1.详解HTTP报文以及HTTPS的加密过程。

HTTP/HTTPS协议的工作（核心传输）：

应用层生成HTTP请求消息：

GET / HTTP/1.1 // 方法：获取首页

Host: www.example.com // 主机名

User-Agent: Chrome/120.0 // 浏览器信息

（HTTPS会先握手加密密钥，确保数据安全。）

往下传给传输层（TCP），TCP加段头、建立连接。

服务器收到后，应用层解析请求，生成响应：

HTTP/1.1 200 OK // 成功

Content-Type: text/html // 类型：网页

<html><body>欢迎！</body></html> // 实际内容

浏览器应用层收到，解释成网页，渲染给你看。

另一个例子：发微信消息。

应用层用自定义协议（微信的私有协议，基于HTTP）包装消息：“文本+表情+时间戳”。

加用户ID、群ID等元数据。

传给TCP，确保不丢；服务器应用层解析，推给接收者。

做了什么事总结：

格式化与验证：确保数据结构正确（e.g., JSON/XML格式）。

会话管理：跟踪对话（e.g., HTTP的Cookie记住登录状态）。

安全性：集成加密（HTTPS）、认证（OAuth）。

多媒体处理：支持图片、视频的编码（MIME类型）。

HTTP/HTTPS报文内容及含义：

C:\Users\SHE.F>curl -v http://www.baidu.com

\* Host www.baidu.com:80 was resolved.

\* IPv6: (none)

\* IPv4: 39.156.70.239, 39.156.70.46

\* Trying 39.156.70.239:80...

\* Connected to www.baidu.com (39.156.70.239) port 80

\* using HTTP/1.x

> GET / HTTP/1.1

> Host: www.baidu.com

> User-Agent: curl/8.13.0

> Accept: \*/\*

>

\* Request completely sent off

< HTTP/1.1 200 OK

< Accept-Ranges: bytes

< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform

< Connection: keep-alive

< Content-Length: 2381

< Content-Type: text/html

< Date: Sat, 11 Oct 2025 06:11:09 GMT

< Etag: "588604c1-94d"

< Last-Modified: Mon, 23 Jan 2017 13:27:29 GMT

< Pragma: no-cache

< Server: bfe/1.0.8.18

< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/

< Tr\_id: bfe\_10453437151549512691

<

<!DOCTYPE html>

<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html;charset=utf-8><meta http-equiv=X-UA-Compatible content=IE=Edge><meta content=always name=referrer><link rel=stylesheet type=text/css href=http://s1.bdstatic.com/r/www/cache/bdorz/baidu.min.css><title>百度一下，你就知道</title></head> <body link=#0000cc> <div id=wrapper> <div id=head> <div class=head\_wrapper> <div class=s\_form> <div class=s\_form\_wrapper> <div id=lg> <img hidefocus=true src=//www.baidu.com/img/bd\_logo1.png width=270 height=129> </div> <form id=form name=f action=//www.baidu.com/s class=fm> <input type=hidden name=bdorz\_come value=1> <input type=hidden name=ie value=utf-8> <input type=hidden name=f value=8> <input type=hidden name=rsv\_bp value=1> <input type=hidden name=rsv\_idx value=1> <input type=hidden name=tn value=baidu><span class="bg s\_ipt\_wr"><input id=kw name=wd class=s\_ipt value maxlength=255 autocomplete=off autofocus></span><span class="bg s\_btn\_wr"><input type=submit id=su value=百度一下 class="bg s\_btn"></span> </form> </div> </div> <div id=u1> <a href=http://news.baidu.com name=tj\_trnews class=mnav>新闻</a> <a href=http://www.hao123.com name=tj\_trhao123 class=mnav>hao123</a> <a href=http://map.baidu.com name=tj\_trmap class=mnav>地图</a> <a href=http://v.baidu.com name=tj\_trvideo class=mnav>视频</a> <a href=http://tieba.baidu.com name=tj\_trtieba class=mnav>贴吧</a> <noscript> <a href=http://www.baidu.com/bdorz/login.gif?login&amp;tpl=mn&amp;u=http%3A%2F%2Fwww.baidu.com%2f%3fbdorz\_come%3d1 name=tj\_login class=lb>登录</a> </noscript> <script>document.write('<a href="http://www.baidu.com/bdorz/login.gif?login&tpl=mn&u='+ encodeURIComponent(window.location.href+ (window.location.search === "" ? "?" : "&")+ "bdorz\_come=1")+ '" name="tj\_login" class="lb">登录</a>');</script> <a href=//www.baidu.com/more/ name=tj\_briicon class=bri style="display: block;">更多产品</a> </div> </div> </div> <div id=ftCon> <div id=ftConw> <p id=lh> <a href=http://home.baidu.com>关于百度</a> <a href=http://ir.baidu.com>About Baidu</a> </p> <p id=cp>&copy;2017&nbsp;Baidu&nbsp;<a href=http://www.baidu.com/duty/>使用百度前必读</a>&nbsp; <a href=http://jianyi.baidu.com/ class=cp-feedback>意见反馈</a>&nbsp;京ICP证030173号&nbsp; <img src=//www.baidu.com/img/gs.gif> </p> </div> </div> </div> </body> </html>

\* Connection #0 to host www.baidu.com left intact

在报文前，有curl的日志：

\* Host www.baidu.com:80 was resolved.

\* IPv6: (none)

\* IPv4: 39.156.70.239, 39.156.70.46

\* Trying 39.156.70.239:80...

\* Connected to www.baidu.com (39.156.70.239) port 80

\* using HTTP/1.x

第一部分：前置信息（连接建立）

\* Host www.baidu.com:80 was resolved.

\* IPv6: (none)

\* IPv4: 39.156.70.239, 39.156.70.46

\* Trying 39.156.70.239:80...

\* Connected to www.baidu.com (39.156.70.239) port 80

\* using HTTP/1.x

DNS解析：curl先查www.baidu.com的IP（IPv4有俩备选，用了第一个）。这是应用层的DNS协议在后台干的活（我们之前聊过）。

TCP连接：用80端口（HTTP默认）建立连接。这是传输层（TCP三握手）的结果。

HTTP/1.x：确认用1.x版本（1.1）。

第二部分：HTTP请求报文（Request Message）

> GET / HTTP/1.1

> Host: www.baidu.com

> User-Agent: curl/8.13.0

> Accept: \*/\*

>

| **字段** | **内容** | **解释（是什么、为什么、怎么用）** |
| --- | --- | --- |
| **请求行** | GET / HTTP/1.1 | **是什么**：报文的核心指令，格式<方法> <URI> <版本>。 **为什么**：定义动作和目标。 **怎么用**：GET=获取首页（/是根路径）；HTTP/1.1=支持持久连接。百度据此知道“给我首页内容”。 |
| **Host** | [www.baidu.com](http://www.baidu.com/) | **是什么**：主机头。 **为什么**：HTTP/1.1必填，指定域名（虚拟主机）。 **怎么用**：服务器用它路由到正确站点（百度有海量子域）。缺了会400错误。 |
| **User-Agent** | curl/8.13.0 | **是什么**：客户端标识。 **为什么**：服务器优化响应（e.g., curl是命令行，百度可能不加JS）。 **怎么用**：自动生成；浏览器会是“Mozilla/5.0...”长串。用于统计/防爬虫。 |
| **Accept** | / | **是什么**：接受的内容类型。 **为什么**：告诉服务器“我啥都行”。 **怎么用**：MIME通配；浏览器通常指定text/html优先。这里简单，百度回text/html。 |

第三部分：HTTP响应报文（Response Message）

< HTTP/1.1 200 OK

< Accept-Ranges: bytes

< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform

< Connection: keep-alive

< Content-Length: 2381

< Content-Type: text/html

< Date: Sat, 11 Oct 2025 06:11:09 GMT

< Etag: "588604c1-94d"

