该篇Blog和上一篇（C++实例）基本相同，只是面向于我们团队中的Java工程师， 毕竟我们项目的前端部分是基于Android开发的，而且我们研发团队中目前主要使用的开发语言就是C++、Java和Python，其中Python主 要用于编写各种工具程序。然而为了保证该篇Blog的完整性和独立性，我仍然会将上一篇Blog中已经出现的内容再一次赘述，同时对于Java中特有的部 分也会着重介绍。  
      
**一、生成目标语言代码。**  
      下面的命令帮助我们将MyMessage.proto文件中定义的一组Protocol Buffer格式的消息编译成目标语言（Java）的代码。至于消息的内容，我们会在后面以分段的形式逐一列出，同时也会在附件中给出所有源代码。  
      **protoc** -I=./message --java\_out=./src ./MyMessage.proto  
      从上面的命令行参数中可以看出，待编译的文件为MyMessage.proto，他存放在当前目录的message子目录下。--java\_out参数则 指示编译工具我们需要生成目标语言是java，输出目录是当前目录的src子目录。这里需要补充说明的是，因为在MyMessage.proto文件中定 义了option java\_package = "com.lsk.lyphone"的文件级选项，所以输出的目前是src/com/lsk/lyphone，生成的目标代码文件名是 MyMessage.java。  
      
**二、简单message生成的Java代码。**  
      这里先定义一个最简单的message，其中只是包含原始类型的字段。  
      option java\_package = "com.lsk.lyphone";  
      option java\_outer\_classname = "LYPhoneMessage";  
      option optimize\_for = LITE\_RUNTIME;  
      message LogonReqMessage {  
          required int64 acctID = 1;  
          required string passwd = 2;  
      }  
      对于选项**java\_package**和**java\_outer\_classname**的功能，我们已经在之前的一篇Blog（语言规范）中进行了清晰的阐述，这里就不在另行介绍了。然而对于选项**optimize\_for**，这里将结合本例给出一些实用性描述。  
      当前.proto文件中该选项的值为**LITE\_RUNTIME**， 因此由该.proto文件生成的所有Java类的父类均为com.google.protobuf.GeneratedMessageLite，而非 com.google.protobuf.GeneratedMessage，同时与之对应的Builder类则均继承自 com.google.protobuf.MessageLiteOrBuilder，而非 com.google.protobuf.MessageOrBuilder。在之前的博客中已经给出了一些简要的说明，MessageLite接口是 Message的父接口，在MessageLite中将缺少Protocol Buffer对反射的支持，而此功能均在Message接口中提供了接口规范，同时又在其实现类GeneratedMessage中给予了最小功能的实 现。对于我们的项目而言，整个系统相对比较封闭，不会和更多的外部程序进行交互，与此同时，我们的客户端部分又是运行在Android平台，有鉴于此，我 们考虑使用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。下面我们来看一下由message LogonReqMessage生成的Java类的部分声明，以及常用方法的说明性注释。  
      在做各种case的对比性分析之前必须要事先声明的是，Protocol Buffer针对Java语言所生成的代码和C++相比存在一个非常重要的差别，即为每个消息均会生成一个**Builder**接 口和一个与消息对应的实现类，该实现类又将同时实现生成的Builder接口和扩展Protocol Buffer内置的GeneratedMessageLite（或GeneratedMessage）类。这一点对于Protocol Buffer而言，是巧妙的使用了设计模式中的Builder模式。换言之，对于所有消息字段的修改操作均需要通过与其对应的Builder接口辅助完 成。相信我们会通过对下面用例的学习可以得到更为清楚的认识。

1 //用于修改LogonReqMessage消息字段的辅助Builder接口。

2 //该接口会为消息中的每个字段均提供getter和setter方法。

3 public interface LogonReqMessageOrBuilder

4 extends com.google.protobuf.MessageLiteOrBuilder {

5

6 // required int64 acctID = 1;

7 boolean hasAcctID();

8 long getAcctID();

9

10 // required string passwd = 2;

11 boolean hasPasswd();

12 String getPasswd();

13 }

14 //该类为final类，即不可以在被子类化了。这一点在Protocol Buffer的官方文档中给予了明确

15 //的说明，因为子类化将会破坏序列化和反序列化的过程。

16 public static final class LogonReqMessage extends

17 com.google.protobuf.GeneratedMessageLite

18 implements LogonReqMessageOrBuilder {

19

20 // Use LogonReqMessage.newBuilder() to construct.

