Java 生成代码

Translated by Jeff

王晖 译

（译者联系方式为：lonelystellar@gmail.com）

注：关于本文档的翻译错误（包括语法错误，错别字）或中文技术交流，可以发送邮件至译者邮箱：lonelystellar@gmail.com，我们共同研究，共同进步，不胜感激。

- 编译器调用
- 包
- 消息
- 字段
- 枚举
- 扩展
- 服务
- 插件插入点

本页精确的描述了适用于任何给定的协议定义的使用 Protocol Buffers 编译器生成的 Java 代码 This page describes exactly what Java code the protocol buffer compiler generates for any given protocol definition.建议你在阅读本文之前阅读语言指南.

编译器调用

当调用--java_out=命令行时 Protocol Buffers 编译器将产生 Java 输出.参数--java_out= 是表示你想要的 Java 的输出目录.编译器将会为每一个.proto 文件输入创建一个单独的.java 文件.这个文件包含了一个单独的外部类定义和一些内部类以及基于在.proto 文件中定义的静态字段.

外部类的名字的由来如下:如果.proto 文件包含一行这样的代码:

```
option java_outer_classname = "Foo";
```

那么这个外部类的名字将是 Foo.否则,他的名字就是转换.proto 文件的名字为驼峰的方式.例如, foo_bar.proto 的文件将转换为 FooBar.

Java 的包名是从包里(Packages)取的.

选择输出文件是通过串联参数传递给--java_out=,包括包名(用/s 替换.s), 和 .java 文件名.例如,如果按照如下的方式调用编译器:

```
protoc --proto_path=src --java_out=build/gen src/foo.proto
```

如果 foo.proto 的 Java 包名是 com.example, 并且他的外部类名是 FooProtos, 那么 Protocol Buffers 编译器将会生成 build/gen/com/example/FooProtos.java 文件. 他还会根据需要自动创建 build/gen/com 和 build/gen/com/example 目录. 但是,他不会创建 build/gen 或 build 目录; 他们必须先存在. 你可以在档次调用中指定多个.proto 文件;所有的输出文件都会一次性生成.

当输出 Java 代码时,由于许多 Java 工具类都能直接从 JAR 文件中在读取源代码,该 Protocol Buffers 编译器的能力能方便的直接输出到 JAR 归档中.为了输出到 JAR 文件,只需简单的提供一个输出路径结尾是.jar 就可以了.注意,在归档中仅仅只有 Java 源代码,因此还必须单独编译它来生成 java 类文件.

包

生成的类放在基于 java_package 定义的 Java 包中.如果选项被省略了,那么将使用 package 声明.

例如,如果.proto 文件包含:

```
package foo.bar;
```

那么将会使 Java 类被放置在 foo.bar 包里.但是,如果 .proto 文件还包含 java_package 选项,例如:

```
package foo.bar;

option java_package = "com.example.foo.bar";
```

那么类将被放置在 com.example.foo.bar 包里.提供 java_package 选项是因为普通的.proto package 声明不能向后支持字段名.

消息

给定一个简单的消息声明:

```
message Foo {}
```

Protocol Buffers 编译器将会生成一个名叫 Foo 的类,它是继承自 Message 接口.而且类被声明为 final 的;意味着不允许有子类. Foo 还扩展了 GeneratedMessage ,但是这个应当被认为是一个非常详细的实现.因为 Foo 最大限度的复写了 GeneratedMessage 的很多方法,但是如果.proto 文件包含这个:

```
option optimize_for = CODE_SIZE;
```

那么 Foo 将只会复写很少的需要操作和依靠 GeneratedMessage 基于反射的实现的剩余一组方法. 这大大的减少了生成的代码的大小,但是也降低了性能.此外,如果.proto 文件包含:

```
option optimize_for = LITE_RUNTIME;
```

那么 Foo 将会包含所有方法的快速实现,但是实现 MessageLite 接口,它只包含一个子集的 Message 的方法.特别注意,它不支持描述器或反射.但是,在这种模式下,生成的代码需要连结 libprotobuf-lite.jar 而不是 libprotobuf.jar."轻量级"的类库比全类库小多了,更适合于资源受限制的系统例如移动电话.
除了 Message 接口定义的方法, Foo 类还定义了如下的静态方法:

