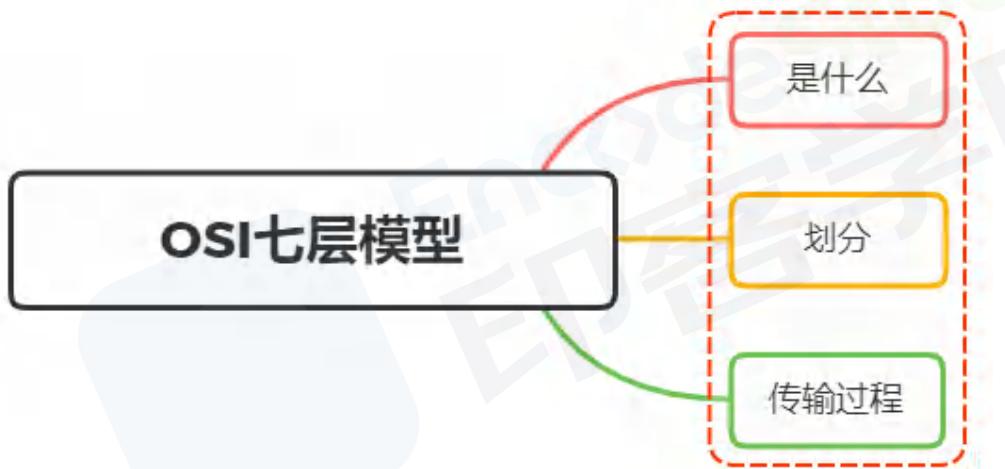


HTTP面试真题（14题）

1. 如何理解OSI七层模型？



1.1. 是什么

OSI (Open System Interconnect) 模型全称为开放式通信系统互连参考模型，是国际标准化组织 (ISO) 提出的一个试图使各种计算机在世界范围内互连为网络的标准框架

OSI 将计算机网络体系结构划分为七层，每一层实现各自的功能和协议，并完成与相邻层的接口通信。即每一层扮演固定的角色，互不打扰

1.2. 划分

OSI 主要划分了七层，如下图所示：

OSI 七层模型



1.2.1. 应用层

应用层位于 OSI 参考模型的第七层，其作用是通过应用程序间的交互来完成特定的网络应用

该层协议定义了应用进程之间的交互规则，通过不同的应用层协议为不同的网络应用提供服务。例如域名系统 DNS，支持万维网应用的 HTTP 协议，电子邮件系统采用的 SMTP 协议等

