## Protobuf是什么

ProtoBuf 是一套接口描述语言（IDL）和相关工具集（主要是 protoc，基于 C++ 实现），类似 Apache 的 Thrift）。用户写好 .proto 描述文件，之后使用 protoc 可以很容易编译成众多计算机语言（C++、Java、Python、C#、Golang 等）的接口代码。

## 源代码

<https://github.com/protocolbuffers/protobuf.git>

## protocol compiler

下载地址为：https://github.com/google/protobuf/releases

## Protobuf Runtime Installation

Protobuf supports several different programming languages. For each programming language, you can find instructions in the corresponding source directory about how to install protobuf runtime for that specific language:

| **Language** | **Source** | **Ubuntu** | **MacOS** | **Windows** |
| --- | --- | --- | --- | --- |
| C++ (include C++ runtime and protoc) | [src](https://github.com/protocolbuffers/protobuf/blob/master/src) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/bazel/continuous) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/cpp/continuous) |  |
| Java | [java](https://github.com/protocolbuffers/protobuf/blob/master/java) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/java_jdk7/continuous) |  |  |
| Python | [python](https://github.com/protocolbuffers/protobuf/blob/master/python) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python27/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python33/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python34/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python35/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python36/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/python37/continuous) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/python/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/python_cpp/continuous) |  |
| Objective-C | [objectivec](https://github.com/protocolbuffers/protobuf/blob/master/objectivec) |  |  |  |
| C# | [csharp](https://github.com/protocolbuffers/protobuf/blob/master/csharp) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/csharp/continuous) |  |  |
| JavaScript | [js](https://github.com/protocolbuffers/protobuf/blob/master/js) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/javascript/continuous) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/javascript/continuous) |  |
| Ruby | [ruby](https://github.com/protocolbuffers/protobuf/blob/master/ruby) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/ruby23/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/ruby24/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/ruby25/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/ruby26/continuous) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/ruby23/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/ruby24/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/ruby25/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/ruby26/continuous) |  |
| Go | [golang/protobuf](https://github.com/golang/protobuf) |  |  |  |
| PHP | [php](https://github.com/protocolbuffers/protobuf/blob/master/php) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/php_all/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/ubuntu/32-bit/continuous) | [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/php5.6_mac/continuous) [Build status](https://fusion.corp.google.com/projectanalysis/current/KOKORO/prod:protobuf/github/master/macos/php7.0_mac/continuous) |  |
| Dart | [dart-lang/protobuf](https://github.com/dart-lang/protobuf) |  |  |  |

## Quick Start

The best way to learn how to use protobuf is to follow the tutorials in our developer guide:

<https://developers.google.com/protocol-buffers/docs/tutorials>

If you want to learn from code examples, take a look at the examples in the [examples](https://github.com/protocolbuffers/protobuf/blob/master/examples) directory.

## Documentation

The complete documentation for Protocol Buffers is available via the web at:

<https://developers.google.com/protocol-buffers/>

https://github.com/protocolbuffers/protobuf/blob/master/src/README.md

## C++ Installation - Unix

To build protobuf from source, the following tools are needed:

* autoconf
* automake
* libtool
* make
* g++
* unzip

On Ubuntu/Debian, you can install them with:

$ sudo apt-get install autoconf automake libtool curl make g++ unzip

On other platforms, please use the corresponding package managing tool to install them before proceeding.

To get the source, download one of the release .tar.gz or .zip packages in the release page:

https://github.com/protocolbuffers/protobuf/releases/latest

For example: if you only need C++, download protobuf-cpp-[VERSION].tar.gz; if you need C++ and Java, download protobuf-java-[VERSION].tar.gz (every package contains C++ source already); if you need C++ and multiple other languages, download protobuf-all-[VERSION].tar.gz.

You can also get the source by "git clone" our git repository. Make sure you have also cloned the submodules and generated the configure script (skip this if you are using a release .tar.gz or .zip package):

git clone https://github.com/protocolbuffers/protobuf.git

cd protobuf

git submodule update --init --recursive

./autogen.sh

To build and install the C++ Protocol Buffer runtime and the Protocol Buffer compiler (protoc) execute the following:

./configure

make

make check

sudo make install

sudo ldconfig # refresh shared library cache.

If "make check" fails, you can still install, but it is likely that some features of this library will not work correctly on your system. Proceed at your own risk.

For advanced usage information on configure and make, please refer to the autoconf documentation:

http://www.gnu.org/software/autoconf/manual/autoconf.html#Running-configure-Scripts

**Hint on install location**

By default, the package will be installed to /usr/local. However, on many platforms, /usr/local/lib is not part of LD\_LIBRARY\_PATH. You can add it, but it may be easier to just install to /usr instead. To do this, invoke configure as follows:

./configure --prefix=/usr

If you already built the package with a different prefix, make sure to run "make clean" before building again.

**Compiling dependent packages**

To compile a package that uses Protocol Buffers, you need to pass various flags to your compiler and linker. As of version 2.2.0, Protocol Buffers integrates with pkg-config to manage this. If you have pkg-config installed, then you can invoke it to get a list of flags like so:

pkg-config --cflags protobuf # print compiler flags

pkg-config --libs protobuf # print linker flags

pkg-config --cflags --libs protobuf # print both

For example:

c++ my\_program.cc my\_proto.pb.cc `pkg-config --cflags --libs protobuf`

Note that packages written prior to the 2.2.0 release of Protocol Buffers may not yet integrate with pkg-config to get flags, and may not pass the correct set of flags to correctly link against libprotobuf. If the package in question uses autoconf, you can often fix the problem by invoking its configure script like:

configure CXXFLAGS="$(pkg-config --cflags protobuf)" \

LIBS="$(pkg-config --libs protobuf)"

This will force it to use the correct flags.

If you are writing an autoconf-based package that uses Protocol Buffers, you should probably use the PKG\_CHECK\_MODULES macro in your configure script like:

PKG\_CHECK\_MODULES([protobuf], [protobuf])

See the pkg-config man page for more info.

If you only want protobuf-lite, substitute "protobuf-lite" in place of "protobuf" in these examples.

**Note for Mac users**

For a Mac system, Unix tools are not available by default. You will first need to install Xcode from the Mac AppStore and then run the following command from a terminal:

sudo xcode-select --install

```shell

To install Unix tools, you can install "port" following the instructions at

https://www.macports.org . This will reside in /opt/local/bin/port for most

Mac installations.

```shell

sudo /opt/local/bin/port install autoconf automake libtool

Then follow the Unix instructions above.

**Note for cross-compiling**

The makefiles normally invoke the protoc executable that they just built in order to build tests. When cross-compiling, the protoc executable may not be executable on the host machine. In this case, you must build a copy of protoc for the host machine first, then use the --with-protoc option to tell configure to use it instead. For example:

./configure --with-protoc=protoc

This will use the installed protoc (found in your $PATH) instead of trying to execute the one built during the build process. You can also use an executable that hasn't been installed. For example, if you built the protobuf package for your host machine in ../host, you might do:

./configure --with-protoc=../host/src/protoc

Either way, you must make sure that the protoc executable you use has the same version as the protobuf source code you are trying to use it with.

**Note for Solaris users**

Solaris 10 x86 has a bug that will make linking fail, complaining about libstdc++.la being invalid. We have included a work-around in this package. To use the work-around, run configure as follows:

./configure LDFLAGS=-L$PWD/src/solaris

See src/solaris/libstdc++.la for more info on this bug.

**Note for HP C++ Tru64 users**

To compile invoke configure as follows:

./configure CXXFLAGS="-O -std ansi -ieee -D\_\_USE\_STD\_IOSTREAM"

Also, you will need to use gmake instead of make.

**Note for AIX users**

Compile using the IBM xlC C++ compiler as follows:

./configure CXX=xlC

Also, you will need to use GNU make (gmake) instead of AIX make.

## C++ Installation - Windows

If you only need the protoc binary, you can download it from the release page:

https://github.com/protocolbuffers/protobuf/releases/latest

In the downloads section, download the zip file protoc-$VERSION-win32.zip. It contains the protoc binary as well as public proto files of protobuf library.

Protobuf and its dependencies can be installed directly by using vcpkg:

>vcpkg install protobuf protobuf:x64-windows

If zlib support is desired, you'll also need to install the zlib feature:

>vcpkg install protobuf[zlib] protobuf[zlib]:x64-windows

See <https://github.com/Microsoft/vcpkg> for more information.

To build from source using Microsoft Visual C++, see [cmake/README.md](https://github.com/protocolbuffers/protobuf/blob/master/cmake/README.md).

To build from source using Cygwin or MinGW, follow the Unix installation instructions, above.

## Binary Compatibility Warning

Due to the nature of C++, it is unlikely that any two versions of the Protocol Buffers C++ runtime libraries will have compatible ABIs. That is, if you linked an executable against an older version of libprotobuf, it is unlikely to work with a newer version without re-compiling. This problem, when it occurs, will normally be detected immediately on startup of your app. Still, you may want to consider using static linkage. You can configure this package to install static libraries only using:

./configure --disable-shared

## Usage

The complete documentation for Protocol Buffers is available via the web at:

https://developers.google.com/protocol-buffers/

## 静态库编译

步骤如下：

编译google protobuff时，在configure 时加上选项：  
configrue --disable-shared  
即可编译成静态库：libprotobuf.a 但是默认的configure文件中，在编译时未加-fPIC ，导致在引用静态库的工程中编译链接时报错误：  
libs/assert.o: relocation R\_X86\_64\_32 against `a local symbol' can not be used when making a shared object; recompile with -fPIC .libs/assert.o: could not read symbols: Bad value  
解决该问题，需要重新编译google protobuff库，并添加编译选项:-fPIC  
以文本形式打开google buff代码目录下的configure文件，在把第2575至2578行修改为如下：  
if test "x${ac\_cv\_env\_CFLAGS\_set}" = "x"; then :   CFLAGS="-fPIC" fi if test "x${ac\_cv\_env\_CXXFLAGS\_set}" = "x"; then :   CXXFLAGS="-fPIC"  
再次执行configure：  
configure --disable-shared  
make   
make install  
编译完成后,使用libprotobuf.a文件

