HTTP2 协议

协议介绍

随着互联网的快速发展，HTTP1.x 协议得到了迅猛发展，但当 App 一个页面包含了数十个请求时，HTTP1.x 协议的局限性便暴露了出来：

- 每个请求与响应需要单独建立链路进行请求(Connection 字段能够解决部分问题)，浪费资源。
- 每个请求与响应都需要添加完整的头信息，应用数据传输效率较低。
- 默认没有进行加密，数据在传输过程中容易被监听与篡改。

HTTP2 正是为了解决 HTTP1.x 暴露出来的问题而诞生的。

说到 HTTP2 不得不提 spdy。

由于 HTTP1.x 暴露出来的问题，Google 设计了全新的名为 spdy 的新协议。spdy 在五层协议栈的 TCP 层与 HTTP 层引入了一个新的逻辑层以提高效率。spdy 是一个中间层，对 TCP 层与 HTTP 层有很好的兼容，不需要修改 HTTP 层即可改善应用数据传输速度。

spdy 通过多路复用技术，使客户端与服务器只需要保持一条链接即可并发多次数据交互，提高了通信效率。

而 HTTP2 便士基于 spdy 的思路开发的。

通过流与帧概念的引入，继承了 spdy 的多路复用，并增加了一些实用特性。

HTTP2 有什么特性呢？HTTP2 的特性不仅解决了上述已暴露的问题，还有一些功能使 HTTP 协议更加好用。

- 多路复用

- 压缩头信息
- 对请求划分优先级
- 支持服务端 Push 消息到客户端

此外，HTTP2 目前在实际使用中，只用于 HTTPS 协议场景下，通过握手阶段 ClientHello 与 ServerHello 的 extension 字段协商而来，所以目前 HTTP2 的使用场景，都是默认安全加密的。

下面介绍 HTTP2 协议协商以及多路复用与压缩头信息两大特性，实现部分采用 okhttp 源码(基于 parent-3.4.2)进行分析与介绍。

okhttp 是目前使用最广泛的支持 HTTP2 的 Android 端开源网络库，以 okhttp 为例介绍 HTTP2 特性也可方便读者提前了解 okhttp，方便后续接入 okhttp。

协议协商

HTTP2 **协议的协商是在握手阶段进行的。**

协商的方式是通过握手协议 extension 扩展字段进行扩展，新增 Application Layer Protocol Negotiation 字段进行协商。

在握手协议的 ClientHello 阶段，客户端将所支持的协议列表填入 Application Layer Protocol Negotiation 字段，供服务端进行挑选。如图所示：

