级时，可以将这些字段/标识符保留(reserved)，这样就不会被重新使用了，protoc 会检查。

```
1 message Foo {
2     reserved 2, 15, 9 to 11;
3     reserved "foo", "bar";
4 }
```

4 字段类型

4.1 标量类型(Scalar)

| proto类型 | go类型 | 备注 | proto类型 | go类型 | 备注 |
|---------|---------|----|---------|---------|----|
| double | float64 | | float | float32 | |
| int32 | int32 | | int64 | int64 | |
| uint32 | uint32 | | uint64 | uint64 | |

| proto类型 | go类型 | 备注 | proto类型 | go类型 | 备注 |
|----------|--------|-------------------|----------|--------|--------------------|
| sint32 | int32 | 适合负数 | sint64 | int64 | 适合负数 |
| fixed32 | uint32 | 固长编码，适合大于2^28的值 | fixed64 | uint64 | 固长编码，适合大于2^56的值 |
| sfixed32 | int32 | 固长编码 | sfixed64 | int64 | 固长编码 |
| bool | bool | | string | string | UTF8 编码，长度不超过 2^32 |
| bytes | []byte | 任意字节序列，长度不超过 2^32 | | | |

标量类型如果没有被赋值，则不会被序列化，解析时，会赋予默认值。

- strings: 空字符串
- bytes: 空序列
- bools: false
- 数值类型: 0

4.2 枚举(Enumerations)

枚举类型适用于提供一组预定义的值，选择其中一个。例如我们将性别定义为枚举类型。

```
1  message Student {
2      string name = 1;
3      enum Gender {
4          FEMALE = 0;
5          MALE = 1;
6      }
7      Gender gender = 2;
8      repeated int32 scores = 3;
9  }
```

- 枚举类型的第一个选项的标识符必须是0，这也是枚举类型的默认值。
- 别名 (Alias) ，允许为不同的枚举值赋予相同的标识符，称之为别名，需要打开 allow_alias 选项。

```
1  message EnumAllowAlias {
2      enum Status {
3          option allow_alias = true;
4          UNKOWN = 0;
5          STARTED = 1;
6          RUNNING = 1;
```

```
7     }  
8 }
```

4.3 使用其他消息类型

Result 是另一个消息类型，在 SearchResponse 作为一个消息字段类型使用。

```
1 message SearchResponse {  
2     repeated Result results = 1;  
3 }  
4  
5 message Result {  
6     string url = 1;  
7     string title = 2;  
8     repeated string snippets = 3;  
9 }
```

嵌套写也是支持的：

```
1 message SearchResponse {  
2     message Result {  
3         string url = 1;  
4         string title = 2;  
5         repeated string snippets = 3;  
6     }  
7     repeated Result results = 1;  
8 }
```

如果定义在其他文件中，可以导入其他消息类型来使用：

```
1 import "myproject/other_protos.proto";
```

4.4 任意类型(Any)

Any 可以表示不在 .proto 中定义任意的内置类型。

```
1 import "google/protobuf/any.proto";  
2  
3 message ErrorStatus {  
4     string message = 1;  
5     repeated google.protobuf.Any details = 2;  
6 }
```

4.5 oneof

```
1  message SampleMessage {
2      oneof test_oneof {
3          string name = 4;
4          SubMessage sub_message = 9;
5      }
6  }
```

4.6 map

```
1  message MapRequest {
2      map<string, int32> points = 1;
3  }
```

5 定义服务(Services)

如果消息类型是用来远程通信的(Remote Procedure Call, RPC), 可以在 .proto 文件中定义 RPC 服务接口。例如我们定义了一个名为 SearchService 的 RPC 服务, 提供了 Search 接口, 入参是 SearchRequest 类型, 返回类型是 SearchResponse

```
1  service SearchService {
2      rpc Search (SearchRequest) returns (SearchResponse);
3  }
```

官方仓库也提供了一个[插件列表](#), 帮助开发基于 Protocol Buffer 的 RPC 服务。

6 protoc 其他参数

命令行使用方法

```
1  protoc --proto_path=IMPORT_PATH --<lang>_out=DST_DIR path/to/file.proto
```

- --proto_path=IMPORT_PATH : 可以在 .proto 文件中 import 其他的 .proto 文件, proto_path 即用来指定其他 .proto 文件的查找目录。如果没有引入其他的 .proto 文件, 该参数可以省略。
- --<lang>_out=DST_DIR : 指定生成代码的目标文件夹, 例如 -go_out=. 即生成 GO 代码在当前文件夹, 另外支持 cpp/java/python/ruby/objc/csharp/php 等语言

