网络RPC编码协议学习

January 13, 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。

二进制编码

主要是TLV编码及其变体。

TLV是Tag-Length-Value的简称，其实它原来的名称为Type-Length-Value。顾名思义，就是把每个字段按照Type-Length-Value的顺序依次进行编码。有这三个信息，接收端就能够正确的将这个字段进行解码。Protobuf/thrift/ASN BER都属于这种。

1. tag: 一般占用1个字节，表示数据类型，有的编码方式（Protobuf/thrift）中tag包含字段的id，有的编码方式（ASN BER）不包含字段的id。包含字段id的序列化方式，id是字段的标志，协议可以灵活的增删字段，只要保证字段id唯一，就能兼容解析，非常适合互联网开发。
2. length：一个整数，表示后面数据块的长度，Protobuf/thrift的序列化不包含length字段，因为大部分数据类型的长度都可以根据tag中的类型信息可以得到。
3. value：真正的数据内容。

TLV编码的特点

- 解析效率高：主要是因为不需要转义字符
- 编码长度低：主要是因为元数据占用的空间很少
- 不易于实现：但是有很多开源的工具，根据IDL自动生成代码，提高开发效率
- 兼容性高：协议双方可以独立升级
- 可读性差：二进制协议，肉眼很难识别

TLV编码例子

1. tag包含id的序列化方式打包解包的例子

说明：只是举个例子说明原理，实际上protocol buffer等协议都做了比较巧妙的实现，比如VInt、zig-zag 编码来尽量减少编码长度。

假设数据包括2个字段：name字段的id为0, 类型为1(string); age字段的id为1, 类型为2(unsigned int)。

字段id	字段类型	字段名
0	string	name

0 string name
1 unsigned int age

需要传输的数据：

```
name = "xxx"
age = 18
```

序列化之后大约是

字段类型(tag的一部分)	字段id(tag的一部分)	字段值(value)
0x01	0x00	xxx
0x02	0x01	0x12

反序列化的时候，逐步解析字节流，先解析字段类型和字段id，再根据字段类型解析出后面的数据内容，得到了一个id和值的映射关系：

```
0 : "xxx"
1 : 18
```

根据协议，id=0的字段表示name，id=1的字段表示age，反序列化之后，就知道传过来的数据是

```
name = "xxx" , age = 18
```

如果协议做了升级，增加了1个字段“gender”，删除一个已经没有意义的字段age，协议变成

```
0 string name
2 string gender
```

需要传输的数据：

```
name = "xxx"
gender = "male"
```

发送方升级了协议，序列化之后大约是

字段类型(tag的一部分)	字段id(tag的一部分)	字段值(value)
0x01	0x00	xxx
0x01	0x02	male

反序列化之后，得到了一个id和值的映射关系

```
0 : "xxx"
2 : "male"
```

反序列化的一方由于没有升级协议，不知道id=2的字段什么意思，直接忽略，没找到id=1的age字段，那么使用默认值，这样单方的升级，完全不影响协议的解析，协议是具有兼容性的。

There are four cases in which version mismatches may occur.

1. *Added field, old client, new server.* In this case, the old client does not

send the new field. The new server recognizes that the field is not set, and implements default behavior for out-of-date requests.
2. *Removed field, old client, new server.* In this case, the old client sends the removed field. The new server simply ignores it.
3. *Added field, new client, old server.* The new client sends a field that the old server does not recognize. The old server simply ignores it and processes as normal.
4. *Removed field, new client, old server.* This is the most dangerous case, as the old server is unlikely to have suitable default behavior implemented for the missing field. It is recommended that in this situation the new server be rolled out prior to the new clients.

可见，只有case 4 才需要先升级server。不过一般来说字段删除是不允许。而且一般也是server先做兼容性升级，client再跟着升级。client先升级一般风险比较高。

2. tag不包含id的序列化方式打包解包的例子

如果tag中没有字段id，那么一般采用字段所在的位置或者字段的名称来替代（注意：前者意味着字段的定义顺序不能随便调整；后者意味着字段的命名不能随意调整，使用id有个好处就是协议包体积更小）。而在带tag的编码方式中，名称并不重要，只是一个标识，用于生成相应的getter和setter方法而已。字段在IDL文件中的所在位置也跟他们的序列化顺序无关，而是由tag决定，所以同一个意义的字段的tag必须相等。 简单起见，这里采用字段的位置作为替代（这也是我们现在App Platform的实现方式）。

协议包括2个字段， 第1个字段name，类型为1（string）；第2个字段age类型为2（unsigned int ）

字段类型	字段名
string	name
unsigned int	age

需要传输的数据：

```
name = "xxx"
age = 18
```

序列化之后大约是

字段类型	字段值
0x01	xxx
0x02	0x12

反序列化程序解析出第1个字段是字符串xxx，第二个字段是整数18，根据协议(一般是autogen出来的stub)，第1个字段是name，第2个字段是age，这时反序列化程序就知道了name是xxx，age是18。

但是相比上面有id的序列化方式，这种方式有个明显的缺陷：一方升级了协议时，另一方很可能需要升级协议才行，协议不具有兼容性。比如协议做了升级，增加了一个字段gender，删除一个已经没有意义的字段age，协议变成

```
string name
```

```
string gender
```

需要传输的数据：

```
name = "xxx"
gender = "male"
```

发送方升级了协议，序列化之后大约是

字段类型	字段值
0x01	xxx
0x01	male

这时接收方如果不升级协议就完全无法理解协议的含义。

可以看出tag包含ID的序列化方式（Protobuf/thrift）兼容性和灵活性方面优于不包含ID的方式（asn-ber）

像Protocol Buffers和Thrift，都是通过定义IDL文件，然后生成各种语言的stub类，这些类包含了serilized和unserilized方法。由于是生成类的方式，所以有时候autogen工具的版本也会是序列化失败的原因之一。另外，需要把生成后的类丢给相应的调用方也是比较麻烦的。

序列化一般要么作为IPC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。

文本流编码

xml/json都属于这种。

基本原理是把每个字段打一个字符串形式的包，通过键值对（key-value）的方式存储数据，key是字段的名字，用于区分不同的字段（对比上面TLV编码采用id的方式标志一个字段），特殊字符特别是非文本字符需要做适当转义，转义为xml/json的合法字符。xml的解析效率低于json，而编码长度高于json，json作为序列化的方式一般是优于xml的。

同样是上面的例子，序列化的结果大概是:

```
<p><name>xxx</name><age>18</age></p>
```

或者:

```
{name:xxx,age:18}
```

文本流编码的特点是：解析效率低，编码长度高，易于实现，可扩展性高，可读性好。

混合型编码

目前主要是Avro，官网上有介绍，非常到位：

> ## Comparison with other systems
>
> Avro provides functionality similar to systems such as Thrift, Protocol Buffers,

etc. Avro differs from these systems in the following fundamental aspects.

- Dynamic typing: Avro does not require that code be generated. Data is always accompanied by a schema that permits full processing of that data without code generation, static datatypes, etc. This facilitates construction of generic data-processing systems and languages.
- Untagged data: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.
- No manually-assigned field IDs: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names

参考文章

1. 通信协议之序列化 从最开始实现方案逐步完善到一个可以扩展方便维护和调试的协议，非常具有启发性。强烈推荐。

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