21 // 由于所有构造函数均为私有方法，由此可见，我们不能直接new LogonReqMessage的对象

22 // 实例，而是只能通过与其对应Builder来构造，或是直接通过反序列化的方式生成。

23 private LogonReqMessage(Builder builder) {

24 super(builder);

25 }

26 //该静态方法为该类Builder接口的工厂方法。返回的Builder实现类在完成各个字段的

27 //初始化后，通过build()方法返回与其对应的消息实现类，即LogonReqMessage。

28 public static Builder newBuilder() { return Builder.create(); }

29 //通过该类的对象获取与其对应的Builder类对象，一般用于通过Builder类完成消息字段的修改。

30 public Builder toBuilder() { return newBuilder(this); }

31

32 private LogonReqMessage(boolean noInit) {}

33 //判断当前对象的所有字段是否都已经被初始化。

34 public final boolean isInitialized() {

35 ... ...

36 }

37 //获取已经被初始化后的对象序列化时所占用的字节空间。

38 public int getSerializedSize() {

39 ... ...

40 }

41 //从内存中饭序列化LogonReqMessage对象。

42 //Protocol Buffer中还提供其他一些接口方法，用于从不同的数据源反序列化对象。

43 public static com.lsk.lyphone.LYPhoneMessage.LogonReqMessage parseFrom(byte[] data)

44 throws com.google.protobuf.InvalidProtocolBufferException {

45 return newBuilder().mergeFrom(data).buildParsed();

46 }

47 //功能和上一个函数相同，只是输入源改为InputStream接口。

48 public static com.lsk.lyphone.LYPhoneMessage.LogonReqMessage parseFrom(java.io.InputStream input)

49 throws java.io.IOException {

50 return newBuilder().mergeFrom(input).buildParsed();

51 }

52

53 // required int64 acctID = 1;

54 // 下面的静态变量对应于该字段在.proto中定义的标签号。该变量的命名规则为：字段(全部大写) + \_FIELD\_NUMBER。

55 public static final int ACCTID\_FIELD\_NUMBER = 1;

56 public boolean hasAcctID() {

57 return ((bitField0\_ & 0x00000001) == 0x00000001);

58 }

59 public long getAcctID() {

60 return acctID\_;

61 }

62

63 // required string passwd = 2;

64 public static final int PASSWD\_FIELD\_NUMBER = 2;

65 public boolean hasPasswd() {

66 return ((bitField0\_ & 0x00000002) == 0x00000002);

67 }

68 public String getPasswd() {

69 ... ...

70 }

71 //每一个Message类都会包含一个静态内部类，即与之对应的Builder类。上面代码中所涉及的Builder类均为该内部类。

72 public static final class Builder extends

73 com.google.protobuf.GeneratedMessageLite.Builder<

74 com.lsk.lyphone.LYPhoneMessage.LogonReqMessage, Builder>

75 implements com.lsk.lyphone.LYPhoneMessage.LogonReqMessageOrBuilder {

76 //清空当前对象中的所有设置。调用该函数之后，本例中的hasAcctID和hasPasswd都会返回false。

77 public Builder clear() {

78 super.clear();

79 acctID\_ = 0L;

80 bitField0\_ = (bitField0\_ & ~0x00000001);

81 passwd\_ = "";

82 bitField0\_ = (bitField0\_ & ~0x00000002);

83 return this;

84 }

85 //克隆出一个Builder对象。

86 public Builder clone() {

87 return create().mergeFrom(buildPartial());

88 }

89 public com.lsk.lyphone.LYPhoneMessage.LogonReqMessage build() {

90 com.lsk.lyphone.LYPhoneMessage.LogonReqMessage result = buildPartial();

91 if (!result.isInitialized()) {

92 throw newUninitializedMessageException(result);

93 }

94 return result;

95 }

96 // Builder类中修改外部消息类的方法。

97 // required int64 acctID = 1;

98 public boolean hasAcctID() {

99 return ((bitField0\_ & 0x00000001) == 0x00000001);

100 }

101 public long getAcctID() {

102 return acctID\_;

103 }

104 //设置AcctID字段，该函数调用后hasAcctID函数将返回true。

105 //这里之所以让返回值为Builder对象，就是可以让调用者在一条代码中方便的连续修改多个字段，

106 //如：myMessage.setAcctID(100).setPasswd("MyName");

107 public Builder setAcctID(long value) {

108 bitField0\_ |= 0x00000001;

109 acctID\_ = value;

110 return this;

111 }

112 //清空AcctID字段，该函数调用后hasAcctID函数返回false。

113 //这里之所以让返回值为Builder对象，就是可以让调用者在一条代码中方便的连续清空多个字段，

114 //如：myMessage.clearAcctID().clearPasswd();

115 public Builder clearAcctID() {

116 bitField0\_ = (bitField0\_ & ~0x00000001);

117 acctID\_ = 0L;

118 return this;

119 }

120

121 // required string passwd = 2;

122 public boolean hasPasswd() {

123 return ((bitField0\_ & 0x00000002) == 0x00000002);

124 }

125 public String getPasswd() {

126 ... ...

