存储优化之probuffer使用与原理详解 预习资料

> 码牛学院只为你能成为一位大牛

Protobuf可能大家对此很陌生，还未接触过，不过不要紧，看完这篇博客，相信你一定有所感触。起初为了节约流量，在我们千里眼后端接口率先使用Protobuf替代Json，支持Java、C++、Python等语言，就尝到甜头了，简单好用还节省内存流量，基于这个特性，英雄岂无用户之地。后面，我们推广到Sqlite、SharedPerference等领域，利用Protobuf进行改造，替换原有的Json或者XML存储方式!

Protobuf是什么

> Protobuf是一种灵活高效可序列化的数据协议，相于XML，具有更快、更简单、更轻量级等特性。支持多种语言，只需定义好数据结构，利用Protobuf框架生成源代码，就可很轻松地实现数据结构的序列化和反序列化。一旦需求有变，可以更新数据结构，而不会影响已部署程序。

从上面我们可以总结出，Protobuf具有以下优点：

1. 代码生成机制

```
syntax = "proto3";
package me.ele.demo.protobuf;
option java_outer_classname = "LoginInfo";
message Login {
    string account = 1;
    string password = 2;
}
```

这是一个用户登录信息的数据结构，通过Protobuf提供的Gradle Plugin就可以在`me.ele.demo.protobuf`目录下编译自动生成LoginInfo类，并有序列化和反序列化等Api。

1. 高效性

用千里眼项目中跑出来的数据进行对比，更具说服力。

序列化时间效率对比：

数据格式	1000条数据	5000条数据
Protobuf	195ms	647ms
Json	515ms	2293ms

序列化空间效率对比：

数据格式	5000条数据
Protobuf	22MB
Json	29MB

从上面的数据可以看出来，Protobuf序列化时，和Json对比，不管在时间和空间上都是更加高效。由于篇幅的原因就不展示反序列化的数据对比了。

1. 支持向后兼容和向前兼容

当客户端和服务器同事使用一块协议的时候， 当客户端在协议中增加一个字节，并不会影响客户端的使用

1. 支持多种编程语言

在Google官方发布的源代码中包含了c++、java、Python三种语言

至于缺点，Protobuf采用了二进制格式进行编码，这直接导致了可读性差；缺乏自描述，Protobuf是二进制格式的协议内容，要是不配合proto结构体根本看不出来什么来。

接入

在项目的根gradle配置如下

```
dependencies {
        classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.0'
}
```

在gradle中配置如下：

```
apply plugin: 'com.google.protobuf'

android {
    sourceSets {
        main {
            // 定义proto文件目录
            proto {
                srcDir 'src/main/proto'
                include '**/*.proto'
            }
        }
    }
}

dependencies {
    // 定义protobuf依赖，使用精简版
    compile "com.google.protobuf:protobuf-lite:3.0.0"
    compile ('com.squareup.retrofit2:converter-protobuf:2.2.0') {
        exclude group: 'com.google.protobuf', module: 'protobuf-java'
    }
}

protobuf {
    protoc {
        artifact = 'com.google.protobuf:protoc:3.0.0'
    }
    plugins {
        javalite {
            artifact = 'com.google.protobuf:protoc-gen-javalite:3.0.0'
        }
    }
    generateProtoTasks {
        all().each { task ->
```

```
            task.plugins {
                javalite {}
            }
        }
    }
}
```

`apply plugin: 'com.google.protobuf'`是Protobuf的Gradle插件，帮助我们在编译时通过语义分析自动生成源码，提供数据结构的初始化、序列化以及反序列等接口。

`compile "com.google.protobuf:protobuf-lite:3.0.0"`是Protobuf支持库的精简版本，在原有的基础上，用public替换set、get方法，减少Protobuf生成代码的方法数目。

