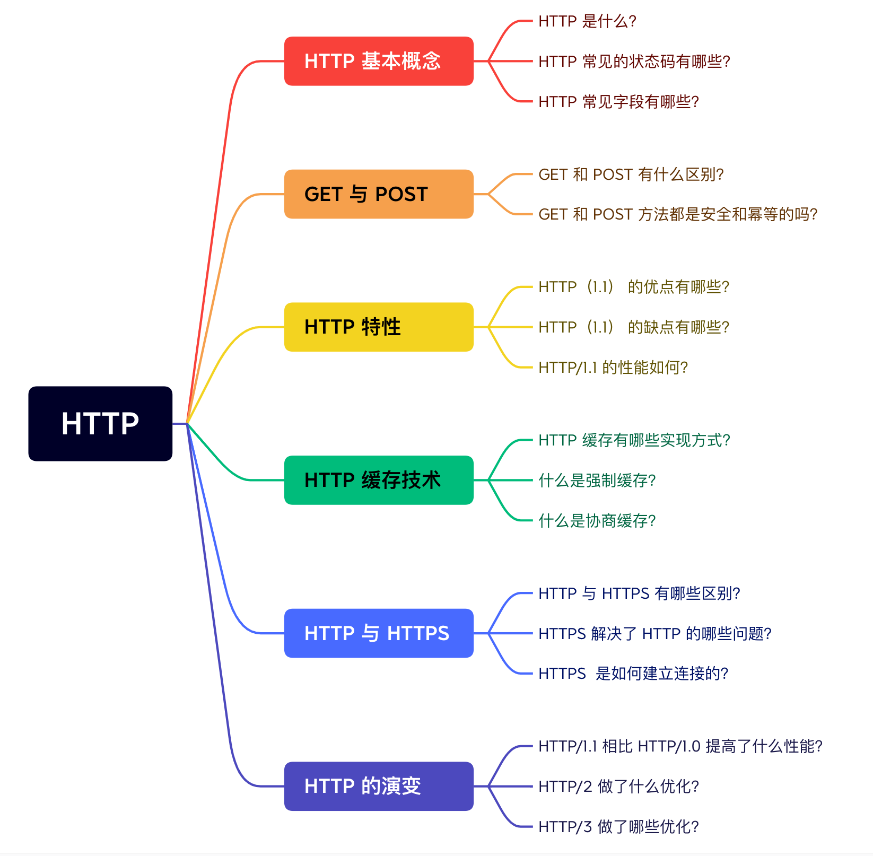
**3.1 HTTP 常见面试题**



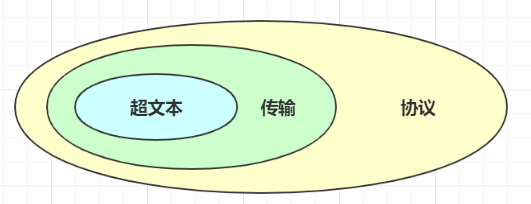
## HTTP 基本概念

### HTTP 是什么？

HTTP 是超文本传输协议（**H**yper**T**ext **T**ransfer **P**rotocol）

详细解释「超文本传输协议」？

拆成三个部分：超文本、传输、协议



**协议**：HTTP 是一个用在计算机世界里的**协议**。它使用计算机能够理解的语言确立了一种计算机之间交流通信的规范（**两个以上的参与者**），以及相关的各种控制和错误处理方式（**行为约定和规范**）。

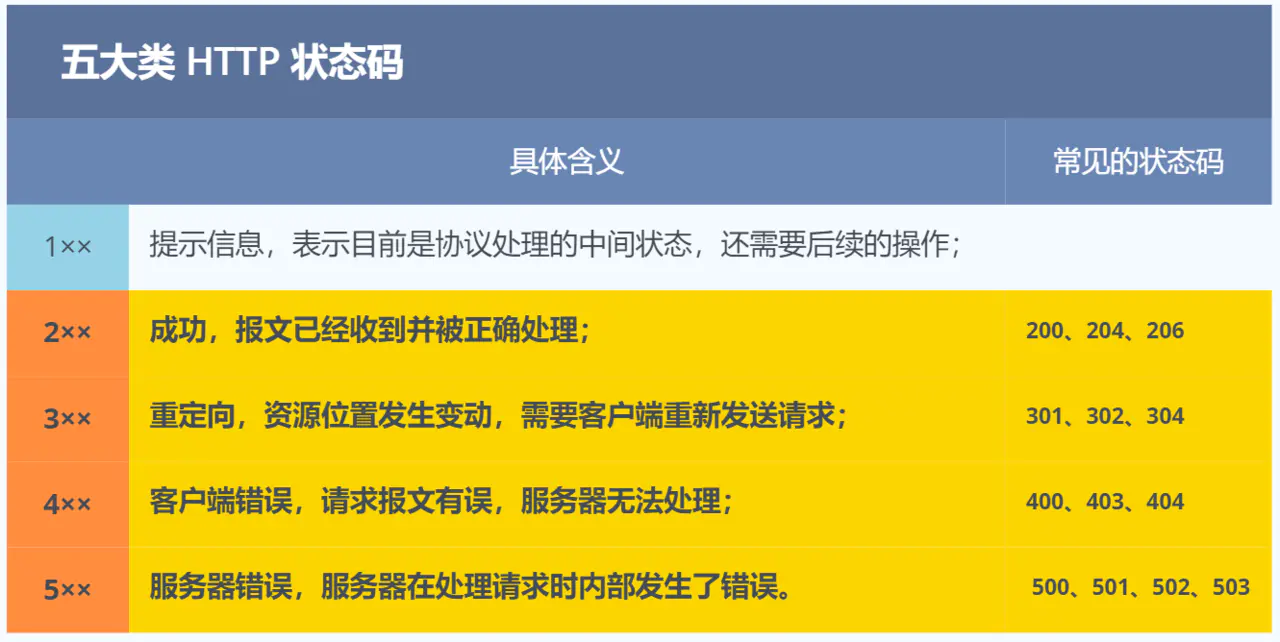
**传输[协议]**：HTTP 是一个在计算机世界里专门用来在**两点之间传输数据**的约定和规范

**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**」

那「HTTP 是用于从互联网服务器传输超文本到本地浏览器的协议」，这种说法正确吗？

这种说法是**不正确**的。因为也可以是「服务器<-->服务器」，所以采用**两点之间**的描述会更准确。

### HTTP 常见的状态码有哪些？



1xx 类状态码属于**提示信息**，是协议处理中的一种**中间状态**，实际用到的比较少。

2xx 类状态码表示服务器**成功处理**了客户端的请求，也是我们最愿意看到的状态。

* 「**200 OK**」是最常见的成功状态码，表示一切正常。如果是非 HEAD 请求，服务器返回的响应头都会有 body 数据。
* 「**204 No Content**」也是常见的成功状态码，与 200 OK 基本相同，但响应头没有 body 数据。
* 「**206 Partial Content**」是应用于 HTTP 分块下载或断点续传，表示响应返回的 body 数据并不是资源的全部，而是其中的一部分，也是服务器处理成功的状态。

3xx 类状态码表示客户端请求的资源发生了变动，需要客户端用新的 URL 重新发送请求获取资源，也就是**重定向**。

* 「**301 Moved Permanently**」表示永久重定向，说明请求的资源已经不存在了，需改用新的 URL 再次访问。
* 「**302 Found**」表示临时重定向，说明请求的资源还在，但暂时需要用另一个 URL 来访问。

301 和 302 都会在响应头里使用字段 Location，指明后续要跳转的 URL，浏览器会**自动**重定向新的 URL。

* 「**304 Not Modified**」不具有跳转的含义，表示资源未修改，重定向已存在的缓冲文件，也称缓存重定向，也就是告诉客户端可以继续使用缓存资源，用于缓存控制。

4xx 类状态码表示**客户端发送的报文有误**，服务器无法处理，也就是错误码的含义。

* 「**400 Bad Request**」表示客户端请求的报文有错误，但只是个笼统的错误。
* 「**403 Forbidden**」表示**服务器**禁止访问资源，并不是客户端的请求出错。
* 「**404 Not Found**」表示请求的资源在服务器上不存在或未找到，所以无法提供给客户端。

5xx 类状态码表示客户端**请求报文正确**，但是**服务器处理时内部发生了错误**，属于服务器端的错误码。

* 「**500 Internal Server Error**」与 400 类型，是个笼统通用的错误码，服务器发生了什么错误，我们并不知道。
* 「**501 Not Implemented**」表示客户端请求的功能还不支持，类似“即将开业，敬请期待”的意思。
* 「**502 Bad Gateway**」通常是服务器作为**网关**或**代理**时返回的错误码，表示服务器自身工作正常，访问**后端服务器**发生了错误。
* 「**503 Service Unavailable**」表示**服务器当前很忙**，暂时无法响应客户端，类似“网络服务正忙，请稍后重试”的意思。

### HTTP 常见字段有哪些

Host 字段

客户端发送请求时，用来指定**服务器的域名。**有了 Host 字段，就可以将请求发往「**同一台**」服务器上的不同网站。

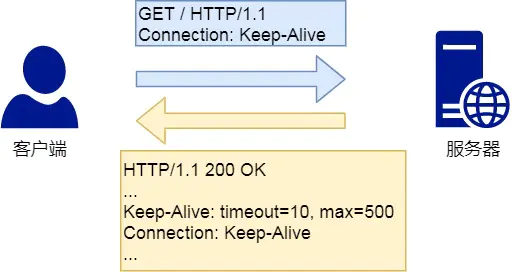
举例：Host: www.A.com

Content-Length字段

服务器在返回数据时，会有 Content-Length 字段，表明**本次回应的数据长度**。

Connection 字段

Connection 字段最常用于客户端要求服务器使用「**HTTP 长连接**」机制，以便其他请求复用。HTTP 长连接的特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。



HTTP/**1.1** 版本的默认连接都是长连接，但为了兼容老版本的 HTTP，需要指定 Connection 首部字段的值为 Keep-Alive（注意和TCP的Keepalive不是同一个东西）

