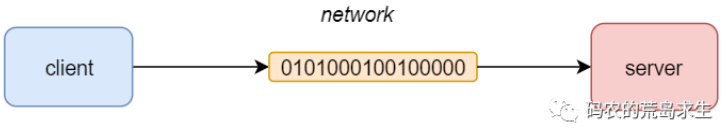
神奇的Google二进制编解码技术：Protobuf

Original 陆小风 码农的荒岛求生 2022-09-04 17:10 Posted on 北京

计算机网络编程中一个非常基本的问题：该怎样表示client与server之间交互的数据，在往下看之前先想一想这个问题。

共识与协议

这个问题可不像看上去的那样简单，因为client进程和server进程运行在不同的机器上，这些机器可能运行在不同的处理器平台、可能运行在不同的操作系统、可能是由不同的编程语言编写的，server要怎样才能识别出client发送的是什么数据呢？就像这样：



client给server发送了一段数据：

```
0101000100100001
```

server怎么能知道该怎样“解读”这段数据呢？

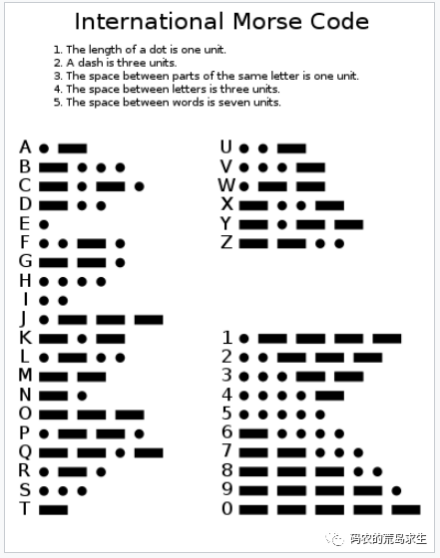
显然，client和server在发送数据之前必须首先达成某种关于怎样解读数据的共识，这就是所谓的**协议**。

这里的协议可以是这样的：“将每8个比特为一个单位解释为无符号数字”，如果协议是这样的，那么server接收到这串二进制后就会将其解析为81(01010001)与33(00100001)。

当然，这里的协议也可以是这样的：“将每8个比特为一个单位解释为ASCII字符”，那么server接收到这串二进制后就将其解析为“Q!”。

可见，同样一串二进制在不同的“上下文/协议”下有完全不一样的解读，**这也是为什么计算机明明只认知0和1但是却能处理非常复杂任务的根本原因，因为一切都可以编码为0和1，同样的我们也可以从0和1中解析出我们想要的信息，这就是所谓的编解码技术。**

实际上不止0和1，我们也可以将信息编码为摩斯密码(Morse code)等，只不过计算机擅长处理0和1而已。

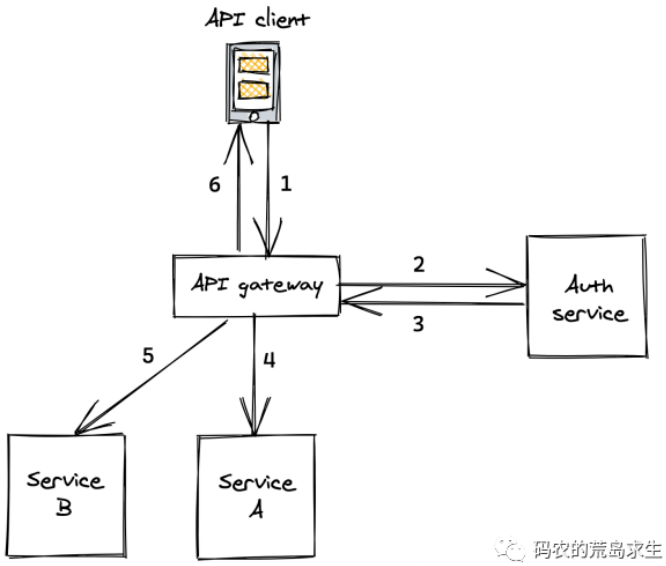


扯远了，回到本文的主题。

远程过程调用：RPC

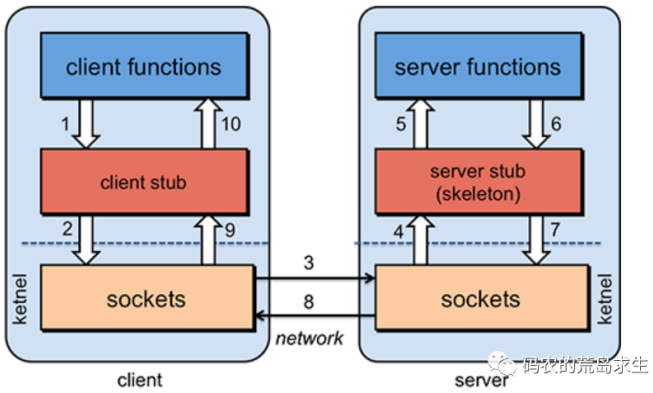
作为程序员我们知道，client以及server之间不会简单传递一串数字以及字符这样简单，尤其在互联网大厂后端服务这种场景下。

