HTTP 面试指南

谈一谈 HTTP 协议优缺点

超文本传输协议，「HTTP 是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范」。

HTTP 特点

1. 「灵活可扩展」。一个是语法上只规定了基本格式，空格分隔单词，换行分隔字段等。另外一个就是传输形式上不仅可以传输文本，还可以传输图片，视频等任意数据。
2. 「请求-应答模式」，通常而言，就是一方发送消息，另外一方要接受消息，或者是做出相应等。
3. 「可靠传输」，HTTP 是基于 TCP/IP，因此把这一特性继承了下来。
4. 「无状态」，这个分场景回答即可。

HTTP 缺点

1. 「无状态」，有时候，需要保存信息，比如像购物系统，需要保留下顾客信息等等，另外一方面，有时候，无状态也会减少网络开销，比如类似直播行业这样子等，这个还是分场景来说。
2. 「明文传输」，即协议里的报文(主要指的是头部)不使用二进制数据，而是文本形式。这让 HTTP 的报文信息暴露给了外界，给攻击者带来了便利。
3. 「队头阻塞」，当 http 开启长连接时，共用一个 TCP 连接，当某个请求时间过长时，其他的请求只能处于阻塞状态，这就是队头阻塞问题。

HTTP/1.0 HTTP1.1 HTTP2.0 版本之间的差异

HTTP 0.9

- 1991 年,原型版本，功能简陋，只有一个命令 GET,只支持纯文本内容，该版本已过时。

HTTP 1.0

- 任何格式的内容都可以发送，这使得互联网不仅可以传输文字，还能传输图像、视频、二进制等文件。
- 除了 GET 命令，还引入了 POST 命令和 HEAD 命令。
- http 请求和回应的格式改变，除了数据部分，每次通信都必须包括头信息（HTTP header），用来描述一些元数据。
- 只使用 header 中的 If-Modified-Since 和 Expires 作为缓存失效的标准。
- 不支持断点续传，也就是说，每次都会传送全部的页面和数据。
- 通常每台计算机只能绑定一个 IP，所以请求消息中的 URL 并没有传递主机名（hostname）

HTTP 1.1

http1.1 是目前最为主流的 http 协议版本，从 1999 年发布至今，仍是主流的 http 协议版本。

- 引入了持久连接（ persistent connection），即 TCP 连接默认不关闭，可以被多个请求复用，不用声明 Connection: keep-alive。长连接的连接时长可以通过请求头中的 `keep-alive` 来设置
- 引入了管道机制（ pipelining），即在同一个 TCP 连接里，客户端可以同时发送多个 请求，进一步改进了 HTTP 协议的效率。
- HTTP 1.1 中新增加了 E-tag，If-Unmodified-Since, If-Match, If-None-Match 等缓存控制标头来控制缓存失效。
- 支持断点续传，通过使用请求头中的 `Range` 来实现。
- 使用了虚拟网络，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。
- 新增方法：PUT、 PATCH、 OPTIONS、 DELETE。

http1.x 版本问题

- 在传输数据过程中，所有内容都是明文，客户端和服务器端都无法验证对方的身份，无法保证数据的安全性。
- HTTP/1.1 版本默认允许复用 TCP 连接，但是在同一个 TCP 连接里，所有数据通信是按次序进行的，服务器通常在处理完一个回应后，才会继续去处理下一个，这样子就会造成队头阻塞。
- http/1.x 版本支持 Keep-alive，用此方案来弥补创建多次连接产生的延迟，但是同样会给服务器带来压力，并且的话，对于单文件被不断请求的服务，Keep-alive 会极大影响性能，因为它在文件被请求之后还保持了不必要的连接很长时间。

HTTP 2.0

- 二进制分帧 这是一次彻底的二进制协议，头信息和数据体都是二进制，并且统称为"帧"：头信息帧和数据帧。
- 头部压缩 HTTP 1.1 版本会出现 「User-Agent、Cookie、Accept、Server、Range」等字段可能会占用几百甚至几千字节，而 Body 却经常只有几十字节，所以导致头部偏重。HTTP 2.0 使用 HPACK 算法进行压缩。
- 多路复用 复用 TCP 连接，在一个连接里，客户端和浏览器都可以同时发送多个请求或回应，且不用按顺序一一对应，这样子解决了队头阻塞的问题。
- 服务器推送 允许服务器未经请求，主动向客户端发送资源，即服务器推送。
- 请求优先级 可以设置数据帧的优先级，让服务端先处理重要资源，优化用户体验。

谈一谈你对 HTTP/2 理解

头部压缩

HTTP 1.1 版本会出现 「User-Agent、Cookie、Accept、Server、Range」等字段可能会占用几百甚至几千字节，而 Body 却经常只有几十字节，所以导致头部偏重。