Content-Type 字段

Content-Type 字段用于**服务器回应**时，告诉客户端，本次数据是什么格式

举例：Content-Type: text/html; Charset=utf-8

Accept 字段

客户端请求的时候，可以使用 Accept 字段声明自己可以接受哪些数据格式

举例：Accept: \*/\*

表示可以接受任意格式

Content-Encoding字段

Content-Encoding 字段说明数据的压缩方法。表示服务器返回的数据使用了什么**压缩格式**

举例：Content-Encoding: gzip

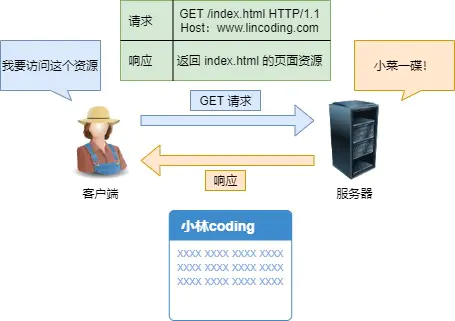
Accept-Encoding 字段

客户端在请求时，用 Accept-Encoding 字段说明自己可以接受哪些压缩方法

举例：Accept-Encoding: gzip, deflate

## GET 与 POST

**GET 的语义是从服务器获取指定的资源**，这个资源可以是静态的文本、页面、图片视频等。GET 请求的参数位置一般是写在 **URL** 中，URL 规定只能支持 ASCII，所以 GET 请求的参数只允许 ASCII 字符 ，而且浏览器会对 URL 的长度有限制



**POST 的语义是根据请求负荷（报文body）对指定的资源做出处理**

POST 请求携带数据的位置一般是写在报文 body 中，body 中的数据可以是任意格式的数据，只要客户端与服务端协商好即可，而且浏览器不会对 body 大小做限制

### GET 和 POST 方法都是安全和幂等的吗？

根据RFC 规范定义的语义：

* GET 方法就是**安全且幂等**的，因为它是「**只读**」操作，无论操作多少次，服务器上的数据都是安全的，且每次的结果都是相同的。所以，可以对 GET 请求的数据做缓存，这个缓存可以做到浏览器本身上（彻底避免浏览器发请求），也可以做到代理上（如nginx），而且在浏览器中 GET 请求可以保存为书签。
* POST 因为是「新增或提交数据」的操作，会修改服务器上的资源，所以是**不安全的**，且**多次提交数据就会创建多个资源**，所以**不是幂等的**。所以，浏览器一般不会缓存 POST 请求，也不能把 POST 请求保存为书签。

但是实际过程中，开发者不一定会按照 RFC 规范定义的语义来实现 GET 和 POST 方法。比如：

* 可以用 GET 方法实现新增或删除数据的请求，这样实现的 GET 方法自然就不是安全和幂等。
* 可以用 POST 方法实现查询数据的请求，这样实现的 POST 方法自然就是安全和幂等。

因为任何请求都可以带 body，也可以带URL 中的查询参数

## HTTP 缓存技术

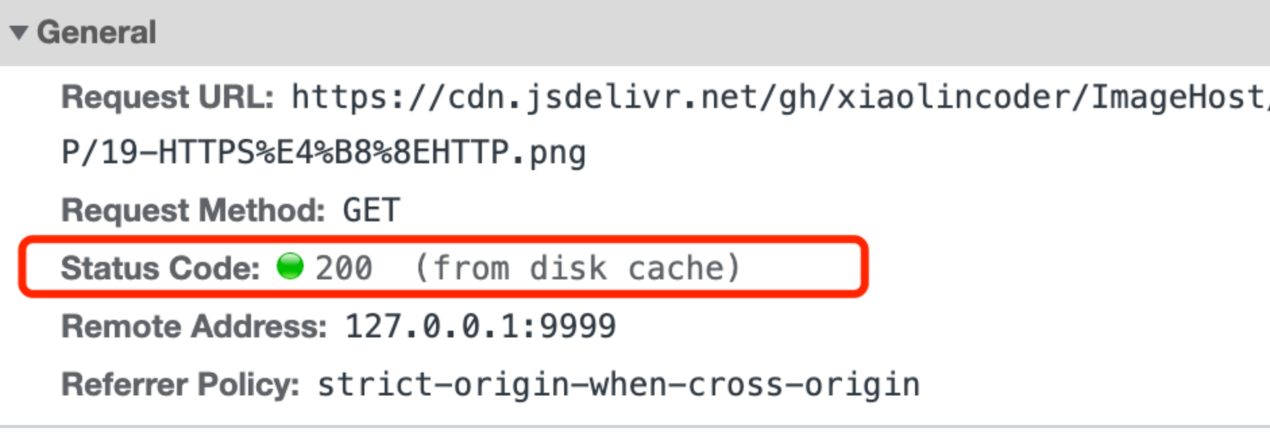
对于一些具有重复性的 HTTP 请求，比如每次请求得到的数据都一样的，我们可以把这对「**请求-响应**」的数据都**缓存在本地**，从而**避免发送 HTTP 请求**

HTTP 缓存有两种实现方式，分别是**强制缓存**和**协商缓存**。

强制缓存决定权在客户端浏览器，协商缓存决定权在服务器

### 强制缓存

**强缓存**指的是只要浏览器判断缓存没有过期，则直接使用浏览器的本地缓存，决定是否使用缓存的主动性在于浏览器这边。



size 项中标识的是 from disk cache，就是使用了**强制缓存**

强缓存是利用下面这两个 HTTP 响应头部（Response Header）字段实现的，它们都用来表示资源在客户端缓存的有效期：

* **Cache-Control**， 是一个**相对时间**；
* **Expires**，是一个**绝对时间**；

如果 HTTP 响应头部同时有 Cache-Control 和 Expires 字段的话，**Cache-Control 的优先级高于 Expires** 。

Cache-control 选项更多一些，设置更加精细，所以建议使用 Cache-Control 来实现强缓存。具体的实现流程如下：

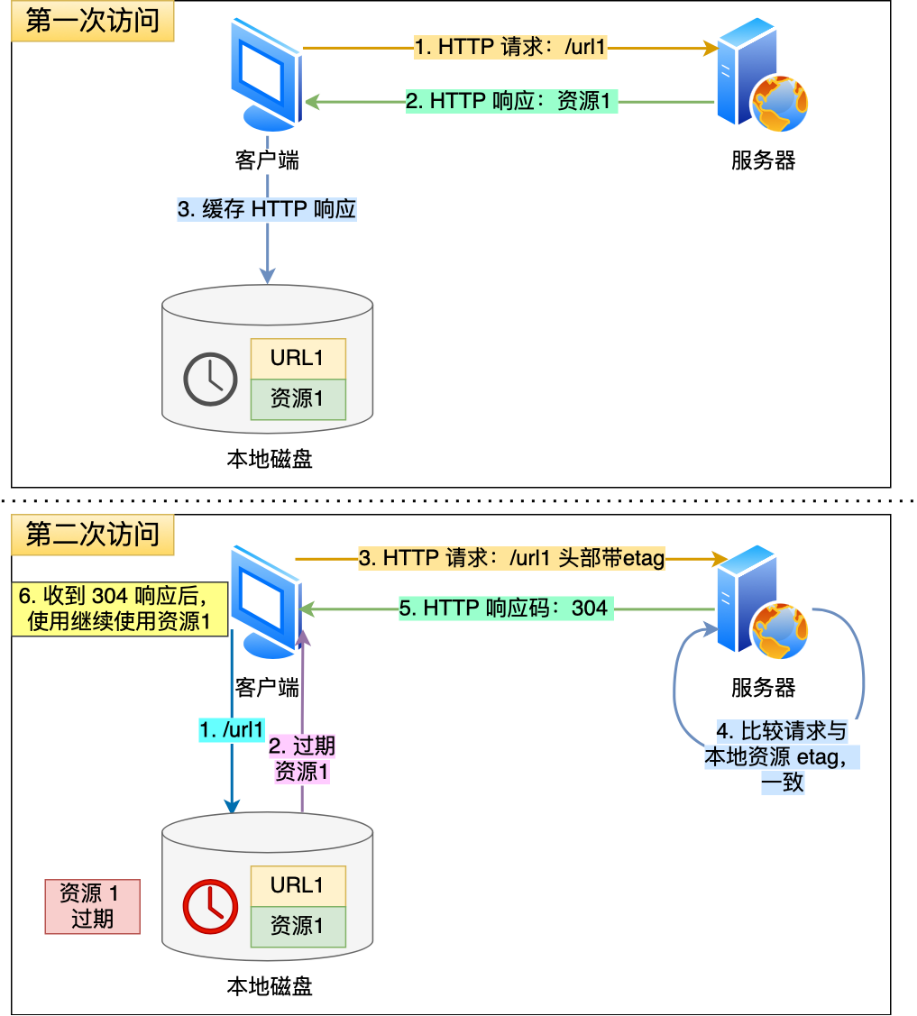
当浏览器**第一次请求**访问服务器资源时，服务器会在返回这个资源的同时，在 Response 头部加上 Cache-Control，Cache-Control 中设置了**过期时间**大小；

浏览器再次请求访问服务器中的该资源时，**会先通过请求资源的时间与 Cache-Control 中设置的过期时间大小，来计算出该资源是否过期，如果没有，则使用该缓存，否则重新请求服务器**；

服务器再次收到请求后，会**再次更新 Response 头部**的 Cache-Control。

### 协商缓存

通过这种方式，服务端告知客户端**是否可以**使用缓存的方式



协商缓存可以基于两种头部来实现：

**第一种**：请求头部中的 **If-Modified-Since** 字段与响应头部中的 **Last-Modified** 字段实现，这两个字段的意思是：

* **响应头部**中的 Last-Modified：标示**这个响应资源的最后修改时间**；
* **请求头部**中的 If-Modified-Since：第一次请求时浏览器会保存**响应头中的 Last-Modified**，**再次发起请求的时候在If-Modified-Since中写上 Last-Modified 的时间**，服务器收到请求后发现有 If-Modified-Since 则与被请求资源的最后修改时间进行对比（Last-Modified），如果**最后修改时间较新（大）**，说明资源又被改过，则**返回最新资源**，HTTP 200 OK；如果**最后修改时间较旧（小）**，说明资源无新修改，**响应 HTTP 304 走缓存**。