< Last-Modified: Mon, 23 Jan 2017 13:27:29 GMT

< Pragma: no-cache

< Server: bfe/1.0.8.18

< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/

< Tr\_id: bfe\_10453437151549512691

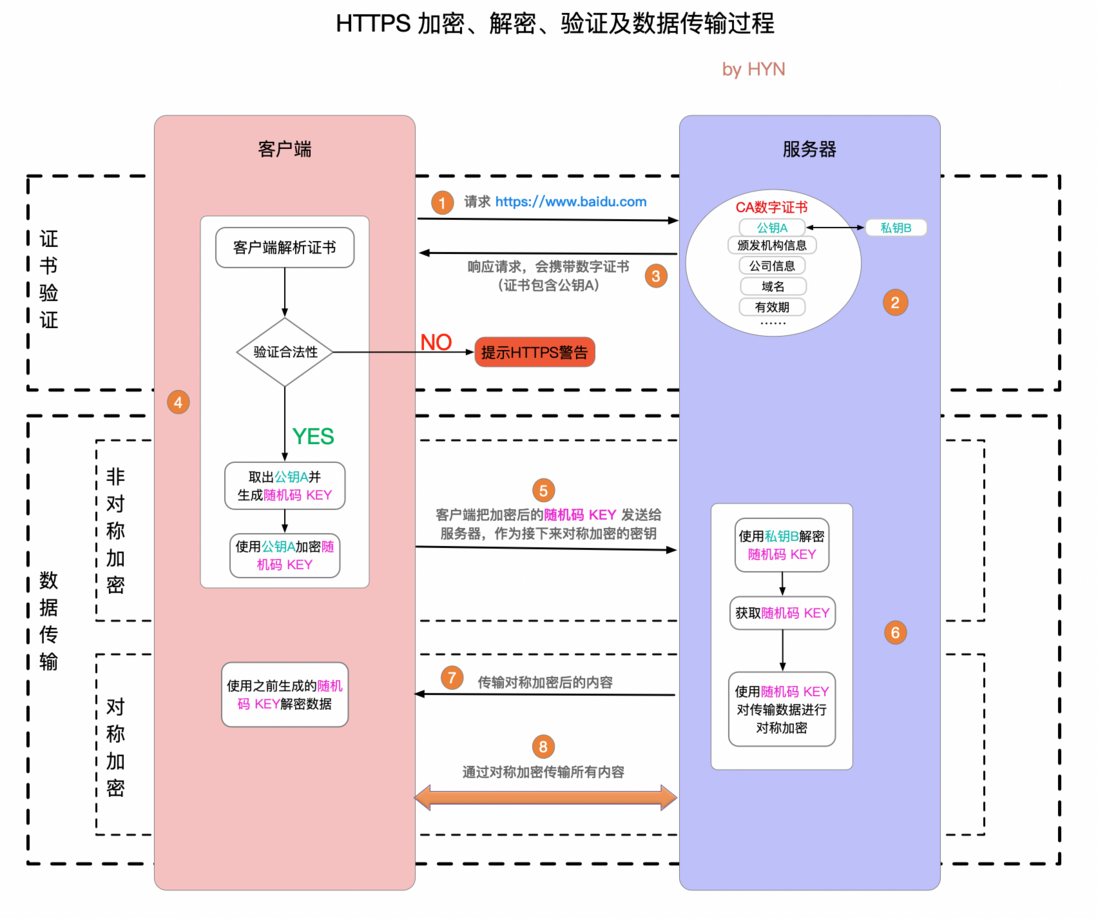
<

| **字段** | **内容** | **解释（是什么、为什么、怎么用）** |
| --- | --- | --- |
| **状态行** | HTTP/1.1 200 OK | **是什么**：响应标题，格式<版本> <状态码> <原因>。 **为什么**：反馈结果。 **怎么用**：200=成功（2xx系列）；OK=人读提示。百度说“一切OK，给你首页”。 |
| **Accept-Ranges** | bytes | **是什么**：支持范围请求。 **为什么**：允许部分下载。 **怎么用**：浏览器用它续传大文件（e.g., 视频从中间开始）。 |
| **Cache-Control** | private, no-cache, no-store, proxy-revalidate, no-transform | **是什么**：缓存指令。 **为什么**：控制浏览器/代理怎么存页面。 **怎么用**：private=只本地缓存；no-cache=每次查新鲜；no-store=别存盘。百度动态页，防旧数据。 |
| **Connection** | keep-alive | **是什么**：连接管理。 **为什么**：保持TCP连接复用。 **怎么用**：HTTP/1.1默认；下一个请求不用重连，省时。 |
| **Content-Length** | 2381 | **是什么**：体字节数。 **为什么**：告诉客户端啥时结束。 **怎么用**：curl据此读2381字节HTML。错位会乱码。 |
| **Content-Type** | text/html | **是什么**：体MIME类型。 **为什么**：指导渲染。 **怎么用**：浏览器解析为HTML；可加charset=utf-8（这里隐含）。 |
| **Date** | Sat, 11 Oct 2025 06:11:09 GMT | **是什么**：响应时间（GMT）。 **为什么**：日志/缓存基准。 **怎么用**：客户端本地时间对比，算延迟。 |
| **Etag** | "588604c1-94d" | **是什么**：资源指纹（哈希）。 **为什么**：精确缓存验证。 **怎么用**：下次请求If-None-Match: 这个值，服务器比对变化。比Last-Modified准。 |
| **Last-Modified** | Mon, 23 Jan 2017 13:27:29 GMT | **是什么**：最后修改时间。 **为什么**：简单缓存检查。 **怎么用**：客户端If-Modified-Since: 这个时间，服务器回304 Not Modified（不重发体）。 |
| **Pragma** | no-cache | **是什么**：旧版缓存指令（HTTP/1.0兼容）。 **为什么**：强制新鲜。 **怎么用**：和Cache-Control类似，防老浏览器缓存。 |
| **Server** | bfe/1.0.8.18 | **是什么**：服务器软件。 **为什么**：调试/指纹。 **怎么用**：Baidu Front End（百度的边缘服务器）；开发者据此优化。 |
| **Set-Cookie** | BDORZ=27315; max-age=86400; domain=.baidu.com; path=/ | **是什么**：设置Cookie。 **为什么**：跟踪会话（e.g., 防刷）。 **怎么用**：BDORZ=随机值，存1天（86400秒），全域有效。下次请求自动带Cookie: BDORZ=...。 |
| **Tr\_id** | bfe\_10453437151549512691 | **是什么**：百度私有跟踪ID。 **为什么**：内部日志/调试。 **怎么用**：追踪这个请求的流转，非标准头。 |

第四部分：响应体（Body）

HTTP变成HTTPS是如何加密的？解释一下其详细过程

HTTPS 的整个通信过程可以分为两大阶段：证书验证和数据传输阶段，数据传输阶段又可以分为非对称加密和对称加密两个阶段。具体流程按图中的序号讲解。



1. 客户端请求 HTTPS 网址，然后连接到 server 的 443 端口 (HTTPS 默认端口，类似于 HTTP 的80端口)。
2. 采用 HTTPS 协议的服务器必须要有一套数字 CA (Certification Authority)证书，证书是需要申请的，并由专门的数字证书认证机构(CA)通过非常严格的审核之后颁发的电子证书 (当然了是要钱的，安全级别越高价格越贵)。颁发证书的同时会产生一个私钥和公钥。私钥由服务端自己保存，不可泄漏。公钥则是附带在证书的信息中，可以公开的。证书本身也附带一个证书电子签名，这个签名用来验证证书的完整性和真实性，可以防止证书被篡改。
3. 服务器响应客户端请求，将证书传递给客户端，证书包含公钥和大量其他信息，比如证书颁发机构信息，公司信息和证书有效期等。Chrome 浏览器点击地址栏的锁标志再点击证书就可以看到证书详细信息。



1. 客户端解析证书并对其进行验证。如果证书不是可信机构颁布，或者证书中的域名与实际域名不一致，或者证书已经过期，就会向访问者显示一个警告，由其选择是否还要继续通信。就像下面这样：



如果证书没有问题，客户端就会从服务器证书中取出服务器的公钥A。然后客户端还会生成一个随机码 KEY，并使用公钥A将其加密。

5.客户端把加密后的随机码 KEY 发送给服务器，作为后面对称加密的密钥。

6.服务器在收到随机码 KEY 之后会使用私钥B将其解密。经过以上这些步骤，客户端和服务器终于建立了安全连接，完美解决了对称加密的密钥泄露问题，接下来就可以用对称加密愉快地进行通信了。