HTTP 2.0 使用 HPACK 算法进行压缩。

那我们看看 HPACK 算法吧

> 从上面看，我们可以看到类似于索引表，每个索引表对应一个值，比如索引为 2 对应头部中的 method 头部信息，这样子的话，在传输的时候，不在是传输对应的头部信息了，而是传递索引，对于之前出现过的头部信息，只需要把「索引」(比如 1，2，...)传给对方即可，对方拿到索引查表就行了。

这种「传索引」的方式，可以说让请求头字段得到极大程度的精简和复用。

其次是对于整数和字符串进行「哈夫曼编码」，哈夫曼编码的原理就是先将所有出现的字符建立一张索引表，然后让出现次数多的字符对应的索引尽可能短，传输的时候也是传输这样的「索引序列」，可以达到非常高的压缩率。

多路复用

HTTP 1.x 中，如果想并发多个请求，必须使用多个 TCP 链接，且浏览器为了控制资源，还会对单个域名有 6-8 个的 TCP 链接请求限制。

HTTP2 中：

- 同域名下所有通信都在单个连接上完成。
- 单个连接可以承载任意数量的双向数据流。
- 数据流以消息的形式发送，而消息又由一个或多个帧组成，多个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装，也就是 **Stream ID**，流标识符，有了它，接收方就能从乱序的二进制帧中选择 ID 相同的帧，按照顺序组装成请求/响应报文。

服务器推送

浏览器发送一个请求，服务器主动向浏览器推送与这个请求相关的资源，这样浏览器就不用发起后续请求。

相比较 http/1.1 的优势

- 推送资源可以由不同页面共享
- 服务器可以按照优先级推送资源
- 客户端可以缓存推送的资源
- 客户端可以拒收推送过来的资源

二进制分帧

之前是明文传输，不方便计算机解析，对于回车换行符来说到底是内容还是分隔符，都需要内部状态机去识别，这样子效率低，HTTP/2 采用二进制格式，全部传输 01 串，便于机器解码。

这样子一个报文格式就被拆分为一个个二进制帧，用「**Headers 帧**」存放头部字段，「**Data 帧**」存放请求体数据。这样子的话，就是一堆乱序的二进制帧，它们不存在先后关系，因此不需要排队等待，解决了 HTTP 队头阻塞问题。

在客户端与服务器之间，双方都可以互相发送二进制帧，这样子「**双向传输的序列**」，称为**流**，所以 HTTP/2 中以流来表示一个 TCP 连接上进行多个数据帧的通信，这就是多路复用概念。

那乱序的二进制帧，是如何组装成对于的报文呢？

- 所谓的乱序，值的是不同 ID 的 Stream 是乱序的，对于同一个 Stream ID 的帧是按顺序传输的。
- 接收方收到二进制帧后，将相同的 Stream ID 组装成完整的请求报文和响应报文。
- 二进制帧中有一些字段，控制着优先级和流量控制等功能，这样子的话，就可以设置数据帧的优先级，让服务器处理重要资源，优化用户体验。

介绍一下 HTTP 常见状态码

RFC 规定 HTTP 的状态码为「三位数」，第一个数字定义了响应的类别，被分为五类：

- 「1xx」：代表请求已被接受，需要继续处理。
- 「2xx」：表示成功状态。
- 「3xx」：重定向状态。
- 「4xx」：客户端错误。
- 「5xx」：服务器端错误。

1xx 信息类

接受的请求正在处理，信息类状态码。

2xx 成功

- 200 OK 表示从客户端发来的请求在服务器端被正确请求。
- 204 No content，表示请求成功，但没有资源可返回。
- 206 Partial Content，该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的 GET 请求 响应报文中包含由 「Content-Range」 指定范围的实体内容。

3xx 重定向

- 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL，这时应该按 Location 首部字段提示的 URI 重新保存。
- 302 found，临时性重定向，表示资源临时被分配了新的 URL。
- 303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源。
- 304 not modified，当协商缓存命中时会返回这个状态码。
- 307 temporary redirect，临时重定向，和 302 含义相同,不会改变 method

当 301、302、303 响应状态码返回时，几乎所有的浏览器都会把 POST 改成 GET，并删除请求报文内的主体，之后

请求会自动再次发送 301、302 标准是禁止将 POST 方法改变成 GET 方法的，但实际使用时大家都会这么做

4XX 客户端错误

- 400 bad request，请求报文存在语法错误。
- 401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息。
- 403 forbidden，表示对请求资源的访问被服务器拒绝。
- 404 not found，表示在服务器上没有找到请求的资源。
- 405 Method Not Allowed，服务器禁止使用该方法，客户端可以通过 options 方法来查看服务器允许的访问方法，如下

```
Access-Control-Allow-Methods →GET,HEAD,PUT,PATCH,POST,DELETE
```

复制代码

5XX 服务器错误

- 500 internal sever error，表示服务器端在执行请求时发生了错误。
- 502 Bad Gateway，服务器自身是正常的，访问的时候出了问题，具体啥错误我们不知道。
- 503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求。