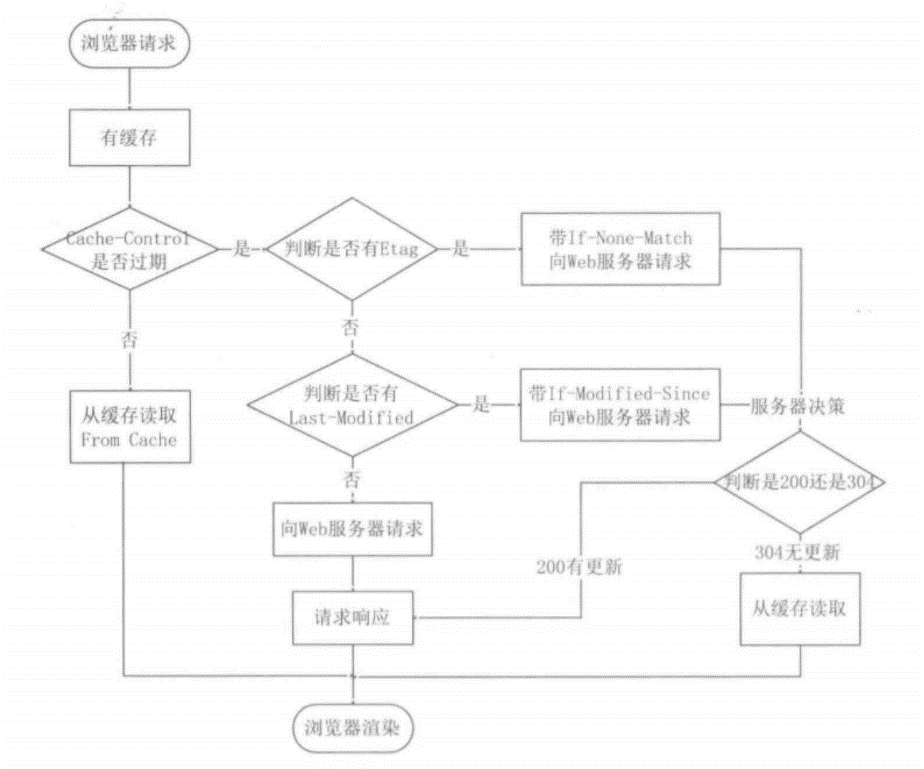
**第二种**：请求头部中的 If-None-Match 字段与响应头部中的**ETag**字段，这两个字段的意思是：

* **响应头部**中 Etag：唯一标识响应资源；
* 请求头部中的 If-None-Match：当资源过期时，浏览器发现响应头里有 Etag，则再次向服务器发起请求时，会将请求头 If-None-Match 值设置为 Etag 的值。服务器收到请求后和当前的资源进行比对（比对请求头中的Etag和真实的Etag），如果资源没有变化返回 **304**，如果资源变化了返回 **200**。

第一种实现方式是**基于时间**实现的，第二种实现方式是**基于一个唯一标识**实现的，相对来说后者可以更加准确地判断文件内容是否被修改，避免由于时间篡改导致的不可靠问题。

如果在第一次请求资源的时候，服务端返回的 HTTP 响应头部同时有 Etag 和 Last-Modified 字段，那么客户端再下一次请求的时候，如果带上了 ETag 和 Last-Modified 字段信息给服务端，**这时 Etag 的优先级更高**，也就是服务端先会判断 Etag 是否变化了，如果 Etag 有变化就不用再判断 Last-Modified 了，如果 Etag 没有变化，然后再看 Last-Modified

**协商缓存这两个字段都需要配合强制缓存中 Cache-Control 字段来使用，只有在未能命中强制缓存的时候，才能发起带有协商缓存字段的请求**



先用Cache-Control判断资源是否过期，如果没过期直接用缓存；如果过期则按顺序用Etag和Last-Modified让服务器判断资源是否更改，再决定发送新的还是继续用缓存。

## HTTP 特性

### HTTP/1.1 的优点

**1. 简单**

HTTP 基本的**报文格式就是 header + body**，**头部信息也是 key-value**简单文本的形式，**易于理解**，降低了学习和使用的门槛。

**2. 灵活和易于扩展**

HTTP 协议里的各类请求方法、URI/URL、状态码、头字段等**每个组成要求都没有被固定死**，都允许开发人员**自定义和扩充**。

同时 HTTP 由于是工作在应用层（ OSI 第七层），则它**下层可以随意变化**，比如：

* HTTPS 就是在 HTTP 与 TCP 层之间增加了 SSL/TLS 安全传输层；
* HTTP/1.1 和 HTTP/2.0 传输协议使用的是 TCP 协议，而到了 HTTP/3.0 传输协议改用了 UDP 协议。

**3. 应用广泛和跨平台**

互联网发展至今，HTTP 的应用范围非常的广泛，从台式机的浏览器到手机上的各种 APP，从看新闻、刷贴吧到购物、理财、吃鸡，HTTP 的应用遍地开花，同时天然具有**跨平台**的优越性。

### HTTP/1.1 的缺点

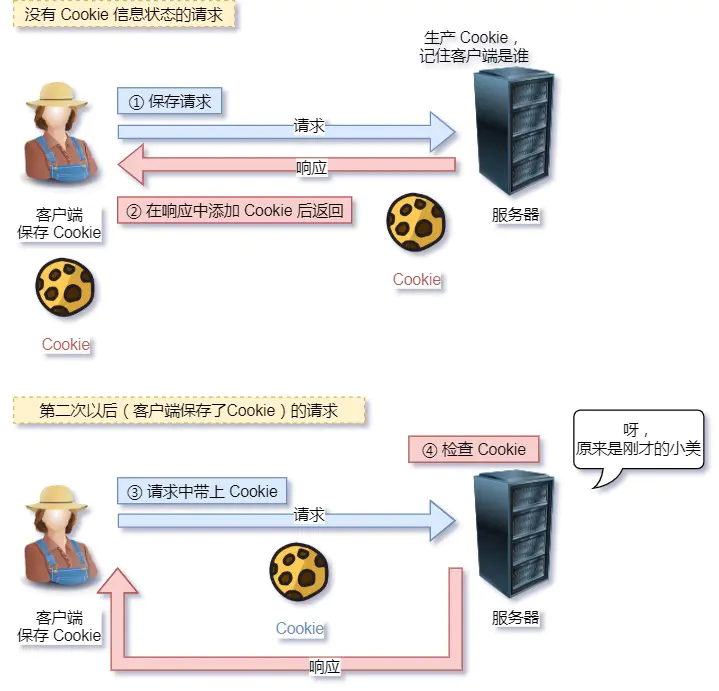
1. 无状态双刃剑

**好处**：因为服务器不会去记忆 HTTP 的状态，所以**不需要额外的资源来记录状态**信息，这能减轻服务器的负担，能够把更多的 CPU 和内存用来对外提供服务。

**坏处**：既然服务器没有记忆能力，它在**完成有关联性的操作时会非常麻烦**。

例如登录->添加购物车->下单->结算->支付，服务器不知道这些请求是有关联的，每次都要问一遍身份信息。

解决方法： **Cookie** 技术。通过在请求和响应报文中写入 Cookie 信息来控制客户端的状态。



2. 明文传输双刃剑

**好处**：方便阅读

**坏处**：信息容易被窃取

3. 不安全

容易被窃听、通信对方身份伪装、报文内容篡改

解决方法：HTTPS

### HTTP/1.1 的性能如何

1. 长连接

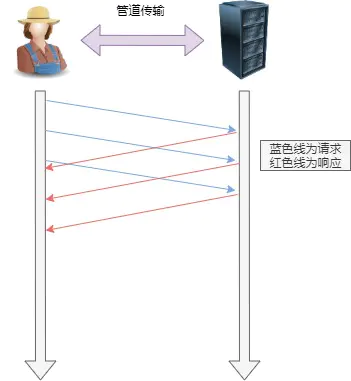
HTTP 协议是基于 **TCP/IP**，并且使用了「**请求 - 应答**」的通信模式，所以性能的关键就在这**两点**里

早期的 HTTP/1.0每发起一个请求，都要新建一次 TCP 连接（三次握手），而且是串行请求。HTTP/1.1 提出了**长连接**的通信方式，也叫持久连接，减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载。

只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。如果某个 HTTP 长连接超过一定时间没有任何数据交互，服务端就会主动断开这个连接。

2. 管道网络传输

即可**在同一个 TCP 连接里面**，客户端可以发起多个请求，只要第一个请求发出去了，**不必等其回来**，就可以发第二个请求出去，可以**减少整体的响应时间。**



**服务器必须按照接收请求的顺序发送对这些管道化请求的响应。**如果服务端在处理 A 请求时耗时比较长，那么后续的请求的处理都会被阻塞住，这称为「**队头堵塞**」。

**HTTP/1.1 管道解决了请求的队头阻塞，但是没有解决响应的队头阻塞**

浏览器基本都没有支持HTTP/1.1管道化技术

## HTTP 与 HTTPS

HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 **SSL/TLS 安全协议**，使得报文能够加密传输。HTTPS 在 TCP 三次握手之后，还需进行 **SSL/TLS 的握手过程**，才可进入加密报文传输。

