gRPC系列(三) 如何借助HTTP2实现传输


渔人
让知识更容易传播

本系列分为四大部分：

- gRPC系列(一) 什么是RPC?
- gRPC系列(二) 如何用Protobuf组织内容
- **gRPC系列(三)如何借助HTTP2实现传输**
- gRPC系列(四) 框架如何赋能分布式系统

回顾

在系列二中，我们一起学习了gRPC如何使用Protobuf来组织数据，达到高效编解码、高压缩率的目标。本文我们将更进一步，看看这些数据是如何在网络中被传输的，达到以更低的资源实现更高效传输的目标。内容将围绕以下几点展开：

- HTTP2 要解决的问题，HTTP1.1的缺点
- HTTP2 的原理，它是如何降低传输成本，借此我们更深入理解何为 二进制编码 ； 同时它是如何提高网络资源利用效率，重温 多路复用 的思想
- 拉通Protobuf和HTTP2，通过抓包，从 数据和协议 角度洞悉gRPC调用

网络传输的目标

数据的传输，都是被切割成一个个小块，包在层层网络协议头里，通过一个个路由器依次转发，最终到达目的地，被重新组装起来。这是网络传输的基本原理，在这个过程中，有两个亘古不变的目标：

- 更快的传输。快的背后就是少，传输的数据越少、越小，整体的速度也就越快。
- 更低的资源消耗。这背后是资源的高效利用，就像cpu那样，压榨的越厉害，就越节约资源。

随着行业的发展，对上述两个目标的追求也更加极致，要想传输速度更快，传输的数据体积要小。

如何降低HTTP1.1在协议上的消耗

HTTP1.1 被视为差生

HTTP1.1以其简单、可读性高、超高普及率、历史悠久，作为经典的存在，为互联网的普及做出了重要贡献。但在当今超高的流量、超高的使用频率背景下，打开一个页面动辄几十个请求，使得速度已经难以满足贪婪人类的需求。这主要表现在以下几个方面：

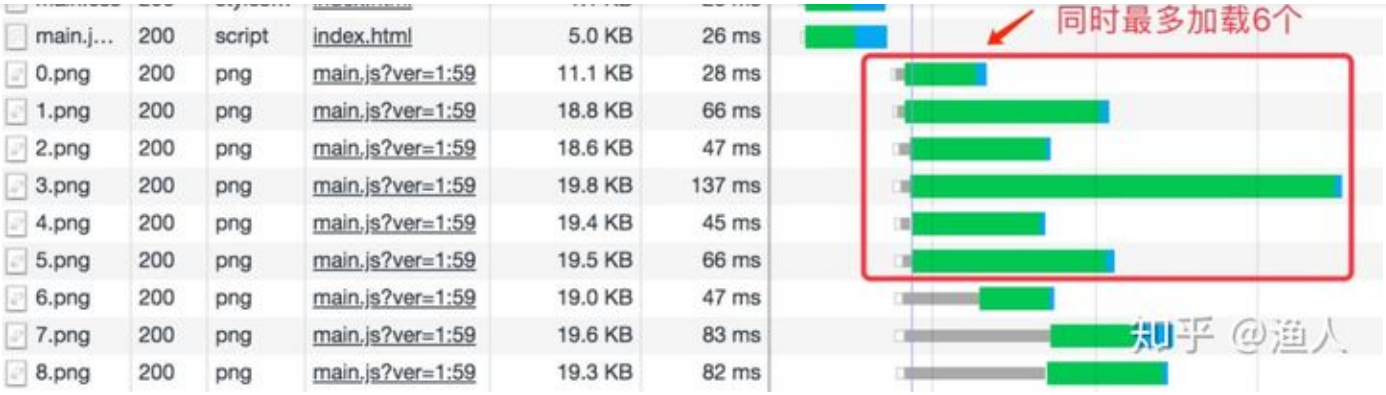
一、 **冗余文本过多，导致传输体积很大**
作为一款经典的无状态协议，它使得Web后端可以灵活地转发、横向扩展，但其代价是每个请求都会带上冗余重复的Header，这些文本内容会消耗很多空间，和 更快传输 的目标相左。

二、 **并发能力差，网络资源利用率低**
HTTP1.1 是基于文本的协议，请求的内容打包在header/body中，内容通过\r\n来分割，**同一个TCP连接中，无法区分request/response是属于哪个请求**，所以无法通过一个TCP连接并发地发送多个请求，只能等上一个请求的response回来了，才能发送下一个请求，否则无法区分谁是谁。

于是H1.1提出了一个pipeline的特性，允许请求方一口气并发多个request，但对服务方有一个变态的要求，需要对应的response按照request的顺序严格排列，因为不按顺序排列就分不清楚response是属于哪个request的。这给Proxy(Nginx等)带来了复杂性，同时如果第一个请求迟迟不返回，那后面的请求都会受影响，所以普及率不高。