DNS 如何工作的

DNS 协议提供的是一种主机名到 IP 地址的转换服务，就是我们常说的域名系统。是应用层协议，通常该协议运行在 UDP 协议之上，使用的是 53 端口号。

「我们通过一张图来看看它的查询过程吧」

这张图很生动的展示了 DNS 在本地 DNS 服务器是如何查询的，**「一般向本地 DNS 服务器发送请求是递归查询的」**

本地 DNS 服务器向其他域名服务器请求的过程是迭代查询的过程

DNS 查询

递归查询和迭代查询

-

递归查询指的是查询请求发出后，域名服务器代为向下一级域名服务器发出请求，最后向用户返回查询的最终结果。使用递归 查询，用户只需要发出一次查询请求。

-
-

迭代查询指的是查询请求后，域名服务器返回单次查询的结果。下一级的查询由用户自己请求。使用迭代查询，用户需要发出 多次的查询请求。

-

所以一般而言，**「本地服务器查询是递归查询」**，而**「本地 DNS 服务器向其他域名服务器请求的过程是迭代查询的过程」**。

DNS 缓存

缓存也很好理解，在一个请求中，当某个 DNS 服务器收到一个 DNS 回答后，它能够回答中的信息缓存在本地存储器中。**「返回的资源记录中的 TTL 代表了该条记录的缓存的时间。」**

DNS 实现负载平衡

它是如何实现负载均衡的呢？首先我们得清楚 DNS 是可以用于在冗余的服务器上实现负载平衡。

原因：这是因为一般的大型网站使用多台服务器提供服务，因此一个域名可能会对应 多个服务器地址。

举个例子来说

- 当用户发起网站域名的 DNS 请求的时候，DNS 服务器返回这个域名所对应的服务器 IP 地址的集合
- 在每个回答中，会循环这些 IP 地址的顺序，用户一般会选择排在前面的地址发送请求。
- 以此将用户的请求均衡的分配到各个不同的服务器上，这样来实现负载均衡。

总结

- DNS 域名系统，是应用层协议，运行 UDP 协议之上，使用端口 43。
- 查询过程，本地查询是递归查询，依次通过浏览器缓存 –>> 本地 hosts 文件 –>> 本地 DNS 解析器 –>>本地 DNS 服务器 –>> 其他域名服务器请求。 接下来的过程就是迭代过程。
- 递归查询一般而言，发送一次请求就够，迭代过程需要用户发送多次请求。

DNS 为什么使用 UDP 协议作为传输层协议？

「DNS 使用 UDP 协议作为传输层协议的主要原因是为了避免使用 TCP 协议时造成的连接时延。」

- 为了得到一个域名的 IP 地址，往往会向多个域名服务器查询，如果使用 TCP 协议，那么每次请求都会存在连接时延，这样使 DNS 服务变得很慢。
- 大多数的地址查询请求，都是浏览器请求页面时发出的，这样会造成网页的等待时间过长。

介绍一下 Connection:keep-alive

什么是 keep-alive

我们知道 HTTP 协议采用“请求-应答”模式，当使用普通模式，即非 KeepAlive 模式时，每个请求/应答客户和服务器都要新建一个连接，完成 之后立即断开连接（HTTP 协议为无连接的协议）；

当使用 Keep-Alive 模式（又称持久连接、连接重用）时，Keep-Alive 功能使客户端到服 务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。

为什么要使用 keep-alive

keep-alive 技术的创建目的，能在多次 HTTP 之前重用同一个 TCP 连接，从而减少创建/关闭多个 TCP 连接的开销（包括响应时间、CPU 资源、减少拥堵等），参考如下示意图（来源：维基百科）：

客户端如何开启

在 HTTP/1.0 协议中，默认是关闭的，需要在 http 头加入"Connection: Keep-Alive”，才能启用 Keep-Alive；

```
Connection: keep-alive
```
复制代码

http 1.1 中默认启用 Keep-Alive，如果加入"Connection: close “，才关闭。

```
Connection: close
```
复制代码

目前大部分浏览器都是用 http1.1 协议，也就是说默认都会发起 Keep-Alive 的连接请求了，所以是否能完成一个完整的 Keep- Alive 连接就看服务器设置情况。

介绍 HTTP 缓存策略

这个跟之前的浏览器缓存原理一样，我直接拿我之前梳理过的吧。

我在我之前的那一篇中已经详细的说过了，**点这里传送门聊一聊浏览器缓存**

我们来梳理一下吧

强缓存

强缓存两个相关字段，「Expires」，「Cache-Control」。

「强缓存分为两种情况，一种是发送 HTTP 请求，一种不需要发送。」

首先检查强缓存，这个阶段**不需要发送 HTTP 请求。**通过查找不同的字段来进行，不同的 HTTP 版本所以不同。

- HTTP1.0 版本，使用的是 Expires，HTTP1.1 使用的是 Cache-Control

Expires

Expires 即过期时间，时间是相对于服务器的时间而言的，存在于服务端返回的响应头中，在这个过期时间之前可以直接从缓存里面获取数据，无需再次请求。比如下面这样：

```
Expires:Mon, 29 Jun 2020 11:10:23 GMT
```
复制代码