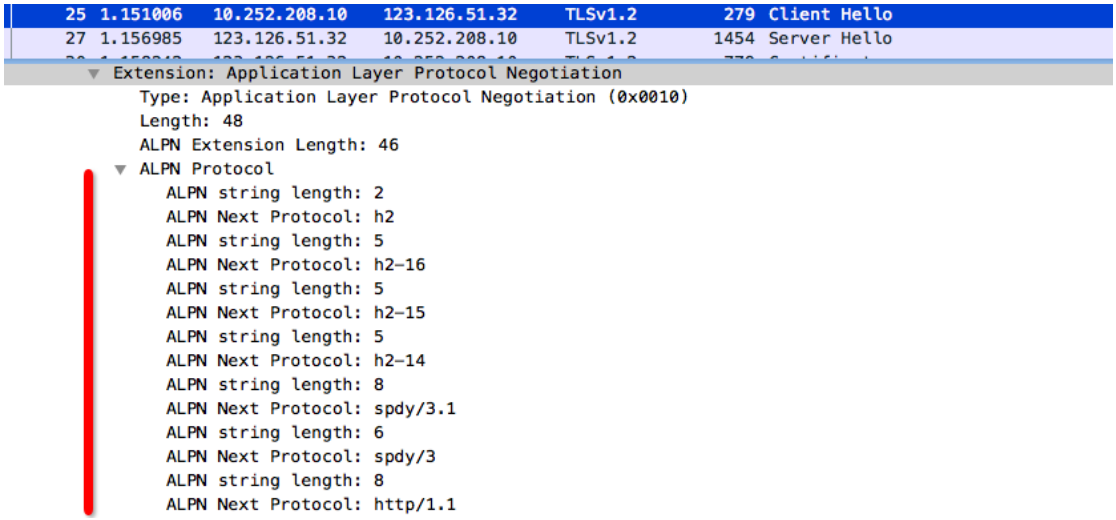

图 ALPN1

服务端收到 ClientHello 消息后，在客户端所支持的协议列表中选择适当协议作为后续应用层协议。如图所示：

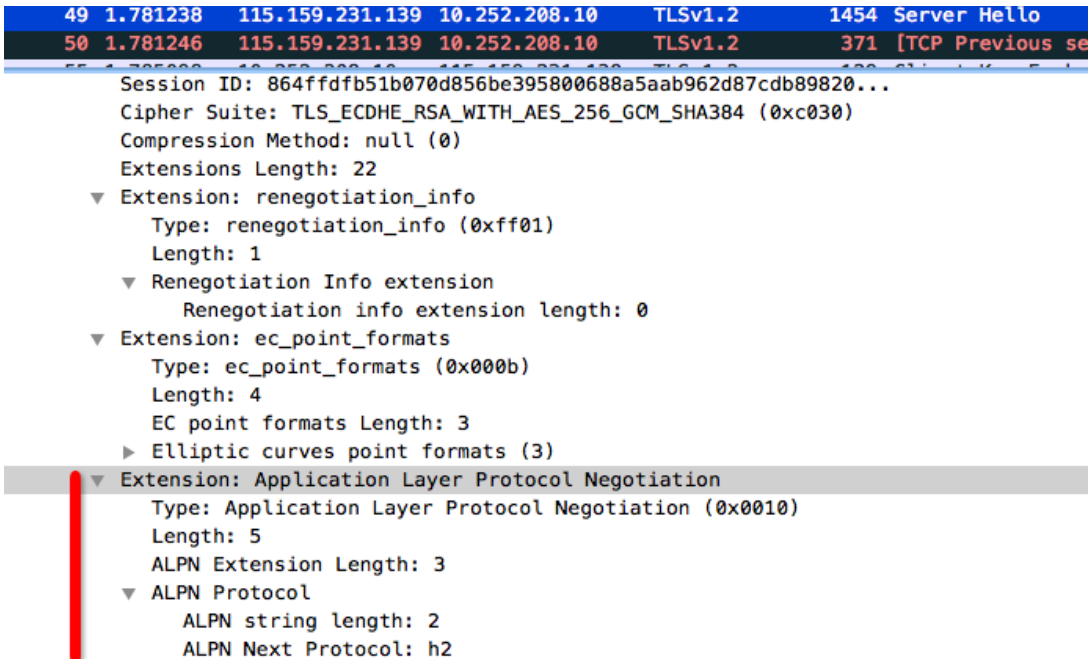

图 ALPN2

这样，两端便完成了 HTTP2 协议的协商。

在 HTTP2 未出现时，spdy 也是通过扩展字段，扩展出 next_protocol_negotiation 字段，以 NPN 协议进行 spdy 的协商。不过由于 NPN 协议协商过于复杂，对 https 协议侵入性较强，在出现 ALPN 协商协议后，便逐渐被淘汰了。所以，本文协议协商并为对 NPN 协议协商做介绍。

协议特性之多路复用

http2 为了优化 http1.x 对 TCP 性能的浪费，提出了多路复用的概念。

多路复用的含义

在 HTTP2 中，同一域名下的请求，可通过同一条 TCP 链路进行传输，使多个请求不必单独建立链路，节省建立链路的开销。

为了达到这个目的，HTTP2 提出了流与帧的概念，流代表请求与响应，而请求与响应具体的数据则包装为帧，对链路中传输的数据通过流 ID 与帧类型进行区分处理。下图便是多路复用的抽象图，每个块代表一帧，而相同颜色的块则代表是同一个流。

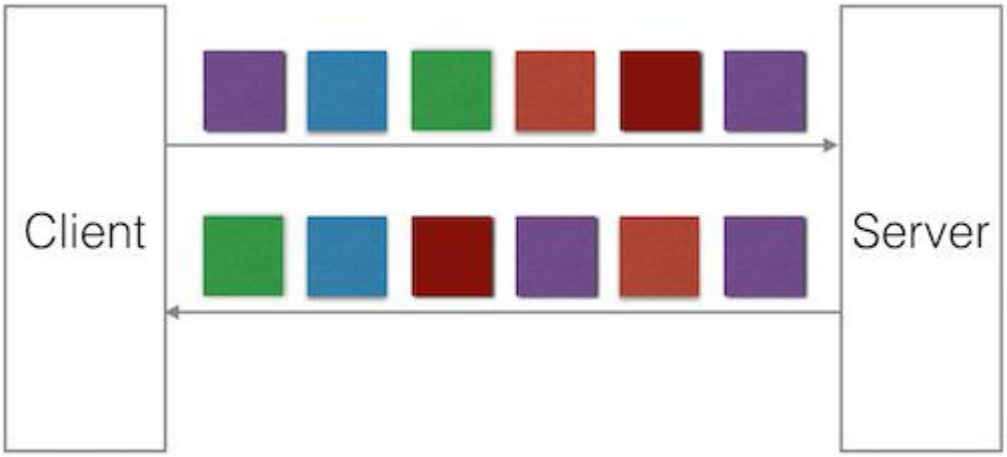

图 http2_stream

那么 HTTP2 的多路复用是如何实现的呢？

由于网络请求的场景很多，我们选择其中一个路径来介绍：

客户端与服务端在某个域名的 TCP 通道已建立

新创建的客户端请求通过已连接的 TCP 通道进行请求发送与响应处理

多路复用实现

默认我们已经添加各参数创建了 Request 对象 r，并通过 Request 对象创建了 Call 对象 c。并在独立线程中，调用 c.execute()方法，进行同步请求操作。

okhttp 调用 execute 方法后，实际上是由一系列的 interceptor 来负责执行的。

interceptor 根据添加顺序依此执行，其中我们关注的是 RetryAndFollowUpInterceptor、ConnectInterceptor0、CallServerInterceptor。