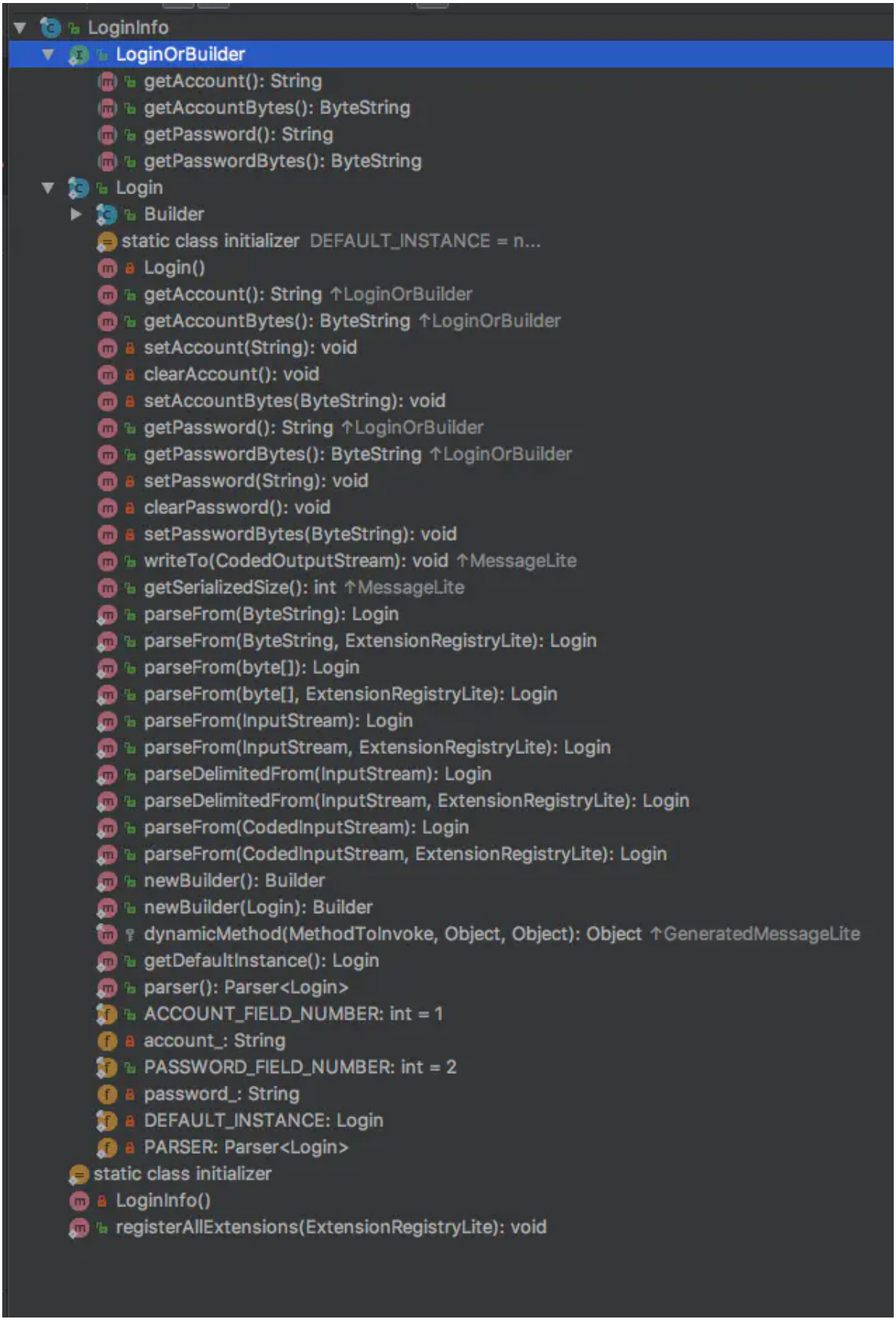
定义数据结构

还是以上面的例子来展开：

```
syntax = "proto3";
package me.ele.demo.protobuf;
option java_outer_classname = "LoginInfo";
message Login {
    string account = 1;
    string password = 2;
}
```