表示该资源在 2020 年 7 月 29 日 11:10:23 过期，过期时就会重新向服务器发起请求。

这个方式有一个问题：「服务器的时间和浏览器的时间可能并不一致」，所以 HTTP1.1 提出新的字段代替它。

Cache-Control

HTTP1.1 版本中，使用的就是该字段，这个字段采用的时间是过期时长，对应的是 max-age。

```
Cache-Control:max-age=6000
```
复制代码

上面代表该资源返回后 6000 秒，可以直接使用缓存。

当然了，它还有其他很多关键的指令，梳理了几个重要的

注意点：

- 当 Expires 和 Cache-Control 同时存在时，优先考虑 Cache-Control。
- 当然了，当缓存资源失效了，也就是没有命中强缓存，接下来就进入协商缓存

协商缓存

强缓存失效后，浏览器在请求头中携带响应的缓存 Tag 来向服务器发送请求，服务器根据对应的 tag，来决定是否使用缓存。

缓存分为两种，「Last-Modified」 和 「ETag」。两者各有优势，并不存在谁对谁有绝对的优势，与上面所讲的强缓存两个 Tag 所不同。

Last-Modified

这个字段表示的是「最后修改时间」。在浏览器第一次给服务器发送请求后，服务器会在响应头中加上这个字段。

浏览器接收到后，「如果再次请求」，会在请求头中携带 If-Modified-Since 字段，这个字段的值也就是服务器传来的最后修改时间。

服务器拿到请求头中的 If-Modified-Since 的字段后，其实会和这个服务器中该资源的最后修改时间对比：

- 如果请求头中的这个值小于最后修改时间，说明是时候更新了。返回新的资源，跟常规的 HTTP 请求响应的流程一样。
- 否则返回 304，告诉浏览器直接使用缓存。

ETag

ETag 是服务器根据当前文件的内容，对文件生成唯一的标识，比如 MD5 算法，只要里面的内容有改动，这个值就会修改，服务器通过把响应头把该字段给浏览器。

浏览器接受到 ETag 值，会在下次请求的时候，将这个值作为「If-None-Match」这个字段的内容，发给服务器。

服务器接收到「If-None-Match」后，会跟服务器上该资源的「ETag」进行比对

- 如果两者一样的话，直接返回 304，告诉浏览器直接使用缓存
- 如果不一样的话，说明内容更新了，返回新的资源，跟常规的 HTTP 请求响应的流程一样

两者对比

- 性能上，Last-Modified 优于 ETag，Last-Modified 记录的是时间点，而 Etag 需要根据文件的 MD5 算法生成对应的 hash 值。
- 精度上，ETag 优于 Last-Modified。ETag 按照内容给资源带上标识，能准确感知资源变化，Last-Modified 在某些场景并不能准确感知变化，比如
    - 编辑了资源文件，但是文件内容并没有更改，这样也会造成缓存失效。
    - Last-Modified 能够感知的单位时间是秒，如果文件在 1 秒内改变了多次，那么这时候的 Last-Modified 并没有体现出修改了。

最后，「如果两种方式都支持的话，服务器会优先考虑 ETag」。

缓存位置

接下来我们考虑使用缓存的话，缓存的位置在哪里呢？

浏览器缓存的位置的话，可以分为四种,优先级从高到低排列分别

- Service Worker
- Memory Cache
- Disk Cache
- Push Cache

Service Worker

这个应用场景比如 PWA，它借鉴了 Web Worker 思路，由于它脱离了浏览器的窗体，因此无法直接访问 DOM。它能完成的功能比如：**离线缓存**、**消息推送**和**网络代理**，其中**离线缓存**就是**「Service Worker Cache」**。

Memory Cache

指的是内存缓存，从效率上讲它是最快的，从存活时间来讲又是最短的，当渲染进程结束后，内存缓存也就不存在了。

Disk Cache

存储在磁盘中的缓存，从存取效率上讲是比内存缓存慢的，优势在于存储容量和存储时长。

Disk Cache VS Memory Cache

两者对比，主要的策略

内容使用率高的话，文件优先进入磁盘

比较大的 JS，CSS 文件会直接放入磁盘，反之放入内存。

Push Cache

推送缓存，这算是浏览器中最后一道防线吧，它是 HTTP/2 的内容。具体我也不是很清楚，有兴趣的可以去了解。

总结

- 首先检查 Cache-Control， 尝鲜，看强缓存是否可用
- 如果可用的话，直接使用
- 否则进入协商缓存，发送 HTTP 请求，服务器通过请求头中的 If-Modified-Since 或者 If-None-Match 字段检查资源是否更新
- 资源更新，返回资源和 200 状态码。
- 否则，返回 304，直接告诉浏览器直接从缓存中去资源。