在应用层交互的数据单元我们称之为报文

1.2.2. 表示层

表示层的作用是使通信的应用程序能够解释交换数据的含义，其位于 OSI 参考模型的第六层，向上为应用层提供服务，向下接收来自会话层的服务

该层提供的服务主要包括数据压缩，数据加密以及数据描述，使应用程序不必担心在各台计算机中表示和存储的内部格式差异

1.2.3. 会话层

会话层就是负责建立、管理和终止表示层实体之间的通信会话

该层提供了数据交换的定界和同步功能，包括了建立检查点和恢复方案的方法

1.2.4. 传输层

传输层的主要任务是为两台主机进程之间的通信提供服务，处理数据包错误、数据包次序，以及其他一些关键传输问题

传输层向高层屏蔽了下层数据通信的细节。因此，它是计算机通信体系结构中关键的一层

其中，主要的传输层协议是 **TCP** 和 **UDP**

1.2.5. 网络层

两台计算机之间传送数据时其通信链路往往不止一条，所传输的信息甚至可能经过很多通信子网

网络层的主要任务就是选择合适的网间路由和交换节点，确保数据按时成功传送

在发送数据时，网络层把传输层产生的报文或用户数据报封装成分组和包，向下传输到数据链路层

在网络层使用的协议是无连接的网际协议（Internet Protocol）和许多路由协议，因此我们通常把该层简单地称为 **IP 层**

1.2.6. 数据链路层

数据链路层通常也叫做链路层，在物理层和网络层之间。两台主机之间的数据传输，总是在一段一段的链路上传送的，这就需要使用专门的链路层协议

在两个相邻节点之间传送数据时，数据链路层将网络层交下来的 **IP** 数据报组装成帧，在两个相邻节点间的链路上传送帧

每一帧的数据可以分成：报头 **head** 和数据 **data** 两部分：

- **head** 标明数据发送者、接受者、数据类型，如 MAC 地址
- **data** 存储了计算机之间交互的数据

通过控制信息我们可以知道一个帧的起止比特位置，此外，也能使接收端检测出所收到的帧有无差错，如果发现差错，数据链路层能够简单的丢弃掉这个帧，以避免继续占用网络资源

1.2.7. 物理层

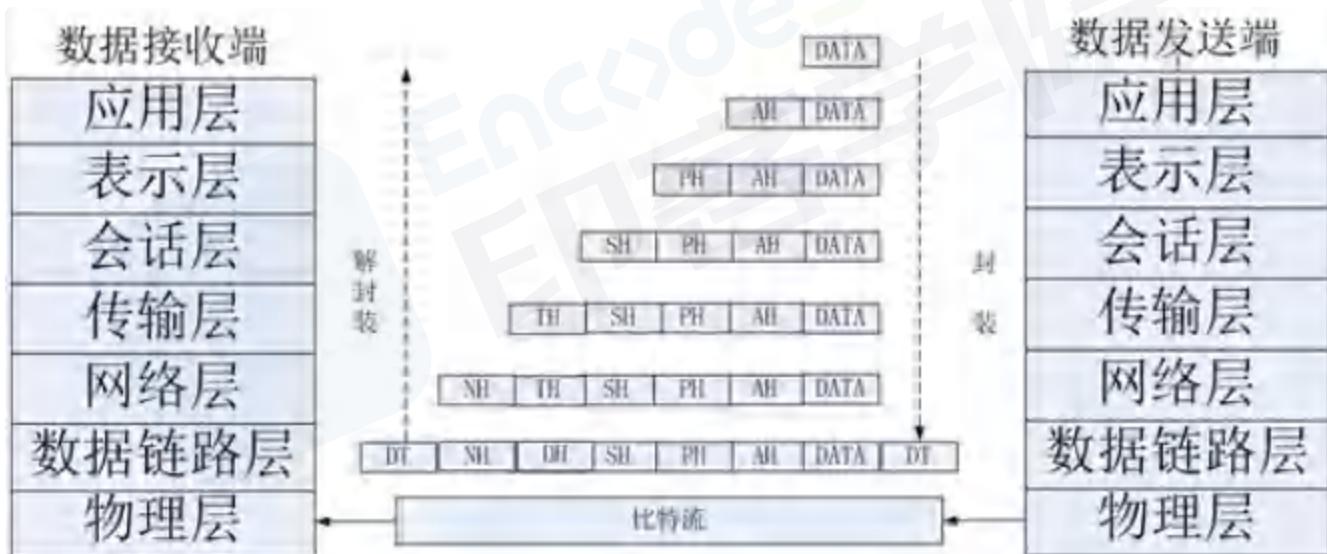
作为 **OSI** 参考模型中最低的一层，物理层的作用是实现计算机节点之间比特流的透明传送