## Protocol Buffers 3.0 技术手册

# proto3语法

## 定义一个 Message

首先我们来定义一个 Search 请求，在这个请求里面，我们需要给服务端发送三个信息：

query：查询条件

page\_number：你想要哪一页数据

result\_per\_page：每一页有多少条数据

于是我们可以这样定义：

// 指定使用proto3，如果不指定的话，编译器会使用proto2去编译

syntax = "proto3"; //[proto2|proto3]

message SearchRequests {

// 定义SearchRequests的成员变量，需要指定：变量类型、变量名、变量Tag

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

## 定义多个 message 类型

一个 proto 文件可以定义多个 message ，比如我们可以在刚才那个 proto 文件中把服务端返回的消息结构也一起定义：

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

message SearchResponse {

repeated string result = 1;

}

message 可以嵌套定义，比如 message 可以定义在另一个 message 内部

message SearchResponse {

message Result {

string url = 1;

string title = 2;

repeated string snippets = 3;

}

repeated Result results = 1;

}

定义在 message 内部的 message 可以这样使用：

message SomeOtherMessage {

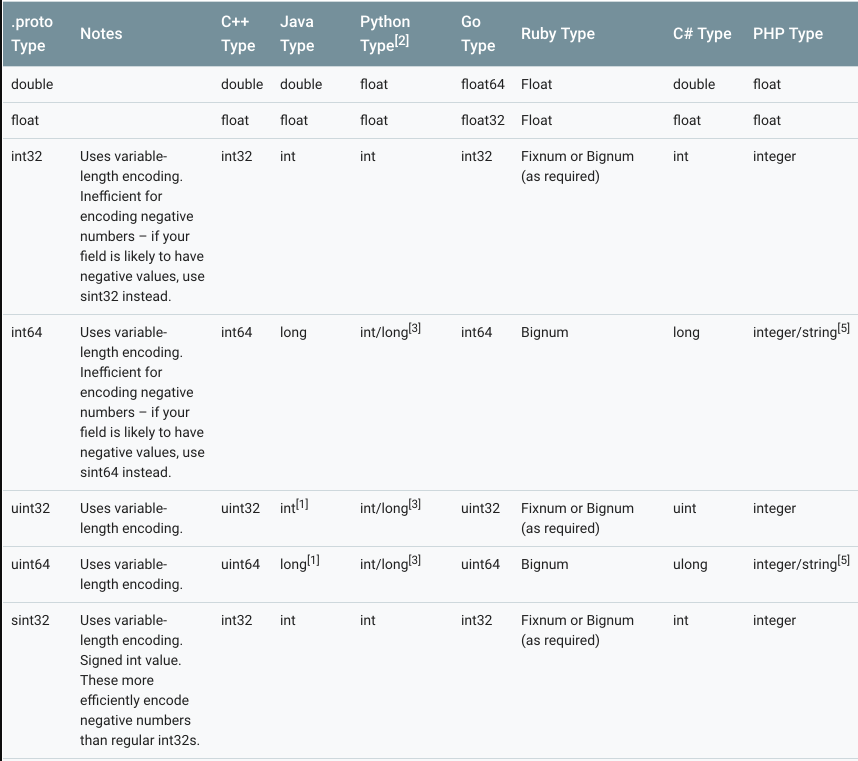
SearchResponse.Result result = 1;

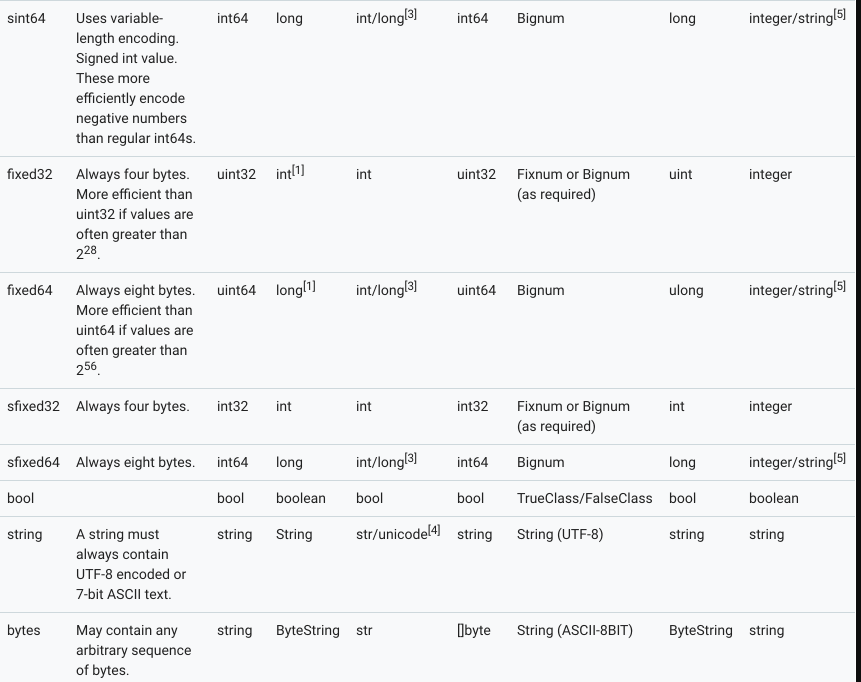
}

## 定义变量类型

在刚才的例子之中，我们使用了2个标准值类型： string 和 int32，除了这些标准类型之外，变量的类型还可以是复杂类型，比如自定义的枚举和自定义的 message

这里我们把标准类型列举一下protobuf内置的标准类型以及跟各平台对应的关系：



****

## 分配Tag

每一个变量在message内都需要自定义一个唯一的数字Tag，protobuf会根据Tag从数据中查找变量对应的位置，具体原理跟protobuf的二进制数据格式有关。Tag一旦指定，以后更新协议的时候也不能修改，否则无法对旧版本兼容。

Tag的取值范围最小是1，最大是229229-1，但 19000~19999 是 protobuf 预留的，用户不能使用。

虽然 Tag 的定义范围比较大，但不同 Tag 也会对 protobuf 编码带来一些影响：

1 ~ 15：单字节编码

16 ~ 2047：双字节编码

使用频率高的变量最好设置为1 ~ 15，这样可以减少编码后的数据大小，但由于Tag一旦指定不能修改，所以为了以后扩展，也记得为未来保留一些 1 ~ 15 的 Tag

## 指定变量规则

在 proto3 中，可以给变量指定以下两个规则：

singular：0或者1个，但不能多于1个

repeated：任意数量（包括0）

当构建 message 的时候，build 数据的时候，会检测设置的数据跟规则是否匹配

在proto2中，规则为：

required：必须有一个

optional：0或者1个

repeated：任意数量（包括0）

## 注释

用//表示注释开头，如

message SearchRequest {

string query = 1;

int32 page\_number = 2; // Which page number do we want

int32 result\_per\_page = 3; // Number of results to return per page

}

## 保留变量不被使用

上面我们说到，一旦 Tag 指定后就不能变更，这就会带来一个问题，假如在版本1的协议中，我们有个变量：

int32 number = 1；

在版本2中，我们决定废弃对它的使用，那我们应该如何修改协议呢？注释掉它？删除掉它？如果把它删除了，后来者很可能在定义新变量的时候，使新的变量 Tag = 1 ，这样会导致协议不兼容。那有没有办法规避这个问题呢？我们可以用 reserved 关键字，当一个变量不再使用的时候，我们可以把它的变量名或 Tag 用 reserved 标注，这样，当这个 Tag 或者变量名字被重新使用的时候，编译器会报错

message Foo {

// 注意，同一个 reserved 语句不能同时包含变量名和 Tag

reserved 2, 15, 9 to 11;

reserved "foo", "bar";

}

## 默认值

当解析 message 时，如果被编码的 message 里没有包含某些变量，那么根据类型不同，他们会有不同的默认值：

string：默认是空的字符串

byte：默认是空的bytes

bool：默认为false

numeric：默认为0

enums：定义在第一位的枚举值，也就是0

messages：根据生成的不同语言有不同的表现，参考generated code guide

注意，收到数据后反序列化后，对于标准值类型的数据，比如bool，如果它的值是 false，那么我们无法判断这个值是对方设置的，还是对方压根就没给这个变量设置值。

## 定义枚举 Enumerations

在 protobuf 中，我们也可以定义枚举，并且使用该枚举类型，比如：

message SearchRequest {

string query = 1;

int32 page\_number = 2; // Which page number do we want

int32 result\_per\_page = 3; // Number of results to return per page

enum Corpus {

UNIVERSAL = 0;

WEB = 1;

IMAGES = 2;

LOCAL = 3;

NEWS = 4;

PRODUCTS = 5;

VIDEO = 6;

}

Corpus corpus = 4;

}

枚举定义在一个消息内部或消息外部都是可以的，如果枚举是 定义在 message 内部，而其他 message 又想使用，那么可以通过 MessageType.EnumType 的方式引用。定义枚举的时候，我们要保证第一个枚举值必须是0，枚举值不能重复，除非使用 option allow\_alias = true 选项来开启别名。如：

enum EnumAllowingAlias {

option allow\_alias = true;

UNKNOWN = 0;

STARTED = 1;

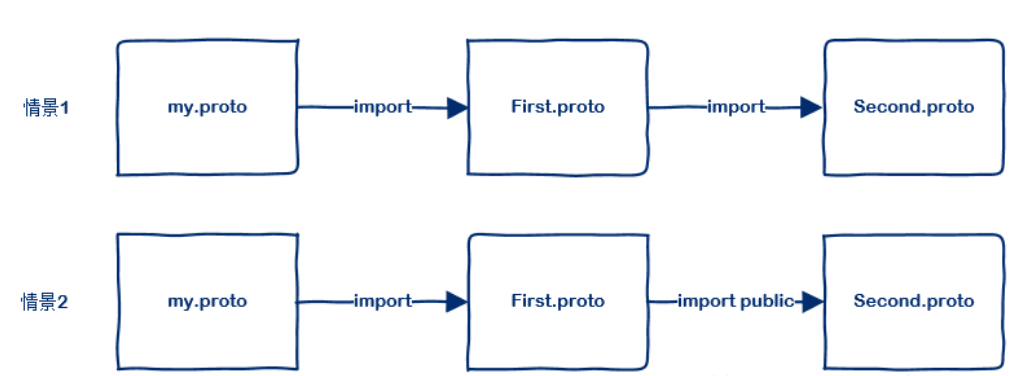
RUNNING = 1;

}

枚举值的范围是32-bit integer，但因为枚举值使用变长编码，所以不推荐使用负数作为枚举值，因为这会带来效率问题。

## 如何引用其他 proto 文件

在proto语法中，有两种引用其他 proto 文件的方法： import 和 import public，这两者有什么区别呢？下面举个例子说明：



在情景1中， my.proto 不能使用 second.proto 中定义的内容

在情景2中， my.proto 可以使用 second.proto 中定义的内容

情景1和情景2中，my.proto 都可以使用 first.proto

情景1和情景2中，first.proto 都可以使用 second.proto

// my.proto

import "first.proto";

// first.proto

//import "second.proto";

import public "second.proto";

// second.proto

...

