该系列Blog的内容主体主要源自于Protocol Buffer的官方文档，而代码示例则抽取于当前正在开发的一个公司内部项目的Demo。这样做的目的主要在于不仅可以保持Google文档的良好风格和系统性，同时再结合一些比较实用和通用的用例，这样就更加便于公司内部的培训，以及和广大网友的技术交流。需要说明的是，Blog的内容并非line by line的翻译，其中包含一些经验性总结，与此同时，对于一些不是非常常用的功能并未予以说明，有兴趣的开发者可以直接查阅Google的官方文档。

一、为什么使用**Protocol Buffer**？

在回答这个问题之前，我们还是先给出一个在实际开发中经常会遇到的系统场景。比如：我们的客户端程序是使用Java开发的，可能运行自不同的平台，如：Linux、Windows或者是Android，而我们的服务器程序通常是基于Linux平台并使用C++开发完成的。在这两种程序之间进行数据通讯时存在多种方式用于设计消息格式，如：

1. 直接传递C/C++语言中一字节对齐的结构体数据，只要结构体的声明为定长格式，那么该方式对于C/C++程序而言就非常方便了，仅需将接收到的数据按照结构体类型强行转换即可。事实上对于变长结构体也不会非常麻烦。在发送数据时，也只需定义一个结构体变量并设置各个成员变量的值之后，再以char*的方式将该二进制数据发送到远端。反之，该方式对于Java开发者而言就会非常繁琐，首先需要将接收到的数据存于ByteBuffer之中，再根据约定的字节序逐个读取每个字段，并将读取后的值再赋值给另外一个值对象中的域变量，以便于程序中其他代码逻辑的编写。对于该类型程序而言，联调的基准是必须客户端和服务器双方均完成了消息报文构建程序的编写后才能展开，而该设计方式将会直接导致Java程序开发的进度过慢。即便是Debug阶段，也会经常遇到Java程序中出现各种域字段拼接的小错误。

2. 使用SOAP协议(WebService)作为消息报文的格式载体，由该方式生成的报文是基于文本格式的，同时还存在大量的XML描述信息，因此将会大大增加网络IO的负担。又由于XML解析的复杂性，这也会大幅降低报文解析的性能。总之，使用该设计方式将会使系统的整体运行性能明显下降。

对于以上两种方式所产生的问题，Protocol Buffer均可以很好的解决，不仅如此，Protocol Buffer还有一个非常重要的优点就是可以保证同一消息报文新旧版本之间的兼容性。至于具体的方式我们将会在后续的博客中给出。

二、定义第一个**Protocol Buffer**消息。

创建扩展名为.proto的文件，如：MyMessage.proto，并将以下内容存入该文件中。

```
message LogonReqMessage {
   required int64 acctID = 1;
   required string passwd = 2;
}
```

这里将给出以上消息定义的关键性说明。

1. message是消息定义的关键字，等同于C++中的struct/class，或是Java中的class。

2. LogonReqMessage为消息的名字，等同于结构体名或类名。

3. required前缀表示该字段为必要字段，既在序列化和反序列化之前该字段必须已经被赋值。与此同时，在Protocol Buffer中还存在另外两个类似的关键字，optional和repeated，带有这两种限定符的消息字段则没有required字段这样的限制。相比于optional，repeated主要用于表示数组字段。具体的使用方式在后面的用例中均会一一列出。

4. int64和string分别表示长整型和字符串型的消息字段，在Protocol Buffer中存在一张类型对照表，既

Protocol Buffer中的数据类型与其他编程语言(C++/Java)中所用类型的对照。该对照表中还将给出在不同的数据场景下，哪种类型更为高效。该对照表将在后面给出。

5. acctID和passwd分别表示消息字段名，等同于Java中的域变量名，或是C++中的成员变量名。

6. 标签数字**1**和**2**则表示不同的字段在序列化后的二进制数据中的布局位置。在该例中，passwd字段编码后的数据一定位于acctID之后。需要注意的是该值在同一message中不能重复。另外，对于Protocol Buffer而言，标签值为1到15的字段在编码时可以得到优化，既标签值和类型信息仅占有一个byte，标签范围是16到2047的将占有两个bytes，而Protocol Buffer可以支持的字段数量则为2的29次方减一。有鉴于此，我们在设计消息结构时，可以尽可能考虑让repeated类型的字段标签位于1到15之间，这样便可以有效的节省编码后的字节数量。

三、定义第二个（含有枚举字段）Protocol Buffer消息。

```
//在定义Protocol Buffer的消息时，可以使用和C++/Java代码同样的方式添加注释。
enum UserStatus {
   OFFLINE = 0;  //表示处于离线状态的用户
   ONLINE = 1;   //表示处于在线状态的用户
}
message UserInfo {
   required int64 acctID = 1;
   required string name = 2;
   required UserStatus status = 3;
}
```