- static Foo getDefaultInstance(): 返回一个 Foo 的单例, 这跟你调用 Foo.newBuilder().build() 得到的是相同的()which is identical to what you'd get if you calledFoo.newBuilder().build() (因此所有的不同的(singular)字段都是未设置的,所有的重复的(repeated)字段都是空的). 注意消息的默认实例都通过调用 newBuilderForType()方法来被当作一个工厂.
- static Descriptor getDescriptor(): 返回类型的描述器.它包含类型的信息,它有的字段和它们的类型.这可以用在 Message 的反射射的方法里,如 getField().
- static Foo parseFrom(...): 从给定的源解析一个 Foo 的类型的消息并返回它. parseFrom 方法对应每一个在 Message.Builder 接口中的 mergeFrom() 方法的变形.注意 parseFrom() 永远不会抛出 UninitializedMessageException 异常;它会在解析消息缺少必填字段时抛出 InvalidProtocolBufferException. 这使它与调用 Foo.newBuilder().mergeFrom(...).build()方法有些微妙的不同.
- Foo.Builder newBuilder(): 创建一个新的建造器 (在下面说明).

- Foo.Builder newBuilder(Foo prototype): 创建一个新的建造器在所有字段还是原型(prototype) 的时候就全部初始化成相同的值.由于嵌入的消息和字符串对象都是不可变的,他们分析原始和拷贝之间的东西.

建造器

消息对象-比如上面说的 Foo 类的实例-都是不可变的,就像 Java 的 String .为了构造一个消息对象,就必须使用*建造器(builder)* .每个消息类都有它自己的建造器类-因此在 Foo 示例中,Protocol Buffers 编译器生成一个内部类 Foo.Builder ,它能被用来编译一个 Foo. Foo.Builder 类实现了 Message.Builder 接口.继承了 GeneratedMessage.Builder 类,但是这也只能被认为是一个详细实现.如 Foo, Foo.Builder 可能依赖在 GeneratedMessage.Builder 中的通用的方法实现,或者当使用了 optimize_for 选项时生成的自定义代码将会更快.

Foo.Builder 不定义任何静态方法.它的接口跟在 Message.Builder 接口一样,只是返回的异常更加具体: Foo.Builder 的方法修改建造器返回类型为 Foo.Builder ,并且 build() 方法返回类型 Foo.

方法修改建造器 的内容-包含字段的 set 方法-他总是返回一个建造器的引用(如: return this;).这允许多个方法在一行代码中来链式的调用.例

如: builder.mergeFrom(obj).setFoo(1).setBar("abc").clearBaz();

嵌套类型

一个消息能定义在另一个消息中.例如: message Foo { message Bar { } }

这样的话,编译器就会简单的生成一个内部类 Bar 嵌套在类 Foo 中.

字段

除了在前面章节中的说过方法,Protocol Buffers 编译器还能为在.proto 消息文件中定义的每个字段生成一组方法访问器.这些方法能读取到的被定义在消息类和他的相对应的建造器中的字段值;但是它只能修改被定义在建造器的字段值.

注意方法的名字总是用驼峰命名法的.即使在.proto 文件中的字段名使用带下划线小写字母.字母转换器能做到:

1. 在名字中有下划线的,下划线被删除,并且紧跟下划线的字母被改成大写.
2. 如果名字中有前缀(如:"get"),除去前缀的第一个字母被大写.其他的小写.

因此,字段 foo_bar_baz 变成 fooBarBaz.如果有前缀 get,就变成 getFooBarBaz.
除了访问器方法,编译器还为每一个字段生成一个包含字段编号的整型常量.字段名就是常量名字大写后加上 _FIELD_NUMBER. 例如,给定一个字段 optional int32 foo_bar = 5;, 那么编译器将会生成一个常量定义 public static final int FOO_BAR_FIELD_NUMBER = 5;.