127 }

128 public Builder setPasswd(String value) {

129 ... ...

130 }

131 public Builder clearPasswd() {

132 bitField0\_ = (bitField0\_ & ~0x00000002);

133 passwd\_ = getDefaultInstance().getPasswd();

134 return this;

135 }

136 void setPasswd(com.google.protobuf.ByteString value) {

137 bitField0\_ |= 0x00000002;

138 passwd\_ = value;

139 }

140 }

141 }

      在上面生成的代码中并没有列出与序列化相关的函数，这部分代码基本都是在父类中实现的，我们将在下面的例子中给出一些最基本的用法，有兴趣的开发者可以直接看Protocol Buffer中的源码，这部分代码比较通俗易懂。  
      下面是读写LogonReqMessage对象的Java测试代码和说明性注释。

1 private static void testSimpleMessage() {

2 System.out.println("==================This is simple message.================");

3 //如前所述，不能直接构造该消息类对象，只能通过他的内部Builder类构造并完成所有字段的初始化。

4 LogonReqMessage.Builder logonReqBuilder = LogonReqMessage.newBuilder();

5 logonReqBuilder.setAcctID(20);

6 logonReqBuilder.setPasswd("Hello World");

7 //builder对象初始化完毕后，再通过build方法生成与之对应的消息类对象。

8 LogonReqMessage logonReq = logonReqBuilder.build();

9 int length = logonReq.getSerializedSize();

10 System.out.println("The result length is " + length);

11 //直接序列化到内存中，之后可对该内存进行二次加工后再写到本地文件或发送到远端，如加密。

12 byte[] buf = logonReq.toByteArray();

13

14 try {

15 LogonReqMessage logonReq2 = LogonReqMessage.parseFrom(buf);

16 System.out.println("acctID = " + logonReq2.getAcctID() + "\tpassword = " + logonReq2.getPasswd());

17 } catch (InvalidProtocolBufferException e) {

18 e.printStackTrace();

19 }

20 //需要说明的是，文件中的内容是由之前C++实例代码写入的，这里这样写主要是一种验证。

21 System.out.println("Reading data from local file generated by C++");

22 try {

23 LogonReqMessage logonReq3 = LogonReqMessage.parseFrom(new FileInputStream("C:/Mine/LogonReq.dat"));

24 System.out.println("acctID = " + logonReq3.getAcctID() + "\tpassword = " + logonReq3.getPasswd());

25 } catch (FileNotFoundException e) {

26 e.printStackTrace();

27 } catch (IOException e) {

28 e.printStackTrace();

29 }

30 }

**三、嵌套message生成的Java代码。**  
      enum UserStatus {  
          OFFLINE = 0;  
          ONLINE = 1;  
      }  
      enum LoginResult {  
          LOGON\_RESULT\_SUCCESS = 0;  
          LOGON\_RESULT\_NOTEXIST = 1;  
          LOGON\_RESULT\_ERROR\_PASSWD = 2;  
          LOGON\_RESULT\_ALREADY\_LOGON = 3;  
          LOGON\_RESULT\_SERVER\_ERROR = 4;  
      }  
      message UserInfo {  
          required int64 acctID = 1;  
          required string name = 2;  
          required UserStatus status = 3;  
      }  
      message LogonRespMessage {  
          required LoginResult logonResult = 1;  
          required UserInfo userInfo = 2; //这里嵌套了UserInfo消息。  
      }  
      对于上述消息生成的Java代码，UserInfo因为只是包含了原始类型字段，因此和上例中的LogonReqMessage没有太多的差别，这里也就 不在重复列出了。由于LogonRespMessage消息中嵌套了UserInfo类型的字段，在这里我们将仅仅给出该消息生成的Java代码和关键性 注释。

1 public static final class LogonRespMessage extends

2 com.google.protobuf.GeneratedMessageLite

3 implements LogonRespMessageOrBuilder {

4

5 //Message类的通用性函数定义。

6 ... ...