但当今的Web页面有玲琅满目的图片、js、css，如果让请求一个个串行执行，那页面的渲染会变得极慢。于是只能同时创建多个TCP连接，实现并发下载数据，快速渲染出页面。这会给浏览器造成较大的资源消耗，电脑会变卡。很多浏览器为了兼顾下载速度和资源消耗，会对同一个域名限制并发的TCP连接数量，如Chrome是6个左右，剩下的请求则需要排队，Network下的Waterfall就可以观察排队情况 (见下图右边的颜色条)。

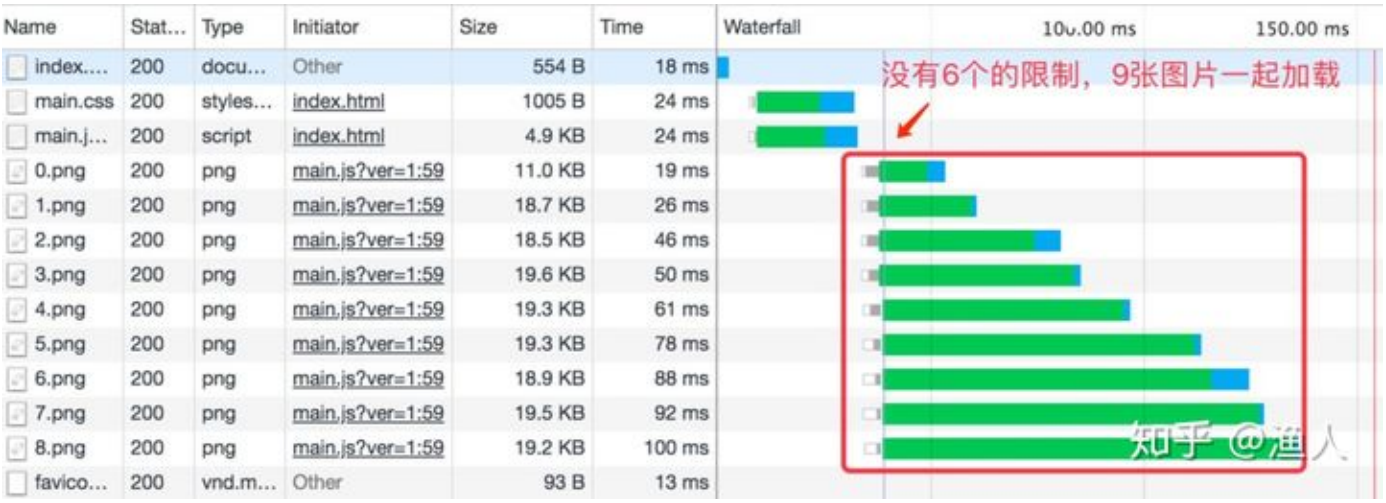
H1.1时，有6个并发连接，可以看到最下面三个请求在排队：



main.j...	200	script	index.html	5.0 KB	26 ms
0.png	200	png	main.js?ver=1:59	11.1 KB	28 ms
1.png	200	png	main.js?ver=1:59	18.8 KB	66 ms
2.png	200	png	main.js?ver=1:59	18.6 KB	47 ms
3.png	200	png	main.js?ver=1:59	19.8 KB	137 ms
4.png	200	png	main.js?ver=1:59	19.4 KB	45 ms
5.png	200	png	main.js?ver=1:59	19.5 KB	66 ms
6.png	200	png	main.js?ver=1:59	19.0 KB	47 ms
7.png	200	png	main.js?ver=1:59	19.6 KB	83 ms
8.png	200	png	main.js?ver=1:59	19.3 KB	82 ms

图片来源[5]

HTTP2中，可以看出请求时同时发出的，没有排队，且只占用一个连接：



Name	Stat...	Type	Initiator	Size	Time	Waterfall
index....	200	docu...	Other	554 B	18 ms	
main.css	200	styles...	index.html	1005 B	24 ms	
main.j...	200	script	index.html	4.9 KB	24 ms	
0.png	200	png	main.js?ver=1:59	11.0 KB	19 ms	
1.png	200	png	main.js?ver=1:59	18.7 KB	26 ms	
2.png	200	png	main.js?ver=1:59	18.5 KB	46 ms	
3.png	200	png	main.js?ver=1:59	19.6 KB	50 ms	
4.png	200	png	main.js?ver=1:59	19.3 KB	61 ms	
5.png	200	png	main.js?ver=1:59	19.3 KB	78 ms	
6.png	200	png	main.js?ver=1:59	18.9 KB	88 ms	
7.png	200	png	main.js?ver=1:59	19.5 KB	92 ms	
8.png	200	png	main.js?ver=1:59	19.2 KB	100 ms	
favico...	200	vnd.m...	Other	93 B	13 ms	

图片来源[5]

狡猾的人类为了避开这个数量限制，将图片、css、js等资源放在不同域名下(或二级域名)，避开排队导致的渲染延迟。快速下载的目标实现了，但这和 更低的资源消耗 目标相违背，背后都是高昂的带宽、CDN成本。