该层的主要任务是确定与传输媒体的接口的一些特性（机械特性、电气特性、功能特性，过程特性）

该层主要是和硬件有关，与软件关系不大

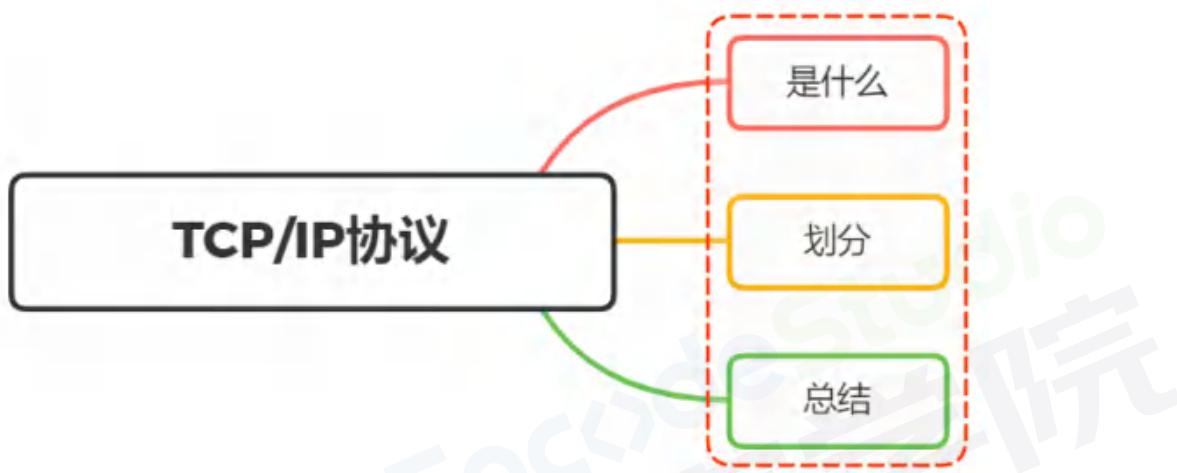
1.3. 三、传输过程

数据在各层之间的传输如下图所示：



- 应用层报文被传送到运输层
- 在最简单的情况下，运输层收取到报文并附上附加信息，该首部将被接收端的运输层使用
- 应用层报文和运输层首部信息一道构成了运输层报文段。附加的信息可能包括：允许接收端运输层向上向适当的应用程序交付报文的信息以及差错检测位信息。该信息让接收端能够判断报文中的比特是否在途中已被改变
- 运输层则向网络层传递该报文段，网络层增加了如源和目的端系统地址等网络层首部信息，生成了网络层数据报
- 网络层数据报接下来被传递给链路层，在数据链路层数据包添加发送端 MAC 地址和接收端 MAC 地址后被封装成数据帧
- 在物理层数据帧被封装成比特流，之后通过传输介质传送到对端
- 对端再一步步解开封装，获取到传送的数据

2. 如何理解TCP/IP协议？



2.1. 是什么

TCP/IP，传输控制协议/网际协议，是指能够在多个不同网络间实现信息传输的协议簇

- TCP（传输控制协议）

一种面向连接的、可靠的、基于字节流的传输层通信协议

- IP（网际协议）

用于封包交换数据网络的协议

TCP/IP协议不仅仅指的是TCP和IP两个协议，而是指一个由FTP、SMTP、TCP、UDP、I P等协议构成的协议簇，

只是因为在TCP/IP协议中TCP协议和IP协议最具代表性，所以通称为TCP/IP协议族（英语：TCP/IP Protocol Suite，或TCP/IP Protocols）