这里将给出以上消息定义的关键性说明（仅包括上一小节中没有描述的）。

1. enum是枚举类型定义的关键字，等同于C++/Java中的enum。

2. UserStatus为枚举的名字。

3. 和C++/Java中的枚举不同的是，枚举值之间的分隔符是分号，而不是逗号。

4. OFFLINE/ONLINE为枚举值。

5. 0和1表示枚举值所对应的实际整型值，和C/C++一样，可以为枚举值指定任意整型值，而无需总是从0开始定义。如：

```
enum OperationCode {
   LOGON_REQ_CODE = 101;
   LOGOUT_REQ_CODE = 102;
   RETRIEVE_BUDDIES_REQ_CODE = 103;

   LOGON_RESP_CODE = 1001;
   LOGOUT_RESP_CODE = 1002;
   RETRIEVE_BUDDIES_RESP_CODE = 1003;
}
```

四、定义第三个（含有嵌套消息字段）Protocol Buffer消息。

我们可以在同一个.proto文件中定义多个message，这样便可以很容易的实现嵌套消息的定义。如：

```
enum UserStatus {
   OFFLINE = 0;
   ONLINE = 1;
}
```

```
message UserInfo {
    required int64 acctID = 1;
    required string name = 2;
    required UserStatus status = 3;
}
message LogonRespMessage {
    required LoginResult logonResult = 1;
    required UserInfo userInfo = 2;
}
```

这里将给出以上消息定义的关键性说明（仅包括上两小节中没有描述的）。

1. LogonRespMessage消息的定义中包含另外一个消息类型作为其字段，如UserInfo userInfo。

2. 上例中的UserInfo和LogonRespMessage被定义在同一个.proto文件中，那么我们是否可以包含在其他.proto文件中定义的message呢？Protocol Buffer提供了另外一个关键字import，这样我们便可以将很多通用的message定义在同一个.proto文件中，而其他消息定义文件可以通过import的方式将该文件中定义的消息包含进来，如：

```
import "myproject/CommonMessages.proto"
```

五、限定符(required/optional/repeated)的基本规则。

1. 在每个消息中必须至少留有一个required类型的字段。

2. 每个消息中可以包含0个或多个optional类型的字段。

3. repeated表示的字段可以包含0个或多个数据。需要说明的是，这一点有别于C++/Java中的数组，因为后两者中的数组必须包含至少一个元素。

4. 如果打算在原有消息协议中添加新的字段，同时还要保证老版本的程序能够正常读取或写入，那么对于新添加的字段必须是optional或repeated。道理非常简单，老版本程序无法读取或写入新增的required限定符的字段。

六、类型对照表。

.proto Type	Notes	C++ Type	Java Type
double		double	double
float		float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long
u		u	

七、Protocol Buffer消息升级原则。

在实际的开发中会存在这样一种应用场景，既消息格式因为某些需求的变化而不得不进行必要的升级，但是有些使用原有消息格式的应用程序暂时又不能被立刻升级，这便要求我们在升级消息格式时要遵守一定的规则，从而可以保证基于新老消息格式的新老程序同时运行。规则如下：

1. 不要修改已经存在字段的标签号。

2. 任何新添加的字段必须是optional和repeated限定符，否则无法保证新老程序在互相传递消息时的消息兼容性。

3. 在原有的消息中，不能移除已经存在的required字段，optional和repeated类型的字段可以被移除，但是他们之前使用的标签号必须被保留，不能被新的字段重用。

4. int32、uint32、int64、uint64和bool等类型之间是兼容的，sint32和sint64是兼容的，string和bytes是兼容的，fixed32和sfixed32，以及fixed64和sfixed64之间是兼容的，这意味着如果想修改原有字段的类型时，为了保证兼容性，只能将其修改为与其原有类型兼容的类型，否则就将打破新老消息格式的兼容性。

5. optional和repeated限定符也是相互兼容的。

八、Packages。

我们可以在.proto文件中定义包名，如：

```
package ourproject.lyphone;
```

该包名在生成对应的C++文件时，将被替换为名字空间名称，既namespace ourproject { namespace lyphone。而在生成的Java代码文件中将成为包名。

九、Options。

Protocol Buffer允许我们在.proto文件中定义一些常用的选项，这样可以指示Protocol Buffer编译器帮助我们生成更为匹配的目标语言代码。Protocol Buffer内置的选项被分为以下三个级别：