不同的字段(Singular Fields)
定义可以是下面的任何一个:

```
optional int32 foo = 1;

required int32 foo = 1;
```

编译器将会在消息类和他的建造器中生成如下的访问器方法:

- bool hasFoo(): 返回 true ,如果设置了字段的话.
- int getFoo(): 返回字段的当前值.如果字段没有设置值,那么返回默认值.

编译器仅会在消息的建造器中生成如下的方法:

- Builder setFoo(int value): 设置字段的值.调用它之后, hasFoo() 将返回 true , getFoo() 将返回他的值(value).
- Builder clearFoo(): 清除字段的值.调用它之后 , hasFoo() 将返回 false , getFoo() 将返回默认值.

对于简单的字段类型,编译器将会按照标值类型表(scalar value types table)选择相应的 Java 基本类型.对于消息和枚举类型,值类型将会被替换成消息或这枚举类.
对于消息类型, setFoo() 还能接受消息的建造器的实例作为参数.它只是一种快捷方式等同于在建造器上调用.build() 方法并把结果传递给方法.

重复的字段(Repeated Fields)
定义如下:

```
repeated int32 foo = 1;
```

编译器将会在消息类和他的建造器中生成如下的访问器方法:

- int getFooCount(): 得到字段当前的元素数量.

- int getFoo(int index): 得到指定的元素(第一个位置的索引是 0).
- List<Integer> getFooList(): 得到一整个字段的不变列表.

编译器仅会在消息的建造器中生成如下的方法:

- Builder setFoo(int index, int value):设置元素的值根据给定的索引.
- Builder addFoo(int value): 添加一个给定值的新元素到字段中.
- Builder addAllFoo(List<Integer> value): 添加给定列表中的所有元素到字段中.
- Builder clearFoo(): 删除所有的字段中的所有元素,调用它之后, getFooCount() 会返回 0.

对于简单的字段类型,编译器将会按照标值类型表(scalar value types table)选择相应的 Java 基本类型.对于消息和枚举类型,值类型将会被替换成消息或这枚举类.
对于消息类型, setFoo() 还能接受消息的建造器的实例作为参数.它只是一种快捷方式等同于在建造器上调用.build() 方法并把结果传递给方法.

枚举

枚举的定义如下:

```
enum Foo{
 VALUE_A = 1;
 VALUE_B = 5;
 VALUE_C = 1234;
}
```

写缓冲区编译器将会生成一个叫 Foo 的 Java 的枚举类型,并且值也是它.另外,这个枚举类型的值还有如下的方法:

- int getNumber(): 得到定义在.proto 文件中的对象的数字值.
- EnumValueDescriptor getValueDescriptor(): 得到值的描述器,其中包含值的名字,编号,和类型.
- EnumDescriptor getDescriptorForType(): 得到枚举类型的描述器,其中包含每个被定义的值的信息.

另外, Foo 枚举类型包含如下的静态方法:

- static Foo valueOf(int value): 得到枚举对象相对应的数值.
- static Foo valueOf(EnumValueDescriptor descriptor): 得到枚举对象相对应的值描述器.可能比 valueOf(int)方法快.
- EnumDescriptor getDescriptor(): 得到枚举类型的描述器,其中包含其中包含每个被定义的值的信息(此方法不同于 getDescriptorForType() ,因为后者是静态方法).

注意.proto 语言允许多个枚举标记拥有同样的数值,同样数值的枚举标记是同义词.例如:

```
enum Foo{
 BAR = 1;
 BAZ = 1;
}
```

在这种情况下, BAZ 是 BAR 的同义词. 在 Java 中, BAZ 将会被定义成一个静态的最终字段,如:

```
static final Foo BAZ = BAR;
```

因此, BAR 和 BAZ 相比较是相等的, 并且 BAZ 不能出现在 switch 代码块中. 编译器总是选择第一个有数值的标记作为"规范的"版本;而在其后面的所有与它的值相同的标记都只是它的别名.

一个枚举能被定义在一个消息的内部.编译器将在消息类型类中生成一个嵌套的 Java 枚举.