1. 在 RetryAndFollowUpInterceptor 中，okhttp 为我们创建了一个 StreamAllocation 对象，StreamAllocation 中含有基于 url 创建的 Address 对象。

Address 类的 url 字段与 Request 类的 url 字段不同，Address 类的 url 字段不包括 path 与 query 字段，只含有 scheme 与 authority 部分，这点在进行 Connection 复用的 equal 操作时起了很大作用。

2.在 ConnectInterceptor 中，StreamAllocation 对象的 Address 与连接池中每个 Connection 对象的 Address 依次进行匹配，匹配成功并满足一些条件的 Connection 便可复用。基于匹配出的 Connection 创建 Http2xStream，用于后续读写操作。

与连接池中 Address 匹配主要通过 Address 的 url，url 由于只含有 scheme 与 authority 所以可用于域名的匹配，这便是 okhttp 基于域名层面多路复用的基础。

实际上真正进行流读写操作的是 FramedConnection 与 FramedStream，Connection 与 Http2xStream 是抽象于具体操作的类，以方便上层使用。

3.在 CallServerInterceptor 中，Http2xStream 创建 FramedStream 用于 Request 发送，并将 FramedStream 与对应的 StreamID 绑定缓存下来，以便 Response 到来时，能够根据 StreamID 索引到对应的 FramedSteam 进行后续操作。

在 FramedStream 发送完 Request 后，执行 readResponseHeaders 方法时进行调用了 wait，将当前线程挂起。

并在 FramedConnection 读线程收到 StreamID 消息时，在缓存中查询 FramedStream 并将对应线程唤醒进行 Response 解码。

归纳下 okhttp 的多路复用实现思路：

通过请求的 Address 与连接池中现有连接 Address 依次匹配，选出可用的 Connection。

通过 Http2xStream 创建的 FramedStream 在发送了请求后，将 FramedStream 对象与 StreamID 的映射关系缓存到 FramedConnection 中。

收到消息后，FramedConnection 解析帧信息，在 Map 中通过解析的 StreamID 选出缓存的 FramedStream，并唤醒 FramedStream 进行 Response 的处理。

在笔者看来，HTTP2 便是一个良好兼容 http 协议格式的自定义协议，通过 Stream 将数据分发到各请求，通过 Frame 将请求数据详细细分。

协议特性之压缩头信息

HTTP2 为了解决 HTTP1.x 中头信息过大导致效率低下的问题，提出的解决方案便是压缩头部信息。具体的压缩方式，则引入了 HPACK。

HPACK 压缩算法是专门为 HTTP2 头部压缩服务的。为了达到压缩头部信息的目的，HPACK 将头部字段缓存为索引，通过索引 ID 代表头部字段。客户端与服务端维护索引表，通信过程中尽可能采用索引进行通信，收到索引后查询索引表，才能解析出真正的头部信息。

HPACK 索引表划分为动态索引表与静态索引表，动态索引表是 HTTP2 协议通信过程中两端动态维护的索引表，而静态索引表是硬编码进协议中的索引表。

作为分析 HPACK 压缩头信息的基础，需要先介绍 HPACK 对索引以及头部字符串的表示方式。

索引

索引以整型数字表示，由于 HPACK 需要考虑压缩与编解码问题，所以整型数字结构定义如图所示：

图 int_strut

类别标识：通过类别标识进行 HPACK 类别分类，指导后续编解码操作，常见的有 1，01，01000000 等八个类别。

首字节低位整型：首字节排除类别标识的剩余位，用于表示低位整型。若数值大于剩余位所能表示的容量，则需要后续字节表示高位整型。

结束标识：表示此字节是否为整型解析终止字节。

高位整型：字节余下 7bit，用于填充整型高位。

“结束标识+高位整型”字节可能有 0 个、也有可能有多个，依据数据大小而定。譬如，若想表示类别为 1，索引为 2，则使用 10000010 即可,不需要额外字节增加高位整型。