说一说 HTTP 的请求方法？

- HTTP1.0 定义了三种请求方法： GET, POST 和 HEAD 方法
- HTTP1.1 新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT

http/1.1 规定了以下请求方法(注意，都是大写):

- GET： 请求获取 Request-URI 所标识的资源
- POST： 在 Request-URI 所标识的资源后附加新的数据
- HEAD： 请求获取由 Request-URI 所标识的资源的响应消息报头
- PUT： 请求服务器存储一个资源，并用 Request-URI 作为其标识（修改数据）
- DELETE： 请求服务器删除对应所标识的资源
- TRACE： 请求服务器回送收到的请求信息，主要用于测试或诊断
- CONNECT： 建立连接隧道，用于代理服务器
- OPTIONS： 列出可对资源实行的请求方法，用来跨域请求

谈一谈 GET 和 POST 的区别

本质上，只是语义上的区别，GET 用于获取资源，POST 用于提交资源。

想装逼请参考 https://zhuanlan.zhihu.com/p/22536382

具体差别

- 从缓存角度看，GET 请求后浏览器会主动缓存，POST 默认情况下不能。

- 从参数角度来看，GET 请求一般放在 URL 中，因此不安全，POST 请求放在请求体中，相对而言较为安全，但是在抓包的情况下都是一样的。
- 从编码角度看，GET 请求只能经行 URL 编码，只能接受 ASCII 码，而 POST 支持更多的编码类型且不对数据类型限值。
- GET 请求幂等，POST 请求不幂等，幂等指发送 M 和 N 次请求（两者不相同且都大于 1），服务器上资源的状态一致。
- GET 请求会一次性发送请求报文，POST 请求通常分为两个 TCP 数据包，首先发 header 部分，如果服务器响应 100(continue)， 然后发 body 部分。

从应用场景角度来看，Get 多用于无副作用，幂等的场景，例如搜索关键字。Post 多用于副作用，不幂等的场景，例如注册。

---

options 方法有什么用？

-

OPTIONS 请求与 HEAD 类似，一般也是用于客户端查看服务器的性能。

-
-

这个方法会请求服务器返回该资源所支持的所有 HTTP 请求方法，该方法会用'*'来代替资源名称，向服务器发送 OPTIONS 请求，可以测试服务器功能是否正常。

-
-

JS 的 XMLHttpRequest 对象进行 CORS 跨域资源共享时，对于复杂请求，就是使用 OPTIONS 方法发送嗅探请求，以判断是否有对指定资源的访问权限。

-

谈一谈你对 URL 理解

统一资源定位符的简称，Uniform Resource Locator，常常被称为网址，是因特网上标准的资源地址。

组成

通用的格式：scheme://host[:port]/path/…/?query#anchor

名称	功能
scheme	访问服务器以获取资源时要使用哪种协议，比如：http，https 和 FTP 等
host	HTTP 服务器的 IP 地址或者域名
port	HTTP 服务器的默认端口是 80，HTTPS 默认端口是 443，这种情况下端口号可以省略，如果使用了别的端口，必须指明。不同的端口，你可以认为是不同的应用程序。
path	访问资源的路径
query-string	发给 http 服务器的数据
anchor	锚点

举个例子

```
https://www.baidu.com/s?tn=baidu&bar=&wd=TianTian
```

复制代码

这个 URL 中，https 就是协议，www.baidu.com 就是域名，默认端口是 443，/s 就是请求的 path，tn=baidu&bar=&wd=TianTian 这个就是 query

URL 编码

- URL 只能使用 ASCII 字符集来通过因特网进行发送。
- 由于 URL 常常会包含 ASCII 集合之外的字符，URL 必须转换为有效的 ASCII 格式。
- URL 编码使用 "%" 其后跟随两位的十六进制数来替换非 ASCII 字符。
- URL 不能包含空格。URL 编码通常使用 + 来替换空格。

举个例子

天天转换为有效的 ASCII 格式就是 %CC%EC%CC%EC

谈一谈队头阻塞问题

什么是队头阻塞？

对于每一个 HTTP 请求而言，这些任务是会被放入一个任务队列中串行执行的，一旦队首任务请求太慢时，就会阻塞后面的请求处理，这就是 **HTTP 队头阻塞** 问题。

有什么解决办法吗

并发连接

我们知道对于一个域名而言，是允许分配多个长连接的，那么可以理解成增加了任务队列，也就是说不会导致一个任务阻塞了该任务队列的其他任务，在 **RFC 规范** 中规定客户端最多并发 2 个连接，不过实际情况就是要比这个还要多，举个例子，Chrome 中是 6 个。

域名分片

顾名思义，我们可以在一个域名下分出多个二级域名出来，而它们最终指向的还是同一个服务器，这样子的话就可以并发处理的任务队列更多，也更好的解决了队头阻塞的问题。

举个例子，比如 **TianTian.com**，可以分出很多二级域名，比如 **Day1.TianTian.com**，**Day2.TianTian.com**,**Day3.TianTian.com**,这样子就可以有效解决队头阻塞问题。