7

8 // required .LoginResult logonResult = 1;

9 public static final int LOGONRESULT\_FIELD\_NUMBER = 1;

10 public boolean hasLogonResult() {

11 return ((bitField0\_ & 0x00000001) == 0x00000001);

12 }

13 public com.lsk.lyphone.LYPhoneMessage.LoginResult getLogonResult() {

14 return logonResult\_;

15 }

16

17 // required .UserInfo userInfo = 2;

18 public static final int USERINFO\_FIELD\_NUMBER = 2;

19 public boolean hasUserInfo() {

20 return ((bitField0\_ & 0x00000002) == 0x00000002);

21 }

22 public com.lsk.lyphone.LYPhoneMessage.UserInfo getUserInfo() {

23 return userInfo\_;

24 }

25 //Message类的通用性函数定义。可参照上一小节中的代码和注释。

26 ... ...

27

28 public static final class Builder extends

29 com.google.protobuf.GeneratedMessageLite.Builder<

30 com.lsk.lyphone.LYPhoneMessage.LogonRespMessage, Builder>

31 implements com.lsk.lyphone.LYPhoneMessage.LogonRespMessageOrBuilder {

32

33 //一些适用于绝大多数Builder对象的通用性方法。

34 ... ...

35

36 //当前示例中Builder生成的代码和上一小节中生成的代码非常类似，这里就不一一赘述了。

37 //和前面的例子相比一个重要的差别是setUserInfo函数多提供了一种函数签名，其参数为

38 //UserInfo类的Builder对象。这样调用者在使用时可以直接将Builder对象作为参数传入。

39 public Builder setUserInfo(com.lsk.lyphone.LYPhoneMessage.UserInfo.Builder builderForValue) {

40 userInfo\_ = builderForValue.build();

41 bitField0\_ |= 0x00000002;

42 return this;

43 }

44 }

45 }

      下面是读写LogonRespMessage对象的Java测试代码和说明性注释。

1 private static void testNestedMessage() {

2 System.out.println("==================This is nested message.================");

3 LogonRespMessage.Builder logonRespBuilder = LogonRespMessage.newBuilder();

4 logonRespBuilder.setLogonResult(LoginResult.LOGON\_RESULT\_SUCCESS);

5 UserInfo.Builder userInfo = UserInfo.newBuilder();

6 userInfo.setAcctID(200);

7 userInfo.setName("Tester");

8 userInfo.setStatus(UserStatus.OFFLINE);

9 //这里也可以直接传递userInfo对象作为参数。因为LogonRespBuilder类提供了setUserInfo的方法重载。

10 logonRespBuilder.setUserInfo(userInfo.build());

11 LogonRespMessage logonResp = logonRespBuilder.build();

12 int length = logonResp.getSerializedSize();

13 System.out.println("The result length is " + length);

14 byte[] buf = logonResp.toByteArray();

15

16 try {

17 LogonRespMessage logonResp2 = LogonRespMessage.parseFrom(buf);

18 UserInfo userInfo2 = logonResp2.getUserInfo();

19 System.out.println("LogonResult = " + logonResp2.getLogonResult().toString() + " acctID = "

20 + userInfo2.getAcctID() + " name = " + userInfo2.getName() + " status = " + userInfo2.getStatus().toString());

21 } catch (InvalidProtocolBufferException e) {

22 e.printStackTrace();

23 }

24 System.out.println("Reading data from local file generated by C++");

25 try {

26 LogonRespMessage logonResp3 = LogonRespMessage.parseFrom(new FileInputStream("C:/Mine/LogonResp.dat"));

27 UserInfo userInfo3 = logonResp3.getUserInfo();

28 System.out.println("LogonResult = " + logonResp3.getLogonResult().toString() + " acctID = "

29 + userInfo3.getAcctID() + " name = " + userInfo3.getName() + " status = " + userInfo3.getStatus().toString());

30 } catch (FileNotFoundException e) {

31 e.printStackTrace();

32 } catch (IOException e) {

33 e.printStackTrace();

34 }

35 }

**四、repeated嵌套message生成的Java代码。**  
      message BuddyInfo {  
          required UserInfo userInfo = 1;  
          required int32 groupID = 2;  
      }  
      message RetrieveBuddiesResp {  
          required int32 buddiesCnt = 1;  
          repeated BuddyInfo buddiesInfo = 2;  
      }  
      对于上述消息生成的代码，我们将只是针对RetrieveBuddiesResp消息所对应的Java代码进行详细说明，其余部分和前面小节的例子基本相 同，可直接参照。而对于RetrieveBuddiesResp类中的代码，我们也仅仅是对buddiesInfo字段生成的代码进行更为详细的解释。

1 public static final class RetrieveBuddiesResp extends

2 com.google.protobuf.GeneratedMessageLite

3 implements RetrieveBuddiesRespOrBuilder {

4 //这里均为Protocol Buffer生成的通用性代码。

5 ... ...