HTTP2 当救世主

H1.1 在速度和成本上的权衡让人纠结不已，HTTP2的出现就是为了优化这些问题，在 更快的传输 和 更低的成本 两个目标上更进了一步。有以下几个基本点：

- HTTP2 未改变HTTP的语义(如GET/POST等)，只是在传输上做了优化

核心可分为 头部压缩 和 多路复用 。这两个点都服务于 更快的传输 、 更低的资源消耗 这两个目标，与上文呼应。

头部压缩

现在的web页面，大多比较复杂，新打开一个地址，动辄产生几十个请求，这会发送大量的header，大部分内容都是一样的内容，以baidu为例:

```
request:
GET  HTTP/1.1
Host: www.baidu.com
Cache-Control: no-cache
Postman-Token: a9702bac-94c4-c7da-2041-7c7ac5f85b6e

response:
Access-Control-Allow-Credentials: true
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html; charset=UTF-8
Server: Apache
Transfer-Encoding: chunked
Vary: Accept-Encoding
```

这些文本内容一次次重复地发送，占用了大量的带宽，如何将这些成本降下去，而又保留HTTP无状态的优点呢?

基于这个想法，诞生了HPACK[2]，全称为 HTTP2头部压缩 ，它以极富创造力的方式，提供了两种方式极大地降低了header的传输占用。

一、**将高频使用的Header编成一个静态表**，每个header对应一个数组索引，每次只用传这个索引，而不是冗长的文本。表总共有61项，下图是前30项:

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	accept	
20	access-control-allow-origin	
21	age	
22	allow	
23	authorization	
24	cache-control	
25	content-disposition	
26	content-encoding	
27	content-language	
28	content-length	
29	content-location	
30	content-range	

图片来源[3]

- 传 3 代表 "POST"，这用一个字节表示了原来4个字节
- 传28代表content-length，这用一个字节表示了原来14个字节（value下文会讨论）

可以预见这种方式，在大量的请求环境下，可以明显降低传输内容。服务端根据内容查表，就可以还原出header。

二、**支持动态地**在表中增加header

header，例如：

```
62   Host: www.baidu.com
```

在请求发起前，通过协议将上面Header添加到表中，则后面的请求都只用发送62即可，不用再发送文本，这又节约了大量空间。(请求方/服务方的 表成员 会保持同步一致)

上面两个分别被成为静态表和动态表。静态表是协议级别的约定，是不变的内容。动态表则是基于当前TCP连接进行协商的结果，发送请求时会相互设置好header，让请求方和服务方维护同一份动态表，后续的请求可复用。连接销毁时，动态表也会注销。

多路复用

H1.1核心的尴尬点在于，在同一个TCP连接中，没办法区分response是属于哪个请求，一旦多个请求返回的文本内容混在一起，就天下大乱，所以请求只能一个个串行排队发送。这直接导致了TCP资源的闲置。

HTTP2为了解决这个问题，提出了 流 的概念，每一次请求对应一个流，有一个唯一ID，用来区分不同的请求。基于流的概念，进一步提出了 帧 ，一个请求的数据会被分成多个帧，方便进行数据分割传输，每个帧都唯一属于某一个流ID，将帧按照流ID进行分组，即可分离出不同的请求。这样同一个TCP连接中就可以同时并发多个请求，不同请求的帧数据可穿插在一起，根据流ID分组即可。**这样直接解决了H1.1的核心痛点，通过这种复用TCP连接的方式，不用再同时建多个连接，提升了TCP的利用效率。** 这也是 多路复用 思想的一种落地方式，在很多消息队列协议中也广泛存在，如AMQP[4]，其 `channel` 的概念和 流 如出一辙，大道相通。

在HTTP2中，流是一个逻辑上的概念，实际上就是一个int类型的ID，可顺序自增，只要不冲突即可，每条 帧 数据都会携带一个流ID，当一串串帧在TCP通道中传输时，通过其流ID，即可区分出不同的请求。

帧则有更多较为复杂的作用，HTTP2几乎所有数据交互，都是以帧为单位进行的，包括header、body、约定配置(除了Magic串)，这天然地就需要给帧进行分类，于是协议约定了以下帧类型：

- HEADERS：帧仅包含 HTTP header信息。
- DATA：帧包含消息的所有或部分请求数据。

- PING：检测信号和往返时间。（流ID为0）[会ACK]
- GOAWAY：停止为当前连接生成流的停止通知。
- WINDOW_UPDATE：用于流控制，约定发送窗口大小。
- CONTINUATION：用于继续传送header片段序列。

一次HTTP2的请求有以下过程：

- 通过一个或多个SETTINGS帧约定一些数据（会有ACK机制，确认约定内容）
- 请求方通过HEADERS帧将 请求A header打包发出
- 请求B可穿插…
- 请求方通过DATA帧将 请求A request数据打包发出
- 服务方通过HEADERS帧将 请求A response header打包发出
- 请求C可穿插…
- 服务方通过DATA帧将 请求A response数据打包发出