7.服务器使用密钥 (随机码 KEY)对数据进行对称加密并发送给客户端，客户端使用相同的密钥 (随机码 KEY)解密数据。

1. 双方使用对称加密愉快地传输所有数据。

HTTPS 和 HTTP 的区别：

最最重要的区别就是安全性，HTTP 明文传输，不对数据进行加密安全性较差。HTTPS (HTTP + SSL / TLS)的数据传输过程是加密的，安全性较好。

使用 HTTPS 协议需要申请 CA 证书，一般免费证书较少，因而需要一定费用。证书颁发机构如：Symantec、Comodo、DigiCert 和 GlobalSign 等。

HTTP 页面响应速度比 HTTPS 快，这个很好理解，由于加了一层安全层，建立连接的过程更复杂，也要交换更多的数据，难免影响速度。

由于 HTTPS 是建构在 SSL / TLS 之上的 HTTP 协议，所以，要比 HTTP 更耗费服务器资源。

HTTPS 和 HTTP 使用的是完全不同的连接方式，用的端口也不一样，前者是 443，后者是 80。

HTTPS 的缺点：

在相同网络环境中，HTTPS 相比 HTTP 无论是响应时间还是耗电量都有大幅度上升。

HTTPS 的安全是有范围的，在黑客攻击、服务器劫持等情况下几乎起不到作用。

在现有的证书机制下，中间人攻击依然有可能发生。

HTTPS 需要更多的服务器资源，也会导致成本的升高。

1、SSL / TLS 以及 SSL / TLS 握手的概念

SSL 和 TLS 协议可以为通信双方提供识别和认证通道，从而保证通信的机密性和数据完整性。TLS 协议是从Netscape SSL 3.0协议演变而来的，不过这两种协议并不兼容，SSL 已经逐渐被 TLS 取代，所以下文就以 TLS 指代安全层。 TLS 握手是启动 HTTPS 通信的过程，类似于 TCP 建立连接时的三次握手。 在 TLS 握手的过程中，通信双方交换消息以相互验证，相互确认，并确立它们所要使用的加密算法以及会话密钥 (用于对称加密的密钥)。可以说，TLS 握手是 HTTPS 通信的基础部分。

2、TLS 握手过程中发生了什么

我们已经知道 TLS 握手的目的是建立安全连接，那么通信双方在这个过程中究竟干了什么呢？下面就是答案：

商定双方通信所使用的的 TLS 版本 (例如 TLS1.0, 1.2, 1.3等等)；

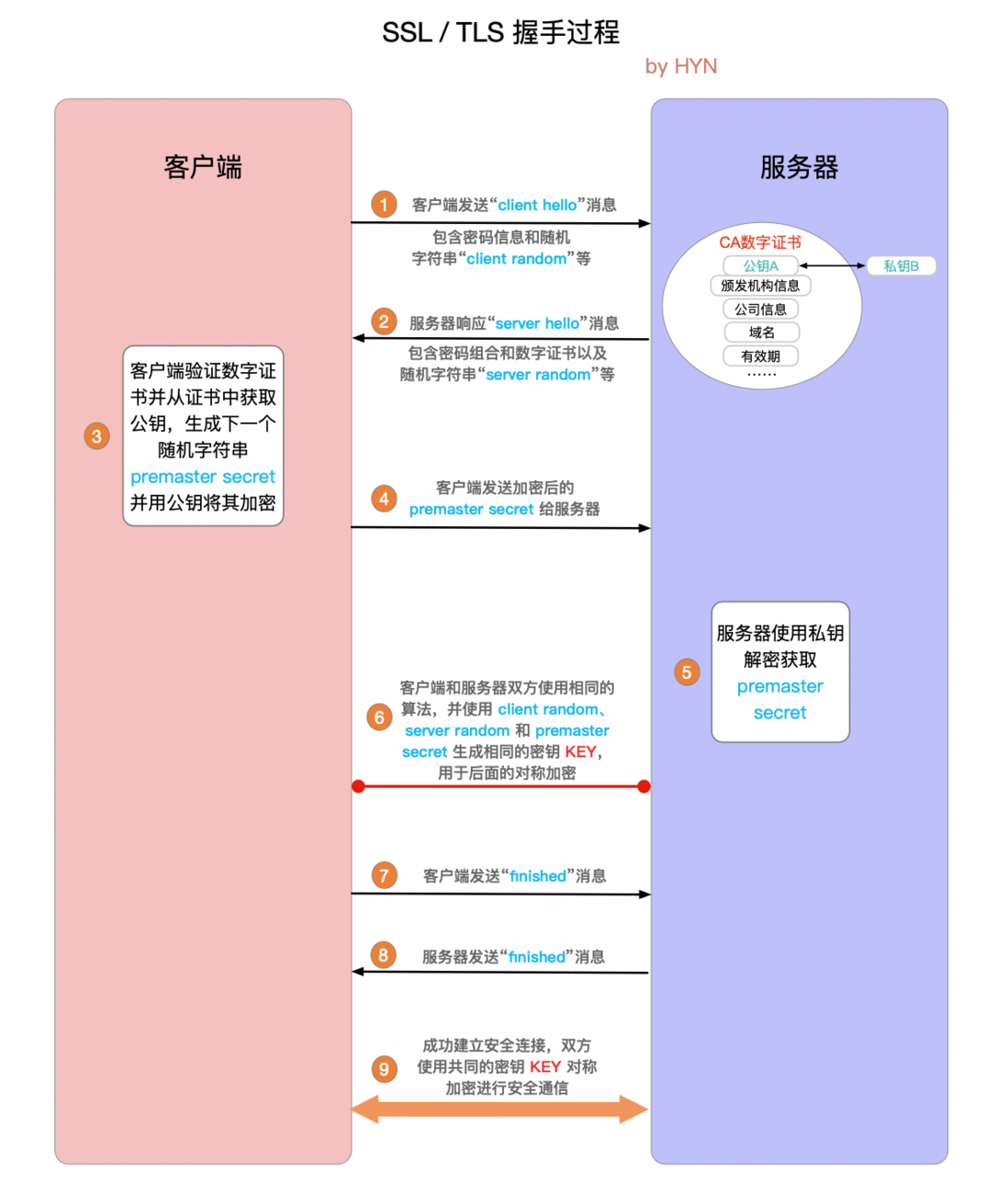
确定双方所要使用的密码组合；

客户端通过服务器的公钥和数字证书 (上篇文章已有介绍)上的数字签名验证服务端的身份；

生成会话密钥，该密钥将用于握手结束后的对称加密。

3、TLS 握手详细过程

下面来看 TLS 握手的详细过程 (注：此图与HTTPS详解一中的 HTTPS 原理图的流程大致相同，不同的是此图把重点放在了TLS握手的相关概念上)：



SSL / TLS 握手详细过程

1. ****"client hello"消息：****客户端通过发送"client hello"消息向服务器发起握手请求，该消息包含了客户端所支持的 TLS 版本和密码组合以供服务器进行选择，还有一个"client random"随机字符串。
2. ****"server hello"消息：****服务器发送"server hello"消息对客户端进行回应，该消息包含了数字证书，服务器选择的密码组合和"server random"随机字符串。

****验证：****客户端对服务器发来的证书进行验证，确保对方的合法身份，验证过程可以细化为以下几个步骤：

* 1. 检查数字签名
  2. 验证证书链 (这个概念下面会进行说明)
  3. 检查证书的有效期
  4. 检查证书的撤回状态 (撤回代表证书已失效)

1. ****"premaster secret"字符串：****客户端向服务器发送另一个随机字符串"premaster secret (预主密钥)"，这个字符串是经过服务器的公钥加密过的，只有对应的私钥才能解密。
2. ****使用私钥：****服务器使用私钥解密"premaster secret"。
3. ****生成共享密钥****：客户端和服务器均使用 client random，server random 和 premaster secret，并通过相同的算法生成相同的共享密钥 ****KEY****。
4. ****客户端就绪：****客户端发送经过共享密钥 ****KEY****加密过的"finished"信号。
5. ****服务器就绪：****服务器发送经过共享密钥 ****KEY****加密过的"finished"信号。
6. ****达成安全通信：****握手完成，双方使用对称加密进行安全通信。

4、TLS 握手过程中的一些重要概念

数字证书 (digital certificate)：在非对称加密通信过程中，服务器需要将公钥发送给客户端，在这一过程中，公钥很可能会被第三方拦截并替换，然后这个第三方就可以冒充服务器与客户端进行通信，这就是传说中的“中间人攻击”(man in the middle attack)。解决此问题的方法是通过受信任的第三方交换公钥，具体做法就是服务器不直接向客户端发送公钥，而是要求受信任的第三方，也就是证书认证机构 (Certificate Authority, 简称 CA)将公钥合并到数字证书中，然后服务器会把公钥连同证书一起发送给客户端，私钥则由服务器自己保存以确保安全。数字证书一般包含以下内容：

证书所有者的公钥

证书所有者的专有名称

证书颁发机构的专有名称

证书的有效起始日期

证书的过期日期

证书数据格式的版本号

序列号，这是证书颁发机构为该证书分配的唯一标识符

... ...