扩展

扩展范围的定义如下:

```
message Foo{
 extensions 100 to 199;
}
```

Protocol Buffers 编译器将会使 Foo 继承 GeneratedMessage.ExtendableMessage 而不是通常认为的 GeneratedMessage.同样的, Foo 的建造器将会继承 GeneratedMessage.ExtendableBuilder,永远不要通过名字指向这些基类型(GeneratedMessage 被认为是一个详细实现).但是,这些父类定义了一些附加的方法能使你操作扩展.

特别是 Foo 和 Foo.Builder 将会把方法 hasExtension(), getExtension(), getExtensionCount()继承过来.另外, Foo.Builderw 将把方法 setExtension() 和 clearExtension()继承过来. 每个这些方法都用它的第一个参数来作为扩展标识(在下面说明),用来标识一个扩展字段.剩下的参数和返回值都和那些相对应的访问器方法可能生成的普通(非扩展的)字段的同类型扩展标识一样.

一个扩展定义实例:

```
extend Foo {

 optional int32 bar = 123;

}
```

Protocol Buffers 编译器将会生成一个名为 bar 的"扩展标识", 可以使用 Foo 的扩展访问器来访问这些扩展,例如:

```
Foo foo =
 Foo.newBuilder()
    .setExtension(bar, 1)
    .build();
assert foo.hasExtension(bar);
assert foo.getExtension(bar) == 1;
```

(详细的扩展标识实现是非常复杂的并且还有涉及泛型的妙用-但是,不需要担心扩展标识怎么来使用它们)

注意 bar 将会被声明成对.proto 文件外部类的一个静态字段,就像上面所描述的;我们在实例中省略了外部类的名字.

扩展能被声明成嵌套在其他类型内部.例如,下面是一个通用做法的模式:

```
message Baz {

 extend Foo {
```

```
    optional Baz foo_ext = 124;

  }

}
```

在这里,扩展标识 foo_ext 被定义在 Baz 内部,能通过下面这些语句来使用它:

```
Baz baz = createMyBaz();
Foo foo =
  Foo.newBuilder()
    .setExtension(Baz.fooExt, baz)
    .build();
```

服务

如果 .proto 文件有如下一行语句:

```
option java_generic_services = true;
```

那么 Protocol Buffers 编译器将会基于在本节描述的服务定义来生成代码.但是,这些生成的代码可能不符合任何特定的 RPC 系统, 因此需要更多间接层级的代码来针对一个系统.如果你不需要这些代码被生成,添加如下的一行到文件中::

```
option java_generic_services = false;
```

目前,如果上面的一行没有给出,那么选项默认的值是 true .但是,这些将会在未来的版本中修改,抽象服务将会被废弃.

RPC 系统基于.proto-语言服务定义应该提供插件来生成适合系统的代码.这些插件可能需要的抽象服务被禁用了,因此它们能生成与它名字相同的类.插件是版本 2.3.0(2010 年 1 月)中的新东西.

本节其他部分描述了 Protocol Buffers 编译器在抽象服务被打开时所生成的东西.

接口

接口的定义如下:

```
service Foo {
```

```
  rpc Bar(FooRequest) returns(FooResponse);

}
```

Protocol Buffers 编译器将会生成一个抽象类 Foo 来代表这个服务. Foo 拥有一个抽象方法包含每个在服务定义中定义的方法.因此, 方法 Bar 定义如下:

```
abstract void bar(RpcController controller, FooRequest request,
                  RpcCallback<FooResponse> done);
```

参数和 Service.CallMethod()的参数一样,除了 method 参数是隐式的并且 request 和 done 都指向的是它们的准确类型.

Foo 是 Service 接口的子类. Protocol Buffers 编译器自动生成 Service 方法的实现如下::

- getDescriptorForType: 获得服务的 ServiceDescriptor.
- callMethod: 根据所提供的方法描述器决定哪个方法将被直接调用,向下包装(down-casting)请求消息并回调给正确的类型.
- getRequestPrototype 和 getRequestPrototype: 得到所提供方法的请求或响应的默认实例的正确类型.

下面的静态方法也是生成出来的::

- static ServiceDescriptor getDescriptor(): 得到类型的描述器,其中包含这些服务所拥有的方法和它们的输入和输出类型.