当我们在电商App搜索商品、打车App呼叫出租车以及刷短视频时，每一次请求的背后在后端都涉及大量服务之间的交互，就像这样：



完成一次客户端请求gateway这个服务要调用N多个下游服务，所谓调用是说A服务向B服务发送一段数据（请求），B服务接收到这段数据后执行相应的函数，并将结果返回给A服务。

只不过对于服务A来说并不想关心网络传输这样的底层细节，如果能像调用本地函数一样调用远程服务就好了，这就是所谓的RPC，经典的实现方式是这样的：



RPC对上层提供和普通函数一样的接口，只不过在实现上封装了底层复杂的网络通信，RPC框架是当前互联网后端的基石之一，很多所谓互联网后端的职位无非就是在此基础之上堆业务逻辑。

本文我们不关心其中的细节，这里我们只关心在网络层client是怎样对请求参数进行编码、server怎样对请求参数进行解码的，也就是本文开头提出的问题。

信息的编解码

在思考怎样进行编解码之前我们必须意识到：

- **client**和**server**可能是用不同语言编写的，你的编解码方案必须通用且不能和语言绑定
- 编解码方法的性能问题，尤其是对时间要求苛刻的服务

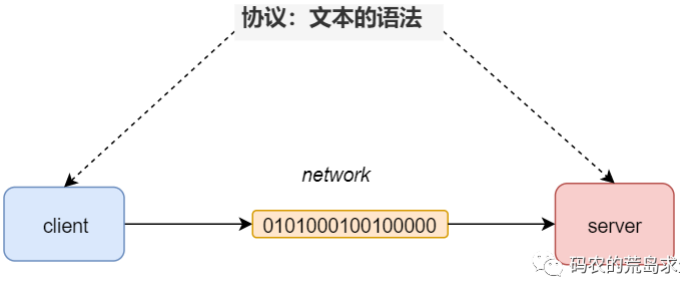
首先，我们最应该能想到的就是以纯文本的形式来表示。

纯文本从来都是一种非常有友好的信息载体，为什么？很简单，因为人类(我们)可以直接看懂，就像这段：

```
{
 "widget": {
  "window": {
   "title": "Sample Konfabulator Widget",
   "name": "main_window",
   "width": 500,
   "height": 500
```

```
    },
    "image": {
     "src": "Images/Sun.png",
     "name": "sun1",
     "hOffset": 250,
     "vOffset": 250,
    },
   }
  }
```

是不是很清晰，一目了然，只要我们实现约定好文本的结构(也就是语法)，那么client和server就能利用这种文本进行信息的编码以及解码，不管client和server是运行在x86还是Arm、是32位的还是64位的、运行在Linux上还是windows上、是大端还是小端，都可以无障碍交流。