## 升级 proto 文件正确的姿势

升级更改 proto 需要遵循以下原则

* 不要修改任何已存在的变量的 Tag
* 如果你新增了变量，新生成的代码依然能解析旧的数据，但新增的变量将会变成默认值。相应的，新代码序列化的数据也能被旧的代码解析，但旧代码会自动忽略新增的变量。
* 废弃不用的变量用 reserved 标注
* int32、 uint32、 int64、 uint64 和 bool 是相互兼容的，这意味你可以更改这些变量的类型而不会影响兼容性
* sint32 和 sint64 是兼容的，但跟其他类型不兼容
* string 和 bytes 可以兼容，前提是他们都是UTF-8编码的数据
* fixed32 和 sfixed32 是兼容的, fixed64 和 sfixed64是兼容的

## Any 的使用

Any可以让你在 proto 文件中使用未定义的类型，具体里面保存什么数据，是在上层业务代码使用的时候决定的，使用 Any 必须导入 import google/protobuf/any.proto

import "google/protobuf/any.proto";

message ErrorStatus {

string message = 1;

repeated google.protobuf.Any details = 2;

}

## Oneof 的使用

Oneof 类似union，如果你的消息中有很多可选字段，而同一个时刻最多仅有其中的一个字段被设置的话，你可以使用oneof来强化这个特性并且节约存储空间，如

message LoginReply {

oneof test\_oneof {

string name = 3;

string age = 4;

}

required string status = 1;

required string token = 2;

}

这样，name 和 age 都是 LoginReply 的成员，但不能给他们同时设置值（设置一个oneof字段会自动清理其他的oneof字段）。

## Maps 的使用

protobuf 支持定义 map 类型的成员，如：

map<key\_type, value\_type> map\_field = N;

// 举例：map<string, Project> projects = 3;

* key\_type:必须是string或者int
* value\_type：任意类型

使用 map 要注意：

* Map 类型不能使 repeated
* Map 是无序的
* 以文本格式展示时，Map 以 key 来排序
* 如果有相同的键会导致解析失败

## Packages 的使用

为了防止不同消息之间的命名冲突，你可以对特定的.proto文件提指定 package 名字。在定义消息的成员的时候，可以指定包的名字：

package foo.bar;

message Open { ... }

message Foo {

...

// 带上包名

foo.bar.Open open = 1;

...

}

## Options

Options 分为 file-level options（只能出现在最顶层，不能在消息、枚举、服务内部使用）、 message-level options（只能在消息内部使用）、field-level options（只能在变量定义时使用）

* java\_package (file option)：指定生成类的包名，如果没有指定此选项，将由关键字package指定包名。此选项只在生成 java 代码时有效
* java\_multiple\_files (file option)：如果为 true， 定义在最外层的 message 、enum、service 将作为单独的类存在
* java\_outer\_classname (file option)：指定最外层class的类名，如果不指定，将会以文件名作为类名
* optimize\_for (file option)：可选有 [SPEED|CODE\_SIZE|LITE\_RUNTIME] ，分别是效率优先、空间优先，第三个lite是兼顾效率和代码大小，但是运行时需要依赖 libprotobuf-lite
* cc\_enable\_arenas (file option):启动arena allocation，c++代码使用
* objc\_class\_prefix (file option)：Objective-C使用
* deprecated (field option)：提示变量已废弃、不建议使用

option java\_package = "com.example.foo";

option java\_multiple\_files = true;

option java\_outer\_classname = "Ponycopter";

option optimize\_for = CODE\_SIZE;

int32 old\_field = 6 [deprecated=true];

## 定义 Services

这个其实和gRPC相关，详细可参考：gRPC， 这里做一个简单的介绍

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

## 简单RPC

客户端使用 Stub 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，这是一个阻塞型的调用

// Obtains the feature at a given position.

rpc GetFeature(Point) returns (Feature) {}

## 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) {}

## 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) {}

## 双向流式 RPC

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) {}

## 代码生成

使用 protoc 工具可以把编写好的 proto 文件“编译”为Java, Python, C++, Go, Ruby, JavaNano, Objective-C,或C#代码， protoc 可以从点击这里进行下载。protoc 的使用方式如下：

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

参数说明：

IMPORT\_PATH：指定 proto 文件的路径，如果没有指定， protoc 会从当前目录搜索对应的 proto 文件，如果有多个路径，那么可以指定多次--proto\_path

指定各语言代码的输出路径

* –cpp\_out：生成c++代码
* java\_out ：生成java代码
* python\_out ：生成python代码
* go\_out ：生成go代码
* ruby\_out ：生成ruby代码
* javanano\_out ：适合运行在有资源限制的平台（如Android）的java代码
* objc\_out ：生成 Objective-C代码
* csharp\_out ：生成C#代码
* php\_out ：生成PHP代码

# 示例

高性能C++ IoC网络框架Nebula <https://github.com/Bwar/Nebula>

<https://github.com/Bwar/Nebula/blob/master/src/codec/CodecProto.cpp>

## TCP通讯协议设计

Bwar设计用于IM系统的协议包括两个protobuf message：MsgHead和MsgBody

同时满足了埋点数据采集、实时计算、消息推送等业务需要，更为通用。正因其通用性和高扩展性，采用proactor模型的IoC网络框架 Nebula 才会选用这个协议，通过这个协议，框架层将网络通信工作从业务应用中完全独立出来，基于Nebula框架的应用开发者甚至可以不懂网络编程也能开发出高并发的分布式服务。

MsgHead和MsgBody的protobuf定义如下：

syntax = "proto3";

// import "google/protobuf/any.proto";

/\*\*

\* @brief 消息头

\* @note MsgHead为固定15字节的头部，当MsgHead不等于15字节时，消息发送将出错。

\* 在proto2版本，MsgHead为15字节总是成立，cmd、seq、len都是required；

\* 但proto3版本，MsgHead为15字节则必须要求cmd、seq、len均不等于0，否则无法正确进行收发编解码。

\*/

message MsgHead

{

fixed32 cmd = 1; ///< 命令字（压缩加密算法占高位1字节）

fixed32 seq = 2; ///< 序列号

sfixed32 len = 3; ///< 消息体长度

}

/\*\*

\* @brief 消息体

\* @note 消息体主体是data，所有业务逻辑内容均放在data里。req\_target是请求目标，用于

\* 服务端接入路由，请求包必须填充。rsp\_result是响应结果，响应包必须填充。

\*/

message MsgBody

{

oneof msg\_type

{

Request req\_target = 1; ///< 请求目标（请求包必须填充）

Response rsp\_result = 2; ///< 响应结果（响应包必须填充）

}

bytes data = 3; ///< 消息体主体

bytes add\_on = 4; ///< 服务端接入层附加在请求包的数据（客户端无须理会）

string trace\_id = 5; ///< for log trace

message Request

{

uint32 route\_id = 1; ///< 路由ID

string route = 2; ///< 路由ID（当route\_id用整型无法表达时使用）

}

message Response

{

int32 code = 1; ///< 错误码

bytes msg = 2; ///< 错误信息

}

}

MsgBody的data字段存储消息主体，任何自定义数据均可以二进制数据流方式写入到data。

  msg\_type用于标识该消息是请求还是响应（所有网络数据流都可归为请求或响应），如果是请求，则可以选择性填充Request里的route\_id或route，如果填充了，则框架层无须解析应用层协议（也无法解析）就能自动根据路由ID转发，而无须应用层解开data里的内容再根据自定义逻辑转发。如果是响应，则定义了统一的错误标准，也为业务无关的错误处理提供方便。

  add\_on是附在长连接上的业务数据，框架并不会解析但会在每次转发消息时带上，可以为应用提供极其方便且强大的功能。比如，IM用户登录时客户端只发送用户ID和密码到服务端，服务端在登录校验通过后，将该用户的昵称、头像等信息通过框架提供的方法SetClientData()将数据附在服务端接入层该用户对应的长连接Channel上，之后所有从该连接过来的请求都会由框架层自动填充add\_on字段，服务端的其他逻辑服务器只从data中得到自定义业务逻辑（比如聊天消息）数据，却可以从add\_on中得到这个发送用户的信息。add\_on的设计简化了应用开发逻辑，并降低了客户端与服务端传输的数据量。

  trace\_id用于分布式日志跟踪。分布式服务的错误定位是相当麻烦的，Nebula分布式服务解决方案提供了日志跟踪功能，协议里的trace\_id字段的设计使得Nebula框架可以在完全不增加应用开发者额外工作的情况下（正常调用LOG4\_INFO写日志而无须额外工作）实现所有标记着同一trace\_id的日志发送到指定一台日志服务器，定义错误时跟单体服务那样登录一台服务器查看日志即可。比如，IM用户发送一条消息失败，在用户发送的消息到达服务端接入层时就被打上了trace\_id标记，这个id会一直传递到逻辑层、存储层等，哪个环节发生了错误都可以从消息的发送、转发、处理路径上查到。

  MsgHead和MsgBody的编解码实现见Nebula框架的 https://github.com/Bwar/Nebula/blob/master/src/codec/CodecProto.cpp 。