深入HTTP2

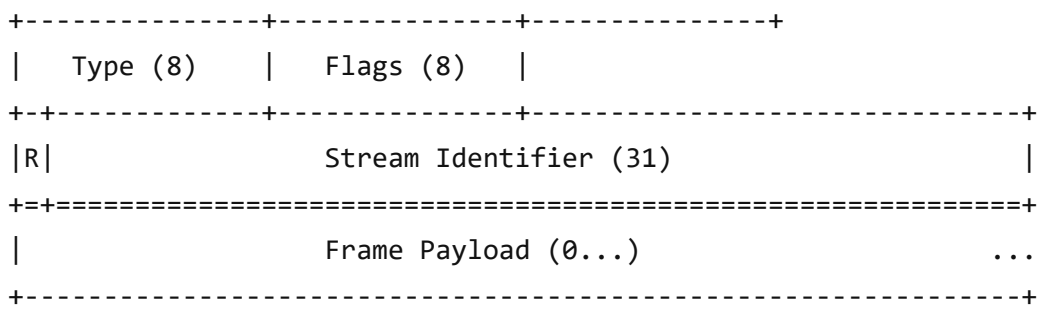
前文简单介绍了，头部压缩和多路复用的具体思路和解决问题的方法，接下来我们深入HTTP2，看看这两个特性是如何落地的，在数据上形成直观地把握，也借此了解何为 二进制编码 。

任何一个应用层的传输协议，都需要解决一个问题，那就是如何表示数据结尾，如何分割数据。在H1.1中，我们知道，它粗暴地先发Header，再发body，每个header通过 `\r\n` 文本内容来分割，header和body通过 `\r\n\r\n` 来分割，通过content-length的值读取body，一个请求的内容就成功结束。

```
// 一次请求的返回
200 OK\r\nHeader1:Value1\r\nHeader2:Value2\r\nHeader3:Value3\r\n\r\nI am body
// 网络中实际传输的是上面文本的ascii编码
```

HTTP2 为了降低协议占用，不会使用文本分割，也不会使用文本来表示header。它是如何表示一帧开始、一帧结束、header传完了、body传完了呢?

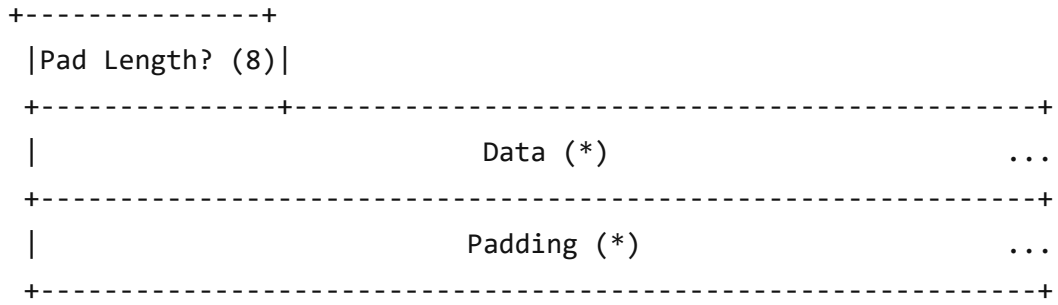
下面是帧格式，所有帧都是一个固定的 9 字节头部 (payload 之前) 跟一个指定长度的数据 (payload):



- Length 代表整个帧的长度，用一个 24 位无符号整数表示。头部的 9 字节不算在这个长度里。从payload开始读Length这么多字节，一帧数据也就读完结束。
- Type 定义 帧 的类型，用 8 bits 表示。帧类型决定了帧的格式和语义，不同类型有差异
- Flags 是为帧类型相关而预留的布尔标识。标识对于不同的帧类型赋予了不同的语义，例如下面会提到的Padding
- R 是一个保留的比特位。这个比特的语义没有定义，发送时它必须被设置为 (0x0), 接收时需要忽略。
- Stream Identifier 唯一标示一个流，用 31 位无符号整数表示。客户端建立的 sid 必须为奇数，服务端建立的 sid 必须为偶数，值 (0x0) 保留给与整个连接相关联的帧 (连接控制消息)，而不是单个流
- Frame Payload 是主体内容，由帧类型决定（上面的9个字节都是协议本身的消耗，payload才是请求本身的主要内容）

不同的帧类型，有不同的Payload格式，我们分别介绍DATA帧和HEADDERS帧：

DATA帧的Payload:

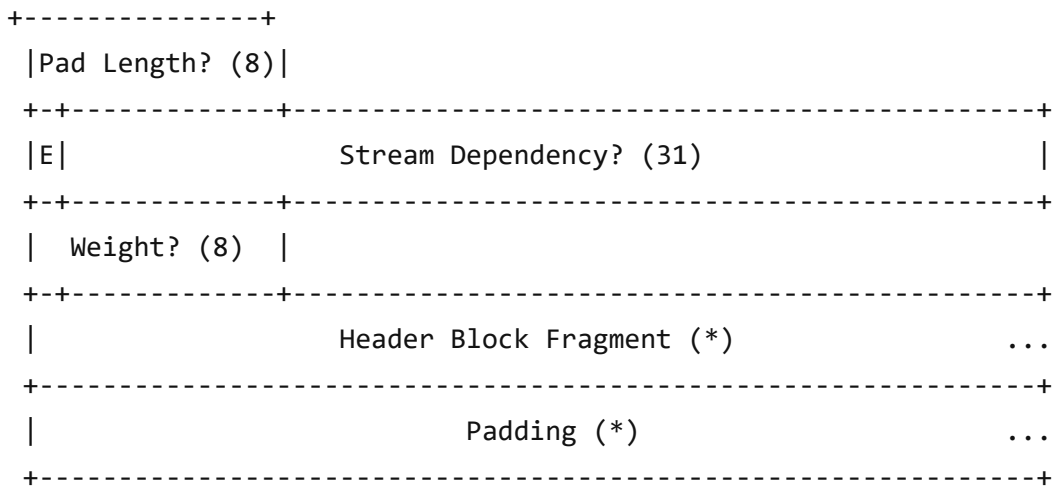


- Pad Length: ? 表示此字段的出现时有条件的，当帧的Flags(8)的第三位为1时，才有效，否则会被忽略

Data帧的Flags(8)目前有两个位有意义：

- END_STREAM: bit 0 设为 1 代表当前流的最后一帧，告诉接收方**请求数据发送完毕**，否则还要继续等下一帧(接收方)
- PADDED: bit 3 设为 1 代表存在 Padding

HEADER帧Payload：



- Pad Length: 同DATA帧
- E: 一个比特位声明流的依赖性是否是排他的，存在则代表 PRIORITY flag 被设置
- Stream Dependency: 指定一个 stream identifier，代表当前流所依赖的流的 id，存在则代表 PRIORITY flag 被设置
- Weight: 一个无符号 8 为整数，代表当前流的优先级权重值 (1~256)，存在则代表 PRIORITY flag 被设置
- Header Block Fragment: header 块片段，header依次打包排列在里面
- Padding: 同DATA帧

HEADERS 帧有以下标识 (flags):

- END_STREAM: bit 0 设为 1 代表当前请求 header 发送完了(可能有CONTINUATION帧，可以认为是HEADERS的一部分)
- END_HEADERS: bit 2 设为 1 代表 header 块结束
- PADDED: bit 3 设为 1 代表 Pad 被设置，存在 Pad Length 和 Padding

由于上面头部压缩的内容，我们知道header可以存在于静态表、动态表中。此时只需要传一个index即可表达对应的header，减少传输内容。 请求传递的header情况有以下几种：

- header 的key、value 在静态表/动态表中，**此时只需要传递一个index即可**
- header 的key 在静态、动态表中，而value由于多种多样，不在表中(如Host)，此时key可以由index表示，但value需要传递原内容
- header 的key、value完全不在静态、动态表中，key、value都需要传递原内容(字符串)
- 希望将本次传递的header写入动态表中，下次只需要传index 即可
- 不希望本次传递的header写入动态表中

Header Block Fragment 中打包header的方式也就是按照上面几种情况展开，具体篇幅较多，本文找一个复杂点的例子： **key、value都不在表中，且需要添加进表中**的情况进行举例：(更详细HPACK细节可见[6]、[7])

```
+---+---+---+---+---+---+---+---+
| 0 | 1 |           0           |          // 通过头8个bit表示是哪种case
+---+---+-----------------------+
| H |     Key Length (7+)       |
+---+---------------------------+
|  Key String (Length octets)   |
+---+---------------------------+
| H |     Value Length (7+)     |
+---+---------------------------+
| Value String (Length octets)  |
+-------------------------------+
```

- 头8个bit中 `01 000000` 表达了两点

1. header的key不在表中(`000000`)、value也不在(需要传文本内容)
2. 希望将此header追加到动态表中，供下次使用(`01` 开头表示需要追加到表中)

- Value Length 代表对应value的长度，借此可读取完整的Value String
- 其余的情况都可以用头8个bit表示[7]
- 多个上面的结构前后拼接在一起，就可以在一个HEADERS帧中表示多个header了

- 数字的表达。 key的Index、key/value文本内容的长度
- 字符串的表达。key的内容(如custom-key)、value的内容(custom-value)

对于 `custom-key: custom-header` 表达示例：(来源[10])

```
编码数据的十六进制表示：
400a 6375 7374 6f6d 2d6b 6579 0d63 7573 | @.custom-key.cus
746f 6d2d 6865 6164 6572                | tom-header

解码过程：

40                                  | == Literal indexed ==    （01000000表示要追加到表中）
0a                                  |   Literal name (len = 10) （得到key长度）
6375 7374 6f6d 2d6b 6579            | custom-key
0d                                  |   Literal value (len = 13) （得到value长度）
6375 7374 6f6d 2d68 6561 6465 72    | custom-header            （一个key:value 读取完毕）


解码结果可得header：      custom-key:custom-header
并将其加入动态表，下次直接只传index
```

上图中有 `Key Length (7+)` 和 `Value Length (7+)`，这是上面提到的 数字的表达 ，可以看到有 `7+` 这个表示。这里面有一个扩展问题，如果Value的长度比较大，7个bit表示不了咋办。

```
0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| ? | ? | ? | 1   1   1   1   1 |       第一个字节  N = 5
+---+---+---+-------------------+
| 1 |    Value-(2^N-1) LSB      |
+---+---------------------------+
               ...
+---+---------------------------+
| 0 |    Value-(2^N-1) MSB      |
+---+---------------------------+
```