因此在这里，文本的语法就是一种协议。顺便说一句，**你都规定好了文本的语法，实际上就相当于发明了一种语言**。

这里用来举例用的语言就是所谓的Json，只不过json这种语言不是用来表示逻辑(代码)而是用来存储数据的。

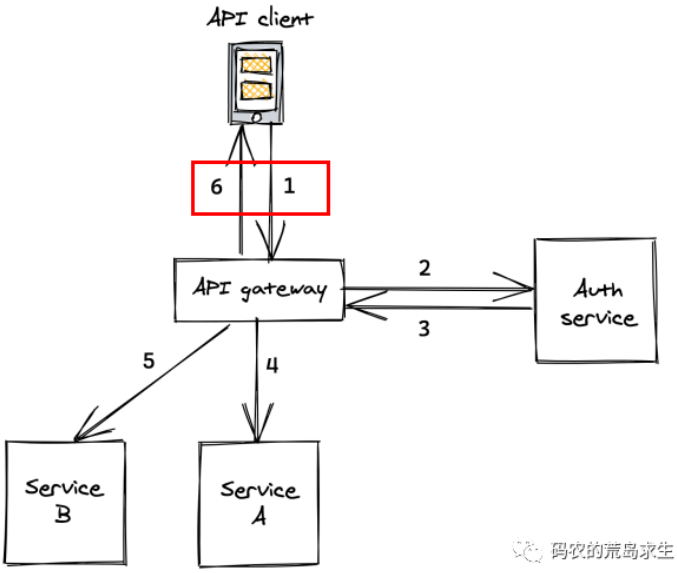
Json就是这个老头提出来的：

除了Json，另一种利用文本存储数据的表示方法是XML，来一段感受下：

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

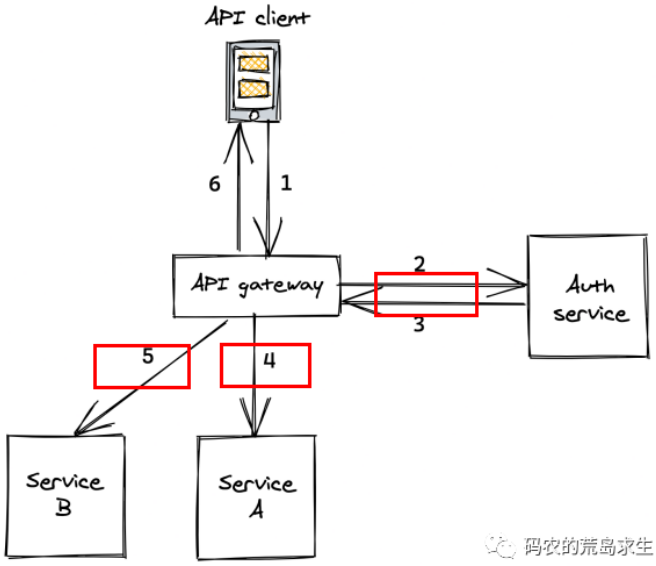
相对Json来说是不是就没那么容易看懂了，Json出现后在web领域逐渐取代了XML。

当两段数据量很少的时候——就像浏览器和服务端的交互，Json可以工作的非常好，这个场景就是这里：



在这里是json的天下。

但对于后端服务之间的交互来说就不一样了，后端服务之间的RPC调用可能会传输大量数据，如果全部用纯文本的形式来表示数据那么不管是网络带宽还是性能可能都会差强人意。



在这种场景下，Json并不是最好的选项，主要原因之一就在于性能以及数据的体积。

我们知道，文本表示对人类是最友好的，对机器来说则不是这样，对机器来说最好的还是01二进制。

那么有没有二进制的编码方法吗？答案是肯定的，这就是当前互联网后端中流行的protobuf，Google公司开源项目。

那么protobuf有什么神奇之处吗？

假设client端想给server端传输这样一段信息：“我有一个id，其值为43”，那么在XML下是这样表示的：

```
<id>43</id>
```

数一数这这段数据占据了多少字节，很显然是11字节；

而如果用json来表示呢？

```
{"id":43}
```

数一数这段数据占据了多少字节，显然是9字节；

而如果用protobuf来表示呢？是这样的：

```
// 消息定义
message Msg {
  optional int32 id = 1;
}

// 实例化
Msg msg;
msg.set_id(43);
```

其中Msg的定义看上去比Json和XML更加复杂了，但这些只是给人看的，这些还会被protbuf进一步处理，最终被编码为：

也就是0x08与0x2b，这占据了多少字节呢？答案是2字节。

从json的9字节到protobuf的2字节，数据大小减少了4倍多，数据量的减少意味着：

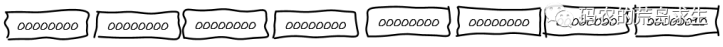
- 更少的网络带宽
- 更快的解析速度

那么protobuf是怎样做到这一点的呢？

protobuf是怎样实现的？

首先，我们来思考最简单的情况，该怎样表示数字。

你可能会想这还不简单，统一用固定长度，比如用64个比特(8字节)，这种方法可行，但问题是不论一个数字有多小，比方2，那么用这种方法表示2也需要占据64个比特(8字节)：



明明只要一个字节就能表示而我们却用了8个，前面的全都是0，这也太奢侈太浪费了吧。

显然，**在这里我们不能使用固定长度来表示数字，而需要使用变长方法来表示。**

什么叫变长？意思是说如果数字本身比较大，那么其使用的比特位可以较多，但如果数字很小那么就应该使用较少的比特位来表示，这就叫变长，随机应变，不死板。

那怎样变长呢？

我们规定：对于每一个字节来说，**第一个比特位如果是1那么表示接下来的一个比特依然要用来解释为一个数字，如果第一个比特为0，那么说明接下来的一个字节不是用来表示该数字的。**