6 // repeated .BuddyInfo buddiesInfo = 2;

7 public static final int BUDDIESINFO\_FIELD\_NUMBER = 2;

8 //对于repeated类型的字段，均返回类型参数为字段类型的泛型容器对象。

9 public java.util.List<com.lsk.lyphone.LYPhoneMessage.BuddyInfo> getBuddiesInfoList() {

10 return buddiesInfo\_;

11 }

12 public java.util.List<? extends com.lsk.lyphone.LYPhoneMessage.BuddyInfoOrBuilder> getBuddiesInfoOrBuilderList() {

13 return buddiesInfo\_;

14 }

15 public int getBuddiesInfoCount() {

16 return buddiesInfo\_.size();

17 }

18 public com.lsk.lyphone.LYPhoneMessage.BuddyInfo getBuddiesInfo(int index) {

19 return buddiesInfo\_.get(index);

20 }

21 public com.lsk.lyphone.LYPhoneMessage.BuddyInfoOrBuilder getBuddiesInfoOrBuilder(int index) {

22 return buddiesInfo\_.get(index);

23 }

24

25 //这里仍有一些Protocol Buffer生成的通用性代码。

26 ... ...

27

28 public static final class Builder extends

29 com.google.protobuf.GeneratedMessageLite.Builder<

30 com.lsk.lyphone.LYPhoneMessage.RetrieveBuddiesResp, Builder>

31 implements com.lsk.lyphone.LYPhoneMessage.RetrieveBuddiesRespOrBuilder {

32

33 //这里仅列出和操作repeated字段相关的方法，其他的方法和前面的例子基本一致。

34 // repeated .BuddyInfo buddiesInfo = 2;

35 //本来打算给出比较详细的说明，但是看到Google为每个函数的命名之后就放弃这个想法，

36 //这样一来不仅可以避免画蛇添足，而且也节省了时间。:)

37 public java.util.List<com.lsk.lyphone.LYPhoneMessage.BuddyInfo> getBuddiesInfoList() {

38 return java.util.Collections.unmodifiableList(buddiesInfo\_);

39 }

40 public int getBuddiesInfoCount() {

41 return buddiesInfo\_.size();

42 }

43 public com.lsk.lyphone.LYPhoneMessage.BuddyInfo getBuddiesInfo(int index) {

44 return buddiesInfo\_.get(index);

45 }

46 public Builder setBuddiesInfo(int index, com.lsk.lyphone.LYPhoneMessage.BuddyInfo value) {

47 ... ...

48 }

49 public Builder setBuddiesInfo(int index, com.lsk.lyphone.LYPhoneMessage.BuddyInfo.Builder builderForValue) {

50 ... ...

51 }

52 public Builder addBuddiesInfo(com.lsk.lyphone.LYPhoneMessage.BuddyInfo value) {

53 ... ...

54 }

55 public Builder addBuddiesInfo(int index, com.lsk.lyphone.LYPhoneMessage.BuddyInfo value) {

56 ... ...

57 }

58 public Builder addBuddiesInfo(com.lsk.lyphone.LYPhoneMessage.BuddyInfo.Builder builderForValue) {

59 ... ...

60 }

61 public Builder addBuddiesInfo(

62 int index, com.lsk.lyphone.LYPhoneMessage.BuddyInfo.Builder builderForValue) {

63 ... ...

64 }

65 public Builder addAllBuddiesInfo(

66 java.lang.Iterable<? extends com.lsk.lyphone.LYPhoneMessage.BuddyInfo> values) {

67 ... ...

68 }

69 public Builder clearBuddiesInfo() {

70 ... ...

71 }

72 public Builder removeBuddiesInfo(int index) {

73 ... ...