2.2. 划分

TCP/IP协议族按层次分别了五层体系或者四层体系

五层体系的协议结构是综合了OSI和TCP/IP优点的一种协议，包括应用层、传输层、网络层、数据链路层和物理层

五层协议的体系结构只是为介绍网络原理而设计的，实际应用还是TCP/IP四层体系结构，包括应用层、传输层、网络层（网际互联层）、网络接口层

如下图所示：

TCP/IP 五层模型



TCP/IP 四层模型



2.2.1. 五层体系

2.2.1.1. 应用层

TCP/IP 模型将 OSI 参考模型中的会话层、表示层和应用层的功能合并到一个应用层实现，通过不同的应用层协议为不同的应用提供服务

如：FTP、Telnet、DNS、SMTP 等

2.2.1.2. 传输层

该层对应于 OSI 参考模型的传输层，为上层实体提供源端到对端主机的通信功能

传输层定义了两个主要协议：传输控制协议（TCP）和用户数据报协议（UDP）

其中面向连接的 TCP 协议保证了数据的传输可靠性，面向无连接的 UDP 协议能够实现数据包简单、快速地传输

2.2.1.3. 网络层

负责为分组网络中的不同主机提供通信服务，并通过选择合适的路由将数据传递到目标主机

在发送数据时，网络层把运输层产生的报文段或用户数据封装成分组或包进行传送

2.2.1.4. 数据链路层

数据链路层在两个相邻节点传输数据时，将网络层交下来的IP数据报组装成帧，在两个相邻节点之间的链路上传送帧

2.2.1.5. 物理层

保数据可以在各种物理媒介上进行传输，为数据的传输提供可靠的环境

2.2.2. 四层体系

TCP/IP 的四层结构则如下表所示：

层次名称	单位	功 能	协 议
网络接口层	帧	负责实际数据的传输，对应OSI参考模型的下两层	HDLC（高级链路控制协议）PPP（点对点协议）SLIP（串行线路接口协议）
网络层	数据报	负责网络间的寻址数据传输，对应OSI参考模型的第三层	IP（网际协议）ICMP（网际控制消息协议）ARP（地址解析协议）RARP（反向地址解析协议）
传输层	报文段	负责提供可靠的传输服务，对应OSI参考模型的第四层	TCP（控制传输协议）UDP（用户数据报协议）
应用层		负责实现一切与应用程序相关的功能，对应OSI参考模型的上三层	FTP（文件传输协议）HTTP（超文本传输协议）DNS（域名服务器协议）SMTP（简单邮件传输协议）NFS（网络文件系统协议）