- 选择一个N，如上面N=5，将第一个字节的后N位全部设为1，则第一个字节表达了 2^N - 1，剩下的len - (2^N - 1) 用后面的字节表示。
- 将len - (2^N - 1) 用二进制表示出来，将二进制位分别分给下面的字节
- 只占用后面字节的后7位
- 如果第一位为0，则表示表达完毕，为1 则表示下一个字节还在继续表示len

示例：（来源[10]）

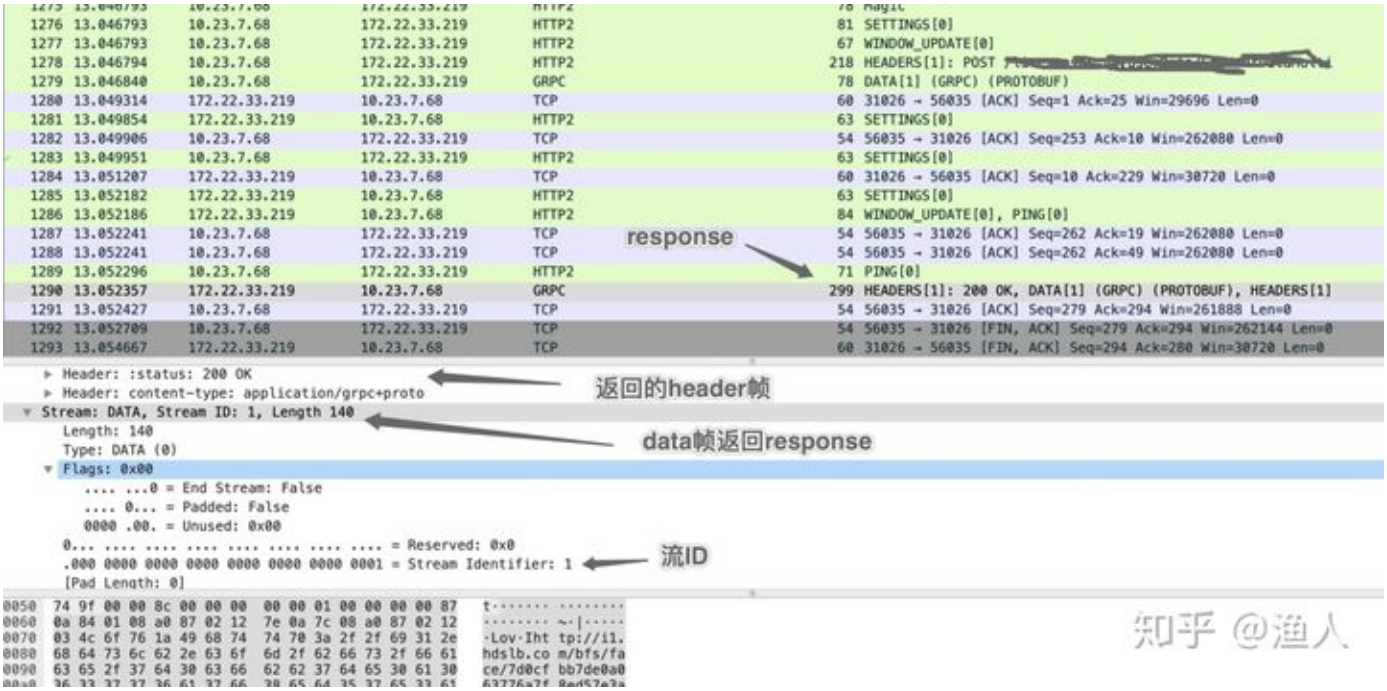
- 表达长度为： 1337，设N = 5
- 1337 大于 31（2^5-1），并使用 5 位前缀表示。5 位前缀使用其最大值（31）填充
- 除第一个字节外，后面字节表达 1337 - 31 = 1306
- 1036 二进制串为： 010100011010，用多个字节表达

```
 0   1   2   3   4   5   6   7
   +---+---+---+---+---+---+---+---+
   | X | X | X | 1 | 1 | 1 | 1 | 1 |  第一个字节表达了 2^N - 1 = 31，下面的字节表达 1337 -
   | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |   后面一截： 0011010   （低位）
   | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |   前面一截： 01010      （高位）
   +---+---+---+---+---+---+---+---+
```