头部字符串

头部字符串需要显式声明长度，所以数据首字节由“类型标识 + 数据长度”组成。如图 11 所示：

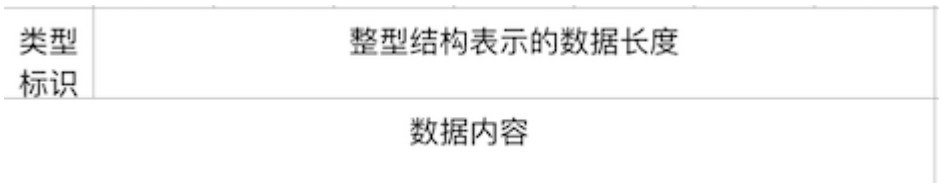

图 11 string_strut

- 类型标识：是否选用哈夫曼编码，1 为选用，0 为不选用，okhttp 默认不选用哈夫曼编码。
- 数据长度：标识数据长度，采用上面提到的整型表示法表示。
- 数据内容：二进制数据。
-

解码实例

下面综合 okhttp 源码分析 HPACK 解码头部字段过程。

对编码部分感兴趣的读者，可以查阅 RFC 7541 或直接分析 OkHttp 源码。

当我们需要解码头部字段时，首先解析头部字段首字节(HPACK 头部字段首字节分为 8 个类别，摘选其中 3 个类别说明)，首字节用于指导当前头部字段的解析规则：

- 1xxxxxxx：类别标识为 1，代表收到一条 K、V 均为索引的头部字段。
- K、V 值：通过解析 HPACK 整型获取 KV 对的索引值，并根据索引值映射对应的头部原字段即可，压缩效率最高。
- 01xxxxxx：类别标识为 01，代表收到一条 K 为索引、V 为原字段，且需要加入动态索引表的头部字段。
- K 值：通过解析 HPACK 整型获取 K 值索引值，并通过索引值映射对应的头部原字段。
- V 值：通过解析 HPACK 字符串获取 V 值原字段。
- 获取 K、V 值后还需插入动态索引表中。
- 01000000：01000000 代表收到一条 K、V 均为原字段，且需要加入动态索引表的头部字段。

- K、V 值：通过解析 HPACK 字符串获取 K、V 原字段，并插入动态索引表中。

还有不加入动态索引表、调整索引表大小等类别，这里就不展开了，感兴趣的可以看 okhttp 源码实现。

okhttp 解析头信息的核心方法实现如下：

```
void readHeaders() throws IOException {
  while (!source.exhausted()) {
    int b = source.readByte() & 0xff;
    if (b == 0x80) { // 10000000
      //类别标识为 1，但索引为 0
      throw new IOException("index == 0");
    } else if ((b & 0x80) == 0x80) { // 1NNNNNNN
      //类别为 1，通过 readIndexedHeader 解析整型 index。
      int index = readInt(b, PREFIX_7_BITS);
      //通过 index 获取完整头部字段
      readIndexedHeader(index - 1);
    } else if (b == 0x40) { // 01000000
      //01000000 代表 KV 均为原字段，解析字符串依次获取 K 值、V 值，并插入动态表中
      readLiteralHeaderWithIncrementalIndexingNewName();
    } else if ((b & 0x40) == 0x40) {  // 01NNNNNN
      //01xxxxxx 代表 K 值为索引，V 值为原字符串，依次解析整型 index 与字符串，并插入动态表中
      int index = readInt(b, PREFIX_6_BITS);
      readLiteralHeaderWithIncrementalIndexingIndexedName(index - 1);
    } else if ((b & 0x20) == 0x20) {  // 001NNNNN
```

```
      //类别为 001，含义是更新动态列表容量
      maxDynamicTableByteCount = readInt(b, PREFIX_5_BITS);
      if (maxDynamicTableByteCount < 0
          || maxDynamicTableByteCount > headerTableSizeSett
ing) {
        throw new IOException("Invalid dynamic table size up
date " + maxDynamicTableByteCount);
      }
      adjustDynamicTableByteCount();
    } else if (b == 0x10 || b == 0) { // 000?0000 - Ignore n
ever indexed bit.
      //这个类别代表 KV 均为原字符串，依次解析字符串，并不对解析后的 K
V 值插入动态表。
      readLiteralHeaderWithoutIndexingNewName();
    } else { // 000?NNNN - Ignore never indexed bit.
      //与上一类别类似，但 K 值为索引，V 值为原字符串
      int index = readInt(b, PREFIX_4_BITS);
      readLiteralHeaderWithoutIndexingIndexedName(index -
1);
    }
  }
}
```

压缩效果

K 值为“accept-encoding”、V 值为“gzip, deflate”的头部字段在 HTTP2 中可通过索引值 15 代替，从而达到头部字段压缩的效果。

“accept-charset”头部字段则通过 14 代表头部 K 值，而 Value 值根据 HPACK 规则编码写入流中。

通过 HPACK，一个头部字段变化较少的 App，每个头部字段将会缩减至 4 字节以内，压缩效果非常明显。