HTTP 默认端口号是 **80**，HTTPS 默认端口号是 **443**

HTTPS解决了HTTP的：

**窃听风险：通过信息加密**

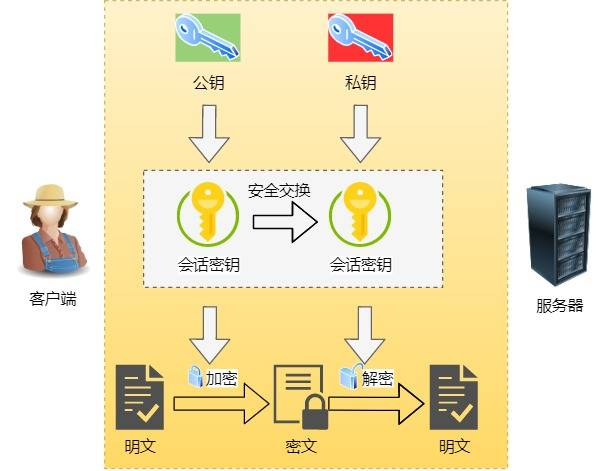
**篡改风险：通过校验机制**

**冒充风险：通过身份证书**

HTTPS 是如何解决上面的三个风险的？

* **混合加密**的方式实现信息的**机密性**，解决了窃听的风险。
* **摘要算法**的方式来实现**完整性**，它能够为数据生成独一无二的「指纹」，指纹用于校验数据的完整性，解决了篡改的风险。
* 将服务器公钥放入到**数字证书**中，解决了冒充的风险。

1. 混合加密



HTTPS 采用的是**对称加密**和**非对称加密**结合的「混合加密」方式：

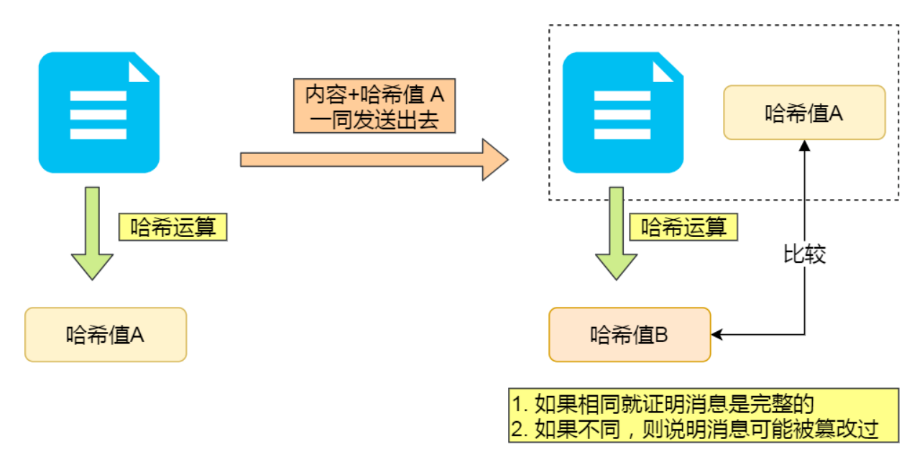
* 在通信建立前采用**非对称加密**的方式交换「会话秘钥」，后续就不再使用非对称加密。
* 在通信过程中全部使用**对称加密**的「会话秘钥」的方式加密明文数据。

采用「混合加密」的方式的原因：

* **对称加密**只使用一个密钥，运算速度快，密钥必须保密，无法做到安全的密钥交换。
* **非对称加密**使用两个密钥：公钥和私钥，公钥可以任意分发而私钥保密，解决了密钥交换问题但速度慢。

2. 摘要算法 + 数字签名

那么，在计算机里会**用摘要算法（哈希函数）来计算出内容的哈希值**，也就是内容的「指纹」，这个**哈希值是唯一的，且无法通过哈希值推导出内容**。



通过哈希算法可以确保内容不会被篡改，**但是并不能保证「内容 + 哈希值」不会被中间人替换，因为这里缺少对客户端收到的消息是否来源于服务端的证明**。

公钥和私钥可以**双向加解密：**

* **公钥加密，私钥解密**。这个目的是为了**保证内容传输的安全**，因为被公钥加密的内容，其他人是无法解密的，**只有持有私钥的人**，才能解密出实际的内容；
* **私钥加密，公钥解密**。这个目的是为了**保证消息不会被冒充**，因为私钥是不可泄露的，如果公钥能正常解密出私钥加密的内容，就能证明这个消息是来源于持有私钥身份的人发送的。

非对称加密的用途主要在于**通过「私钥加密，公钥解密」的方式，来确认消息的身份**，我们常说的**数字签名算法**，就是用的是这种方式，不过私钥加密内容不是内容本身，而是**对内容的哈希值加密**。私钥是由服务端保管，然后服务端会向客户端颁发对应的公钥。如果客户端收到的信息，能被公钥解密，就说明该消息是由服务器发送的。

3. 数字证书

前面**还缺少身份验证的环节**，万一公钥是被伪造的呢



将服务器公钥放在数字证书（由数字证书认证机构颁发）中，只要证书是可信的，公钥就是可信的。

### HTTPS 是如何建立连接的？其间交互了什么？

SSL/TLS 协议基本流程：

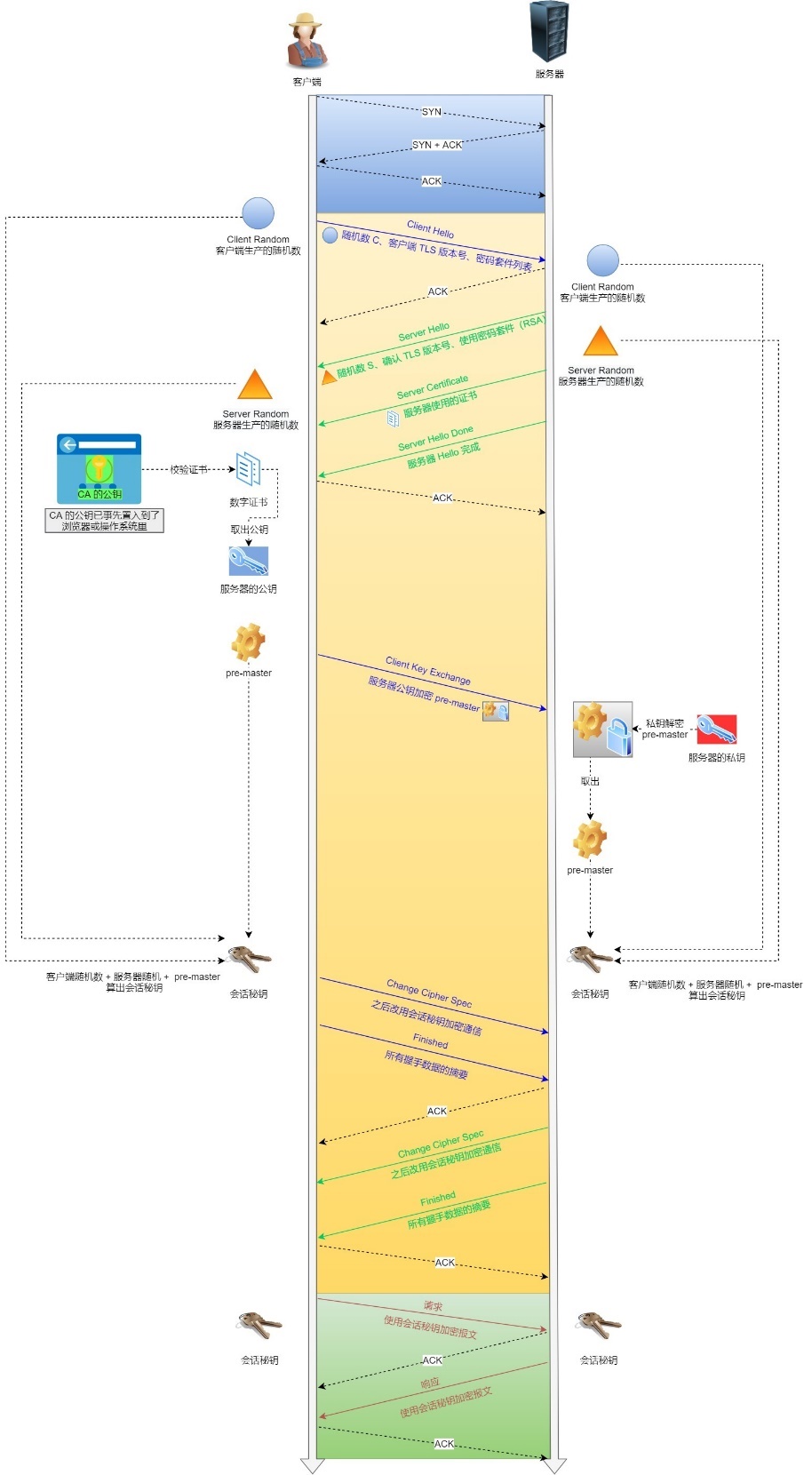
* 客户端向服务器索要并验证服务器的公钥。
* 双方协商生产「会话秘钥」。
* 双方采用「会话秘钥」进行加密通信。

前两步也就是 SSL/TLS 的建立过程，也就是 TLS 握手阶段。

TLS 的「握手阶段」涉及**四次**通信，使用不同的密钥交换算法，TLS 握手流程也会不一样的，现在常用的密钥交换算法有两种：[RSA 算法](https://xiaolincoding.com/network/2_http/https_rsa.html" \t "_blank)和 [ECDHE 算法](https://xiaolincoding.com/network/2_http/https_ecdhe.html)

**基于 RSA 算法的 TLS 握手过程**：

TLS 协议建立的详细流程：



1. ClientHello

首先，由客户端向服务器发起加密通信请求，也就是 ClientHello 请求。

在这一步，客户端主要向服务器发送以下信息：

（1）客户端支持的 **TLS 协议版本**，如 TLS 1.2 版本。

（2）客户端生产的**随机数**（Client Random），后面用于生成「会话秘钥」条件之一。

（3）客户端支持的**密码套件列表**，如 RSA 加密算法。

2. SeverHello

服务器收到客户端请求后，向客户端发出响应，也就是 SeverHello。服务器回应的内容有如下内容：

（1）**确认 TLS 协议版本**，如果浏览器不支持，则关闭加密通信。

（2）服务器生产的**随机数**（Server Random），也是后面用于生产「会话秘钥」条件之一。

（3）**确认的密码套件列表**，如 RSA 加密算法。

（4）服务器的**数字证书**。

3.客户端回应

客户端收到服务器的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务器的数字证书的真实性。

如果证书没有问题，客户端会**从数字证书中取出服务器的公钥**，然后使用它加密报文，向服务器发送如下信息：

（1）一个**随机数**（pre-master key）。该随机数会被服务器公钥加密。

（2）**加密通信算法改变通知**，表示随后的信息都将用「会话秘钥」加密通信。

（3）客户端**握手结束通知**，表示客户端的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供服务端校验。