数字签名 (digital signature)：这个概念很好理解，其实跟人的手写签名类似，是为了确保数据发送者的合法身份，也可以确保数据内容未遭到篡改，保证数据完整性。与手写签名不同的是，数字签名会随着文本数据的变化而变化。具体到数字证书的应用场景，数字签名的生成和验证流程如下：

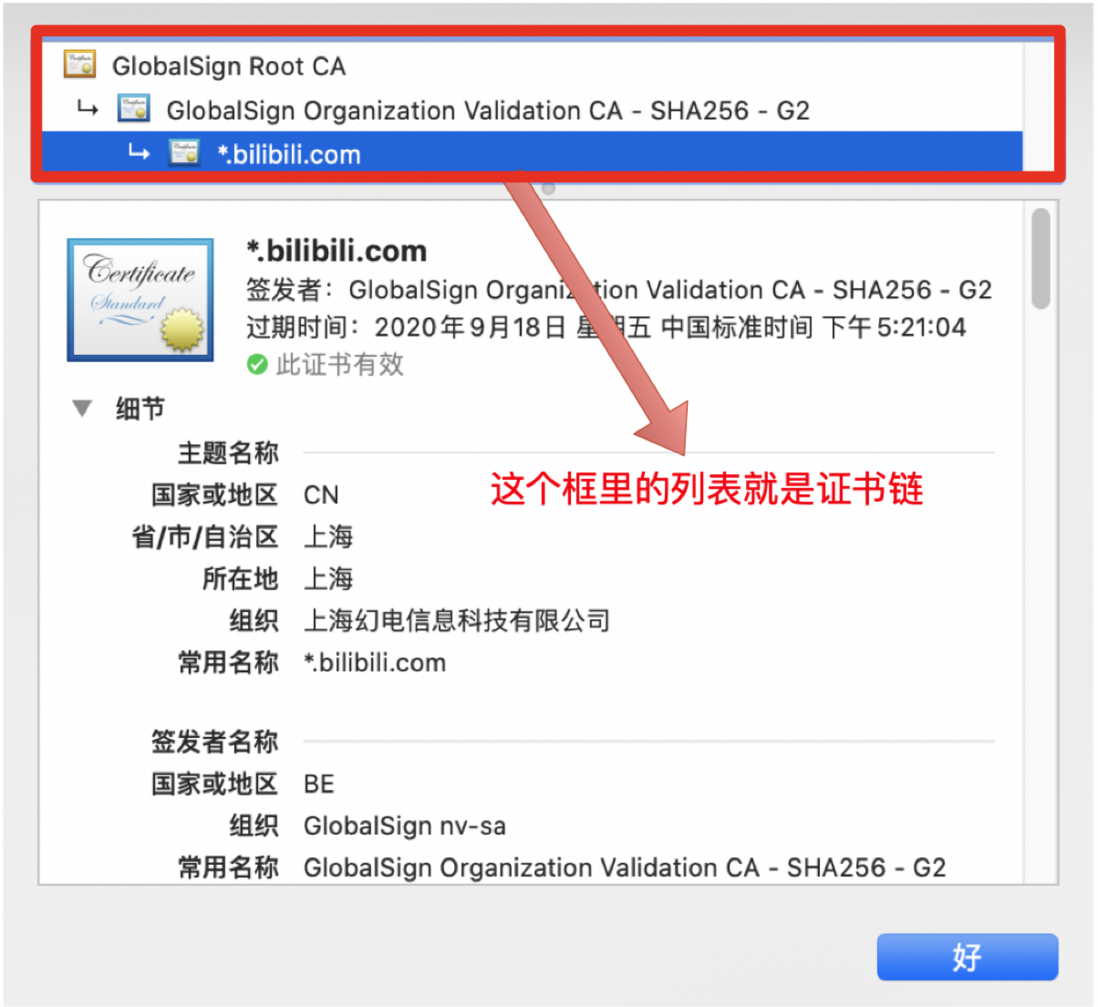
服务器对证书内容进行信息摘要计算 (常用算法有 SHA-256等)，得到摘要信息，再用私钥把摘要信息加密，就得到了数字签名

服务器把数字证书连同数字签名一起发送给客户端

客户端用公钥解密数字签名，得到摘要信息

客户端用相同的信息摘要算法重新计算证书摘要信息，然后对这两个摘要信息进行比对，如果相同，则说明证书未被篡改，否则证书验证失败

证书链 (certificate chain)：证书链，也称为证书路径，是用于认证实体合法身份的证书列表，具体到 HTTPS 通信中，就是为了验证服务器的合法身份。之所以使用证书链，是为了保证根证书 (root CA certificate)的安全，中间层可以看做根证书的代理，起到了缓冲的作用，如下图所示，这里还以 B 站证书为例：



证书链

证书链从根证书开始，并且证书链中的每一级证书所标识的实体都要为其下一级证书签名，而根证书自身则由证书颁发机构签名。客户端在验证证书链时，必须对链中所有证书的数字签名进行验证，直到达到根证书为止。

4.****密码规范和密码组合 (CipherSpecs 和 CipherSuites)：****通信双方在安全连接中所使用的算法必须符合密码安全协议的规定，CipherSpecs 和 CipherSuites 正好定义了合法的密码算法组合。CipherSpecs 用于认证加密算法和信息摘要算法的组合，通信双方必须同意这个密码规范才能进行通信。而 CipherSuites 则定义了 SSL / TLS 安全连接中所使用的加密算法的组合，该组合包含三种不同的算法：

1. 握手期间所使用的的密钥交换和认证算法 (最常用的是 RSA 算法)
2. 加密算法 (用于握手完成后的对称加密，常用的有 AES、3DES等)
3. 信息摘要算法 (常用的有 SHA-256、SHA-1 和 MD5 等)
4. 解释DNS域名解析的整个过程。