## Http通讯协议设计

上面的讲解的是protobuf应用于TCP数据流通信，接下来将描述protobuf在http通信上的应用。

  在Web服务中通常会用Nginx做接入层的反向代理，经过Nginx转发到后续业务逻辑层的tomcat、apache或nginx上，接入层和业务逻辑层至少做了两次http协议解析，http协议是文本协议，传输数据量大解析速度慢。Nebula框架不是一个web服务器，但支持http协议，在只需提供http接口的应用场景（比如完全前后端分离的后端）基于Nebula的单进程http服务端并发量就可以是tomcat的数十倍。这一定程度上得益于Nebula框架在http通信上protobuf的应用。Nebula框架解析http文本协议并转化为HttpMsg在服务内部处理，应用开发者填充HttpMsg，接入层将响应的HttpMsg转换成http文本协议发回给请求方，不管服务端内部经过多少次中转，始终只有一次http协议的decode和一次http协议的encode。

syntax = "proto3";

message HttpMsg

{

int32 type = 1; ///< http\_parser\_type 请求或响应

int32 http\_major = 2; ///< http大版本号

int32 http\_minor = 3; ///< http小版本号

int32 content\_length = 4; ///< 内容长度

int32 method = 5; ///< 请求方法

int32 status\_code = 6; ///< 响应状态码

int32 encoding = 7; ///< 传输编码（只在encode时使用，当 Transfer-Encoding: chunked 时，用于标识chunk序号，0表示第一个chunk，依次递增）

string url = 8; ///< 地址

map<string, string> headers = 9; ///< http头域

bytes body = 10; ///< 消息体（当 Transfer-Encoding: chunked 时，只存储一个chunk）

map<string, string> params = 11; ///< GET方法参数，POST方法表单提交的参数

Upgrade upgrade = 12; ///< 升级协议

float keep\_alive = 13; ///< keep alive time

string path = 14; ///< Http Decode时从url中解析出来，不需要人为填充（encode时不需要填）

bool is\_decoding = 15; ///< 是否正在解码（true 正在解码， false 未解码或已完成解码）

message Upgrade

{

bool is\_upgrade = 1;

string protocol = 2;

}

}

HttpMsg的编解码实现见Nebula框架的 https://github.com/Bwar/Nebula/blob/master/src/codec/CodecHttp.cpp

## 数据库代理服务协议设计

如果上面描述的protobuf在网络通信上应用算不错的话，那以下将protobuf用于数据代理上的协议设计则绝对是让人眼前一亮。 有的公司规定web服务不得直接访问MySQL数据库，甚至不允许在web逻辑层拼接SQL语句。如果有这种出于安全性考虑而做的限制，在web逻辑层后面再增加一层业务逻辑层成本未免太高了，那么解决办法应该是增加一层业务逻辑无关的代理服务层。这个代理服务层不是简单的转发SQL语句这么简单，因为web逻辑层可能不允许拼接SQL，由此引出我们这个用于数据库代理的protobuf协议设计。这个协议是将SQL逻辑融入整个协议之中，数据库代理层接收并解析这个协议后生成SQL语句或用binding方式到数据库去执行。数据库代理层只有协议解析和转化逻辑，无其他任何业务逻辑，业务逻辑还在web逻辑层，区别只在于从拼接SQL变成了填充protobuf协议。

syntax = "proto2";

package dbagent;

/\*\*

\* @brief DB Agent消息

\*/

message DbAgentMsg

{

enum E\_TYPE

{

UNDEFINE = 0; ///< 未定义

REQ\_CONNECT = 1; ///< 连接DB请求

RSP\_CONNECT = 2; ///< 连接DB响应

REQ\_QUERY = 3; ///< 执行SQL请求

RSP\_QUERY = 4; ///< 执行SQL响应

REQ\_DISCONNECT = 5; ///< 关闭连接请求

RSP\_DISCONNECT = 6; ///< 关闭连接响应

RSP\_RECORD = 7; ///< 结果集记录

RSP\_COMMON = 8; ///< 通用响应（当请求不能被Server所认知时会做出这个回应）

REQ\_GET\_CONNECT = 9; ///< 获取连接请求

RSP\_GET\_CONNECT = 10; ///< 获取连接响应

}

required E\_TYPE type = 1; ///< 消息/操作 类型

optional RequestConnect req\_connect = 2; ///< 连接请求

optional ResponseConnect rsp\_connect = 3; ///< 连接响应

optional RequestDisconnect req\_disconnect = 4; ///< 关闭请求

optional ResponseDisconnect rsp\_disconnect = 5; ///< 关闭响应

optional RequestQuery req\_query = 6; ///< 执行SQL请求

optional ResponseQuery rsp\_query = 7; ///< 执行SQL响应

optional ResponseRecord rsp\_record = 8; ///< SELECT结果集记录

optional ResponseCommon rsp\_common = 9; ///< 通用响应

optional RequestGetConnection req\_get\_conn = 10; ///< 获取连接请求

optional ResponseGetConnection rsp\_get\_conn = 11; ///< 获取连接响应

}

/\*\*

\* @brief 连接请求

\*/

message RequestConnect

{

required string host = 1; ///< DB所在服务器IP

required int32 port = 2; ///< DB端口

required string user = 3; ///< DB用户名

required string password = 4; ///< DB用户密码

required string dbname = 5; ///< DB库名

required string charset = 6; ///< DB字符集

}

/\*\*

\* @brief 连接响应

\*/

message ResponseConnect

{

required int32 connect\_id = 1; ///< 连接ID （连接失败时，connect\_id为0）

optional int32 err\_no = 2; ///< 错误码 0 表示连接成功

optional string err\_msg = 3; ///< 错误信息

}

/\*\*

\* @brief 关闭连接请求

\*/

message RequestDisconnect

{

required int32 connect\_id = 1; ///< 连接ID （连接失败时，connect\_id为0）

}

/\*\*

\* @brief 关闭连接响应

\*/

message ResponseDisconnect

{

optional int32 err\_no = 2; ///< 错误码 0 表示连接成功

optional string err\_msg = 3; ///< 错误信息

}

/\*\*

\* @brief 执行SQL请求

\*/

message RequestQuery

{

required E\_QUERY\_TYPE query\_type = 1; ///< 查询类型

required string table\_name = 2; ///< 表名

repeated Field fields = 3; ///< 列类型

repeated ConditionGroup conditions= 4; ///< where条件组（由group\_relation指定，若不指定则默认为AND关系）

repeated string groupby\_col = 5; ///< group by字段

repeated OrderBy orderby\_col = 6; ///< order by字段

optional uint32 limit = 7; ///< 指定返回的行数的最大值 (limit 200)

optional uint32 limit\_from = 8; ///< 指定返回的第一行的偏移量 (limit 100, 200)

optional ConditionGroup.E\_RELATION group\_relation = 9; ///< where条件组的关系,条件组之间有且只有一种关系（and或者or）

optional int32 connect\_id = 10; ///< 连接ID，有效连接ID（长连接，当connect后多次执行query可以使用connect\_id）

optional string bid = 11; ///< 业务ID，在CmdDbAgent.json配置文件中配置（短连接，每次执行query时连接，执行完后关闭连接）

optional string password = 12; ///< 业务密码

enum E\_QUERY\_TYPE ///< 查询类型

{

SELECT = 0; ///< select查询

INSERT = 1; ///< insert插入

INSERT\_IGNORE = 2; ///< insert ignore插入，若存在则放弃

UPDATE = 3; ///< update更新

REPLACE = 4; ///< replace覆盖插入

DELETE = 5; ///< delete删除

}

enum E\_COL\_TYPE ///< 列类型

{

STRING = 0; ///< char, varchar, text, datetime, timestamp等

INT = 1; ///< int

BIGINT = 2; ///< bigint

FLOAT = 3; ///< float

DOUBLE = 4; ///< double

}

message Field ///< 字段

{

required string col\_name = 1; ///< 列名

required E\_COL\_TYPE col\_type = 2; ///< 列类型

required bytes col\_value = 3; ///< 列值

optional string col\_as = 4; ///< as列名

}

message Condition ///< where条件

{

required E\_RELATION relation = 1; ///< 关系（=, !=, >, <, >=, <= 等）

required E\_COL\_TYPE col\_type = 2; ///< 列类型

required string col\_name = 3; ///< 列名

repeated bytes col\_values = 4; ///< 列值（当且仅当relation为IN时值的个数大于1有效）

optional string col\_name\_right= 5; ///< 关系右边列名（用于where col1=col2这种情况）

enum E\_RELATION

{

EQ = 0; ///< 等于=

NE = 1; ///< 不等于!=

GT = 2; ///< 大于>

LT = 3; ///< 小于<

GE = 4; ///< 大于等于>=

LE = 5; ///< 小于等于<=

LIKE = 6; ///< like

IN = 7; ///< in (1, 2, 3, 4, 5)

}

}

message ConditionGroup ///< where条件组合

{

required E\_RELATION relation = 1; ///< 条件之间的关系，一个ConditionGroup里的所有Condition之间有且只有一种关系（and或者or）

repeated Condition condition = 2; ///< 条件

enum E\_RELATION

{

AND = 0; ///< and且

OR = 1; ///< or或

}

}

message OrderBy

{

required E\_RELATION relation = 1; ///< 降序或升序

required string col\_name = 2; ///< 列名

enum E\_RELATION

{

ASC = 0;

DESC = 1;

}

}

}

/\*\*

\* @brief 执行SQL响应

\*/

message ResponseQuery

{

required uint32 seq = 1; ///< 数据包序列号（SELECT结果会分包返回，只有一个包的情况或已到达最后一个包则seq=0xFFFFFFFF）

required int32 err\_no = 2; ///< 错误码，0 表示执行成功

optional string err\_msg = 3; ///< 错误信息

optional uint64 insert\_id = 4; ///< mysql\_insert\_id()获取的值（视执行的SQL语句而定，不一定存在）

repeated bytes dict = 5; ///< 结果集字典（视执行的SQL语句而定，不一定存在）

}

/\*\*

\* @brief SELECT语句返回结果集的一条记录

\*/

message ResponseRecord

{

required uint32 seq = 1; ///< 数据包序列号（SELECT结果会分包返回，已到达最后一个包则seq=0xFFFFFFFF）

repeated bytes field = 2; ///< 数据集记录的字段

}

/\*\*

\* @brief 常规响应

\*/

message ResponseCommon

{

optional int32 err\_no = 1; ///< 错误码 0 表示连接成功

optional string err\_msg = 2; ///< 错误信息

}

/\*\*

\* @brief 获取连接请求

\*/

message RequestGetConnection

{

required string bid = 1; ///< 业务ID，在dbproxy配置文件中配置

required string password = 2; ///< 业务密码

}

/\*\*

\* @brief 获取连接响应

\*/

message ResponseGetConnection

{

required int32 connect\_id = 1; ///< 连接ID，有效连接ID，否则执行失败

optional int32 err\_no = 2; ///< 错误码 0 表示连接成功

optional string err\_msg = 3; ///< 错误信息

}

## Redis和MySQL数据代理协议设计

 大部分后台应用只有MySQL是不够的，往往还需要缓存，经常会用Redis来做数据缓存。用缓存意味着数据至少需要同时写到Redis和MySQL，又或者在未命中缓存时从MySQL中读取到的数据需要回写到Redis，这些通常都是由业务逻辑层来做的。也有例外，Nebula提供的分布式解决方案是由数据代理层来做的，业务逻辑层只需向数据代理层发送一个protobuf协议数据，数据代理层就会完成Redis和MySQL双写或缓存未命中时的自动回写（暂且不探讨数据一致性问题）。数据代理层来做这些工作是为了减少业务逻辑层的复杂度，提高开发效率。既然是为了提高开发效率，就得让业务逻辑层低于原来同时操作Redis和MySQL的开发量。

