二、Protobuf

protobuf是google旗下的一款平台无关，语言无关，可扩展的序列化结构数据格式。所以很适合用做数据存储和作为不同应用，不同语言之间相互通信的数据交换格式，只要实现相同的协议格式即同一 proto文件被编译成不同的语言版本，加入到各自的工程中去。这样不同语言就可以解析其他语言通过 protobuf序列化的数据。目前官网提供了 C++,Python,JAVA,GO等语言的支持。google在2008年7月7号将其作为开源项目对外公布。

protoBuf简介

Google Protocol Buffer(简称 Protobuf)是一种轻便高效的结构化数据存储格式，平台无关、语言无关、可扩展，可用于通讯协议和数据存储等领域。

数据交互的格式比较

数据交互xml、json、protobuf格式比较

1、json: 一般的web项目中，最流行的主要还是json。因为浏览器对于json数据支持非常好，有很多内建的函数支持。

2、xml: 在webservice中应用最为广泛，但是相比于json，它的数据更加冗余，因为需要成对的闭合标签。json使用了键值对的方式，不仅压缩了一定的数据空间，同时也具有可读性。

3、protobuf:是后起之秀，是谷歌开源的一种数据格式，适合高性能，对响应速度有要求的数据传输场景。因为profobuf是二进制数据格式，需要编码和解码。数据本身不具有可读性。因此只能反序列化之后得到真正可读的数据。

相对于其它protobuf更具有优势

1：序列化后体积相比Json和XML很小，适合网络传输

2：支持跨平台多语言

3：消息格式升级和兼容性还不错

4：序列化反序列化速度很快，快于Json的处理速速

Protobuf 的优点

Protobuf 有如 XML，不过它更小、更快、也更简单。你可以定义自己的数据结构，然后使用代码生成器生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。只需使用 Protobuf 对数据结构进行一次描述，即可利用各种不同语言或从各种不同数据流中对你的结构化数据轻松读写。

它有一个非常棒的特性，即“向后”兼容性好，人们不必破坏已部署的、依靠“老”数据格式的程序就可以对数据结构进行升级。

Protobuf 语义更清晰，无需类似 XML 解析器的东西（因为 Protobuf 编译器会将 .proto 文件编译生成对应的数据访问类以对 Protobuf 数据进行序列化、反序列化操作）。使用 Protobuf 无需学习复杂的文档对象模型，Protobuf 的编程模式比较友好，简单易学，同时它拥有良好的文档和示例，对于喜欢简单事物的人们而言，Protobuf 比其他的技术更加有吸引力。

ProtoBuf 的不足

Protobuf 与 XML 相比也有不足之处。它功能简单，无法用来表示复杂的概念。

XML 已经成为多种行业标准的编写工具，Protobuf 只是 Google 公司内部使用的工具，在通用性上还差很多。 由于文本并不适合用来描述数据结构，所以 Protobuf 也不适合用来对基于文本的标记文档（如 HTML）建模。另外，由于 XML 具有某种程度上的自解释性，它可以被人直接读取编辑，在这一点上 Protobuf 不行，它以二进制的方式存储，除非你有 .proto 定义，否则你没法直接读出 Protobuf 的任何内容。

Protobuf安装

安装protoBuf

```

#下载 protoBuf:
$ git clone https://github.com/protocolbuffers/protobuf.git
#或者直接将压缩包拖入后解压
unzip protobuf.zip


#安装依赖库
```

```
$ sudo apt-get install autoconf  automake  libtool  curl  make  g++  unzip libffi-dev -y
#安装
$ cd protobuf/
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
$ sudo ldconfig    # 刷新共享库 很重要的一步啊
#安装的时候会比较卡
#成功后需要使用命令测试
$ protoc -h

```

获取 proto包

```
#Go语言的proto API接口
$ go get  -v -u github.com/golang/protobuf/proto

```

安装protoc-gen-go插件

它是一个 go程序，编译它之后将可执行文件复制到\bin目录。

```
#安装
$ go get -v -u github.com/golang/protobuf/protoc-gen-go
#编译
$ cd $GOPATH/src/github.com/golang/protobuf/protoc-gen-go/
$ go build
#将生成的 protoc-gen-go可执行文件，放在/bin目录下
$ sudo cp protoc-gen-go /bin/
```

protobuf的语法

要想使用 protobuf必须得先定义 proto文件。所以得先熟悉 protobuf的消息定义的相关语法。

定义一个消息类型

```
syntax = "proto3";

message PandaRequest {
  string name = 1;
  int32 shengao = 2;
  repeated int32 tizhong = 3;
}
```