7 推荐风格

- 文件(Files)
 - 文件名使用小写下划线的命名风格, 例如 `lower_snake_case.proto`
 - 每行不超过 80 字符
 - 使用 2 个空格缩进
- 包(Packages)
 - 包名应该和目录结构对应, 例如文件在 `my/package/` 目录下, 包名应为 `my.package`
- 消息和字段(Messages & Fields)
 - 消息名使用首字母大写驼峰风格(CamelCase), 例如 `message StudentRequest { ... }`
 - 字段名使用小写下划线的风格, 例如 `string status_code = 1`
 - 枚举类型, 枚举名使用首字母大写驼峰风格, 例如 `enum FooBar`, 枚举值使用全大写下划线隔开的风格(CAPITALS_WITH_UNDERSCORES), 例如 `FOO_DEFAULT=1`
- 服务(Services)
 - RPC 服务名和方法名, 均使用首字母大写驼峰风格, 例如 `service FooService{ rpc GetSomething() }`

附：参考

1. [protobuf 代码仓库 - github.com](#)
2. [golang protobuf 代码仓库 - github.com](#)
3. [Remote procedure call 远程过程调用 - wikipedia.org](#)
4. [Groupcache Go语言版 memcached - github.com](#)
5. [Language Guide \(proto3\) 官方指南 - google.com](#)
6. [Proto Style Guide 代码风格指南 - google.com](#)
7. [Protocol Buffer 插件列表 - github.com](#)

专题: [Go 简明教程](#)

本文发表于 2020-01-11, 最后修改于 2021-02-05。

本站永久域名「[geektutu.com](#)」, 也可搜索「极客兔兔」找到我。

[上一篇](#) « [Go语言动手写Web框架 - Gee第七天 错误恢复\(Panic Recover\)](#)

[下一篇](#) » [Go RPC & TLS 鉴权简明教程](#)

赞赏支持



推荐阅读

Go 语言笔试面试题(并发编程)

发表于2020-09-05, 阅读约11分钟

动手写ORM框架 - GeeORM第五天 实现钩子(Hooks)

发表于2020-03-08, 阅读约14分钟

TensorFlow 2 中文文档 - MNIST 图像分类

发表于2019-07-09, 阅读约14分钟

#关于我 (9) #Go (48) #百宝箱 (2) #Cheat Sheet (1) #Go语言高性能编程 (20) #友链 (1) #Pandas (3)

#机器学习 (9) #TensorFlow (9) #mnist (5) #Python (10) #强化学习 (3) #OpenAI gym (4) #DQN (1)

#Q-Learning (1) #CNN (1) #TensorFlow 2 (10) #官方文档 (10) #Rust (1)

4 条评论

未登录用户 ▾



说点什么

① 支持 Markdown 语法

使用 GitHub 登录

预览



[southpolemonkey](#) 发表于 12 个月前



喜欢博主简单明了的总结



[geektutu](#) 发表于 12 个月前



[@southpolemonkey](#) 感谢你的认可, 这样写适合快速熟悉, 如果需要重度使用, 再参考官方文档就会容易多了~



[HuntSweet](#) 发表于 12 个月前



name, male, scores 是该类型的 2 个字段



geektutu 发表于 12 个月前




@HuntSweet

name, male, scores 是该类型的 2 个字段

感谢指出，今天修正~


Go 语言笔试面试题(基础语法)

6 评论 • 7天前

 **zzhaolei** —— ##### 2. Q13 如何判断 2 个字符串切片 (slice) 是相等的? 针对这个中的`b = b[:len(a)]`, 使用`go 1.15.6`编译, 已经没


Go语言动手写Web框架 - Gee第三天 路由 Router

30 评论 • 2天前

 **GaloisZhou** —— 很棒的学习资料! 有一个问题 `get /a/:b get /a/c /a/x` 也是去到 `/a/c` -


动手写分布式缓存 - GeeCache第六天 防止缓存击穿

17 评论 • 16天前

 **shiluoye** —— 这块代码真的太精巧了, 看完后大呼过瘾的感觉。`wg.Wait()`用来阻塞当前

7天用Go从零实现分布式缓存GeeCache

12 评论 • 16天前

 **longxibendi** —— 文章写的太好, 支持1k on

Gitalk Plus

© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#)  [Star](#)

👁996639 📄14427