Foo 还包含一个内部接口 Foo.Interface. 这是一个纯接口并且还包括在服务定义中对应的每个方法.但是,接口这个接口不继承自 Service 接口.这会有一个问题,因为 RPC 服务器的实现通常是被写成使用抽象的 Service 对象,而不是特定的服务.为了解决这个问题,如果一个对象 impl 实现了 Foo.Interface,可以通过调用 Foo.newReflectiveService(impl) 来构造一个能代理 impl 的 Foo 对象,并且它实现自 Service .

总之,当实现你自己的服务时,有 2 个选项可以选择:

- 子类 Foo 和实现了它的方法是适当的,然后手动把你子类的实例直接传给 RPC 服务器的实现.这通常是最简单的,但是有些人认为这是不那么”纯”的.Subclass Foo and implement its methods as appropriate, then hand instances of your subclass directly to the RPC server implementation. This is usually easiest, but some consider it less "pure".

- 实现 Foo.Interface 和使用 Foo.newReflectiveService(Foo.Interface) 构造一个 Service 来包装它,那么传递这个包装器给你的 RPC 实现.

根(Stub)

Protocol Buffers 编译器还能生成一个所有服务接口的"根"实现,它是被客户端用来发送实现了业务的请求给服务器.对于 Foo 服务(上面所提到的),它的根实现 Foo.Stub 将被定义成一个内部类.

Foo.Stub 是 Foo 的内部类并且还实现了如下的方法:

- Foo.Stub(RpcChannel channel): 构造一个新的根来发送请求给指定的通道.
- RpcChannel getChannel(): 获得这个根的通道,并传递给构造器.

另外根还围绕通道实现了每一个服务的方法来作为一个包装器.调用其中任何一个方法只需简单的调用 channel.callMethod().

Protocol Buffers 类库没有包含 RPC 实现.但是, 它包含了你选择的任意一个 RPC 实现用来生成服务类的所有需要的工具. 你只需要提供 RpcChannel 和 RpcController 的实现.

阻塞接口

RPC 类描述了上面所有的非阻塞语义:但你调用一个方法,你得提供一个在方法完成时会被调用的回调对象.通常它能使编写阻塞语言的代码变得简单(通过可能更少的伸长性),为了适应它,Protocol Buffers 编译器还能生成一个阻塞式版本的服务类. Foo.BlockingInterface 和 Foo.Interface 是等效的,除了每一种方法都会返回和一个值而不是调用一个回调.所以,例如,bar 是按照如下定义的:

```
abstract FooResponse bar(RpcController controller, FooRequest request)
                throws ServiceException;
```

类似与非阻塞服务, Foo.newReflectiveBlockingService(Foo.BlockingInterface) 返回一个 BlockingService 包装了一些 Foo.BlockingInterface 接口. 最后, Foo.BlockingStub 返回一个 Foo.BlockingInterface 的能发送请求给特定 BlockingRpcChannel 的根的实现.

插件插入点

代码生成器插件 需要通过扩展 Java 代码生成器的输出使用给定的插入点名字可能插入如下的类型的代码 which want to extend the output of the Java code generator may insert code of the following types using the given insertion point names.

- outer_class_scope: 成员声明属于文件的外部类.
- class_scope:TYPENAME: 成员声明属于一个消息类. TYPENAME 是完整的原名字, 如: package.MessageType.
- builder_scope:TYPENAME: 成员声明属于以消息的建造器类. TYPENAME 是完整的原名字, 如:package.MessageType.
- enum_scope:TYPENAME: 成员声明属于一个枚举类. TYPENAME 是完整的原名字, 如:package.EnumType.

生成的代码不能包含引用语句,这些可能会与在生成代码里定义的类型名字定义有冲突. 相应的,当引用一个外部类时,必须用全路径名.

在 Java 代码生成器中决定输出文件的逻辑是比较复杂的.你必须看 protoc 的源代码, 特别是 java_headers.cc, 确保你包含了所有的情况.

不要在私有类成员被声明在一个标准的代码生成器中时生成代码,这些实现细节可能在未来的 Protocol Buffers 版本中被修改.