谈一谈 HTTP 数据传输

大概遇到的情况就分为 **「定长数据」** 与 **「不定长数据」** 的处理吧。

定长数据

对于定长的数据包而言，发送端在发送数据的过程中，需要设置 **Content-Length**,来指明发送数据的长度。

当然了如果采用了 Gzip 压缩的话，Content-Length 设置的就是压缩后的传输长度。

我们还需要知道的是

- Content-Length 如果存在并且有效的话，则必须和消息内容的传输长度完全一致，也就是说，如果过短就会截断，过长的话，就会导致超时。
- 如果采用短链接的话，直接可以通过服务器关闭连接来确定消息的传输长度。
- 那么在 HTTP/1.0 之前的版本中，Content-Length 字段可有可无,因为一旦服务器关闭连接，我们就可以获取到传输数据的长度了。
- 在 HTTP/1.1 版本中，如果是 Keep-alive 的话，chunked 优先级高于 Content-Length，若是非 Keep-alive，跟前面情况一样，Content-Length 可有可无。

那怎么来设置 Content-Length

举个例子来看看

```
const server = require('http').createServer();
server.on('request', (req, res) => {
  if(req.url === '/index') {
    // 设置数据类型
    res.setHeader('Content-Type', 'text/plain');
    res.setHeader('Content-Length', 10);
    res.write("你好，使用的是 Content-Length 设置传输数据形式");
  }
})

server.listen(3000, () => {
  console.log("成功启动--TinaTian");
})
```

复制代码

不定长数据

现在采用最多的就是 HTTP/1.1 版本，来完成传输数据，在保存 Keep-alive 状态下，当数据是不定长的时候，我们需要设置新的头部字段

```
Transfer-Encoding: chunked
```

复制代码

通过 chunked 机制，可以完成对不定长数据的处理，当然了，你需要知道的是

- 如果头部信息中有 `Transfer-Encoding`,优先采用 Transfer-Encoding 里面的方法来找到对应的长度。
- 如果设置了 Transfer-Encoding，那么 Content-Length 将被忽视。
- 使用长连接的话，会持续的推送动态内容。

那我们来模拟一下吧

```
const server = require('http').createServer();
server.on('request', (req, res) => {
  if(req.url === '/index') {
   // 设置数据类型
    res.setHeader('Content-Type', 'text/html; charset=utf8');
    res.setHeader('Content-Length', 10);
    res.setHeader('Transfer-Encoding', 'chunked');

    res.write("你好，使用的是 Transfer-Encoding 设置传输数据形式");
    setTimeout(() => {
      res.write("第一次传输数据给您<br/>");
    }, 1000);
    res.write("骚等一下");
    setTimeout(() => {
      res.write("第一次传输数据给您");
      res.end()
    }, 3000);
  }
})

server.listen(3000, () => {
  console.log("成功启动--TinaTian");
})
```

复制代码

上面使用的是 nodejs 中 http 模块，有兴趣的小伙伴可以去试一试，以上就是 HTTP 对「定长数据」和「不定长数据」传输过程中的处理手段。

介绍一下 HTTPS 和 HTTP 区别

HTTPS 要比 HTTPS 多了 secure 安全性这个概念，实际上，HTTPS 并不是一个新的应用层协议，它其实就是 HTTP + TLS/SSL 协议组合而成，而安全性的保证正是 SSL/TLS 所做的工作。

「SSL」

安全套接层（Secure Sockets Layer）

「TLS」

（传输层安全，Transport Layer Security）

现在主流的版本是 TLS/1.2，之前的 TLS1.0、TLS1.1 都被认为是不安全的，在不久的将来会被完全淘汰。

「HTTPS 就是身披了一层 SSL 的 HTTP」。

HTTP 与 HTTPS 区别

那么区别有哪些呢

- HTTP 是明文传输协议，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 协议安全。
- HTTPS 比 HTTP 更加安全，对搜索引擎更友好，利于 SEO,谷歌、百度优先索引 HTTPS 网页。
- HTTPS 标准端口 443，HTTP 标准端口 80。
- HTTPS 需要用到 SSL 证书，而 HTTP 不用。

我觉得记住以下两点 HTTPS 主要作用就行

1. 对数据进行加密，并建立一个信息安全通道，来保证传输过程中的数据安全;
2. 对网站服务器进行真实身份认证。

介绍一个 HTTPS 工作原理

上一节来看，我们可以把 HTTPS 理解成「HTTPS = HTTP + SSL/TLS」

> TLS/SSL 的功能实现主要依赖于三类基本算法：散列函数 、对称加密和非对称加密，其利用非对称加密实现身份认证和密钥协商，对称加密算法采用协商的密钥对数据加密，基于散列函数验证信息的完整性。