上面第一项的随机数是整个握手阶段的第三个随机数，会发给服务端，所以这个随机数客户端和服务端都是一样的。

**服务器和客户端有了这三个随机数（Client Random、Server Random、pre-master key），接着就用双方协商的加密算法，各自生成本次通信的「会话秘钥」**。

4. 服务器的最后回应

服务器收到客户端的第三个随机数（pre-master key）之后，通过协商的加密算法，计算出本次通信的「会话秘钥」。

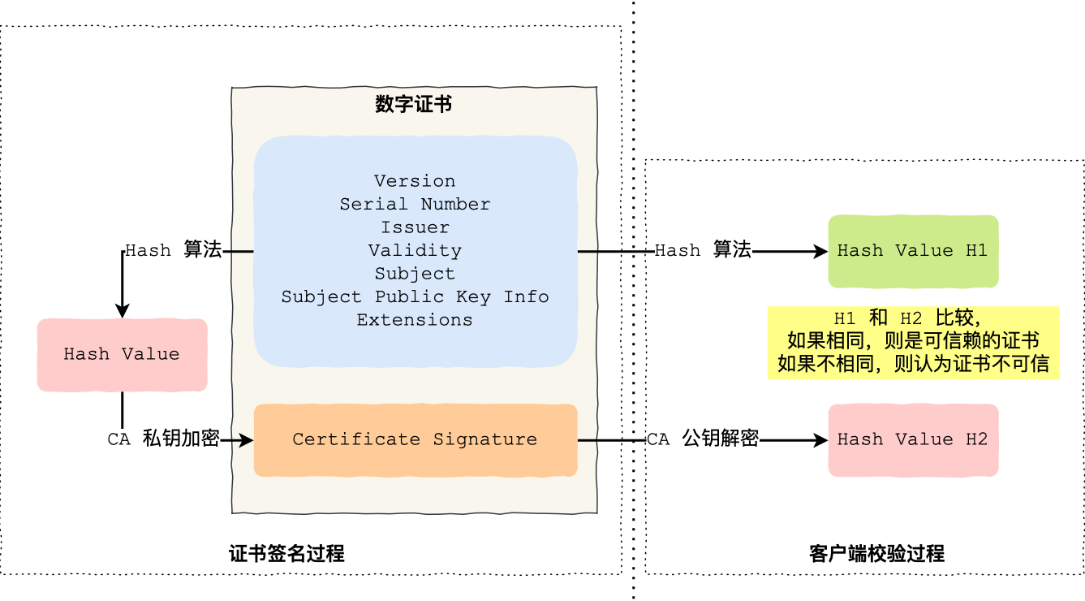
然后，向客户端发送最后的信息：

（1）**加密通信算法改变通知**，表示随后的信息都将用「会话秘钥」加密通信。

（2）服务器**握手结束通知**，表示服务器的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供客户端校验。

至此，整个 TLS 的握手阶段全部结束。接下来，客户端与服务器进入加密通信，就完全是使用普通的 HTTP 协议，只不过用「会话秘钥」加密内容。

客户端校验数字证书的流程是怎样的？



CA **签发证书**的过程，如上图左边部分：

* 首先 CA 会把持有者的**公钥**、**持有者信息**、**证书认证机构（CA）信息**、**数字签名及使用的算法**、**证书有效期**等信息打成一个包，然后对这些信息进行 Hash 计算，得到一个 Hash 值；
* 然后 CA 会使用自己的私钥**将该 Hash 值加密**，生成 Certificate Signature，也就是 CA 对证书做了**签名**；
* 最后将 Certificate Signature 添加在文件证书上，形成数字证书；

客户端**校验服务端的数字证书**的过程，如上图右边部分：

* 首先客户端会使用同样的 Hash 算法获取该证书的 Hash 值 H1；
* 通常**浏览器和操作系统中集成了 CA 的公钥信息**，浏览器收到证书后可以使用 CA 的公钥解密 Certificate Signature 内容，得到一个 Hash 值 H2 ；
* 最后比较 H1 和 H2，如果值相同，则为可信赖的证书，否则则认为证书不可信。

证书的验证过程中**还存在一个证书信任链的问题:**



向 CA 申请的证书一般不是根证书签发的。最开始客户端只信任根证书 GlobalSign Root CA 证书(软件商预载了根证书的 GlobalSign 都可被信任)，然后逐级验证

**为了确保根证书的绝对安全性，将根证书隔离地越严格越好，不然根证书如果失守了，那么整个信任链都会有问题。**

**RSA 算法的缺陷**

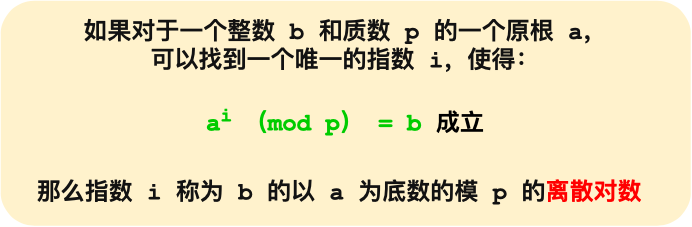
使用 RSA 密钥协商算法的最大问题是**不支持前向保密（不能对以前的信息保密）**

一旦服务端的私钥泄漏了，就可以用私钥解密随机数，过去被第三方截获的所有 TLS 通讯密文都会被破解。

解决方法：**ECDHE 算法**

# ECDHE 握手解析

ECDHE 密钥协商算法是 DH 算法演进过来



底数 a 和模数 p 是离散对数的公共参数，也就说是公开的，b 是真数，i 是对数。知道了对数，就可以用上面的公式计算出真数。但反过来，**知道真数却很难推算出对数**。

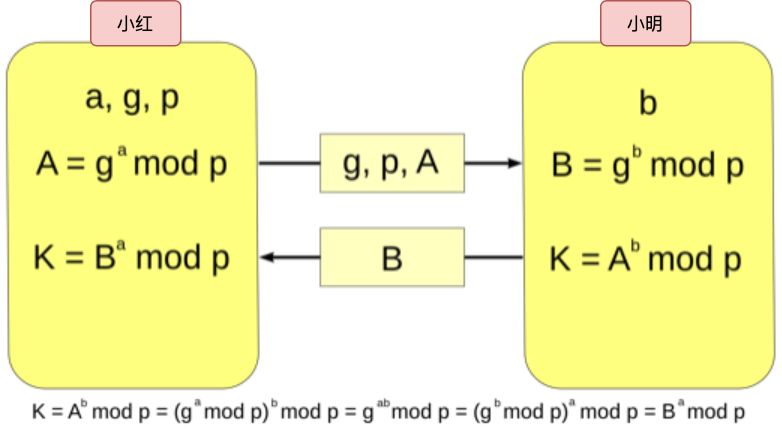
## DH 算法

先确定模数和底数作为算法的参数，这两个参数是公开的，用 P 和 G 来代称。

小红和小明各自生成一个随机整数作为**私钥**，双方的私钥要各自严格保管，不能泄漏，小红的私钥用 a 代称，小明的私钥用 b 代称。

* 小红的公钥记作 A，A = G ^ a ( mod P )；
* 小明的公钥记作 B，B = G ^ b ( mod P )；

A 和 B 也是公开的



K 就是小红和小明之间用的**对称加密密钥**，可以作为会话密钥使用

## DHE 算法

根据私钥生成的方式，DH 算法分为两种实现：

* static DH 算法，这个是已经被废弃了；
* DHE 算法，现在常用的；

static DH 算法里有一方的私钥是静态的，也就说每次密钥协商的时候有一方的私钥都是一样的，一般是**服务器方固定**，即 a 不变，**客户端的私钥则是随机生成的**。

于是，DH 交换密钥时就只有客户端的公钥是变化，而服务端公钥是不变的，那么随着时间延长，黑客就会截获海量的密钥协商过程的数据，因为密钥协商的过程有些数据是公开的，黑客就可以依据这些数据暴力破解出服务器的私钥，然后就可以计算出会话密钥了，于是之前截获的加密数据会被破解，所以 **static DH 算法不具备前向安全性**。

既然固定一方的私钥有被破解的风险，那么干脆就让双方的私钥在每次密钥交换通信时，都是随机生成的、临时的，这个方式也就是 DHE 算法，E 全称是 ephemeral（临时性的）。

所以，即使有个牛逼的黑客破解了某一次通信过程的私钥，其他通信过程的私钥仍然是安全的，因为**每个通信过程的私钥都是没有任何关系的，都是独立的，这样就保证了「前向安全」**。

## ECDHE 算法

DHE 算法由于计算性能不佳，因为需要做大量的乘法

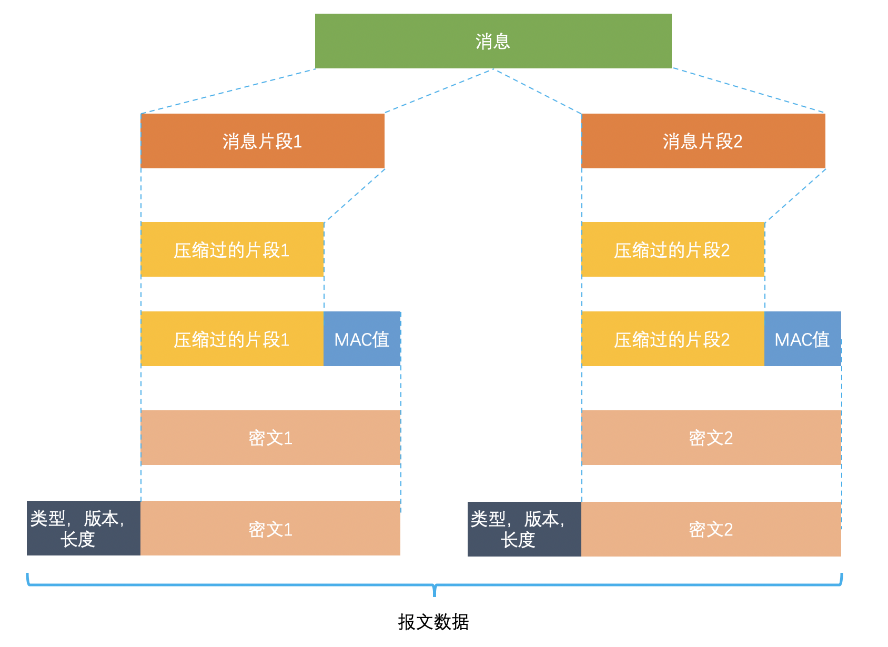
ECDHE 算法是在 DHE 算法的基础上利用了 ECC **椭圆曲线**特性，可以用更少的计算量计算出公钥