2.3. 总结

OSI 参考模型与 TCP/IP 参考模型区别如下：

相同点：

- OSI 参考模型与 TCP/IP 参考模型都采用了层次结构
- 都能够提供面向连接和无连接两种通信服务机制

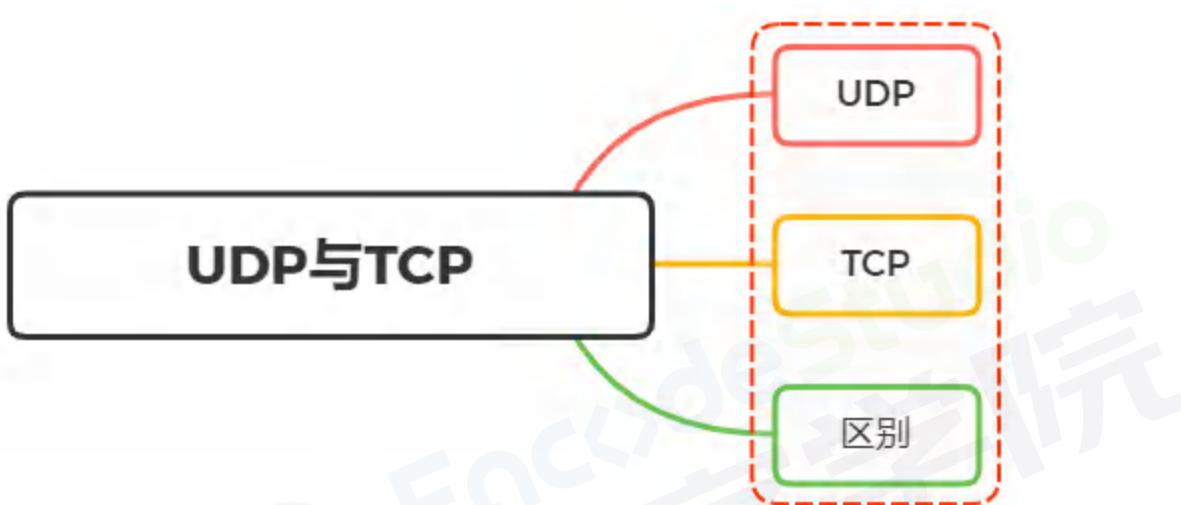
不同点：

- OSI 采用的七层模型； TCP/IP 是四层或五层结构
- TCP/IP 参考模型没有对网络接口层进行细分，只是一些概念性的描述； OSI 参考模型对服务和协议做了明确的区分
- OSI 参考模型虽然网络划分为七层，但实现起来较困难。TCP/IP 参考模型作为一种简化的分层结构是可以的
- TCP/IP 协议去掉表示层和会话层的原因在于会话层、表示层、应用层都是在应用程序内部实现的，最终产出的是一个应用数据包，而应用程序之间是几乎无法实现代码的抽象共享的，这也就造成 OSI 设想中的应用程序维度的分层是无法实现的

三种模型对应关系如下图所示：

区域	TCP/IP 四层模型	TCP/IP 五层模型	OSI 七层模型	单位	地址	功能	对应设备	协议
计算机高层	应用层	应用层	应用层	应用进程	进程号	应用程序与协议	应用程序 (eg: FTP、HTTP)	FTP、NFS
			表示层			数据加密、压缩	编码解码、加密解密	Telnet、SNMP
			会话层			会话的开始、恢复、释放、同步	建立会话、session 验证、断点传输	SMTP、DNS
网络低层	传输层	传输层	传输层	报文/数据段	端口号	端到端的可靠透明传输、保证数据完整性	进程与端口	TCP、UDP
	网络层	网络层	网络层	包/分组	IP地址	服务选择、路径选择、多路复用等(如何选择发送路径、方式)	路由器、防火墙、多层次交换机	IP、ICMP、ARP
	数据链路层	数据链路层	数据链路层	帧	mac地址	差错控制、流量控制 (规定如何进行01发送不会造成错误)	网卡、网桥、交换机	PPP、SLIP
		物理层	物理层	比特流	bit	光纤、电缆、双绞线连接，传送0/1电信号	中继器、集线器、网线	IEEExxxx

3. 如何理解 UDP 和 TCP？区别？应用场景？



3.1. UDP

UDP (User Datagram Protocol)，用户数据包协议，是一个简单的面向数据报的通信协议，即对应用层交下来的报文，不合并，不拆分，只是在其上面加上首部后就交给了下面的网络层

也就是说无论应用层交给 UDP 多长的报文，它统统发送，一次发送一个报文

而对接收方，接到后直接去除首部，交给上面的应用层就完成任务

UDP 报头包括4个字段，每个字段占用2个字节（即16个二进制位），标题短，开销小



特点如下：

- UDP 不提供复杂的控制机制，利用 IP 提供面向无连接的通信服务
- 传输途中出现丢包，UDP 也不负责重发
- 当包的到达顺序出现乱序时，UDP没有纠正的功能。
- 并且它是将应用程序发来的数据在收到的那一刻，立即按照原样发送到网络上的一种机制。即使是在出现网络拥堵的情况下，UDP 也无法进行流量控制等避免网络拥塞行为

3.2. TCP

TCP (Transmission Control Protocol)，传输控制协议，是一种可靠、面向字节流的通信协议，把上面应用层交下来的数据看成无结构的字节流来发送

可以想象成流水形式的，发送方TCP会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着，TCP会根据当前网络的拥塞状态来确定每个报文段的大小