这个同时操作Redis和MySQL的数据代理协议如下：

syntax = "proto3";

package neb;

message Mydis

{

uint32 section\_factor = 1;

RedisOperate redis\_operate = 2;

DbOperate db\_operate = 3;

message RedisOperate

{

bytes key\_name = 1;

string redis\_cmd\_read = 2;

string redis\_cmd\_write = 3;

OPERATE\_TYPE op\_type = 4;

repeated Field fields = 5;

int32 key\_ttl = 6;

int32 redis\_structure = 7; ///< redis数据类型

int32 data\_purpose = 8; ///< 数据用途

bytes hash\_key = 9; ///< 可选hash key，当has\_hash\_key()时用hash\_key来计算hash值，否则用key\_name来计算hash值

enum OPERATE\_TYPE

{

T\_READ = 0;

T\_WRITE = 1;

}

}

message DbOperate

{

E\_QUERY\_TYPE query\_type = 1; ///< 查询类型

string table\_name = 2; ///< 表名

repeated Field fields = 3; ///< 列类型

repeated ConditionGroup conditions = 4; ///< where条件组（由group\_relation指定，若不指定则默认为AND关系）

repeated string groupby\_col = 5; ///< group by字段

repeated OrderBy orderby\_col = 6; ///< order by字段

ConditionGroup.E\_RELATION group\_relation = 7; ///< where条件组的关系,条件组之间有且只有一种关系（and或者or）

uint32 limit = 8; ///< 指定返回的行数的最大值 (limit 200)

uint32 limit\_from = 9; ///< 指定返回的第一行的偏移量 (limit 100, 200)

uint32 mod\_factor = 10; ///< 分表取模因子，当这个字段没有时使用section\_factor

enum E\_QUERY\_TYPE ///< 查询类型

{

SELECT = 0; ///< select查询

INSERT = 1; ///< insert插入

INSERT\_IGNORE = 2; ///< insert ignore插入，若存在则放弃

UPDATE = 3; ///< update更新

REPLACE = 4; ///< replace覆盖插入

DELETE = 5; ///< delete删除

}

message Condition ///< where条件

{

E\_RELATION relation = 1; ///< 关系（=, !=, >, <, >=, <= 等）

E\_COL\_TYPE col\_type = 2; ///< 列类型

string col\_name = 3; ///< 列名

repeated bytes col\_values = 4; ///< 列值（当且仅当relation为IN时值的个数大于1有效）

string col\_name\_right = 5; ///< 关系右边列名（用于where col1=col2这种情况）

enum E\_RELATION

{

EQ = 0; ///< 等于=

NE = 1; ///< 不等于!=

GT = 2; ///< 大于>

LT = 3; ///< 小于<

GE = 4; ///< 大于等于>=

LE = 5; ///< 小于等于<=

LIKE = 6; ///< like

IN = 7; ///< in (1, 2, 3, 4, 5)

}

}

message ConditionGroup ///< where条件组合

{

E\_RELATION relation = 1; ///< 条件之间的关系，一个ConditionGroup里的所有Condition之间有且只有一种关系（and或者or）

repeated Condition condition = 2; ///< 条件

enum E\_RELATION

{

AND = 0; ///< and且

OR = 1; ///< or或

}

}

message OrderBy

{

E\_RELATION relation = 1; ///< 降序或升序

string col\_name = 2; ///< 列名

enum E\_RELATION

{

ASC = 0;

DESC = 1;

}

}

}

}

enum E\_COL\_TYPE ///< 列类型

{

STRING = 0; ///< char, varchar, text, datetime, timestamp等

INT = 1; ///< int

BIGINT = 2; ///< bigint

FLOAT = 3; ///< float

DOUBLE = 4; ///< double

}

message Record

{

repeated Field field\_info = 1; ///< value data

}

message Field ///< 字段

{

string col\_name = 1; ///< 列名

E\_COL\_TYPE col\_type = 2; ///< 列类型

bytes col\_value = 3; ///< 列值

string col\_as = 4; ///< as列名

}

/\*\*

\* @brief 查询结果

\* @note 适用于Redis返回和MySQL返回，当totalcount与curcount相等时表明数据已接收完毕，

\* 否则表示数据尚未接收完，剩余的数据会在后续数据包继续返回。

\*/

message Result

{

int32 err\_no = 1;

bytes err\_msg = 2;

int32 total\_count = 3;

int32 current\_count = 4;

repeated Record record\_data = 5;

int32 from = 6; ///< 数据来源 E\_RESULT\_FROM

DataLocate locate = 7; ///< 仅在DataProxy使用

enum E\_RESULT\_FROM

{

FROM\_DB = 0;

FROM\_REDIS = 1;

}

message DataLocate

{

uint32 section\_from = 1;

uint32 section\_to = 2; ///< 数据所在分段，section\_from < MemOperate.section\_factor <= section\_to

uint32 hash = 3; ///< 用于做分布的hash值（取模运算时，为取模后的结果）

uint32 divisor = 4; ///< 取模运算的除数（一致性hash时不需要）

}

}

 这个协议分了Redis和MySQL两部分数据，看似业务逻辑层把一份数据填充了两份并没有降低多少开发量，实际上这两部分数据有许多是可共用的，只要提供一个填充类就可以大幅降低协议填充开发量。

## 完整示例

在项目中使用protocol buffers需要经历如下三步：

* + 定义消息格式文件，最好以proto作为后缀名
  + 使用Google提供的protocol buffers编译器来生成代码文件，一般为.h和.cc文件，主要是对消息格式以特定的语言方式描述
  + 使用protocol buffers库提供的API来编写应用程序

定义game.proto结构：

syntax = "proto3";

package pt;

option optimize\_for = LITE\_RUNTIME;

message req\_login

{

string username = 1;

string password = 2;

}

message obj\_user\_info

{

string nickname = 1; 　　//昵称

string icon = 2; //头像

int64 coin = 3; //金币

string location = 4; //所属地

}

//游戏数据统计

message obj\_user\_game\_record

{

string time = 1;

int32 kill = 2; //击杀数

int32 dead = 3; //死亡数

int32 assist= 4; //助攻数

}

message rsp\_login

{

enum RET {

SUCCESS = 0;

ACCOUNT\_NULL = 1; //账号不存在

ACCOUNT\_LOCK = 2; //账号锁定

PASSWORD\_ERROR = 3; //密码错误

ERROR = 10;

}

int32 ret = 1;

obj\_user\_info user\_info = 2;

repeated obj\_user\_game\_record record = 3;

}

生成目标语言代码

下面的命令帮助我们将game.proto文件中定义的Protocol Buffer格式的消息编译成C++代码文件。

//SRC\_DIR .proto文件存放目录

//--cpp\_out 指示编译器生成C++代码,DST\_DIR为生成文件存放目录

//game.proto 待编译的协议文件

protoc -I=$SRC\_DIR --cpp\_out=$DST\_DIR $SRC\_DIR/game.proto

由于我们在game.proto文件中定义选项optimize\_for=LITE\_RUNTIME，因此由该文件内生成的所有C++类的父类均为::google::protobuf::MessageLite，而非::google::protobuf::Message。MessageLite类是Message的父类，MessageLite中缺少对反射的支持，而此类功能均在Message类中提供了具体的实现。

使用LITE版本的Protocol Buffer。这样不仅可以得到更高编码效率，而且生成代码编译后所占用的资源也会更少，至于反射所能带来的灵活性和极易扩展性，对于该项目而言完全可以忽略。

Protobuf读和写

//game.cpp

#include <iostream>

#include <string>

#include "game.pb.h"

int main()

{

pt::rsp\_login rsp{};

rsp.set\_ret(pt::rsp\_login\_RET\_SUCCESS);

auto user\_info = rsp.mutable\_user\_info();

user\_info->set\_nickname("dsw");

user\_info->set\_icon("345DS55GF34D774S");

user\_info->set\_coin(2000);

user\_info->set\_location("zh");

for (int i = 0; i < 5; i++) {

auto record = rsp.add\_record();

record->set\_time("2019/9/4 12:13:14");

record->set\_kill(i \* 4);

record->set\_dead(i \* 2);

record->set\_assist(i \* 5);

}

std::string buff{};

rsp.SerializeToString(&buff);

//------------------解析----------------------

pt::rsp\_login rsp2{};

if (!rsp2.ParseFromString(buff)) {

std::cout << "parse error\n";

}

auto temp\_user\_info = rsp2.user\_info();

std::cout << "nickname:" << temp\_user\_info.nickname() << std::endl;

std::cout << "coin:" << temp\_user\_info.coin() << std::endl;

for (int m = 0; m < rsp2.record\_size(); m++) {

auto temp\_record = rsp2.record(m);

std::cout << "time:" << temp\_record.time() << " kill:" << temp\_record.kill() << " dead:" << temp\_record.dead() << " assist:" << temp\_record.assist() << std::endl;

}

}

[root@node2 test]# g++ -std=c++11 -g -Wall -o game.out game.cpp game.pb.cc -lprotobuf