### HTTPS 的应用数据是如何保证完整性的？

TLS 在实现上分为**握手协议**和**记录协议**两层：

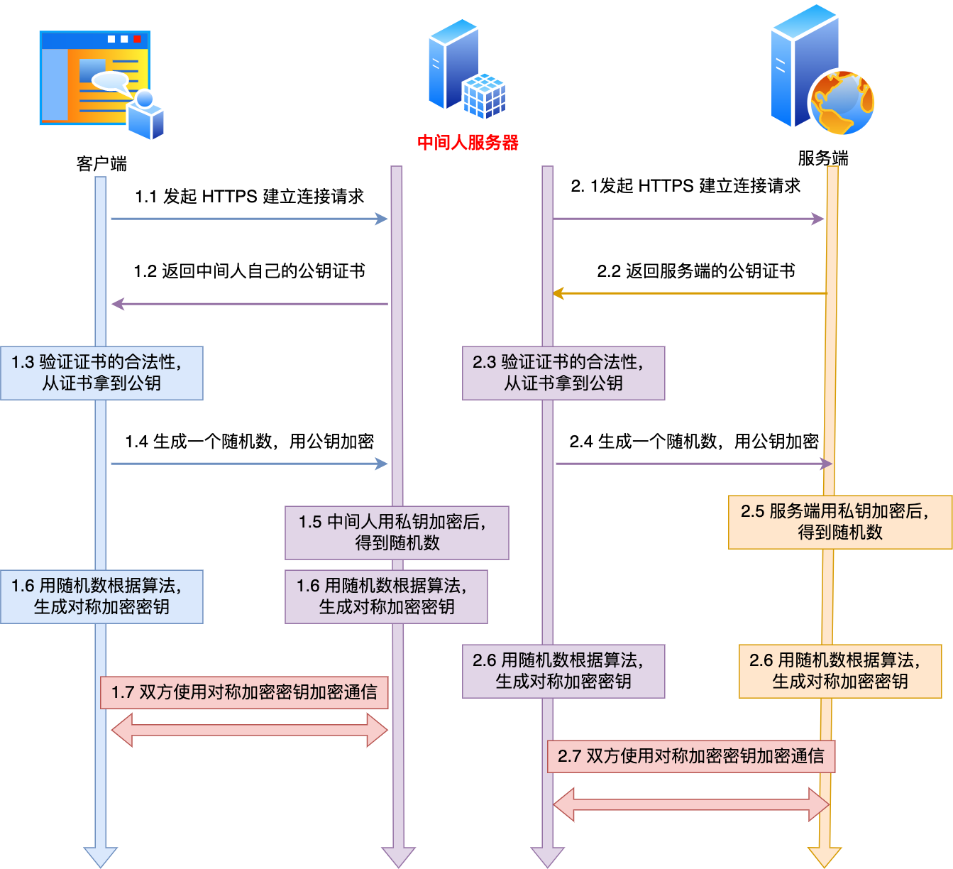
**握手协议：**TLS 四次握手的过程，负责协商加密算法和生成对称密钥，后续用此密钥来保护应用程序数据（即 HTTP 数据）

**记录协议：**负责消息（HTTP 数据）的压缩，加密及数据的认证，



* 首先，消息被分割成多个较短的片段,然后分别对每个片段进行压缩。
* 接下来，经过压缩的片段会被**加上消息认证码（MAC 值，这个是通过哈希算法生成的），这是为了保证完整性，并进行数据的认证**。通过附加消息认证码的 MAC 值，可以识别出篡改。与此同时，为了防止重放攻击，在计算消息认证码时，还加上了片段的编码。
* 再接下来，经过压缩的片段再加上消息认证码会一起通过**对称密码**进行加密。
* 最后，上述经过加密的数据再加上由数据类型、版本号、压缩后的长度组成的报头就是最终的报文数据。

### HTTPS 一定安全可靠吗



中间人服务器与客户端在 TLS 握手过程中，实际上发送了自己伪造的证书给浏览器，而这个伪造的证书是能被浏览器（客户端）识别出是非法的，于是就会提醒用户该证书存在问题。

**HTTPS 协议本身到目前为止还是没有任何漏洞的，即使你成功进行中间人攻击，本质上是利用了客户端的漏洞（用户点击继续访问或者被恶意导入伪造的根证书），并不是 HTTPS 不够安全**。

为什么抓包工具能截取 HTTPS 数据？

自己签发证书，且要被浏览器信任

如何避免被中间人抓取数据？

1. 不要被病毒乘虚而入，而且也不要点击任何证书非法的网站

**2. HTTPS 双向认证：客户端也有自己的证书，服务端也会验证客户端身份**

# HTTPS 如何优化？

## 分析性能损耗

产生性能消耗的两个环节：

* 第一个环节， **TLS 协议握手**过程；
* 第二个环节，握手后的**对称加密报文传输**。

## 硬件优化

**HTTPS 协议是计算密集型，而不是 I/O 密集型**，所以不能把钱花在网卡、硬盘等地方，应该花在 CPU 上。选择可以**支持 AES-NI 特性的 CPU**

## 软件优化

将正在使用的软件升级到最新版本

## 协议优化

### 1.密钥交换算法优化

尽量**选用 ECDHE 密钥交换**算法替换 RSA 算法，尽量**选择 x25519 曲线**

**2. TLS 升级**

把 TLS 1.2 升级成 TLS 1.3，TLS 1.3 大幅度简化了握手的步骤，**完成 TLS 握手只要 1 RTT**，而且安全性更高。**TLS 1.3 把 Hello 和公钥交换这两个消息合并成了一个消息，于是这样就减少到只需 1 RTT 就能完成 TLS 握手**。

TLS1.3**对于密钥交换算法，废除了不支持前向安全性的 RSA 和 DH 算法，只支持 ECDHE 算法**。

## 证书优化

### 1. 证书传输优化

减少证书的大小，**应该选择椭圆曲线（ECDSA）证书，因为在相同安全强度下， ECC 密钥长度比 RSA 短的多**

**2. 证书验证优化**

使用 OCSP来查询证书的有效性，它的工作方式是**向 CA 发送查询请求，让 CA 返回证书的有效状态**。不用查询 CRL（证书吊销列表）

## 会话复用

会话复用分两种：

* 第一种叫 Session ID；
* 第二种叫 Session Ticket

**1. Session ID**

**客户端和服务器首次 TLS 握手连接后，双方会在内存缓存会话密钥，并用唯一的 Session ID 来标识**，Session ID 和会话密钥相当于 key-value 的关系。当客户端再次连接时，hello 消息里会带上 Session ID，服务器收到后就会从内存找，如果找到就直接用该会话密钥恢复会话状态，跳过其余的过程，只用一个消息往返就可以建立安全通信。当然为了安全性，内存中的会话密钥会**定期失效**。

缺点：

* 服务器必须保持每一个客户端的会话密钥，**内存压力也会越大**。
* 网站服务一般是由**多台服务器**通过负载均衡提供服务的，**客户端再次连接不一定会命中上次访问过的服务器**

**2. Session Ticket**

解决 Session ID 的问题。

**服务器不再缓存每个客户端的会话密钥，而是把缓存的工作交给了客户端**

Session ID 和 Session Ticket **都不具备前向安全性**，容易遭受重放攻击**。**避免重放攻击的方式就是需要**对会话密钥设定一个合理的过期时间**

3. **Pre-shared Key**

相当于TLS1.3中的Ticket，在重连时，客户端会把 Ticket 和 HTTP 请求一同发送给服务端

## HTTP/1.1、HTTP/2、HTTP/3 演变

### HTTP/1.1 相比 HTTP/1.0 提高了什么性能？

HTTP/1.1 相比 HTTP/1.0 性能上的改进：

* 使用长连接的方式改善了 HTTP/1.0 短连接造成的性能开销。
* 支持管道网络传输（解决请求队头阻塞）

HTTP/1.1 的性能瓶颈：

* 请求 / 响应头部未经压缩就发送，首部信息越多延迟越大。只能压缩 Body 的部分；
* 每次互相发送**相同的**首部造成的浪费较多
* 响应队头阻塞
* 没有请求优先级控制
* 请求只能从客户端开始，服务器只能被动响应

### HTTP/2 做了什么优化？

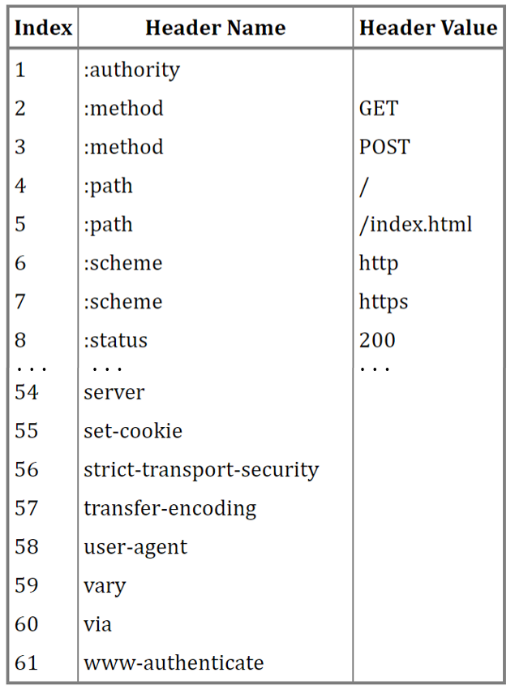
HTTP/2 协议是基于 HTTPS

HTTP/2 相比 HTTP/1.1 性能上的改进：

1. 头部压缩

如果你同时发出多个请求，他们的头是一样的或是相似的，那么协议会帮你**压缩重复的部分**。

HPACK 算法：在客户端和服务器同时维护一张**头信息表**，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送**索引号**，这样就**提高速度**了。



