# 通讯协议序列化思考

February 10, 2014

由于我们现在的RPC框架原来的序列化方式太弱了，导致各种序列化问题。所以最近研究了一下相关的序列化方式。主要有：

1. 二进制类的，一般采用TLV编码方式，如Protocol Buffers, Thrift。
2. 文本类的，如JSON, XML。
3. 混合型的，比如Avro，schema是JSON，data是二进制方式的。

序列化一般要么作为RPC传递，或者作为自描述的数据存储，比如数据库。不过后者由于需要存储多条，而且考虑到搜索和更新问题，所以对每条记录做了一定的组织，一般是B树组织结构。

事实上，实现一个简单的通讯协议序列化是非常简单的（通信协议之序列化）。关键在于一些细节问题，需要好好思考：

**1. 性能与带宽**

二进制协议最大的特点就是节省空间和带宽（特别是移动端对流量和网速要求更高）。Protobuf在这点真的到了吹毛求疵的地步。主要有如下几个技术： 1. VarInt 2. ZigZag

**2. 如何知道一个字段是不是被显示赋值还是默认值？**

protobuf对于每个字段（不管可选还是必填）都提供了`hasXXX`方法（或者通用的`hasField`方法），实现方式是用一个int类型的bitField0_私有变量来标志各个字段的赋值情况。没有被显示赋值的字段是不会被传递的，如果有传递则`bitField0_`就会被标志。这样只传递有效的字段，可以大大减少网络开销。Protobuf在反序列化时会对声明为`required`的字段进行校验，如果不存在则会报`Missing required fields`错误。

thrif的实现也是类似。不过他没有像PB那么吹毛求疵，使用一个int字段搞定一切。而是使用一个 _XXX_isset的struct成员来标志，而且只有`optional`的字段才有对应的bool字段，这个字段就标识这个可选字段是否被赋值。

比如下面这个thrift IDL定义：

```
struct Example {
    1:i32 number=10,
    2:i64 bigNumber,
    3:double decimals,
    4:string name="thrifty"
}
```

会生成如下C++文件：

```
class Example {
    public:
```

```
    Example() :
    number(10),
    bigNumber(0),
    decimals(0),
    name("thrifty") {}
    int32_t number;
    int64_t bigNumber;
    double decimals;
    std::string name;
    struct __isset {
        __isset() :
        number(false),
        bigNumber(false),
        decimals(false),
        name(false) {}
        bool number;
        bool bigNumber;
        bool decimals;
        bool name;
    } __isset;
    ...
}
```

然后在序列化的时候，thrift会对所有required的字段进行序列化（不管有没有被显示赋值），而对所有optional的字段，只有相应的isset字段为true时，才会序列化。反序列化的时候，thrift对required的字段会做校验，如果不存在，则抛出异常。而对于optional的字段，只有存在，才会设置该字段相应的isset为true。

**3. 可选与必填？强制校验?**

PB和Thrift都支持`required`和`optional`声明，并且都对`required`字段进行反序列化校验。

**4. 支持更多的数据类型？比如Constants、Enum、Map、Set等等？支持范型？**

Protobuf支持的类型不多，但是可以说完全够用:

1. bool
2. 32/64-bit integers
3. float
4. double
5. string
6. byte sequence
7. class(message)
8. list(repeated)
9. enum

Thrift相对于PB支持的类型更丰富一些:

1. Set
2. Map<K, V>
3. Constants
4. Exception

并且貌似支持范型。

**5. 支持更多的编程语言？**

PB支持Java、Javascript、Python、C++。Thrift支持更多。

**6. 字段增减，名称变化，顺序，版本升降级。**

> The system must be able to support reading of old data, as well as requests from out-of-date clients to new servers, and vice versa.
>
> - Versioning in Thrift and Protobuf is implemented via field identifiers.
> - The combination of this field identifiers and its type specifier is used to uniquely identify the field.
> - A new compiling isn't necessary.