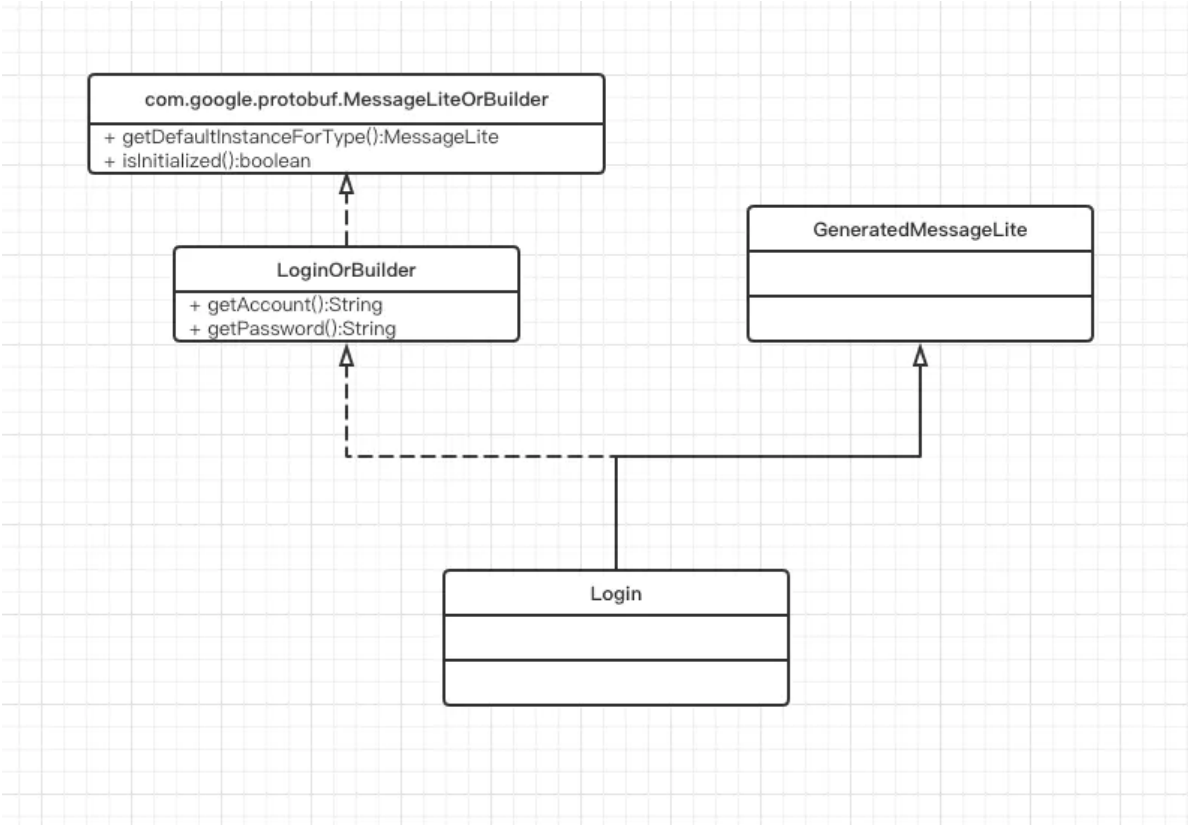
在这里定义了一个`LoginInfo`，我们只是简单的定义了`account`和`password`两个字段。这里注意，在上例中，`syntax = "proto3";`声明proto协议版本，proto2和proto3在定义数据结构时有些差别，`option java_outer_classname = "LoginInfo";`定义了Protobuf自动生成类的类名，`package me.ele.demo.protobuf;`定义了Protobuf自动生成类的包名。