也就是说对于每个8个比特(1字节)来说，它的有效载荷是7个比特，第一个比特仅仅用来标记是否还应该把接下来的一个字节解析为数字。

根据这个规定假设来了这样一串01二进制：

```
1010110000000010
```

根据规定，我们首先取出第一个字节，也就是：

```
10101100
```

此时我们发现第一个比特位是1，因此我们知道接下来的一个字节也属于该数字，将当前字节的1去掉就是：

```
0101100
```

然后我们看下一个字节：

```
00000010
```

我们发现第一个bit为0，因此我们知道下一个字节不属于该数字了。

接下来我们将解析到的0101100(第一个字节去掉第一个比特位)以及第二个字节0000010(第二个字节去掉第一个比特位)翻转之后拼接到一起，这里之所以翻转是因为我们规定数字的高位在后。

这个过程就是：

```
    1010110000000010
->  10101100 | 00000010 // 解析得到两个字节
    _          _

->  0101100  |  0000010  // 各自去掉最高位
->  0000010  |  0101100  // 两个字节翻转顺序

    0000010  +  0101100
->  100101100           // 拼接
```

最后我们得到了100101100，这一串二进制表示数字300。

这种数字的变长表示方法在protobuf中被称之为varint。

因此在这种表示方法下，如果数字较大，那么使用的比特就多，如果数字较小那么使用比特就少，聪明吧。

有的同学看到这里可能会问题，刚才讲解的方法只能表示无符号数字，那么有符号数字该怎么表示呢？比如-2该怎么表示？

有符号数的表示

按照刚才变长编码的思想，-2147483646使用的比特位应该比-2要少。

然而我们知道在计算机世界中负数使用补码表示的，也就是说最高位(最左侧的比特位)一定是1，假设我们使用64位来表示数字，那么如果我们依然用补码来表示数字的话那么无论这个负数有多大还是多小都需要占据10个字节的空间。

为什么是10个字节呢？

不要忘了varint每个字节的有效负荷是7个比特，那么对于需要64位表示的数字来说就需要64/7向上取整也就是10个字节来表示。

这显然不能满足我们对数字变长存储的要求。

该怎么解决这个问题呢？

既然无符号数字可以方便的进行变长编码，那么我们将有符号数字映射称为无符号数字不就可以了，这就是所谓的ZigZag编码，是不是很聪明，就像这样：

原始信息	编码后
0	0
-1	1
1	2
-2	3
2	4
-3	5
3	6
...	...

```
2147483647    4294967294
-2147483648   4294967295
```

这样我们就可以将有符号数字转为无符号数字，接收方接收到该数据后再恢复出有符号数字。

现在数字的问题彻底解决了，但这仅仅是万里长征第一步。

字段名称与字段类型

对于任何一个有用的信息都包含这样几部分：

- 字段名称
- 字段类型
- 字段值

就像C/C++中定义变量时：

```
int i = 100;
```

在这里，字段名称就是i，字段类型是int，字段值是100。

刚才我们用varint以及ZigZag编码解决了字段值表示的问题，那么该怎样表示字段名称和字段类型呢？

首先，对于字段类型还比较简单，因为字段类型就那么多，protobuf中定义了6种字段类型：

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

对于6种字段类型我们使用3个比特位来表示就足够了。

接下来比较有趣的是字段名称该怎么表示呢？假设我们需要传递这样一个字段：

```
int long_long_name = 100;
```

那么我们真的需要把“long_long_name”这么多字符通过网络传递给对端吗?

既然通信双方需要协议，那么“long_long_name”这字段其实是client和server都知道的，它们唯一不知道的就是“**哪些值属于哪些字段**”。

为解决这个问题，**我们给每个字段都进行编号**，比如通信双方都知道“long_long_name”这个字段的编号是2，那么对于：

```
int long_long_name = 100;
```

这个信息我们只需要传递：

- 字段名称：**2 (2对应字段“long_long_name”)**
- 字段类型：**0 (0表示varint类型，参见上图)**
- 字段值：**100**