1. 文件级别，这样的选项将影响当前文件中定义的所有消息和枚举。

2. 消息级别，这样的选项仅影响某个消息及其包含的所有字段。

3. 字段级别，这样的选项仅仅响应与其相关的字段。

下面将给出一些常用的Protocol Buffer选项。

1. option java_package = "com.companyname.projectname";

java_package是文件级别的选项，通过指定该选项可以让生成Java代码的包名为该选项值，如上例中的Java代码包名为com.companyname.projectname。与此同时，生成的Java文件也将会自动存放到指定输出目录下的com/companyname/projectname子目录中。如果没有指定该选项，Java的包名则为package关键字指定的名称。该选项对于生成C++代码毫无影响。

2. option java_outer_classname = "LYPhoneMessage";

java_outer_classname是文件级别的选项，主要功能是显示的指定生成Java代码的外部类名称。如果没有指定该选项，Java代码的外部类名称为当前文件的文件名部分，同时还要将文件名转换为驼峰格式，如：my_project.proto，那么该文件的默认外部类名称将为MyProject。该选项对于生成C++代码毫无影响。

注：主要是因为Java中要求同一个.java文件中只能包含一个Java外部类或外部接口，而C++则不存在此限制。因此在.proto文件中定义的消息均为指定外部类的内部类，这样才能将这些消息生成到同一个Java文件中。在实际的使用中，为了避免总是输入该外部类限定符，可以将该外部类静态引入到当前Java文件中，如：*import static com.company.project.LYPhoneMessage.**。

3. option optimize_for = LITE_RUNTIME;

optimize_for是文件级别的选项，Protocol Buffer定义三种优化级别SPEED/CODE_SIZE/LITE_RUNTIME。缺省情况下是SPEED。

SPEED: 表示生成的代码运行效率高，但是由此生成的代码编译后会占用更多的空间。

CODE_SIZE: 和SPEED恰恰相反，代码运行效率较低，但是由此生成的代码编译后会占用更少的空

int32	Uses variable-length encoding.	int32	int
uint32	Uses variable-length encoding.	int64	long
int64	Uses variable-length encoding.	int64	long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	int32	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	int64	long
sfixed32	Always four bytes.	int32	int
sfixed64	Always eight bytes.	int64	long
bool		bool	boolean

间，通常用于资源有限的平台，如Mobile。

LITE_RUNTIME: 生成的代码执行效率高，同时生成代码编译后的所占用的空间也是非常少。这是以牺牲Protocol Buffer提供的反射功能为代价的。因此我们在C++中链接Protocol Buffer库时仅需链接libprotobuf-lite，而非libprotobuf。在Java中仅需包含protobuf-java-2.4.1-lite.jar，而非protobuf-java-2.4.1.jar。

注：对于LITE_MESSAGE选项而言，其生成的代码均将继承自MessageLite，而非Message。

4. [**pack** = true]: 因为历史原因，对于数值型的repeated字段，如int32、int64等，在编码时并没有得到很好的优化，然而在新近版本的Protocol Buffer中，可通过添加[pack=true]的字段选项，以通知Protocol Buffer在为该类型的消息对象编码时更加高效。如：

repeated int32 samples = 4 [packed=true]。

注：该选项仅适用于**2.3.0**以上的Protocol Buffer。

5. [**default** = default_value]: optional类型的字段，如果在序列化时没有被设置，或者是老版本的消息中根本不存在该字段，那么在反序列化该类型的消息是，optional的字段将被赋予类型相关的缺省值，如bool被设置为false，int32被设置为0。Protocol Buffer也支持自定义的缺省值，如：

optional int32 result_per_page = 3 [default = 10]。

bool		bool	optional String
bytes	A string must always contain UTF-8 encoded or 7-bit ASCII text.	String	String
bytes	May contain any arbitrary sequence of bytes.	String	ByteString

十、命令行编译工具。

protoc *--proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_DIR path/to/file.proto*

这里将给出上述命令的参数解释。

1. protoc为Protocol Buffer提供的命令行编译工具。

2. --proto_path等同于-I选项，主要用于指定待编译的.proto消息定义文件所在的目录，该选项可以被同时指定多个。

3. --cpp_out选项表示生成C++代码，--java_out表示生成Java代码，--python_out则表示生成Python代码，其后的目录为生成后的代码所存放的目录。

4. path/to/file.proto表示待编译的消息定义文件。

注：对于C++而言，通过Protocol Buffer编译工具，可以将每个.proto文件生成出一对.h和.cc的C++代码文件。生成后的文件可以直接加载到应用程序所在的工程项目中。如：MyMessage.proto生成的文件为MyMessage.pb.h和MyMessage.pb.cc。