什么是DNS域名解析

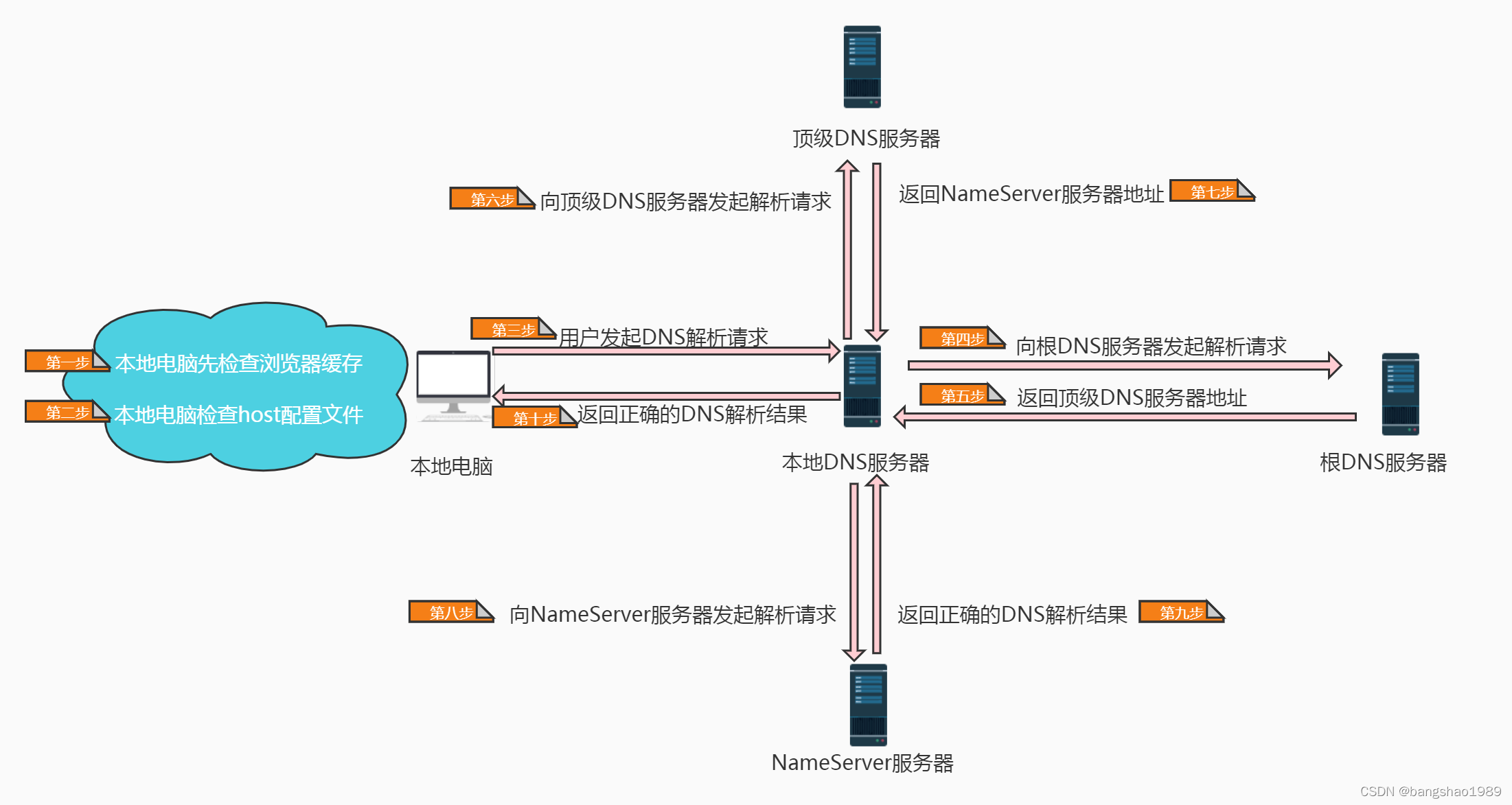
我们首先要了解域名和IP地址的区别。IP地址是互联网上计算机唯一的逻辑地址，通过IP地址实现不同计算机之间的相互通信，每台联网计算机都需要通过IP地址来互相联系和分别。

但由于IP地址是由一串容易混淆的数字串构成，人们很难记忆所有计算机的IP地址，这样对于我们日常工作生活访问不同网站是很困难的。基于这种背景，人们在IP地址的基础上又发展出了一种更易识别的符号化标识，这种标识由人们自行选择的字母和数字构成，相比IP地址更易被识别和记忆，逐渐代替IP地址成为互联网用户进行访问互联的主要入口。这种符号化标识就是域名。

域名虽然更易被用户所接受和使用，但计算机只能识别纯数字构成的IP地址，不能直接读取域名。因此要想达到访问效果，就需要将域名翻译成IP地址。而DNS域名解析承担的就是这种翻译效果。

DNS域名解析过程

当我们在浏览器地址栏中输入www.baidu.com时，DNS解析将会有将近10个步骤，这个过程大体大体由一张图可以表示：



整个过程大体描述如下，其中前两个步骤是在本地电脑内完成的，后8个步骤涉及到真正的域名解析服务器：

第一步、

本地电脑会检查浏览器缓存中有没有这个域名对应的解析过的IP地址，如果缓存中有，这个解析过程就结束。浏览器缓存域名也是有限制的，不仅浏览器缓存大小有限制，而且缓存的时间也有限制，通常情况下为几分钟到几小时不等，域名被缓存的时间限制可以通过TTL属性来设置。这个缓存时间太长和太短都不太好，如果时间太长，一旦域名被解析到的IP有变化，会导致被客户端缓存的域名无法解析到变化后的IP地址，以致该域名不能正常解析，这段时间内有一部分用户无法访问网站。如果设置时间太短，会导致用户每次访问网站都要重新解析一次域名。

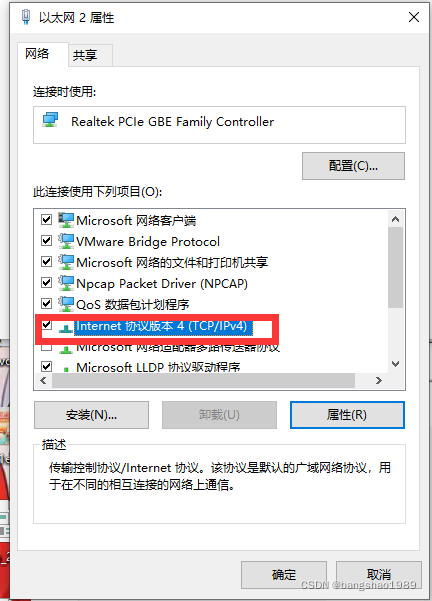
第二步、

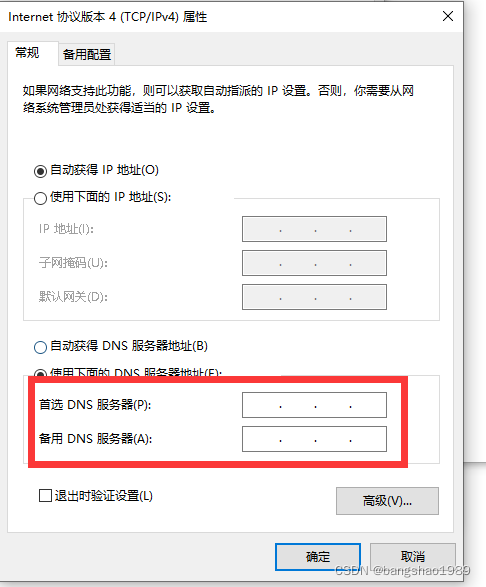
如果浏览器缓存中没有数据，浏览器会查找操作系统缓存中是否有这个域名对应的DNS解析结果。其实操作系统也有一个域名解析的过程，在Linux中可以通过/etc/hosts文件来设置，而在windows中可以通过配置C:\Windows\System32\drivers\etc\hosts文件来设置，用户可以将任何域名解析到任何能够访问的IP地址。例如，我们在测试时可以将一个域名解析到一台测试服务器上，这样不用修改任何代码就能测试到单独服务器上的代码的业务逻辑是否正确。正是因为有这种本地DNS解析的规程，所以有黑客就可能通过修改用户的域名来把特定的域名解析到他指定的IP地址上，导致这些域名被劫持。

第三步、

前两个过程无法解析时，就要用到我们网络配置中的"DNS服务器地址"了。操作系统会把这个域名发送给这个本地DNS服务器。每个完整的内网通常都会配置本地DNS服务器，例如用户是在学校或工作单位接入互联网，那么用户的本地DNS服务器肯定在学校或工作单位里面。它们一般都会缓存域名解析结果，当然缓存时间是受到域名的失效时间控制的。大约80%的域名解析到这里就结束了，后续的DNS迭代和递归也是由本地DNS服务器负责。

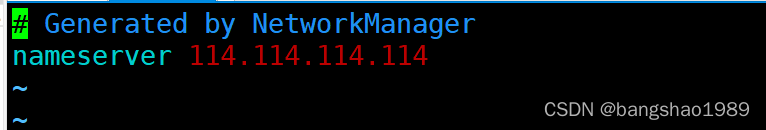
windows在这配置：控制面板-》网络和共享中心-》更改适配器设置-》选中目标适配器右键选择属性-》Internet协议版本4(TCP/IPv4)-》配置DNS地址。





Linux在这设置：/etc/resolv.conf

IMG_256



第四步、

如果本地DNS服务器仍然没有命中，就直接到根DNS服务器请求解析。

第五步、

根DNS服务器返回给本地DNS域名服务器一个顶级DNS服务器地址，它是国际顶级域名服务器，如.com、.cn、.org等，全球只有13台左右。

第六步、

本地DNS服务器再向上一步获得的顶级DNS服务器发送解析请求。

第七步、

接受请求的顶级DNS服务器查找并返回此域名对应的Name Server域名服务器的地址，这个Name Server服务器就是我要访问的网站域名提供商的服务器，其实该域名的解析任务就是由域名提供商的服务器来完成。 比如我要访问www.baidu.com，而这个域名是从A公司注册获得的，那么A公司上的服务器就会有www.baidu.com的相关信息。