PandaRequest消息格式有3个字段，在消息中承载的数据分别对应于每一个字段。其中每个字段都有一个名字和一种类型。

文件的第一行指定了你正在使用proto3语法：如果你没有指定这个，编译器会使用proto2。这个指定语法行必须是文件的非空非注释的第一个行。

在上面的例子中，所有字段都是标量类型：两个整型（shengao和tizhong），一个string类型（name）。

Repeated 关键字表示重复的那么在go语言中用切片进行代表

正如上述文件格式，在消息定义中，每个字段都有唯一的一个标识符。

添加更多消息类型

在一个.proto文件中可以定义多个消息类型。在定义多个相关的消息的时候，这一点特别有用——例如，如果想定义与SearchResponse消息类型对应的回复消息格式的话，你可以将它添加到相同的.proto文件中

```
syntax = "proto3";

message PandaRequest {
  string name = 1;
  int32 shengao = 2;
  int32 tizhong = 3;
}

message PandaResponse {
 ...
}
```

添加注释

向.proto文件添加注释，可以使用C/C++/java/Go风格的双斜杠（//） 语法格式，如：

```
syntax = "proto3";
message PandaRequest {
  string name = 1;          //姓名
  int32 shengao = 2;        //身高
  int32 tizhong = 3;        //体重
}
message PandaResponse {
 ...
}
```

从.proto文件生成了什么?

当用protocol buffer编译器来运行.proto文件时，编译器将生成所选择语言的代码，这些代码可以操作在.proto文件中定义的消息类型，包括获取、设置字段值，将消息序列化到一个输出流中，以及从一个输入流中解析消息。

对C++来说，编译器会为每个.proto文件生成一个.h文件和一个.cc文件，.proto文件中的每一个消息有一个对应的类。

对Python来说，有点不太一样——Python编译器为.proto文件中的每个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（runtime）被用来创建所需的Python数据访问类。

对go来说，编译器会为每个消息类型生成了一个.pd.go文件。

标准数据类型

一个标量消息字段可以含有一个如下的类型——该表格展示了定义于.proto文件中的类型，以及与之对应的、在自动生成的访问类中定义的类型：

.proto Type	Notes	C++ Type	Python Type	Go Type
double		double	float	float64
float		float	float	float32
int32	使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用sint64替代	int32	int	int32
uint32	使用变长编码	uint32	int/long	uint32
uint64	使用变长编码	uint64	int/long	uint64
sint32	使用变长编码，这些编码在负值时比int32高效的多	int32	int	int32
sint64	使用变长编码，有符号的整型值。编码时比通常的int64高效。	int64	int/long	int64
fixed32	总是4个字节，如果数值总是比总是比228大的话，这个类型会比uint32高效。	uint32	int	uint32
fixed64	总是8个字节，如果数值总是比总是比256大的话，这个类型会比uint64高效。	uint64	int/long	uint64
sfixed32	总是4个字节	int32	int	int32
sfixed32	总是4个字节	int32	int	int32
sfixed64	总是8个字节	int64	int/long	int64
bool		bool	bool	bool
string	一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。	string	str/unicode	string
bytes	可能包含任意顺序的字节数据。	string	str	[]byte

默认值

当一个消息被解析的时候，如果被编码的信息不包含一个特定的元素，被解析的对象锁对应的域被设置位一个默认值，对于不同类型指定如下：

对于strings，默认是一个空string

对于bytes，默认是一个空的bytes

对于bools，默认是false

对于数值类型，默认是0

使用其他消息类型

你可以将其他消息类型用作字段类型。例如，假设在每一个PersonInfo消息中包含Person消息，此时可以在相同的.proto文件中定义一个Result消息类型，然后在PersonInfo消息中指定一个Person类型的字段

```
message PersonInfo {
  repeated Person info = 1;
}
message Person {
  string name = 1;
  int32 shengao = 2;
  repeated int32 tizhong = 3;
}
```

嵌套类型

你可以在其他消息类型中定义、使用消息类型，在下面的例子中，Person消息就定义在PersonInfo消息内，如：

```
message PersonInfo {
  message Person {
    string name = 1;
    int32 shengao = 2;
    repeated int32 tizhong = 3;
  }
  repeated Person info = 1;
}
```