gRPC 请求抓包

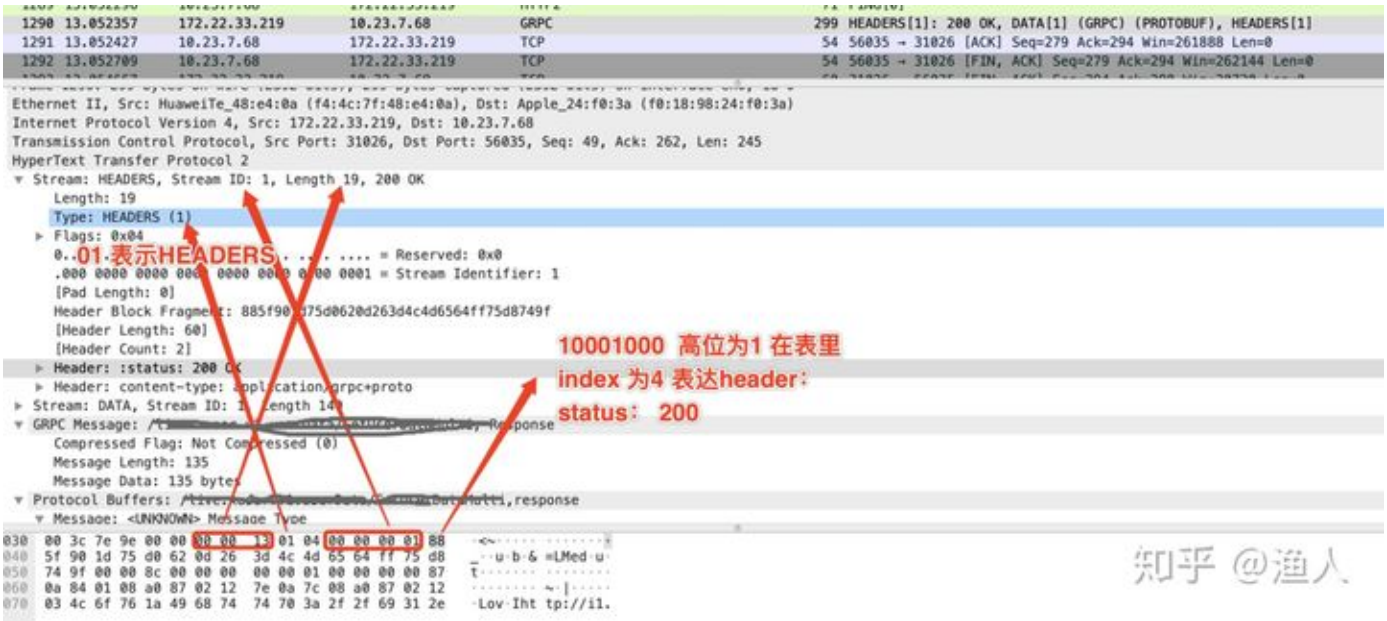
上文已经搞清楚了HTTP2的传输原理，接下来通过wireshark透视一下gRPC调用的过程。

请求内容：



返回抓包

帧示例：



header帧抓包

先约定配置，SETTINGS帧有ACK表达确认

- 返回数据用DATA帧

可见调用语义和HTTP并无差别，但通过协议优化，在很大程度上降低了传输的体积，节省资源的同时，也较好地提升了性能。

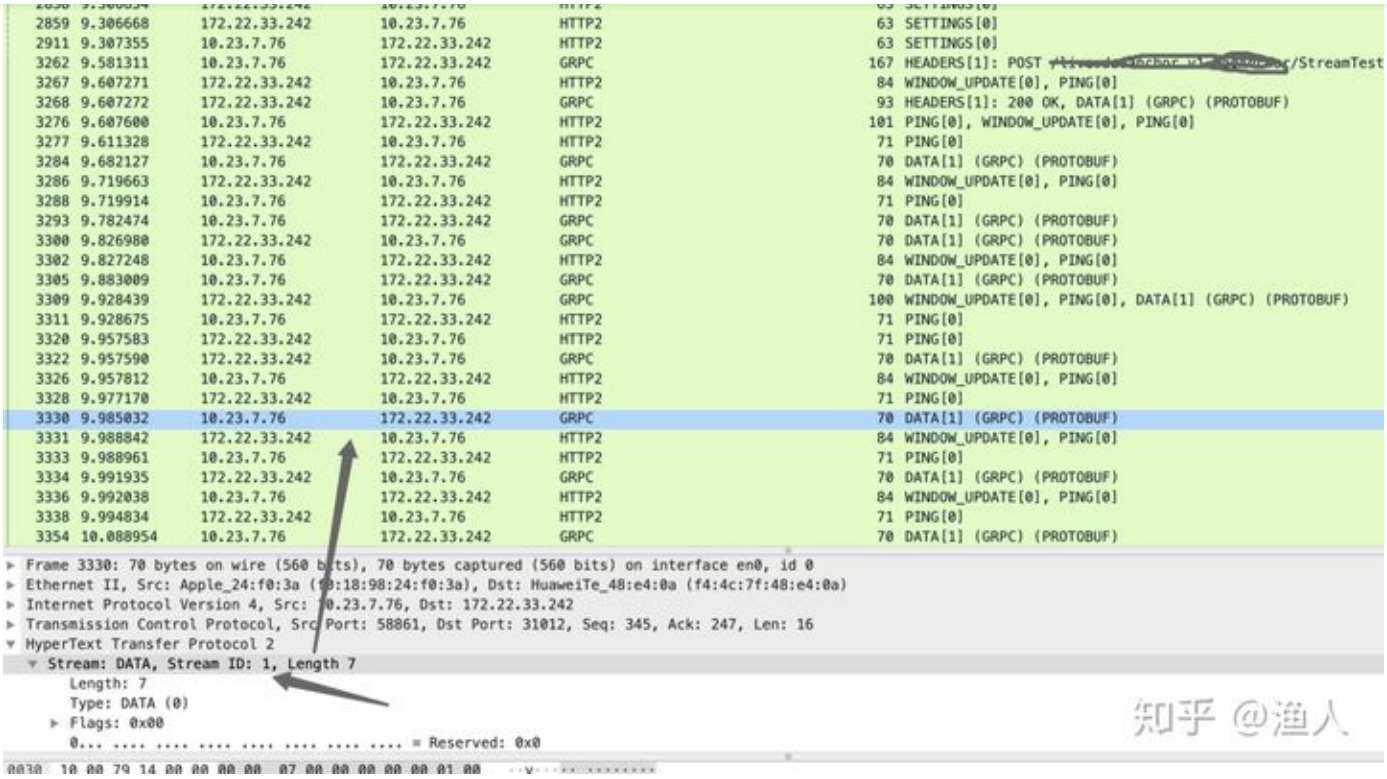
看了单个请求的抓包样例，我们得再看看gRPC的stream是什么鬼，代码约定如下：

```
// proto
service XXX {
    rpc StreamTest(stream StreamTestReq) returns (stream StreamTestResp);
}
message StreamTestReq {
    int64 i = 1;
}
message StreamTestResp {
    int64 j = 1;
}
// server端代码
func (s *XXXService) StreamTest(re v1pb.XXX_StreamTestServer ) (err error) {
    for {
        data, err := re.Recv()
        if err != nil {
            break
        }
             // 将客户端发送来的值乘以10再返回给它
        err = re.Send(&v1pb.StreamTestResp{J: data.I * 10 })
    }
    return
}
// client 端代码
func TestStream(t *testing.T) {
    c, _ := service2.daClient.StreamTest(context.TODO())
    go func(){
        for {
            rec, err := c.Recv()
            if err != nil {
                break
```

```
    for _, x := range []int64{1,2,3,4,5,6,7,8,9}{
        _ = c.Send(&dav1.StreamTestReq{I: x})
        time.Sleep(100*time.Millisecond)
    }
    _ = c.CloseSend()
}
// client端输出结果
resp: 10
resp: 20
resp: 30
resp: 40
resp: 50
resp: 60
resp: 70
resp: 80
resp: 90
```