[root@node2 test]# ./game.out

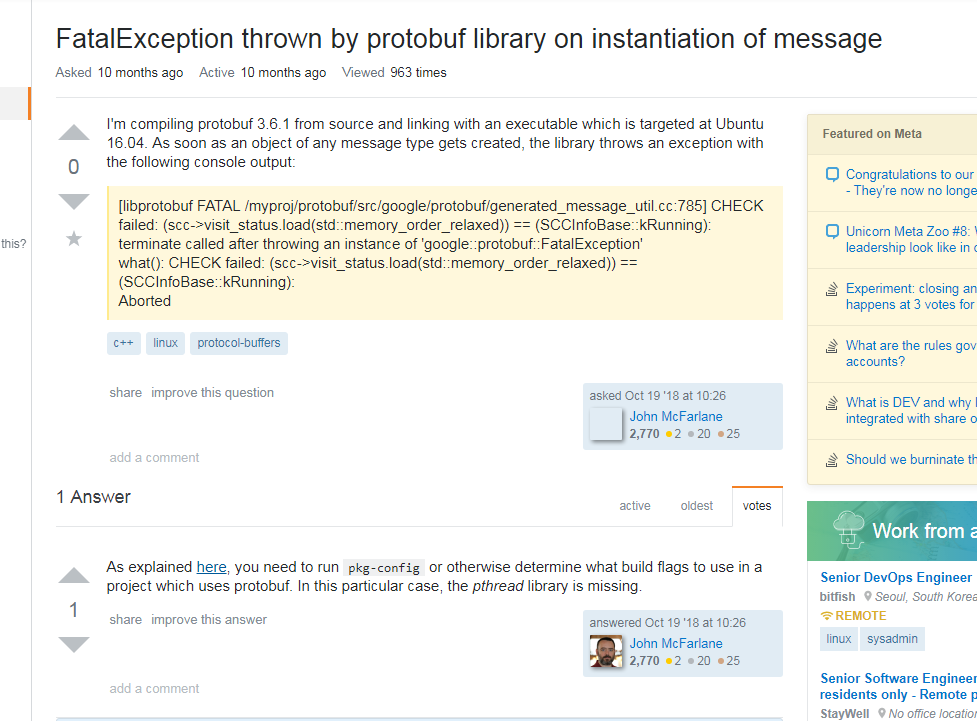
[libprotobuf FATAL google/protobuf/generated\_message\_util.cc:818] CHECK failed: (scc->visit\_status.load(std::memory\_order\_relaxed)) == (SCCInfoBase::kRunning):

terminate called after throwing an instance of 'google::protobuf::FatalException'

what(): CHECK failed: (scc->visit\_status.load(std::memory\_order\_relaxed)) == (SCCInfoBase::kRunning):

Aborted

问题：<https://stackoverflow.com/questions/52890529/fatalexception-thrown-by-protobuf-library-on-instantiation-of-message>



[root@node2 test]# g++ -std=c++11 -g -Wall -o game.out game.cpp game.pb.cc -lprotobuf -lpthread

[root@node2 test]# ./game.out

nickname:dsw

coin:2000

time:2017/4/13 12:22:11 kill:0 dead:0 assist:0

time:2017/4/13 12:22:11 kill:4 dead:2 assist:5

time:2017/4/13 12:22:11 kill:8 dead:4 assist:10

time:2017/4/13 12:22:11 kill:12 dead:6 assist:15

time:2017/4/13 12:22:11 kill:16 dead:8 assist:20

# protobuf实现反射

反射一个应用：<https://github.com/shafreeck/pb2json>

## 何谓反射

所谓反射机制，就是能够在运行时知道任意类的所有属性和方法，能够调用任意对象的任意方法和属性。这种动态获取的信息以及动态调用对象方向的功能称为反射机制。

不像Jave等语言，C++本身没有反射机制，在使用C++版本Protobuf时，protobuf通过proto文件产生响应的message和service，protobuf可以通过proto文件提供反射机制，程序在运行时可以通过proto获取任意message和任意service的属性和方法，也可以在运行时调用message的属性和方法。

## 获取message和service的属性和方法

protobuf通过Descriptor获取任意message或service的属性和方法，Descriptor主要包括了一下几种类型：

|  |  |
| --- | --- |
| FileDescriptor | 获取Proto文件中的Descriptor和ServiceDescriptor |
| Descriptor | 获取类message属性和方法，包括FieldDescriptor和EnumDescriptor |
| FieldDescriptor | 获取message中各个字段的类型、标签、名称等 |
| EnumDescriptor | 获取Enum中的各个字段名称、值等 |
| ServiceDescriptor | 获取service中的MethodDescriptor |
| MethodDescriptor | 获取各个RPC中的request、response、名称等 |

也就是说，如果能够获取到proto文件的FileDescriptor，就能获取proto文件中的所有的内容。那么如何获取proto文件的FileDescriptor呢？protobuf提供两种方法。

* 使用protoc将proto文件生成.h和.cc文件

这种法法直接根据生成的类来获取响应的FileDescriptor，比如现在有test.proto文件，那么可以通过DescriptorPool::generated\_pool()获取到其FileDescriptor

const FileDescriptor\* fileDescriptor = DescriptorPool::generated\_pool()->FindFileByName(file);

并且对于任意的message和service都可以根据其名称，通过DescriptorPool对应的Descriptor和ServiceDescriptor。

* 只使用proto文件，不使用protoc进行编译

这种情况需要手动解析proto文件，再获取FileDescriptor。protobuf提供了响应的解析器compiler，通过compoiler可以方便的获取proto文件的FileDescriptor

const FileDescriptor\* GetFileDescriptorFromProtoFile(const std::string &protoRoot, const std::string &protoFile){

compiler::DeskSourceTree sourceTree;

sourceTree.MapPath("", protoRoot);

FileErrorCollector errorCollector;

compiler::Importer importer(&sourceTree, &errorCollector);

return importer.Import(protoFile);

}

## 调用message的属性和方法

首先获取message的对象：

const Message\* prototype = MessageFactory::generated\_factory()->GetPrototype(descriptor);

if(!prototype){

continue;

}

Message \*message = prototype->New();

有了message对象，因为所有的message都是Message\* 类型的，但不同的message对象的属相和方法是不一样的，所以并不能直接调用其对象和方法。

protobuf是通过Reflection来调用message的属性和方法的。message中的方法只有对各个属性的get和set，而调用message的属性其实也就是调用属性的get。调用message的某一个属性的get和set，就需要该属性的Descriptor，通过Reflection将相应的值写入到message对应的属性：

const FieldDescriptor \* field\_descriptor = message\_descriptor->field(i);

if(field\_descriptor->type() == FieldDescriptor::TYPE\_INT32){

reflection->SetInt32(message, field\_descriptor, 0);

}

## Protobuf反射使用案例

这部分主要是重复上两节的内容，进行完整示例。以下是使用的test.proto文件

package T;

message Test

{

optional int32 id = 1;

}

IDL 编译为cpp

protoc -I=./ --cpp\_out=./ ./test.proto

* 通过类型名字创建出类型对象

预编译好proto模式

//reflect\_test1.cpp

//! 利用类型名字构造对象.

/\*!

\* @Param type\_name 类型名字，比如 "Test.TestMessage".

\* @Return 对象指针，new 出来的，使用者负责释放.

\*/

#include <google/protobuf/descriptor.h>

#include <google/protobuf/message.h>

#include "cpp/test.pb.h" // 这是protoc给你生成的文件

int main()

{

// 先获得类型的Descriptor .

auto descriptor = google::protobuf::DescriptorPool::generated\_pool()->FindMessageTypeByName("T.Test");

if (nullptr == descriptor)

{

return 0 ;

}

// 利用Descriptor拿到类型注册的instance. 这个是不可修改的.

auto prototype = google::protobuf::MessageFactory::generated\_factory()->GetPrototype(descriptor);

if ( nullptr == prototype)

{

return 0 ;

}

// 构造一个可用的消息.

auto message = prototype->New();

// 只有当我们预先编译了test消息并且正确链接才能这么干.

auto test = dynamic\_cast<T::Test\*>(message);

// 直接调用message的具体接口

// 其实这些接口是语法糖接口.所以并没有对应的反射机制来对应调用.

// 反射机制实现了的Set/Get XXX系列接口，是属于Reflection的接口，接收Message作为参数.

test->set\_id(1);

std::cout<<test->Utf8DebugString()<<std::endl;

delete message ;

return 0 ;

}

[root@node2 test]# g++ -std=c++11 -g -Wall -o reflect\_test1.out reflect\_test1.cpp cpp/test.pb.cc -lprotobuf -lpthread

[root@node2 test]# ./reflect\_test1.out

id: 1

* 直接解析proto文件模式

#include <iostream>

#include <google/protobuf/compiler/importer.h>

#include <google/protobuf/dynamic\_message.h>

int main()

{

// 准备配置好文件系统

google::protobuf::compiler::DiskSourceTree sourceTree;

// 将当前路径映射为项目根目录 ， project\_root 仅仅是个名字，你可以你想要的合法名字.

sourceTree.MapPath("project\_root","./");

// 配置动态编译器.

google::protobuf::compiler::Importer importer(&sourceTree, NULL);

// 动态编译proto源文件。 源文件在./source/proto/test.proto .

importer.Import("project\_root/source\_proto/test.proto");

// 现在可以从编译器中提取类型的描述信息.

auto descriptor1 = importer.pool()->FindMessageTypeByName("T.Test");

// 创建一个动态的消息工厂.

google::protobuf::DynamicMessageFactory factory;

// 从消息工厂中创建出一个类型原型.

auto proto1 = factory.GetPrototype(descriptor1);

// 构造一个可用的消息.

auto message1= proto1->New();

// 下面是通过反射接口给字段赋值.

auto reflection1 = message1->GetReflection();

auto filed1 = descriptor1->FindFieldByName("id");

reflection1->SetInt32(message1,filed1,1);

// 打印看看

std::cout << message1->DebugString();

// 删除消息.

delete message1 ;

return 0 ;

}

[root@node2 test]# g++ -std=c++11 -g -Wall -o reflect\_test2.out reflect\_test2.cpp -lprotobuf -lpthread

[root@node2 test]# ./reflect\_test2.out

id: 1

* 通过对象和对象的属性获取名字，修改对应属性

#include "cpp/test.pb.h"

#include <iostream>

int main()

{

// 拿到一个对象，不在乎怎么拿到，可以是通过反射拿到。

// 这里简单直接的创建一个.