第八步、

Name Server服务器会查询存储的域名和IP的映射关系表，再把查询出来的域名和IP地址等等信息，连同一个TTL值返回给本地DNS服务器。

第九步、

返回该域名对应的IP和TTL值，本地DNS服务器会缓存这个域名和IP的对应关系，缓存时间由TTL值控制。

第十步、

把解析的结果返回给本地电脑，本地电脑根据TTL值缓存在本地系统缓存中，域名解析过程结束在实际的DNS解析过程中，可能还不止这10步，如Name Server可能有很多级，或者有一个GTM来负载均衡控制，这都有可能会影响域名解析过程。

递归查询和迭代查询的区别

DNS客户端和本地名称服务器是递归，而本地名称服务器和其他名称服务器之间是迭代。

DNS递归名称解析： 在DNS递归名称解析中，当所配置的本地名称服务器解析不了时，后面的查询工作是由本地名称服务器替代DNS客户端进行的（以“本地名称服务器”为中心），只需要本地名称服务器向DNS客户端返回最终的查询结果即可。

DNS迭代名称解析：（或者叫“迭代查询”）的所有查询工作全部是DNS客户端自己进行（以“DNS客户端”自己为中心）。在条件之一满足时就会采用迭代名称解析方式：

在查询本地名称服务器时，如果客户端的请求报文中没有申请使用递归查询，即在DNS请求报头部的RD字段没有置1。相当于说“你都没有主动要求我为你进行递归查询，我当然不会为你工作了”。

客户端在DNS请求报文中申请使用的是递归查询（也就是RD字段置1了），但在所配置的本地名称服务器上是禁用递归查询（DNS服务器一般默认支持递归查询的），即在应答DNS报文头部的RA字段置0。

域名解析记录

主要分为A记录、MX记录、CNAME记录、NS记录和TXT记录：

1、A记录

A代表Address，用来指定域名对应的IP地址，如将item.taobao.com指定到115.238.23.xxx，将switch.taobao.com指定到121.14.24.xxx。A记录可以将多个域名解析到一个IP地址，但是不能将一个域名解析到多个IP地址

2、MX记录

Mail Exchange，就是可以将某个域名下的邮件服务器指向自己的Mail Server，如taobao.com域名的A记录IP地址是115.238.25.xxx，如果将MX记录设置为115.238.25.xxx，即xxx@taobao.com的邮件路由，DNS会将邮件发送到115.238.25.xxx所在的服务器，而正常通过Web请求的话仍然解析到A记录的IP地址

3、CNAME记录

Canonical Name，即别名解析。所谓别名解析就是可以为一个域名设置一个或者多个别名，如将aaa.com解析到bbb.net、将ccc.com也解析到bbb.net，其中bbb.net分别是aaa.com和ccc.com的别名

4、NS记录

为某个域名指定DNS解析服务器，也就是这个域名由指定的IP地址的DNS服务器取解析

5、TXT记录

为某个主机名或域名设置说明，如可以为ddd.net设置TXT记录为"这是XXX的博客"这样的说明

1. TCP三次握手四次挥手的详细过程，解释其中的关键字段。

| **步骤** | **发送方** | **包类型** | **关键字段** | **接收方状态变化** |
| --- | --- | --- | --- | --- |
| 1 | 客户端A | SYN | SYN=1, seq=x | SYN\_SENT → (等) |
| 2 | 服务器B | SYN+ACK | SYN=1, ACK=1, seq=y, ack=x+1 | SYN\_RCVD → (等) |
| 3 | 客户端A | ACK | ACK=1, seq=x+1, ack=y+1 | ESTABLISHED ← ESTABLISHED |

| **步骤** | **发送方** | **包类型** | **关键字段** | **接收方状态变化** |
| --- | --- | --- | --- | --- |
| 1 | 客户端A | FIN | FIN=1, seq=u | FIN\_WAIT\_1 → FIN\_WAIT\_2 |
| 2 | 服务器B | ACK | ACK=1, ack=u+1 | CLOSE\_WAIT → (继续) |
| 3 | 服务器B | FIN+ACK | FIN=1, ACK=1, seq=v, ack=u+1 | LAST\_ACK → CLOSED |
| 4 | 客户端A | ACK | ACK=1, ack=v+1 | TIME\_WAIT → CLOSED |

### 第二部分：UDP“直接发”怎么知道发给谁、谁该接收、接收到的信息来自哪里？

哈哈，这个疑问超级好！很多人以为UDP“乱扔包”，其实它靠**IP地址 + 端口号**的“双标签系统”精准定位，就像寄信用“邮编 + 门牌号”。UDP无连接，但不等于“无地址”——它借用网络层（IP）和应用层（端口）的力量，确保包“知道去哪、从哪来”。我们一步步拆过程。

**UDP报文结构回顾**（头仅8字节，简单粗暴）：

* **源端口（2字节）**：发包的应用进程ID（e.g., 12345）。
* **目的端口（2字节）**：收包的应用进程ID（e.g., 53=DNS）。
* **长度（2字节）**：整个数据报大小。
* **校验和（2字节）**：简单错误检查（可选）。

**发送过程：怎么知道“发给谁”？**

1. **应用层生成数据**：比如，浏览器要查DNS，应用层（DNS协议）生成查询消息：“www.baidu.com的IP？”。
2. **加UDP头**：传输层加头——源端口（浏览器随机端口，如5353）、目的端口（DNS标准53）。数据=UDP头 + 应用数据。
3. **交给IP层**：网络层加IP头——源IP（你的IP，如192.168.1.100）、目的IP（DNS服务器IP，如8.8.8.8）。
4. **直接发**：整个“IP包”（含UDP）扔到网络接口层，通过路由器跳跃。无握手，无确认——“邮递员”只管投递，不管到没到。

**比喻**：你写明信片（UDP数据），写上“寄给：北京XX路53号（目的IP+端口）”，落款“来自：上海YY街5353号（源IP+端口）”。邮局（IP层）按地址转运，收方一看门牌（端口）知道给谁。

**接收过程：怎么知道“谁该接收、来自哪里”？**

1. **网络层剥IP头**：包到你的主机，IP层查目的IP匹配本地？是→交给传输层；否→丢。
2. **传输层剥UDP头**：UDP层查目的端口（e.g., 53）→找对应应用进程（DNS服务）。同时，源IP+源端口告诉你“来自谁的哪个App”。
3. **应用层处理**：DNS应用收到消息，知道来源（可回包到源IP+源端口），处理后回。
   * 如果端口无对应进程？OS丢包或发ICMP错误。

**关键点**：

* **发给谁**：目的IP（主机）+目的端口（进程）。无连接，但每个包自带“坐标”。
* **谁接收**：主机收到后，端口号像“分机号”，直达正确App（e.g., 80=HTTP服务器）。
* **来自哪里**：源IP+源端口，收方据此回复（e.g., DNS回包用你的源端口作为目的端口）。
* **潜在问题**：丢包？UDP不管，重传靠应用（e.g., DNS超时重问）。乱序？应用自己序。防火墙可滤端口防滥用。