74 }

75 }

76 }

      下面是读写RetrieveBuddiesResp对象的Java测试代码和说明性注释。

1 private static void testRepeatedMessage() {

2 System.out.println("==================This is repeated message.================");

3 RetrieveBuddiesResp.Builder retrieveBuddiesBuilder = RetrieveBuddiesResp.newBuilder();

4 retrieveBuddiesBuilder.setBuddiesCnt(2);

5 BuddyInfo.Builder buddyInfoBuilder = BuddyInfo.newBuilder();

6 buddyInfoBuilder.setGroupID(20);

7 UserInfo.Builder userInfoBuilder = UserInfo.newBuilder();

8 userInfoBuilder.setAcctID(200);

9 userInfoBuilder.setName("user1");

10 userInfoBuilder.setStatus(UserStatus.OFFLINE);

11 buddyInfoBuilder.setUserInfo(userInfoBuilder.build());

12 retrieveBuddiesBuilder.addBuddiesInfo(buddyInfoBuilder.build());

13

14 buddyInfoBuilder = BuddyInfo.newBuilder();

15 buddyInfoBuilder.setGroupID(21);

16 userInfoBuilder = UserInfo.newBuilder();

17 userInfoBuilder.setAcctID(201);

18 userInfoBuilder.setName("user2");

19 userInfoBuilder.setStatus(UserStatus.ONLINE);

20 buddyInfoBuilder.setUserInfo(userInfoBuilder);

21 retrieveBuddiesBuilder.addBuddiesInfo(buddyInfoBuilder);

22 RetrieveBuddiesResp buddiesResp = retrieveBuddiesBuilder.build();

23

24 int length = buddiesResp.getSerializedSize();

25 System.out.println("The result length is " + length);

26 byte[] buf = buddiesResp.toByteArray();

27

28 try {

29 RetrieveBuddiesResp buddiesResp2 = RetrieveBuddiesResp.parseFrom(buf);

30 System.out.println("BuddiesCount = " + buddiesResp2.getBuddiesCnt());

31 System.out.println("Repeated Size = " + buddiesResp2.getBuddiesInfoCount());

32 for (int i = 0; i < buddiesResp2.getBuddiesInfoCount(); ++i) {

33 BuddyInfo buddyInfo = buddiesResp2.getBuddiesInfo(i);

34 UserInfo userInfo = buddyInfo.getUserInfo();

35 System.out.println("GroupID = " + buddyInfo.getGroupID() + " UserInfo.acctID = " + userInfo.getAcctID()

36 + " UserInfo.name = " + userInfo.getName() + " UserInfo.status = " + userInfo.getStatus());

37 }

38

39 } catch (InvalidProtocolBufferException e) {

40 e.printStackTrace();

41 }

42 System.out.println("Reading data from local file generated by C++");

43 try {

44 RetrieveBuddiesResp buddiesResp3 = RetrieveBuddiesResp.parseFrom(new FileInputStream("C:/Mine/RetrieveBuddiesResp.dat"));

45 System.out.println("BuddiesCount = " + buddiesResp3.getBuddiesCnt());

46 System.out.println("Repeated Size = " + buddiesResp3.getBuddiesInfoCount());

47 List<BuddyInfo> buddiesInfo = buddiesResp3.getBuddiesInfoList();

48 for (BuddyInfo buddyInfo : buddiesInfo) {

49 UserInfo userInfo = buddyInfo.getUserInfo();

50 System.out.println("GroupID = " + buddyInfo.getGroupID() + " UserInfo.acctID = " + userInfo.getAcctID()

51 + " UserInfo.name = " + userInfo.getName() + " UserInfo.status = " + userInfo.getStatus());

52 }

53 } catch (FileNotFoundException e) {

54 e.printStackTrace();

55 } catch (IOException e) {

56 e.printStackTrace();

57 }

58 }

      对于Java而言，我们可以通过Maven工具生成两个jar包，其中一个是protobuf-java-2.4.1.jar，主要用于optimize\_for选项为非LITE\_RUNTIME的情况，而另一个protobuf-java-2.4.1-lite.jar文件则恰恰与之相反。另外，我通过Beyond Compare工具对这两个jar包进行了二进制比较后发现，他们是完全相同的。这里之所以仍以LITE版本为例，主要还是因为和之前一篇Blog（C++实例）想匹配。  
      最后需要说明的是，Protocol Buffer仍然提供了很多其它非常有用的功能，特别是针对序列化的目的地，比如文件流和网络流等。与此同时，也提供了完整的官方文档和规范的命名规则，在很多情况下，可以直接通过函数的名字便可获悉函数所完成的工作。  
      该Blog中使用的示例代码均已附件的方式上传，欢迎大家下载并指正。