如果你想在它的父消息类型的外部重用这个消息类型，你需要以PersonInfo.Person的形式使用它，如：

```
message PersonMessage {
  PersonInfo.Person info = 1;
}
```

当然，你也可以将消息嵌套任意多层，如：

```
message Grandpa {                  // Level 0
  message Father {  // Level 1
    message son {   // Level 2
      string name = 1;
```

```
      int32  age = 2;
    }
  }
  message Uncle {  // Level 1
    message Son {   // Level 2
      string name = 1;
      int32  age = 2;
    }
  }
}
```

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 buffer编译器将会根据所选择的不同语言生成服务接口代码及存根。如，想要定义一个RPC服务并具有一个方法，该方法能够接收 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 服务的函数名 （传入参数）返回（返回参数）
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

最直观的使用protocol buffer的RPC系统是gRPC一个由谷歌开发的语言和平台中的开源的RPC系统，gRPC在使用protocl buffer时非常有效，如果使用特殊的protocol buffer插件可以直接为您从.proto文件中产生相关的RPC代码。

如果你不想使用gRPC，也可以使用protocol buffer用于自己的RPC实现.

生成访问类（了解）

可以通过定义好的.proto文件来生成Java,Python,C++, Ruby, JavaNano, Objective-C,或者C# 代码，需要基于.proto文件运行protocol buffer编译器protoc。如果你没有安装编译器，下载安装包并遵照README安装。对于Go,你还需要安装一个特殊的代码生成器插件。你可以通过GitHub上的protobuf库找到安装过程

通过如下方式调用protocol编译器：

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR  --python_out=DST_DIR --
go_out=DST_DIR   path/to/file.proto

#--proto_path=IMPORT_PATH
#IMPORT_PATH声明了一个.proto文件所在的解析import具体目录。如果忽略该值，则使用当前目录。如果有多个目录则可以多次调用--proto_path，它们将会顺序的被访问并执行导入。
#--cpp_out  在目标目录DST_DIR中产生C++代码，可以在C++代码生成参考中查看更多。
#--python_out  在目标目录 DST_DIR 中产生Python代码，可以在Python代码生成参考中查看更多。
#--go_out=  在目标目录 DST_DIR 中产生Go代码，可以在GO代码生成参考中查看更多。
#DST_DIR文件路径
#path/to/file.proto  需要进行编译的文件名
```

测试

protobuf的使用方法是将数据结构写入到 .proto文件中，使用 protoc编译器编译(间接使用了插件）得到一个新的go包，里面包含 go中可以使用的数据结构和一些辅助方法。

编写 test.proto文件

1.$GOPATH/src/创建 myproto文件夹

```
$ cd $GOPATH/src/
$ mkdir myproto
```

2.myproto文件夹中创建 test.proto文件 (protobuf协议文件

```
$ vim test.proto
```

文件内容

```
syntax = "proto3";
package myproto;

message Test {
  string name  = 1;
  int32 stature = 2 ;
  repeated int64 weight  = 3;
  string motto = 4;
}
```

3.编译 :执行

```
$ protoc --go_out=./ *.proto
```

生成 test.pb.go文件 4.使用 protobuf做数据格式转换

```
package main

import (
    "fmt"
    "github.com/golang/protobuf/proto"
    "myproto"
)

func main() {
    test := &myproto.Test{
        Name :   "panda",
        Stature : 180,
```

```
        Weight : []int64{120,125,198,180,150,180},
        Motto : "天行健，地势坤",
    }
    //将Struct test 转换成 protobuf
    data,err:= proto.Marshal(test)
    if err!=nil{
    fmt.Println("转码失败",err)
    }
    //得到一个新的Test结构体 newTest
    newtest:= &myproto.Test{}
    //将data转换为test结构体
    err = proto.Unmarshal(data,newtest)
    if err!=nil {
    fmt.Println("转码失败",err)
    }
    fmt.Println(newtest.String())
    //得到name字段
    fmt.Println("newtest->name",newtest.GetName())
    fmt.Println("newtest->Stature",newtest.GetStature())
    fmt.Println("newtest->Weight",newtest.GetWeight())
    fmt.Println("newtest->Motto",newtest.GetMotto())
}
```