对称加密

加密和解密用同一个秘钥的加密方式叫做对称加密。Client 客户端和 Server 端共用一套密钥，这样子的加密过程似乎很让人理解，但是随之会产生一些问题。

「问题一:」 WWW 万维网有许许多多的客户端，不可能都用秘钥 A 进行信息加密，这样子很不合理，所以解决办法就是使用一个客户端使用一个密钥进行加密。

「问题二:「既然不同的客户端使用不同的密钥，那么」对称加密的密钥如何传输？」 那么解决的办法只能是「一端生成一个秘钥，然后通过 HTTP 传输给另一端」，那么这样子又会产生新的问题。

「问题三:」 这个传输密钥的过程，又如何保证加密？「如果被中间人拦截，密钥也会被获取,」 那么你会说对密钥再进行加密，那又怎么保存对密钥加密的过程，是加密的过程？

到这里，我们似乎想明白了，使用对称加密的方式，行不通，所以我们需要采用非对称加密

非对称加密

通过上面的分析，对称加密的方式行不通，那么我们来梳理一下非对称加密。采用的算法是 RSA，所以在一些文章中也会看见「传统 RSA 握手」，基于现在 TLS 主流版本是 1.2，所以接下来梳理的是「TLS/1.2 握手过程」。

非对称加密中，我们需要明确的点是

- 有一对秘钥，「公钥」和「私钥」。
- 公钥加密的内容，只有私钥可以解开，私钥加密的内容，所有的公钥都可以解开，这里说的「公钥都可以解开，指的是一对秘钥」。
- 公钥可以发送给所有的客户端，私钥只保存在服务器端。

主要工作流程

梳理起来，可以把「TLS 1.2 握手过程」分为主要的五步

图片内容来自浪里行舟

步骤(1)

Client 发起一个 HTTPS 请求，连接 443 端口。这个过程可以理解成是「请求公钥的过程」。

步骤(2)

Server 端收到请求后，通过第三方机构私钥加密，会把数字证书（也可以认为是公钥证书）发送给 Client。

步骤(3)

- 浏览器安装后会自动带一些权威第三方机构公钥，使用匹配的公钥对数字签名进行解密。
- 根据签名生成的规则对网站信息进行本地签名生成，然后两者比对。
- 通过比对两者签名，匹配则说明认证通过，不匹配则获取证书失败。

步骤(4)

在安全拿到「服务器公钥」后，客户端 Client 随机生成一个「对称密钥」，使用「服务器公钥」(证书的公钥)加密这个「对称密钥」，发送给 Server(服务器)。

步骤(5)

Server(服务器)通过自己的私钥，对信息解密，至此得到了「对称密钥」，此时两者都拥有了相同的「对称密钥」。

接下来，就可以通过该对称密钥对传输的信息加密/解密啦，从上面图举个例子

- Client 用户使用该「对称密钥」加密'明文内容 B',发送给 Server(服务器)
- Server 使用该「对称密钥」进行解密消息，得到明文内容 B。

接下来考虑一个问题，「如果公钥被中间人拿到篡改怎么办呢？」

以下图片来自 leocoder

中间人获取公钥

「客户端可能拿到的公钥是假的，解决办法是什么呢？」

第三方认证

客户端无法识别传回公钥是中间人的，还是服务器的，这是问题的根本，我们是不是可以通过某种规范可以让客户端和服务器都遵循某种约定呢？那就是通过「第三方认证的方式」

在 HTTPS 中，通过 「证书」 + 「数字签名」来解决这个问题。

这里唯一不同的是，假设对网站信息加密的算法是 MD5，通过 MD5 加密后，「然后通过第三方机构的私钥再次对其加密，生成数字签名」。

这样子的话，数字证书包含有两个特别重要的信息「某网站公钥+数字签名」

我们再次假设中间人截取到服务器的公钥后，去替换成自己的公钥，因为有数字签名的存在，这样子客户端验证发现数字签名不匹配，这样子就防止中间人替换公钥的问题。

那么客户端是如何去对比两者数字签名的呢？

- 浏览器会去安装一些比较权威的第三方认证机构的公钥，比如 VeriSign、Symantec 以及 GlobalSign 等等。
- 验证数字签名的时候，会直接从本地拿到相应的第三方的公钥，对私钥加密后的数字签名进行解密得到真正的签名。
- 然后客户端利用签名生成规则进行签名生成，看两个签名是否匹配，如果匹配认证通过，不匹配则获取证书失败。

数字签名作用

数字签名：将网站的信息，通过特定的算法加密，比如 MD5,加密之后，再通过服务器的私钥进行加密，形成「加密后的数字签名」。

第三方认证机构是一个公开的平台，中间人可以去获取。

如果没有数字签名的话，这样子可以就会有下面情况

从上面我们知道，如果「只是对网站信息进行第三方机构私钥加密」的话，还是会受到欺骗。

因为没有认证，所以中间人也向第三方认证机构进行申请，然后拦截后把所有的信息都替换成自己的，客户端仍然可以解密，并且无法判断这是服务器的还是中间人的，最后造成数据泄露。