T::Test p\_test ;

// 拿到对象的描述包.

auto descriptor = p\_test.GetDescriptor() ;

// 拿到对象的反射配置.

auto reflecter = p\_test.GetReflection() ;

// 拿到属性的描述包.

auto field = descriptor->FindFieldByName("id");

// 设置属性的值.

reflecter->SetInt32(&p\_test , field , 5 ) ;

// 获取属性的值.

std::cout<<reflecter->GetInt32(p\_test , field)<< std::endl ;

return 0 ;

}

[root@node2 test]# g++ -std=c++11 -g -Wall -o reflect\_test3.out reflect\_test3.cpp cpp/test.pb.cc -lprotobuf -lpthread

[root@node2 test]# ./reflect\_test3.out

5

## 处理Protobuf反射，通过协议字串动态生成协议

下面两部分是针对protobuf通讯协议的内容。下面这两个函数可以动态生成pb的message，其中ProtoMsg是pb package的名字

inline google::protobuf::Message\* CreateMessage(const std::string& msg)

{

google::protobuf::Message\* message = NULL;

const google::protobuf::Descriptor\* descriptor = google::protobuf::DescriptorPool::generated\_pool()->FindMessageTypeByName("ProtoMsg." + msg);

if (descriptor)

{

const google::protobuf::Message\* prototype = google::protobuf::MessageFactory::generated\_factory()->GetPrototype(descriptor);

if (prototype)

{

message = prototype->New();

}

}

return message;

}

inline void ReleaseMessage(google::protobuf::Message\* pMsg)

{

if (NULL != pMsg)

{

pMsg->Clear();

delete pMsg;

}

}

## 通过反射将配置中的值设置进pb字段

可以通过如下方法，自动创建message并对field赋值：

// mstrCurMsg 当前正在执行的协议

google::protobuf::Message\* pMsg = Test::CreateMessage(mstrCurMsg);

const google::protobuf::Descriptor\* pDescriptor = google::protobuf::DescriptorPool::generated\_pool()->FindMessageTypeByName("ProtoMsg." + mstrCurMsg);

assert(NULL != pDescriptor);

// 这一个可以获取到反射类，然后可以将配置中值赋值进去

const google::protobuf::Reflection\* pReflection = pMsg->GetReflection();

assert(NULL != pReflection);

for (int i = 0; i < pDescriptor->field\_count(); ++i)

{

const google::protobuf::FieldDescriptor\* pFieldDescriptor = pDescriptor->field(i);

assert(NULL != pFieldDescriptor);

const std::string& strFieldName = pFieldDescriptor->name();

const TestConfigModule::MsgEntry\* pMsgEntry = pMsgStruct->GetMsgEntry(strFieldName);

assert(NULL != pMsgEntry);

// 读取字段类型，顺带可以做类型检查

assert(pMsgEntry->mnType == pFieldDescriptor->type());

// 设置值

switch (pMsgEntry->mnType)

{

case Test::TYPE\_STRING:

pReflection->SetString(pMsg, pFieldDescriptor, pMsgEntry->mstrValue);

break;

// ...

default:

break;

}

}

std::string strData;

if (!pMsg->SerializeToString(&strData))

{

m\_pLogModule->LogNormal("Test stop, cannot SerializeToString ", mstrCurMsg, \_\_FUNCTION\_\_, \_\_LINE\_\_);

return;

}

Test::ReleaseMessage(pMsg);

## serialize\_message

serialize\_message遍历提取message中各个字段以及对应的值，序列化到string中。主要思路就是通过Descriptor得到每个字段的描述符：字段名、字段的cpp类型。通过Reflection的GetX接口获取对应的value。

void serialize\_message(const google::protobuf::Message& message, std::string\* serialized\_string) {

const google::protobuf::Descriptor\* descriptor = message.GetDescriptor();

const google::protobuf::Reflection\* reflection = message.GetReflection();

for (int i = 0; i field\_count(); ++i) {

const google::protobuf::FieldDescriptor\* field = descriptor->field(i);

bool has\_field = reflection->HasField(message, field);

if (has\_field) {

//arrays not supported

assert(!field->is\_repeated());

switch (field->cpp\_type()) {

#define CASE\_FIELD\_TYPE(cpptype, method, valuetype)

case google::protobuf::FieldDescriptor::CPPTYPE\_##cpptype:{

valuetype value = reflection->Get##method(message, field);

int wsize = field->name().size();

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(field->name().c\_str(), field->name().size());

wsize = sizeof(value);

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(reinterpret\_cast(&value), sizeof(value));

break;

}

CASE\_FIELD\_TYPE(INT32, Int32, int);

CASE\_FIELD\_TYPE(UINT32, UInt32, uint32\_t);

CASE\_FIELD\_TYPE(FLOAT, Float, float);

CASE\_FIELD\_TYPE(DOUBLE, Double, double);

CASE\_FIELD\_TYPE(BOOL, Bool, bool);

CASE\_FIELD\_TYPE(INT64, Int64, int64\_t);

CASE\_FIELD\_TYPE(UINT64, UInt64, uint64\_t);

#undef CASE\_FIELD\_TYPE

case google::protobuf::FieldDescriptor::CPPTYPE\_ENUM: {

int value = reflection->GetEnum(message, field)->number();

int wsize = field->name().size();

//写入name占用字节数

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

//写入name

serialized\_string->append(field->name().c\_str(), field->name().size());

wsize = sizeof(value);

//写入value占用字节数

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

//写入value

serialized\_string->append(reinterpret\_cast(&value), sizeof(value));

break;

}

case google::protobuf::FieldDescriptor::CPPTYPE\_STRING: {

std::string value = reflection->GetString(message, field);

int wsize = field->name().size();

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(field->name().c\_str(), field->name().size());

wsize = value.size();

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(value.c\_str(), value.size());

break;

}

case google::protobuf::FieldDescriptor::CPPTYPE\_MESSAGE: {

std::string value;

int wsize = field->name().size();

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(field->name().c\_str(), field->name().size());

const google::protobuf::Message& submessage = reflection->GetMessage(message, field);

serialize\_message(submessage, &value);

wsize = value.size();

serialized\_string->append(reinterpret\_cast(&wsize), sizeof(wsize));

serialized\_string->append(value.c\_str(), value.size());

break;

}

}

}

}

}

## parse\_message

parse\_message通过读取field/value，还原message对象。

主要思路跟serialize\_message很像，通过Descriptor得到每个字段的描述符FieldDescriptor，通过Reflection的SetX填充message。

void parse\_message(const std::string& serialized\_string, google::protobuf::Message\* message) {

const google::protobuf::Descriptor\* descriptor = message->GetDescriptor();

const google::protobuf::Reflection\* reflection = message->GetReflection();

std::map field\_map;

for (int i = 0; i field\_count(); ++i) {

const google::protobuf::FieldDescriptor\* field = descriptor->field(i);

field\_map[field->name()] = field;

}

const google::protobuf::FieldDescriptor\* field = NULL;

size\_t pos = 0;

while (pos (serialized\_string.substr(pos, sizeof(int)).c\_str()));

pos += sizeof(int);

std::string name = serialized\_string.substr(pos, name\_size);

pos += name\_size;

int value\_size = \*(reinterpret\_cast(serialized\_string.substr(pos, sizeof(int)).c\_str()));

pos += sizeof(int);

std::string value = serialized\_string.substr(pos, value\_size);

pos += value\_size;

std::map::iterator iter =

field\_map.find(name);

if (iter == field\_map.end()) {

fprintf(stderr, "no field found.n");

continue;

} else {

field = iter->second;

}

assert(!field->is\_repeated());

switch (field->cpp\_type()) {

#define CASE\_FIELD\_TYPE(cpptype, method, valuetype)

case google::protobuf::FieldDescriptor::CPPTYPE\_##cpptype: {

reflection->Set##method(

message,

field,

\*(reinterpret\_cast(value.c\_str())));

std::cout name() (value.c\_str())) enum\_type()->FindValueByNumber(\*(reinterpret\_cast(value.c\_str())));

reflection->SetEnum(message, field, enum\_value\_descriptor);

std::cout name() (value.c\_str())) SetString(message, field, value);

std::cout name() MutableMessage(message, field);

parse\_message(value, submessage);

break;

}

default: {

break;