通过Android Studio clean，Protobuf插件会帮助我们自动生成`LoginInfo`类，类结构如下：



LoginInfo类结构

Protobuf帮我们自动生成`LoginOrBuilder`接口，主要声明各个字段的set和get方法；并且生成`Login`类，核心逻辑这个类中，通过`writeTo(CodedOutputStream)`接口序列化到`CodedOutputStream`，通过ParseFrom(InputStream)接口从`InputStream`中反序列化。类图如下：

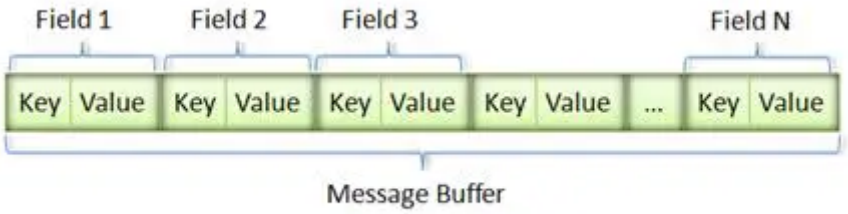


Login类图

原理分析

上文提到，Protobuf不管在时间和空间上更高效，是怎么做到的呢?

消息经过Protobuf序列化后会成为一个二进制数据流，通过Key-Value组成方式写入到二进制数据流，如图所示:



二进制数据流

Key 定义如下:

```
(field_number << 3) | wire_type
```

以上面的例子来说，如字段`account`定义:

```
string account = 1;
```

在序列化时，并不会把字段`account`写进二进制流中，而是把`field_number=1`通过上述`Key`的定义计算后写进二进制流中，这就是Protobuf可读性差的原因，也是其高效的主要原因。

数据类型

.proto Type	Notes	PHP Type
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.（使用变长编码，对于负值的效率很低，如果你的域有可能有负值，请使用sint32替代）	integer
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	integer/string
uint32	Uses variable-length encoding.	integer
uint64	Uses variable-length encoding.	integer/string
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.（使用变长编码，这些编码在负值时比int32高效的多）	integer
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	integer/string
fixed32	.	integer
fixed64	.	integer/string
sfixed32	Always four bytes.	integer
sfixed64	Always eight bytes.	integer/string
bool		boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string
bytes	May contain any arbitrary sequence of bytes.	string

Protobuf数据类型

在Java种对不同类型的选择，其他的类型区别很明显，主要在与int32、uint32、sint32、fixed32中以及对应的64位版本的选择，因为在Java中这些类型都用int(long)来表达，但是protobuf内部使用ZigZag编码方式来处理多余的符号问题，但是在编译生成的代码中并没有验证逻辑，比如uint的字段不能传入负数之类的。而从编码效率上，对fixed32类型，如果字段值大于2^28，它的编码效率比int32更加有效；而在负数编码上sint32的效率比int32要高；uint32则用于字段值永远是正整数的情况。

编码原理

在实现上，Protobuf使用`CodedOutputStream`实现序列化、`CodedInputStream`实现反序列化，他们包含write/read基本类型和`Message`类型的方法，`write`方法中同时包含`fieldNumber`和`value`参数，在写入时先写入由`fieldNumber`和`WireType`组成的tag值(添加这个`WireType`类型信息是为了在对无法识别的字段编码时可以通过这个类型信息判断使用那种方式解析这个未知字段，所以这几种类型值即可)，这个tag值是一个可变长int类型，所谓的可变长类型就是一个字节的最高位(msb，most significant bit)用1表示后一个字节属于当前字段，而最高位0表示当前字段编码结束。在写入tag值后，再写入字段值value，对不同的字段类型采用不同的编码方式：

1. 对int32/int64类型，如果值大于等于0，直接采用可变长编码，否则，采用64位的可变长编码，因而其编码结果永远是10个字节，所有说int32/int64类型在编码负数效率很低。
2. 对uint32/uint64类型，也采用变长编码，不对负数做验证。
3. 对sint32/sint64类型，首先对该值做ZigZag编码，以保留，然后将编码后的值采用变长编码。所谓ZigZag编码即将负数转换成正数，而所有正数都乘2，如0编码成0，-1编码成1，1编码成2，-2编码成3，以此类推，因而它对负数的编码依然保持比较高的效率。