「总结」

- HTTPS 就是使用 SSL/TLS 协议进行加密传输
- 大致流程：客户端拿到服务器的公钥（是正确的），然后客户端随机生成一个「对称加密的秘钥」，使用「该公钥」加密，传输给服务端，服务端再通过解密拿到该「对称秘钥」，后续的所有信息都通过该「对称秘钥」进行加密解密，完成整个 HTTPS 的流程。
- 「第三方认证」，最重要的是「数字签名」，避免了获取的公钥是中间人的。

SSL 连接断开后如何恢复？

一共有两种方法来恢复断开的 SSL 连接，一种是使用 session ID，一种是 session ticket。

通过 session ID

使用 session ID 的方式，每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器如果有这个编号的记录，那么双方就可以继续使用以前的秘钥，而不用重新生成一把。目前所有的浏览器都支持这一种方法。但是这种方法有一个缺点是，session ID 只能够存在一台服务器上，如果我们的请求通过负载平衡被转移到了其他的服务器上，那么就无法恢复对话。

通过 session ticket

另一种方式是 session ticket 的方式，session ticket 是服务器在上一次对话中发送给客户的，这个 ticket 是加密的，只有服务器能够解密，里面包含了本次会话的信息，比如对话秘钥和加密方法等。这样不管我们的请求是否转移到其他的服务器上，当服务器将 ticket 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。

短轮询、长轮询和 WebSocket 间的区别？

短轮询

短轮询的基本思路:

- 浏览器每隔一段时间向浏览器发送 http 请求，服务器端在收到请求后，不论是否有数据更新，都直接进行 响应。
- 这种方式实现的即时通信，本质上还是浏览器发送请求，服务器接受请求的一个过程，通过让客户端不断的进行请求，使得客户端能够模拟实时地收到服务器端的数据的变化。

优缺点

-

优点是比较简单，易于理解。

-
-

缺点是这种方式由于需要不断的建立 http 连接，严重浪费了服务器端和客户端的资源。当用户增加时，服务器端的压力就会变大，这是很不合理的。

-

长轮询

长轮询的基本思路:

-

首先由客户端向服务器发起请求，当服务器收到客户端发来的请求后，服务器端不会直接进行响应，而是先将 这个请求挂起，然后判断服务器端数据是否有更新。

-
-

如果有更新，则进行响应，如果一直没有数据，则到达一定的时间限制才返回。客户端 JavaScript 响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接。

-

优缺点

-

  长轮询和短轮询比起来，它的优点是「明显减少了很多不必要的 http 请求次数」，相比之下节约了资源。

  -
  -

  长轮询的缺点在于，连接挂起也会导致资源的浪费。

  -

WebSocket

- WebSocket 是 Html5 定义的一个新协议，与传统的 http 协议不同，该协议允许由服务器主动的向客户端推送信息。
- 使用 WebSocket 协议的缺点是在服务器端的配置比较复杂。WebSocket 是一个全双工的协议，也就是通信双方是平等的，可以相互发送消息。

说一说正向代理和反向代理

正向代理

我们常说的代理也就是指正向代理，正向代理的过程，它隐藏了真实的请求客户端，服务端不知道真实的客户端是谁，客户端请求的服务都被代理服务器代替来请求。

反向代理

这种代理模式下，它隐藏了真实的服务端，当我们向一个网站发起请求的时候，背后可能有成千上万台服务器为我们服务，具体是哪一台，我们不清楚，我们只需要知道反向代理服务器是谁就行，而且反向代理服务器会帮我们把请求转发到真实的服务器那里去，一般而言反向代理服务器一般用来实现负载平衡。

负载平衡的两种实现方式？

-

一种是使用反向代理的方式，用户的请求都发送到反向代理服务上，然后由反向代理服务器来转发请求到真实的服务器上，以此来实现集群的负载平衡。

-
-

另一种是 DNS 的方式，DNS 可以用于在冗余的服务器上实现负载平衡。因为现在一般的大型网站使用多台服务器提供服务，因此一个域名可能会对应多个服务器地址。当用户向网站域名请求的时候，DNS 服务器返回这个域名所对应的服务器 IP 地址的集合，但在每个回答中，会循环这些 IP 地址的顺序，用户一般会选择排在前面的地址发送请求。以此将用户的请求均衡的分配到各个不同的服务器上，这样来实现负载均衡。这种方式有一个缺点就是，由于 DNS 服务器中存在缓存，所以有可能一个服务器出现故障后，域名解析仍然返回的是那个 IP 地址，就会造成访问的问题。

-

参考

- 听说『99% 的人都理解错了 HTTP 中 GET 与 POST 的区别』？？
- 如何理解 HTTP 响应的状态码？
- HTTP 响应代码 | MDN
- 图解 HTTP 缓存
- 看完这篇 HTTP，跟面试官扯皮就没问题了
- HTTP keep-alive 二三事
- 深入理解 HTTPS 工作原理
- 看图学 HTTPS
- 轮询、长轮询、长连接、websocket
- DNS 原理入门