- 上面是一个双向stream流
- client和server端同时在收发数据
- client连续发送9次后，中断过程。常规的流式服务，如视频编解码，可以一直持续直到结束
- 服务端将client的参数*10后返回

我们不禁要问，这种流式请求和常规的gRPC有没有区别？ 这从抓包便可知分晓：



stream

上面只提取了http2 和grpc的协议内容，否则会被tcp的ack打乱视野，可以从图上看到：

- 请求的method只发送了一次
- 服务端的回复header也只返回了一次(200 OK 那行)
- 剩下的就是： client的data帧和server 端的data帧交替
- 其实全场就只有一次请求(stream ID 未变化)

stream模式，其实就是gRPC从协议层支持了，在一次长请求中，分批地处理小量数据，达到多次请求的效果，像流水一样可以延绵不绝，直到某一方终止。

试想下，如果gRPC内部不支持这种模式，其实也能自己实现流式的服务，只不过在形式上要多调用几次接口而已。 从上面抓包来看，这种封装在无论在性能和语义上都更好。

进一步提升

参见HTTP3，抛弃TCP协议，拥抱QUIC。

- [3] tools.ietf.org/html/rfc...
- [4] rabbitmq.com/tutorials/...
- [5]zhuanlan.zhihu.com/p/34...
- [6] http2.github.io/http2-s...
- [7] github.com/halfrost/Hal...
- [8] zhuanlan.zhihu.com/p/14...
- [9]zh.wikipedia.org/zh-han...
- [10] github.com/halfrost/Hal...

编辑于 01-16

gRPC HTTP/2 网络协议

文章被以下专栏收录

互联网技术
体系化知识转述

推荐阅读

QUIC/HTTP3 协议简析

HTTP、HTTP2 和 HTTP3先和大家来回顾一下 HTTP 的历史，看看 HTTP3 相比 HTTP、HTTP2 都有哪些改进和升级的地方。 HTTP VS HTTP2多路复用：多路复用时，多文件传输有时只需维护一个 TCP...

一文读懂 HTTP/1HTTP/2HTTP/3

前端应该知道的http协议

作为互联网通信协议的一员老将，HTTP 协议走到今天已经经历了三次版本的变动，现在最新的版本是 HTTP2.0，相信大家早已耳熟能详。今天就给大家好好介绍一下 HTTP 的前世今生。1、http的历...

9 条评论

切换为时间排序

写下你的评论...

因为超人不会飞 2020-11-24

博主还更新吗？期待博主继续讲解GRPC

1

渔人 (作者) 回复 因为超人不会飞 2020-11-24

还有一篇，快了😆

2

Bomb 06-23

👍👍👍

赞

叶落花开水自流 03-30

强

赞

SANTA 01-14

催更+1

赞

fumeboy 01-07

太强了

赞

李李李 01-06

赞

Dapor 2020-12-14

催更

赞