4. 对fixed32/sfixed32/fixed64/sfixed64类型，直接将该值以小端模式的固定长度编码。
5. 对double类型，先将double转换成long类型，然后以8个字节固定长度小端模式写入。
6. 对float类型，先将float类型转换成int类型，然后以4个字节固定长度小端模式写入。
7. 对bool类型，写0或1的一个字节。
8. 对String类型，使用UTF-8编码获取字节数组，然后先用变长编码写入字节数组长度，然后写入所有的字节数组。
9. 对bytes类型(ByteString)，先用变长编码写入长度，然后写入整个字节数组。
10. 对枚举类型(类型值`WIRETYPE_VARINT`)，用int32编码方式写入定义枚举项时给定的值（因而在给枚举类型项赋值时不推荐使用负数，因为int32编码方式对负数编码效率太低）。
11. 对内嵌`Message`类型(类型值`WIRETYPE_LENGTH_DELIMITED`)，先写入整个`Message`序列化后字节长度，然后写入整个`Message`。

> ZigZag编码实现：`(n << 1) ^ (n >> 31) / (n << 1) ^ (n >> 63);`在`CodedOutputStream`中还存在一些用于计算某个字段可能占用的字节数的`compute`静态方法，这里不再详述。

在Protobuf的序列化中，所有的类型最终都会转换成一个可变长int/long类型、固定长度的int/long类型、byte类型以及byte数组。对byte类型的写只是简单的对内部buffer的赋值：

```
public void writeRawByte(final byte value) throws IOException {
  if (position == limit) {
    refreshBuffer();
  }
  buffer[position++] = value;
}
```

对32位可变长整形实现为：

```
public void writeRawVarint32(int value) throws IOException {
  while (true) {
    if ((value & ~0x7F) == 0) {
      writeRawByte(value);
      return;
    } else {
      writeRawByte((value & 0x7F) | 0x80);
      value >>>= 7;
    }
  }
}
```

对于定长，Protobuf采用小端模式，如对32位定长整形的实现：

```
public void writeRawLittleEndian32(final int value) throws IOException {
    writeRawByte((value      ) & 0xFF);
    writeRawByte((value >>  8) & 0xFF);
    writeRawByte((value >> 16) & 0xFF);
    writeRawByte((value >> 24) & 0xFF);
}
```

对byte数组，可以简单理解为依次调用`writeRawByte()`方法，只是`CodedOutputStream`在实现时做了部分性能优化。这里不详细介绍。对`CodedInputStream`则是根据`CodedOutputStream`的编码方式进行解码，因而也不详述，其中关于ZigZag的解码：

```
(n >>> 1) ^ -(n & 1)
```

repeated字段编码

对于`repeated`字段，一般有两种编码方式：

1. 每个项都先写入tag，然后写入具体数据。
2. 先写入tag，后count，再写入count个项，每个项包含length|data数据。

从编码效率的角度来看，个人感觉第二中情况更加有效，然而不知道处于什么原因考虑，Protobuf采用了第一种方式来编码，个人能想到的一个理由是第一种情况下，每个消息项都是相对独立的，因而在传输过程中接收端每接收到一个消息项就可以进行解析，而不需要等待整个`repeated`字段的消息包。对于基本类型，Protobuf也采用了第一种编码方式，后来发现这种编码方式效率太低，因而可以添加`[packed = true]`的描述将其转换成第三种编码方式(第二种方式的变种，对基本数据类型，比第二种方式更加有效)

1. 先写入tag，后写入字段的总字节数，再写入每个项数据。

目前Protobuf只支持基本类型的`packed`修饰，因而如果将`packed`添加到非`repeated`字段或非基本类型的`repeated`字段，编译器在编译proto文件时会报错。