**静态表编码**

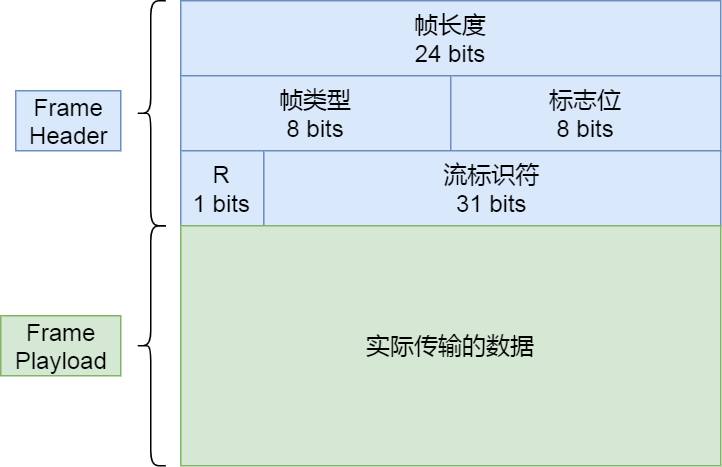
HTTP/2 为高频出现在头部的字符串和字段建立了一张**静态表，从1-61**

**动态表编码**

不在静态表范围内的头部字符串就要自行构建**动态表**，它的 Index 从 62 起步，会在编码解码的时候随时更新。前提：**必须同一个连接上，重复传输完全相同的 HTTP 头部**

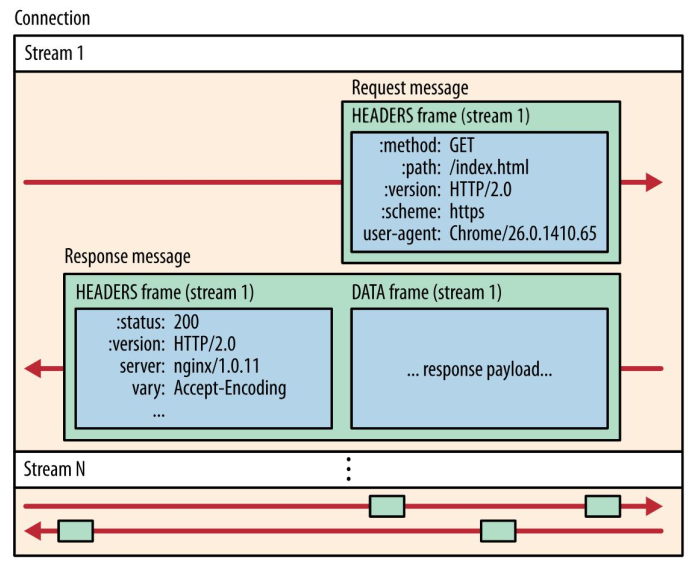
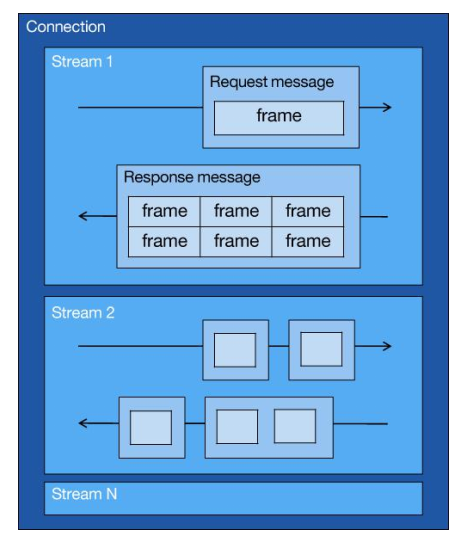
2. 二进制格式

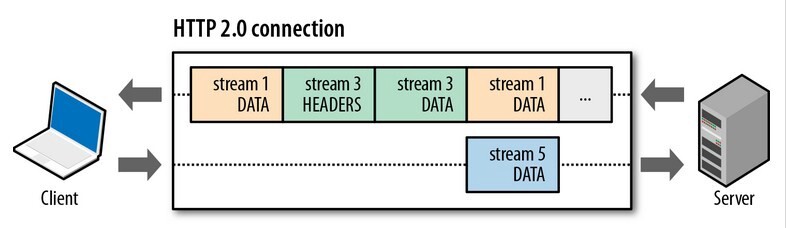
头信息和数据体都是二进制，并且统称为帧（frame）：**头信息帧（Headers Frame）和数据帧（Data Frame）**



3. 并发传输

引出了 Stream 概念，多个 Stream 复用在一条 TCP 连接. 同一个 HTTP 请求与响应是跑在同一个 Stream 中，HTTP 消息可以由多个 Frame 构成， 一个 Frame 可以由多个 TCP 报文构成。





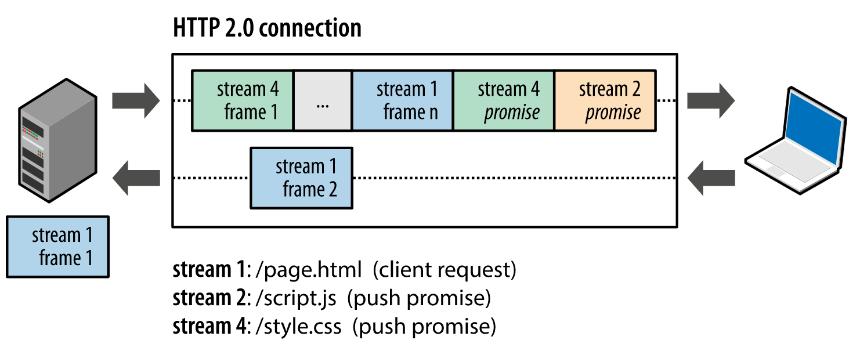
**针对不同的 HTTP 请求用独一无二的 Stream ID 来区分，接收端可以通过 Stream ID 有序组装成 HTTP 消息，不同 Stream 的帧是可以乱序发送的，因此可以并发不同的 Stream ，也就是 HTTP/2 可以并行交错地发送请求和响应**。**同一 Stream 内部的帧必须是严格有序的**。

**当 HTTP/2 实现 100 个并发 Stream 时，只需要建立一次 TCP 连接，而 HTTP/1.1 需要建立 100 个 TCP 连接**

还可以对每个 Stream 设置不同**优先级，让服务器先传递优先级高的**

4、服务器推送

改善了传统的「请求-应答」工作模式，服务端不再是被动地响应，可以**主动**向客户端发送消息（js、css等文件）

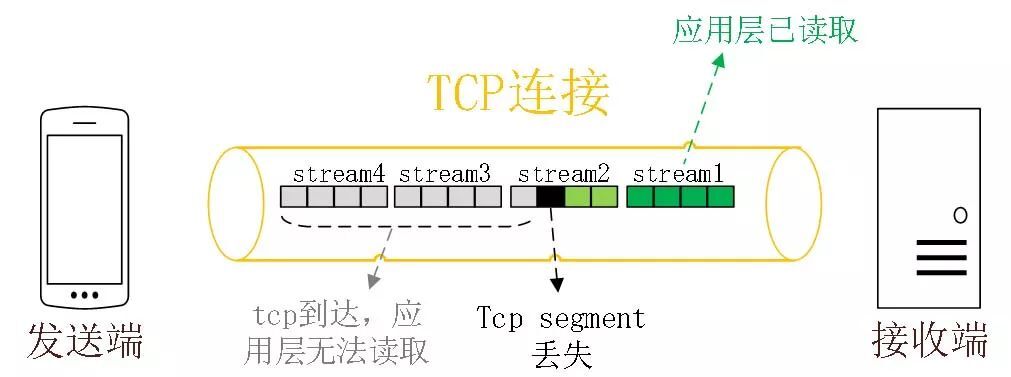


Stream 1 是客户端向服务端请求的资源，属于客户端建立的 Stream，所以该 Stream 的 ID 是奇数（数字 1）；Stream 2 和 4 都是服务端主动向客户端推送的资源，属于服务端建立的 Stream，所以这两个 Stream 的 ID 是偶数（数字 2 和 4）

HTTP/2 有什么缺陷？

 HTTP/2 还是存在“队头阻塞”的问题，只不过问题不在 HTTP 这一层面，而在 **TCP** 这一层

**TCP 层必须保证收到的字节数据是完整且连续的，这样内核才会将缓冲区里的数据返回给 HTTP 应用，那么当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据，这就是 HTTP/2 队头阻塞问题。**



一旦发生了丢包现象，就会触发 TCP 的重传机制，这样在**一个 TCP 连接**中的**所有的 HTTP 请求都必须等待这个丢了的包被重传回来**

### HTTP/3 做了哪些优化？

HTTP/1.1： HTTP 层队头阻塞

HTTP/2： TCP 层队头阻塞

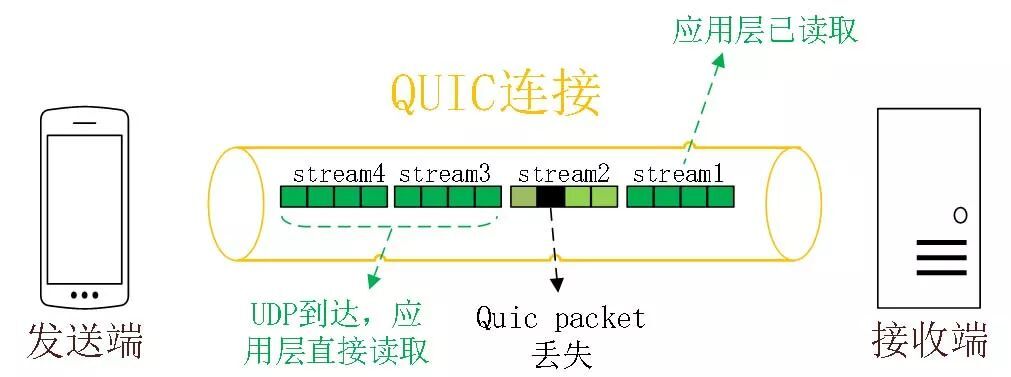
**HTTP/3 把 HTTP 下层的 TCP 协议改成了 UDP**

UDP 发送是不管顺序，也不管丢包的。为了保证可靠性，基于 UDP 协议在「应用层」实现了**QUIC 协议**，它具有类似 TCP 的连接管理、拥塞窗口、流量控制的网络特性，相当于将不可靠传输的 UDP 协议变成“可靠”的了。