**例子**：玩游戏UDP发“位置更新”——源：你的IP+游戏端口1234；目的：服务器IP+游戏端口7777。服务器收到，知道“玩家X从IP Y发来”，回包到你的坐标。快！但网抖=丢帧。

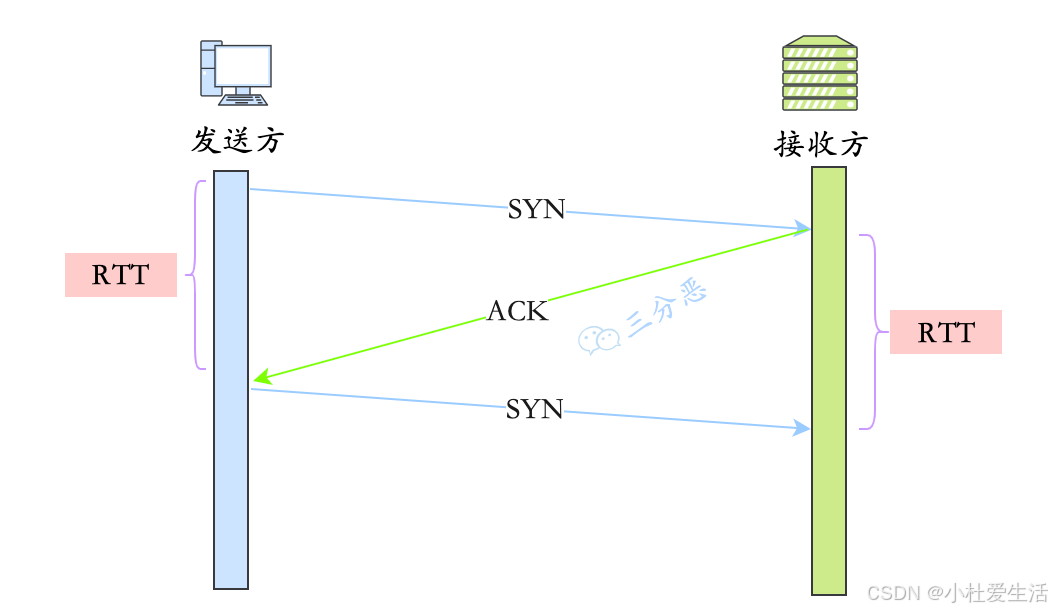
4.TCP的重传机制：

重传包括超时重传、快速重传、带选择确认的重传（SACK）、重复 SACK 四种。

超时重传

超时重传，是 TCP 协议保证数据可靠性的另一个重要机制，其原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的 ACK 报文，那么就重新发送数据，直到发送成功为止。

超时时间设置为多少合适捏？



RTT 就是数据完全发送完，到收到确认信号的时间，即数据包的一次往返时间。

超时重传时间，就是 RTO（Retransmission Timeout)。那么，RTO 到底设置多大呢？

如果 RTO 设置很大，等了很久都没重发，这样肯定就不行。

如果 RTO 设置很小，那很可能数据都没有丢失，就开始重发了，这会导致网络阻塞，从而恶性循环，导致更多的超时出现。

一般来说，RTO 略微大于 RTT，效果是最佳的。

其实，RTO 有个标准方法的计算公式，也叫 Jacobson / Karels 算法。

1.首先计算 SRTT（即计算平滑的 RTT）

SRTT = (1 - α) \* SRTT + α \* RTT //求 SRTT 的加权平均

（这一次的SRTT是上一次的SRTT与这一次的RTT计算得来，初始SRTT等于RTT）

1. 其次，计算 RTTVAR (round-trip time variation)

RTTVAR = (1 - β) \* RTTVAR + β \* (|RTT - SRTT|) //计算 SRTT 与真实值的差距

3.最后，得出最终的 RTO

RTO = µ \* SRTT + ∂ \* RTTVAR = SRTT + 4·RTTVAR

在 Linux 下，α = 0.125，β = 0.25， μ = 1，∂ = 4。别问这些参数是怎么来的，它们是大量实践，调出的最优参数。

超时重传不是十分完美的重传方案，它有这些缺点：

当一个报文丢失时，会等待一定的超时周期，才重传分组，增加了端到端的时延。

当一个报文丢失时，在其等待超时的过程中，可能会出现这种情况：其后的报文段已经被接收端接收但却迟迟得不到确认，发送端会认为也丢失了，从而引起不必要的重传，既浪费资源也浪费时间。

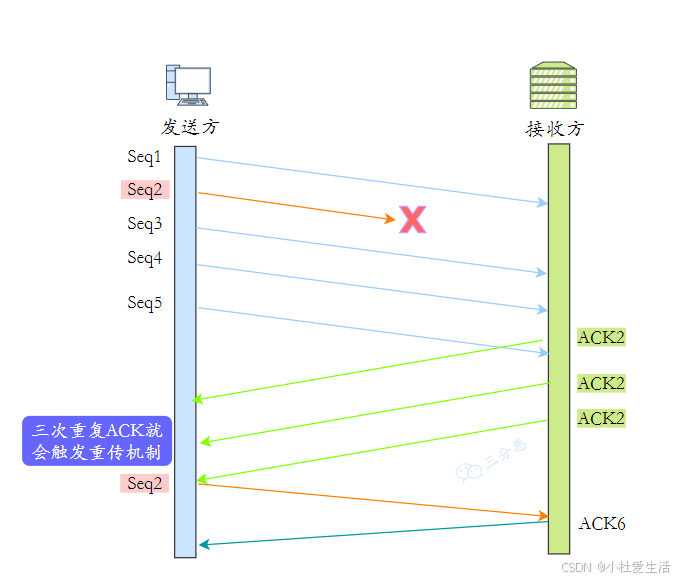
并且，对于 TCP，如果发生一次超时重传，时间间隔下次就会加倍。

快速重传

TCP 还有另外⼀种快速重传（Fast Retransmit）机制，它不以时间为驱动，⽽是以数据驱动重传。

它不以时间驱动，而是以数据驱动。它是基于接收端的反馈信息来引发重传的。

可以用它来解决超时重发的时间等待问题，快速重传流程如下：



在上图，发送⽅发出了 1，2，3，4，5 份数据：

第⼀份 Seq1 先送到了，于是就 Ack 回 2；

结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；

后⾯的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；

发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。

最后，收到了 Seq2，此时因为 Seq3，Seq4，Seq5 都收到了，于是 Ack 回 6 。

快速重传机制只解决了⼀个问题，就是超时时间的问题，但是它依然⾯临着另外⼀个问题。就是重传的时候，是重传之前的⼀个，还是重传所有的问题。

⽐如对于上⾯的例⼦，是重传 Seq2 呢？还是重传 Seq2、Seq3、Seq4、Seq5 呢？因为发送端并不清楚这连续的三个 Ack 2 是谁传回来的。

根据 TCP 不同的实现，以上两种情况都是有可能的。可⻅，这是⼀把双刃剑。

为了解决不知道该重传哪些 TCP 报⽂，于是就有 SACK ⽅法。

一、传统 TCP ACK 的局限性

先复习下普通的 ACK 是怎么工作的：

假设发送端连续发送了 5 个包：

[1][2][3][4][5]

如果包 3 丢失，接收方收到：

✅1 ✅2 ❌3 ✅4 ✅5

接收方会发送：

ACK = 3

ACK = 3

ACK = 3

ACK = 3 ...

表示“我希望从 3 开始的数据”。

👉 但它并不能告诉发送方：我其实已经收到了 4、5。

结果：

发送方只能根据重复 ACK 猜测“也许 3 丢了”，触发快速重传；

可是一旦丢的不止一个包（例如丢 3 和 4），事情就复杂了；

发送方不知道哪些数据已经收到、哪些没收到，只能笼统地重传多个包，效率低。

二、SACK（Selective Acknowledgment，选择确认）

目标：让接收方告诉发送方——我具体收到了哪些片段。

1️⃣ 工作原理

SACK 是 TCP 选项的一种扩展字段，由接收方在 ACK 报文中携带：

每个 SACK 块表示一个已收到的数据范围：

SACK = [Left Edge, Right Edge)

举例：

假设：

丢失情况：包3丢了，但包4、5收到了

接收方发送：

ACK = 3

SACK = [4,6)

意思是：

“我希望从 3 开始的数据（即包3）；

但我其实已经收到了 4 和 5。”

2️⃣ 发送方收到 SACK 后会：

知道哪些数据已经被确认；

只重传“未被确认”的那部分（例如包3），不再重复发送包4、5。

这样就极大提升了网络利用率，尤其在高丢包或长延迟环境下。

3️⃣ SACK 机制图解

发送端： [1][2][3][4][5]

接收端收到： 1,2,4,5

接收端回复： ACK=3 + SACK=[4,6)

发送端得知：3丢失，4~5已收，重传3即可。

三、DSACK（Duplicate SACK，重复选择确认）

1️⃣ 出现背景

SACK 能告诉发送方“哪些收到了”，

但有时候 发送方会误以为某个包丢了（例如乱序造成重复 ACK），从而多重传了一次。

为了让发送方知道自己“白白重传了”，就有了 DSACK。

2️⃣ 工作原理

接收方在检测到“重复的数据段”时，会在 SACK 选项中携带重复的区间。

例子：

发送方发出： [1][2][3][4]