}

}

}

}

# Protobuf序列化原理

## Protobuf定义

一种 结构化数据 的数据存储格式（类似于 XML、Json ）

Google 出品 （开源）

Protocol Buffer 目前有两个版本：proto2 和 proto3

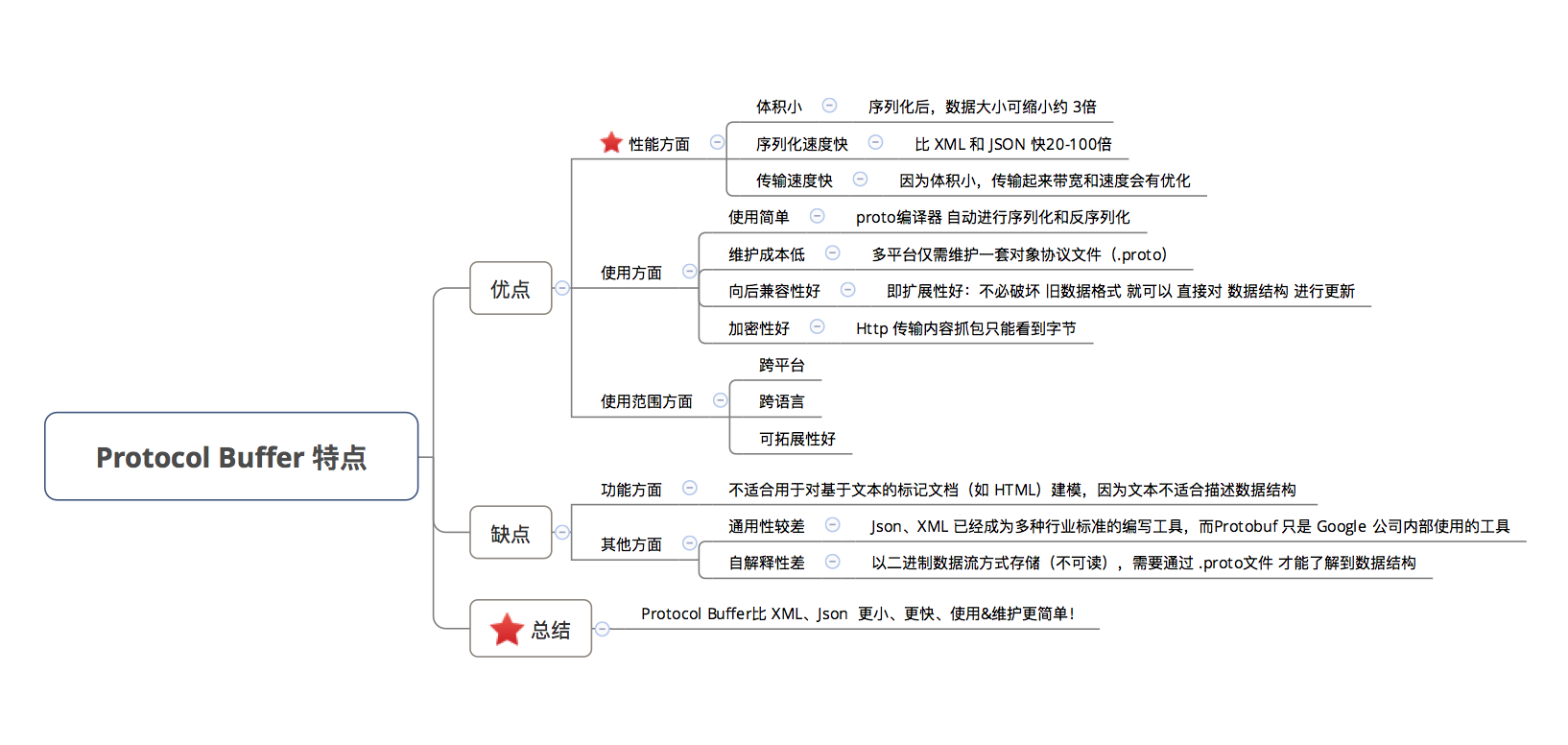
## 作用

通过将 结构化的数据 进行 串行化（**序列化**），从而实现 **数据存储 / RPC 数据交换**的功能

* 序列化： 将 数据结构或对象 转换成 二进制串 的过程
* 反序列化：将在序列化过程中所生成的二进制串 转换成 数据结构或者对象 的过程

## 特点

对比于 常见的 XML、Json 数据存储格式，Protocol Buffer有如下特点：



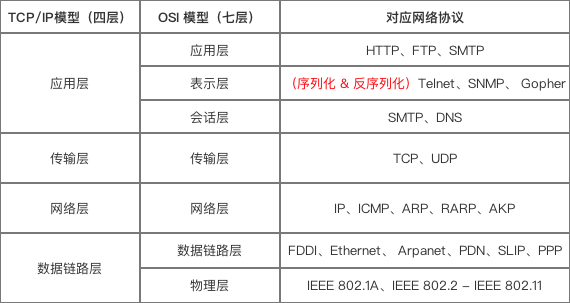
## 应用场景

传输数据量大 & 网络环境不稳定 的**数据存储、RPC 数据交换** 的需求场景

如 即时IM （QQ、微信）的需求场景

在 传输数据量较大的需求场景下，Protocol Buffer比XML、Json 更小、更快、使用 & 维护更简单！

## 序列化与反序列化在协议栈中的位置



* （序列化）把 应用层的对象 转换成 二进制串
* （反序列化）把 二进制串 转换成 应用层的对象

## 数据结构、对象与二进制串

不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

1. 对于数据结构和对象

* 对于面向对象的语言（如Java）：对象 = Object = 类的实例化；在Java中最接近数据结构 即 POJO（Plain Old Java Object），或Javabean（只有 setter/getter 方法的类）
* 对于半面向对象的语言（如C++），对象 = class，数据结构 = struct

b. 二进制串

* 对于C++，因为具有内存操作符，所以 二进制串 容易理解：C++的字符串可以直接被传输层使用，因为其本质上就是以 '\0' 结尾的存储在内存中的二进制串
* 对于 Java，二进制串 = 字节数组 =byte[]

byte 属于 Java 的八种基本数据类型

二进制串 容易和 String混淆：String 一种特殊对象（Object）。对于跨语言间的通讯，序列化后的数据当然不能是某种语言的特殊数据类型。

## T - L - V 的数据存储方式

* 定义  
  即 Tag - Length - Value，**标识 - 长度 - 字段值** 存储方式
* 作用  
  以 **标识 - 长度 - 字段值** 表示单个数据，最终将所有数据拼接成一个 **字节流**，从而 实现 数据存储 的功能

其中 Length可选存储，如 储存Varint编码数据就不需要存储Length

* 示意图



* 优点  
  从上图可知，T - L - V 存储方式的优点是

1. 不需要分隔符 就能 分隔开字段，减少了 分隔符 的使用
2. 各字段 存储得非常紧凑，存储空间利用率非常高
3. 若 字段没有被设置字段值，那么该字段在序列化时的数据中是完全不存在的，即不需要编码
4. 相应字段在解码时才会被设置为默认值

## 序列化原理解析

请记住Protocol Buffer的 三个关于数据存储 的重要结论：

* 结论1  
  Protocol Buffer将 消息里的每个字段 进行编码后，再利用T - L - V 存储方式 进行数据的存储，最终得到的是一个 二进制字节流

序列化 = 对数据进行编码 + 存储

* 结论2  
  Protocol Buffer对于不同数据类型 采用不同的 序列化方式（编码方式 & 数据存储方式），如下图：



从上表可以看出：

1. 对于存储Varint编码数据，就不需要存储字节长度 Length，所以实际上Protocol Buffer的存储方式是 T - V；
2. 若Protocol Buffer采用其他编码方式（如LENGTH\_DELIMITED）则采用T - L - V

* 结论3：

因为 Protocol Buffer对于数据字段值的 独特编码方式 & T - L - V数据存储方式，使得 Protocol Buffer序列化后数据量体积如此小

## Varint编码

varint是一种对数字进行编码的方案，编码后的数据是不定长的，值越小的数字使用越小的字节数，编码后的一般占在1~5个字节。

编码规则：

* 最高位表示是否继续，继续是1，代表后面7位仍然表示数字，否则为  
  0，后面7位用原码补齐。
* protobuf使用的是小字节序

上面的例子：400对应的二进制为00000001 10010000（原码）

具体步骤：

* 每个字节保留后7位，去掉最高位  
  0000011 0010000
* 因为protobuf使用的是小字节序，所以要把低位字节写到高字节，最后一个字节高位补0，其余各字节高位补1  
  10010000 0000011

400，所以经过varint编码后10010000 0000011,即0x90 0x03

varint的缺点:

负数需要10个字节显示（因为负数最高位是1，会被当作很大的整数处理）

## ZigZag编码

ZigZag是将符号数统一映射到无符号号数的一种编码方案，比如：对于0 -1 1 -2 2映射到无符号数 0 1 2 3 4。

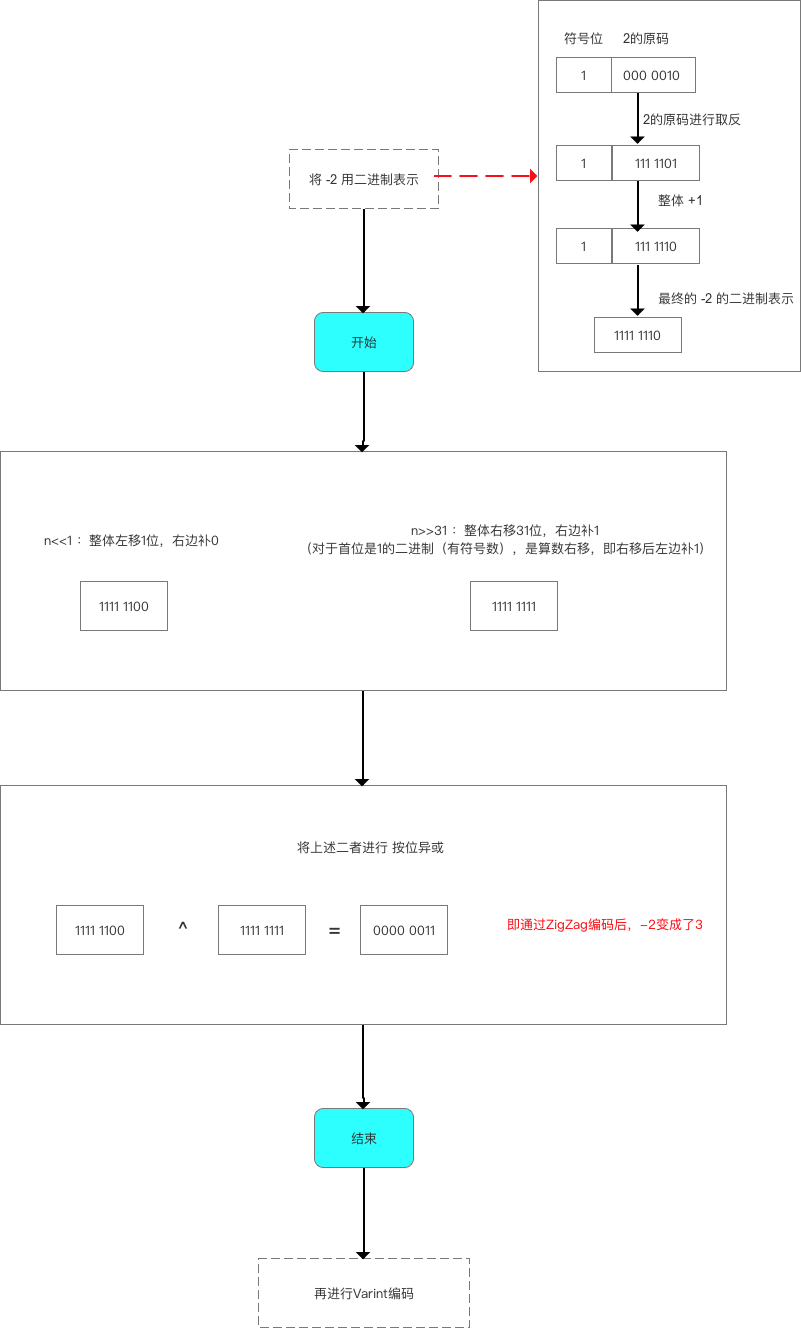
如：

| **原始值** | **映射值** |
| --- | --- |
| 0 | 0 |
| -1 | 1 |
| 1 | 2 |
| 2 | 3 |
| -2 | 4 |

* 定义：一种变长的编码方式
* 原理：使用 无符号数 来表示 有符号数字；
* 作用：使得绝对值小的数字都可以采用较少 字节 来表示；

特别是对 表示负数的数据 能更好地进行数据压缩

实例说明：将 -2进行 Zigzag编码：



Zigzag 编码 是补充 Varint编码在 表示负数 的不足，从而更好的帮助 Protocol Buffer进行数据的压缩

所以，**如果提前预知字段值是可能取负数的时候，记得采用**sint32 / sint64**数据类型**

总结

Protocol Buffer 通过Varint和Zigzag编码后大大减少了字段值占用字节数。

附录

message Test

{

required int32 id1 = 1；

required int32 id2 = 2；

}

// 在代码中给id1 附上1个字段值：296

// 在代码中给id2 附上1个字段值：296

Test.setId1（300）；

Test.setId2（296）；

// 编码结果为：二进制字节流 = [8，-84，2，16, -88, 2]

全编码过程：