所以我们可以看到，**无论你用多么复杂的字段名称也不会影响编码后占据的空间，字段名称根本就不会出现在编码后的信息中，**so clever。

从宏观上看

我们已经在protobuf中看到了数字以及字段名称以及字段类型是怎么表示了，现在是时候从宏观角度来看看多个字段该怎么编码了。

从本质上讲，protobuf被编码后形成一系列的key-value，每个key-value对应一个proto中的字段。

也就是键值对：

其中value比较简单，也就是字段值；而字段名称和字段类型会被拼接成key，protobuf中共有6种类型，因此只需要3个比特位即可；字段名称只需要存储对应的编号，这样可以就可以这样编码：

```
(字段编号 << 3) | 字段类型
```

假设server接收到了一个key为0x08，其二进制的表示为：

```
0000 1000
```

由于key也是利用varint编码的，因此需要将第一个比特位去掉，这样我的得到：

```
000 1000
```

根据key的编码方式，其后三个比特位表示字段类型，即：

```
000
```

也就是0，这样我们知道该key的类型是Varint(第0号类型)，而字段编号为抹掉后3个比特位的值，即：

```
0001
```

这样，我们就知道了该key对应的字段编号为1，得到编号我们就能根据编号找到对应的编号名称。

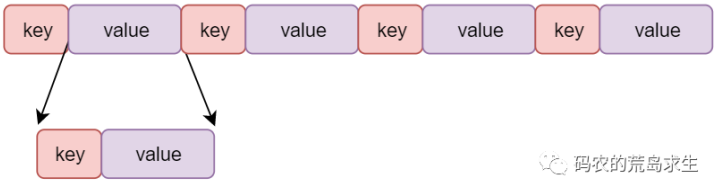
嵌套数据

与Json和XML类似，protobuf中也支持嵌套消息，就像这样：

```
message SubMsg {
  optional int32 id = 1;
}

message Msg {
  optional SubMsg msg = 1;
}
```

其实现也比较简单，这依然遵循被编码后形成一系列的key-value，只不过对于嵌套类型的key来说，其value是由子消息的key-value组成。



protobuf与编译语言

与Json一样，protobuf也是一门语言，兼具了文本的可读性以及二进制的高效。

protobuf之所以能做到这一点就好比C语言与机器指令。

C语言是给程序员看的，可读性好，而机器指令是给硬件使用的，性能好，编译器会将C语言程序转为机器可执行的机器指令。

而protobuf也一样，protobuf也是一门语言，会将可读性较好的消息编码为二进制从而可以在网络中进行传播，而对端也可以将其解码回来。

在这里protobuf中定义的消息就好比C语言，编码后的二进制消息就好比机器指令。

而protobuf作为事实上语言必然有自己的语法，其语法就是这样：

```
message   := (tag value)*     You can think of this as "key value"

tag       := (field << 3) BIT_OR wire_type, encoded as varint
value     := (varint|zigzag) for wire_type==0 |
             fixed32bit      for wire_type==5 |
             fixed64bit      for wire_type==1 |
             delimited       for wire_type==2 |
             group_start     for wire_type==3 | This is like "open parenthesis"
             group_end       for wire_type==4   This is like "close parenthesis"

varint       := int32 | int64 | uint32 | uint64 | bool | enum, encoded as
                varints
zigzag       := sint32 | sint64, encoded as zig-zag varints
fixed32bit   := sfixed32 | fixed32 | float, encoded as 4-byte little-endian;
                memcpy of the equivalent C types (u?int32_t, float)
fixed64bit   := sfixed64 | fixed64 | double, encoded as 8-byte little-endian;
                memcpy of the equivalent C types (u?int64_t, double)

delimited := size (message | string | bytes | packed), size encoded as varint
message   := valid protobuf sub-message
string    := valid UTF-8 string (often simply ASCII); max 2GB of bytes
bytes     := any sequence of 8-bit bytes; max 2GB
packed    := varint* | fixed32bit* | fixed64bit*,
             consecutive values of the type described in the protocol definition

varint encoding: sets MSB of byte to 1 to indicate that there a
zigzag encoding: sint32 and sint64 types use zigzag encoding.
```


怎么样，还觉得编译原理没什么用吗？

不理解编译原理是不可能发明protobuf这种技术的。

总结

我在写这篇文章时不断感叹，Google的这项技术节省了多少程序员的时间，同时我们也能看到这种基石般的技术依赖的底层原理却非常古老：

- 信息的编解码
- 编译原理

怎么样，这些是不是远远没有IT界各种流行的技术听上去时髦有趣，而正是这种朴素的技术支撑起了工业界，现在你也应该能明白底层技术的重要性了吧。