**QUIC 协议的特点：**

1、无队头阻塞

也有类似 HTTP/2 Stream 与多路复用的概念。**当某个流发生丢包时，只会阻塞这个流，其他流不会受到影响，因此不存在队头阻塞问题**。



2、更快的连接建立

HTTP/1 和 HTTP/2：分批次来握手，先 TCP 握手（传输层），再 TLS 握手（表示层）。

HTTP/3：QUIC 内部包含了 TLS

3、连接迁移

基于 TCP 传输协议的 HTTP 协议，由于是通过四元组（源 IP、源端口、目的 IP、目的端口）确定一条 TCP 连接。



**当IP 地址变化了，那么就必须要断开连接，然后重新建立连接（重新握手等等）**。

而 QUIC 协议没有用四元组的方式来“绑定”连接，而是通过**连接 ID** 来标记通信的两个端点，客户端和服务器**可以各自选择一组 ID 来标记自己**，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息（比如连接 ID、TLS 密钥等），就可以“无缝”地复用原连接，消除重连的成本，没有丝毫卡顿感，达到了**连接迁移**的功能。

# 3.2 HTTP/1.1 如何优化？

1.尽量避免发送 HTTP 请求

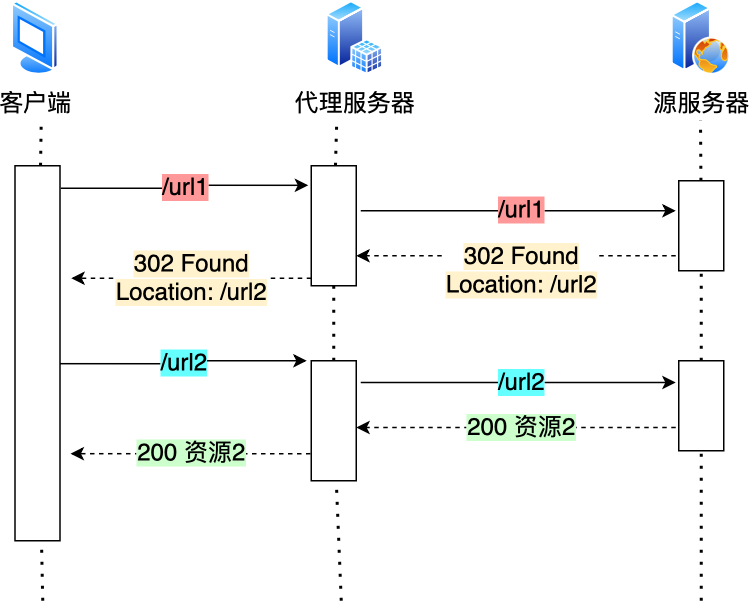
通过使用**缓存技术**

2.在需要发送 HTTP 请求时，考虑如何减少请求次数

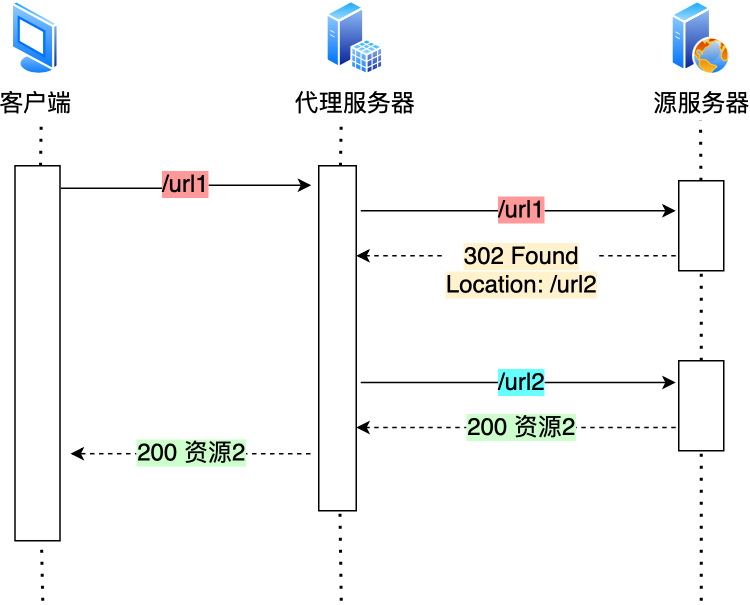
2.1减少重定向请求次数

当一个资源从 url1 移至 url2 后，客户端不知情，它还是继续请求 url1，这时服务器通过 302 响应码和 Location 头部，告诉客户端该资源已经迁移至 url2 了，于是客户端需要再发送 url2 请求以获得服务器的资源。

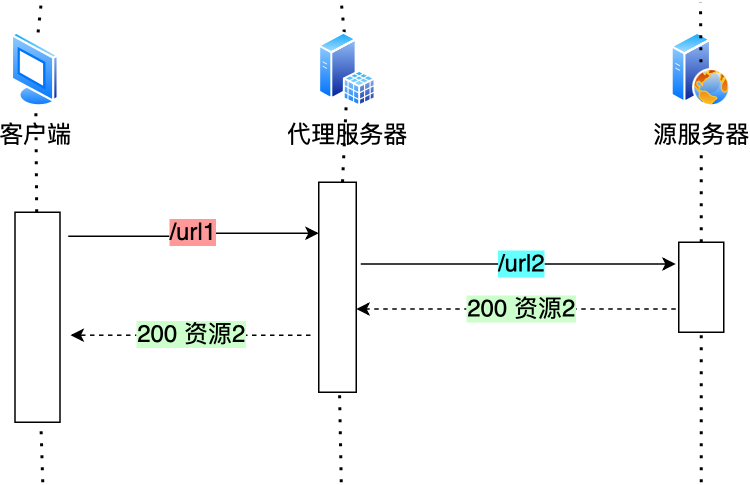
如果源服务器上一级是代理服务器，然后代理服务器才与客户端通信，这时客户端重定向就会导致客户端与代理服务器之间需要 2 次消息传递



如果**重定向的工作交由代理服务器完成，就能减少 HTTP 请求次数了**



当代理服务器知晓了**重定向规则**后，可以**进一步减少消息传递次数**



2.2合并请求

为了防止单个请求的阻塞，所以**一般浏览器会同时发起 5-6 个请求，每一个请求都是不同的 TCP 连接**，那么如果合并了请求，也就会**减少 TCP 连接的数量，因而省去了 TCP 握手和慢启动过程耗费的时间**。

**合并请求的方式就是合并资源，以一个大资源的请求替换多个小资源的请求：**

① 使用 CSS Image Sprites 技术把多个小图片合成一个大图片

② 使用 webpack 等打包工具将 js、css 等资源合并打包成大文件

③ 将图片的二进制数据用 base64 编码后，以URL的形式嵌入到HTML文件

2.3*延迟发送请求*

HTML 里会含有很多 HTTP 的 URL，当前不需要的资源没必要获取过来。请求网页的时候，只获取当前用户所看到的页面资源，当用户向下滑动页面的时候，再向服务器获取接下来的资源，这样就达到了延迟发送请求的效果。

3.减少服务器的 HTTP 响应的数据大小

考虑对响应的资源进行**压缩**

3.1无损压缩

无损压缩是指资源经过压缩后，信息不被破坏，还能**完全恢复**到压缩前的原样，适合用在**文本文件**、**程序可执行文件**、**程序源代码**。

生成二进制比特序列一般是「**霍夫曼编码**」算法

3.2有损压缩

经过此方法压缩，解压的数据会与原始数据不同但是非常接近，经常用于压缩**多媒体数据**，比如**音频**、**视频、图片**。

压缩比较高的是 Google 推出的 **WebP 格式**

## HTTP 和 RPC

**RPC**（**R**emote **P**rocedure **C**all），又叫做**远程过程调用**。它本身并不是一个具体的协议，而是一种**调用方式**。多年以前，**HTTP 主要用于 B/S 架构，而 RPC 更多用于 C/S 架构。但现在其实已经没分那么清了，B/S 和 C/S 在慢慢融合。**

### 服务发现

即知道**IP 地址和端口**

**HTTP** : **DNS 服务**

**RPC** : **中间服务(Consul 或者 Etcd，甚至是 Redis)**

### 底层连接形式

**HTTP/1.1** ：长连接

**RPC：**长连接+**连接池（**建立多条连接放在池内，要发数据的时候就从池里取一条连接出来，**用完放回去，下次再复用）**

### 传输的内容

**HTTP/1.1：Json** 来**序列化**结构体数据。内容非常多的**冗余**，显得**非常啰嗦**

**RPC：**定制化程度更高，采用体积更小的 Protobuf 或其他序列化协议去保存结构体数据，**性能也会更好**

## HTTP和WebSocket

TCP 连接的两端，同一时间里，双方都可以主动向对方发送数据。这就是所谓的全双工

HTTP/1.1: **同一时间里**，客户端和服务器**只能有一方主动**发数据，这就是所谓的**半双工。**对于登录页面这样的简单场景，可以使用**定时轮询或者长轮询**的方式实现**服务器推送**

WebSocket: 完美继承TCP 协议的**全双工**能力，并且提供了解决粘包的方案。适用于**需要服务器和客户端（浏览器）频繁交互**的大部分场景

解决粘包：HTTP：使用特殊字符作为边界

WebSocket：自定义消息结构

连接方式： WebSocket 会先利用HTTP协议加上一些特殊的 header 头进行握手升级操作，升级成功后就跟 HTTP 没有任何关系了，之后就用 WebSocket 的数据格式进行收发数据。