PB和Thrift都是采用key-value形式编码，而不是简单的固定value值编码。这样可以根据key来进行灵活反序列化。key一般是tag（不采用fieldName，因为空间更小）和type，还有length（如果需要的话）。

**7. 字符串编码**

作为一个内部RPC框架，其实应该统一字符串编码的，这不是不民主，混用编码很容易导致各种奇怪的乱码问题。然而编码问题很多时候是因为遗留系统的问题，所以为了兼容不同的编码问题，提供这种机制。

欧美国家一般没有这个问题，因为他们都是使用英文(ASCII编码)，一般都使用UTF-8，因为UTF-8无缝兼容ASCII。但是亚洲国家的编码就比较复杂，有GBK，有EUC-KR，有EUC-JP等等。在中国一般是GBK和UTF-8混用。所以有时候还是得需要支持多种编码。

Protobuf是对String类型统一编码为UTF-8的:

> **Strings**
>
> A wire type of 2 (length-delimited) means that the value is a varint encoded length followed by the specified number of bytes of data.
>
> ```
> message Test2 {
>   required string b = 2;
> }
> ```
>
> Setting the value of b to "testing" gives you:
>
> ```
> 12 07 74 65 73 74 69 6e 67
> ```
>
> The red bytes are the UTF8 of "testing". The key here is 0x12 → tag = 2, type = 2. The length varint in the value is 7 and lo and behold, we find seven bytes following it – our string.

从autogen生成的代码也可以看出来：

```
/**
 * <code>required string name = 1;</code>
 */
public com.google.protobuf.ByteString
```

```
    getNameBytes() {
  java.lang.Object ref = name_;
  if (ref instanceof java.lang.String) {
    com.google.protobuf.ByteString b =
        com.google.protobuf.ByteString.copyFromUtf8(
            (java.lang.String) ref);
    name_ = b;
    return b;
  } else {
    return (com.google.protobuf.ByteString) ref;
  }
}
```

其中com.google.protobuf.ByteString.copyFromUtf8方法java doc为：Encodestextinto a sequence of UTF-8 bytes and returns the result as a ByteString.

查看了一下Thrift的文档，Thrift对于String也是采用UTF-8编码。

### 8. idl with autogen(stub)或者idl with metadata(schema)

想自动化升级的话，像Avro这种有schema，不需要预先生成stub的方式是必须的。但是对于后台，特别是强类型的语言，有stub还是很方便的。

### 9. 支持import? 复杂性 vs. 模型重复定义？

Multiple structs/messages can be defined and referred to within the same .thrift/.proto file.

### 10. 是否包含service定义？参数和返回值是统一对象还是打散的参数列表？另外，需要提供接口让应用获取协议头吗？

Protobuf和Thrift都支持service定义，但是两者的接口定义有所不同，protobuf的接口参数和返回值只支持message类型，这意味着接口参数不是打散的参数，而是统一的req和resp对象:

```
service Foo {
  rpc Bar(FooRequest) returns(FooResponse);
}
```

生成:

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

现在的app-platform协议C++这边是采用打散的参数列表，非常不利于升级，增减参数意味着接口变化。java由于统一封装为req和resp对象，所以字段升级就很简单了。

### 11. 包含SLA，比如调用超时时间？

每个接口的SLA其实都不同，如果仅仅IDL仅仅考虑序列化，那么是不需要考虑Service定义和SLA的，但是如果要把IDL作为RPC的输入，而且都支持定义service定义了，而且都支持同步异步了，再支持其他SLA是很正常的。

### 12. 支持抛异常？

Thrift支持异常类型:

```
i32 calculate(1:32 logid, 2:Work w) throws (1:InvalidOperation ouch)
```

### 13. UDP和TCP? Thrift有个oneway关键字，表示client不等待。

> */* * This method has a oneway modifier. That means the client only makes * a request and does not listen for any response at all. Oneway methods * must be void. /* oneway void zip()
>
> Additionally, an async modifier keyword may be added to a void function, which will generate code that does not wait for a response. Note that a pure void function will return a response to the client which guarantees that the operation has completed on the server side. With async method calls the client will only be guaranteed that the request succeeded at the transport layer. Async method calls of the same client may be executed in parallel/out of order by the server.

现在的app-platform中，TCP和UDP的接口定义都是一样的，其实不合理。UDP应该没有Response。或者应该是异步response。

# 参考文档


1. Protocol Buffers
2. PB vs. Thrift vs. Avro
3. Thrift: Scalable Cross-Language Services Implementation
4. thrift 的required、optional探究
5. 通信协议之序列化
6. Protocol Buffers-Encoding

## Related Posts

- 24 Jul 2015 » Tomcat调优
- 22 Jul 2015 » 记一次MySQL主从同步错误处理
- 03 Jul 2015 » Metric监控系统