包2乱序晚到

接收方先收到：1,3,4

→ 回复 ACK=2 + SACK=[3,5)

发送方误以为包2丢失 → 重传2

稍后包2原本的副本也到了

接收方现在收到重复的包2

→ 回复 DSACK = [2,3)

意思是：

“我收到了重复的 [2,3) 这段数据。”

发送方看到 DSACK，就知道：

啊，原来我重传的包2其实早已送达；

所以网络中可能只是乱序而不是丢包。

| **特性** | **SACK** | **DSACK** |
| --- | --- | --- |
| 全称 | Selective Acknowledgment | Duplicate Selective Acknowledgment |
| 目的 | 告诉发送方哪些数据**已成功收到** | 告诉发送方哪些数据**被重复收到** |
| 帮助 | 减少**不必要的重传** | 帮助发送方判断**是否错误重传**或**发生乱序** |
| 是否由接收方发出 | ✅ 是 | ✅ 是 |
| 是否基于 SACK 扩展 | ✅ 是 | ✅ 是（SACK 的一种特殊用法） |

TCP 发送方确实会根据 DSACK 调整之后的行为。

它不会仅仅“知道错了”，而是会据此修正自己的网络判断模型。

TCP 发送方在整个连接期间，会持续做三件事：

| **任务** | **依赖什么信息** |
| --- | --- |
| 估算 RTT、RTO | 根据 ACK 到达时间 |
| 估算丢包率 | 根据重复 ACK、超时、SACK |
| 调整发送速度（拥塞窗口 cwnd） | 根据拥塞控制算法（如 Reno、CUBIC 等） |

这些算法全都依赖于一个假设：

“重复 ACK 代表丢包。”

但现实中，这个假设有时是错的（例如包乱序）。

于是，DSACK 提供了一种“反馈纠正机制”。

| **时刻** | **没有 DSACK** | **有 DSACK** |
| --- | --- | --- |
| 网络轻微乱序 | 发送方误以为丢包，进入快速恢复，cwnd减半 | DSACK 告诉发送方：其实没丢，只是乱序 |
| 后续行为 | TCP 拥塞窗口长期降低，速度恢复慢 | TCP 恢复 cwnd，避免过度反应 |

实际行为（以 Linux TCP 为例）：

检测到 DSACK 后，内核会认为最近的“丢包”其实是误判；

内核记录一次“spurious retransmission（虚假重传）”；

如果连续发生这种情况，TCP 会：

增加乱序容忍度；

延长重传超时时间 RTO；

调整触发快速重传的阈值；

在某些实现中，还会恢复 cwnd（称为 DSACK-based cwnd undo）。

换句话说：

TCP 会逐步“学会”这个网络有点乱，

不要一看到重复 ACK 就惊慌地减速。

5.解释HTTP协议版本的发展过程，每一次改进了什么方面？目前常用的HTTP协议是什么？

| **版本** | **出现时间** | **主要特征** | **是否常用** |
| --- | --- | --- | --- |
| **HTTP/0.9** | 1991年 | 最早版本，只支持纯文本传输 | ❌ 几乎淘汰 |
| **HTTP/1.0** | 1996年 | 增加请求头/响应头，支持多种媒体类型 | ⚠️ 仍用于学习 |
| **HTTP/1.1** | 1997年 | 长连接、管线化、缓存控制、Host头 | ✅ 仍广泛使用 |
| **HTTP/2** | 2015年 | 二进制分帧、多路复用、头部压缩 | ✅ 常用 |
| **HTTP/3** | 2022年（正式标准化） | 基于 QUIC（UDP），减少延迟 | 🚀 快速普及中 |

二、各版本的改进细节

HTTP/0.9（1991）

🌱 “最原始的 HTTP，只能拿文本。”

特点：

只有一个命令：GET

无请求头、无响应头

只能传输 HTML 文本

连接：一次请求一个 TCP 连接，完毕即断开

GET /index.html

<html>...</html>

问题：

无法传输图片、音频、视频等多媒体内容

无法区分内容类型、编码

无缓存机制、无状态控制

HTTP/1.0（1996）

🌿 “现代 Web 的雏形。”

改进：

增加请求头和响应头（允许携带元信息）

支持 Content-Type, User-Agent, Status Code 等

支持多媒体类型传输

可传输图片、音频等

引入缓存机制

使用 Expires、Last-Modified 等头

增加状态码（如 404, 500）

每次请求仍需重新建立 TCP 连接

问题：

TCP 连接频繁建立/断开 → 性能低

服务器无法区分同一 IP 下多个站点（虚拟主机）

HTTP/1.1（1997）

🌾 “真正的主流标准版本”，至今仍然大量使用。

重大改进：

持久连接（Persistent Connection）

默认 Connection: keep-alive

同一 TCP 连接可发送多个请求（减少握手开销）

管线化（Pipelining）

允许在一个连接上并行发送多个请求（但响应仍需按顺序返回）

实际上由于“队头阻塞”问题（Head-of-line blocking）→ 很少被用

引入 Host 头

Host: www.example.com

支持虚拟主机（一个服务器多个域名）

缓存优化

Cache-Control, ETag, If-Modified-Since

分块传输编码（Chunked Transfer Encoding）

支持流式传输，不必提前知道数据长度

更多方法支持

如 PUT、DELETE、OPTIONS 等

问题：

多个请求仍在同一 TCP 流中排队（队头阻塞）

头部体积越来越大（无压缩）

并发性能差（需多 TCP 连接）

HTTP/2（2015，RFC 7540）

🌲 “性能革命版本”，解决 HTTP/1.1 的关键性能瓶颈。

核心改进：

二进制分帧（Binary Framing）

请求与响应被分为独立的二进制帧，减少解析成本

多路复用（Multiplexing）

在一个 TCP 连接上并行传输多个请求/响应流（无队头阻塞）

头部压缩（HPACK）

减少重复的 HTTP 头传输成本

服务器推送（Server Push）

服务端可主动推送资源（如 CSS、JS）

优先级和依赖机制

客户端可指定流的优先级以优化加载顺序

问题：

仍基于 TCP → 一旦丢包，仍然存在底层队头阻塞（HOL）

HTTP/3（2022，RFC 9114）

🚀 “新时代的 HTTP”，基于 QUIC（UDP）。

关键创新：

基于 QUIC 协议（UDP之上）

内建加密（TLS 1.3）

消除了 TCP 队头阻塞问题

更快的连接建立（0-RTT）

多路复用无阻塞

各请求独立传输，即使丢包也不影响其他流

更快的握手

第一次通信仅需 1-RTT（甚至 0-RTT）

内置加密

不再区分 HTTP 与 HTTPS

迁移能力强

QUIC 的连接可在 IP 变化（如 Wi-Fi → 5G）后保持

现状：

已被 Chrome、Firefox、Edge 等主流浏览器支持

大型网站（Google、Facebook、Cloudflare）已普及

| **特性** | **HTTP/1.0** | **HTTP/1.1** | **HTTP/2** | **HTTP/3** |
| --- | --- | --- | --- | --- |
| 连接复用 | ❌ | ✅ | ✅ | ✅ |
| 管线化 | ❌ | ⚠️（少用） | ✅ | ✅ |
| 二进制帧 | ❌ | ❌ | ✅ | ✅ |
| 头部压缩 | ❌ | ❌ | ✅（HPACK） | ✅（QPACK） |
| 队头阻塞 | ✅ | ✅ | ⚠️（TCP层） | ❌（QUIC） |
| 加密 | 可选（TLS） | 可选 | 通常使用 TLS | 强制 TLS 1.3 |
| 基于协议 | TCP | TCP | TCP | UDP（QUIC） |

目前最常用的 HTTP 版本

当前主流：HTTP/1.1 与 HTTP/2 共存

浏览器默认使用 HTTP/2（若服务器支持）

不支持 HTTP/2 的服务器则自动回退至 HTTP/1.1

HTTP/3 正在快速普及中

大多数现代浏览器和 CDN 已支持

预计未来几年成为主流