TCP 报文首部有20个字节，额外开销大

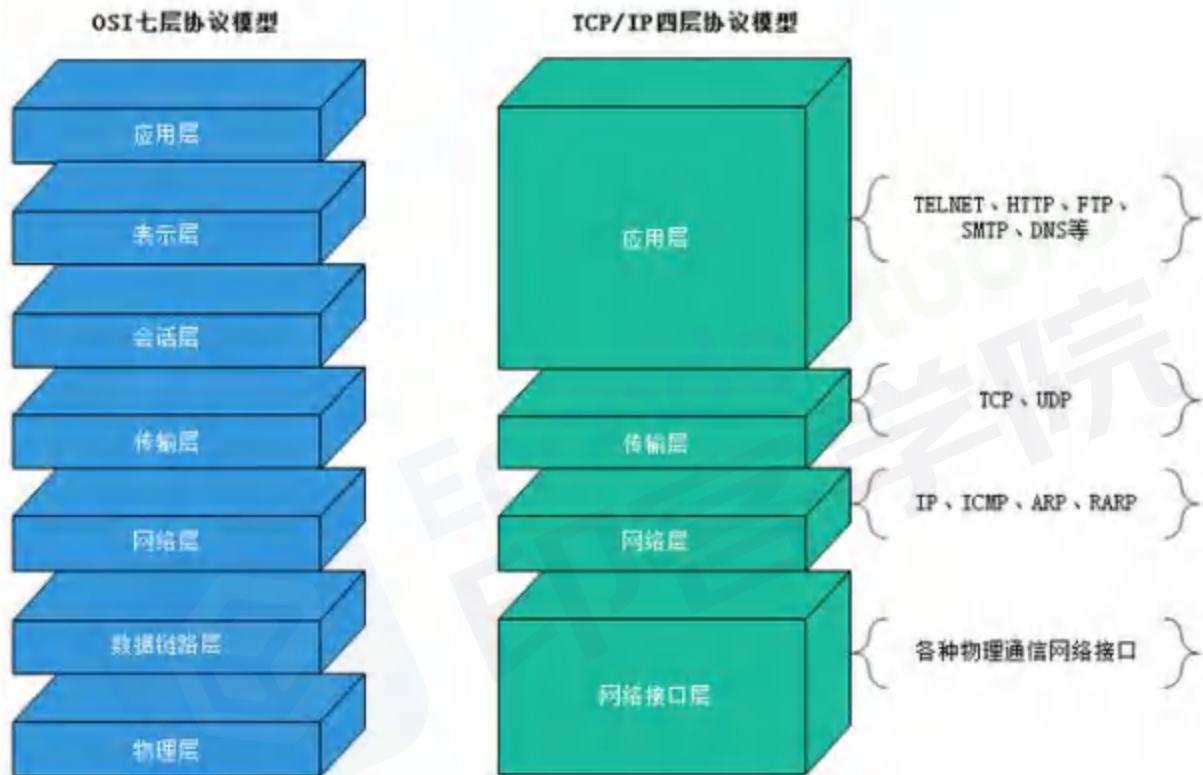


特点如下：

- TCP充分地实现了数据传输时各种控制功能，可以进行丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。而这些在 UDP 中都没有。
- 此外，TCP 作为一种面向有连接的协议，只有在确认通信对端存在时才会发送数据，从而可以控制通信流量的浪费。
- 根据 TCP 的这些机制，在 IP 这种无连接的网络上也能够实现高可靠性的通信（主要通过检验和、序列号、确认应答、重发控制、连接管理以及窗口控制等机制实现）

3.3. 区别

UDP 与 TCP 两者的都位于传输层，如下图所示：



两者区别如下表所示：

	TCP	UDP
可靠性	可靠	不可靠
连接性	面向连接	无连接
报文	面向字节流	面向报文
效率	传输效率低	传输效率高
双工性	全双工	一对一、一对多、多对一、多对多
流量控制	滑动窗口	无
拥塞控制	慢开始、拥塞避免、快重传、快恢复	无
传输效率	慢	快

- TCP 是面向连接的协议，建立连接3次握手、断开连接四次挥手，UDP是面向无连接，数据传输前后不连接连接，发送端只负责将数据发送到网络，接收端从消息队列读取
- TCP 提供可靠的服务，传输过程采用流量控制、编号与确认、计时器等手段确保数据无差错，不丢

失。UDP 则尽可能传递数据，但不保证传递交付给对方

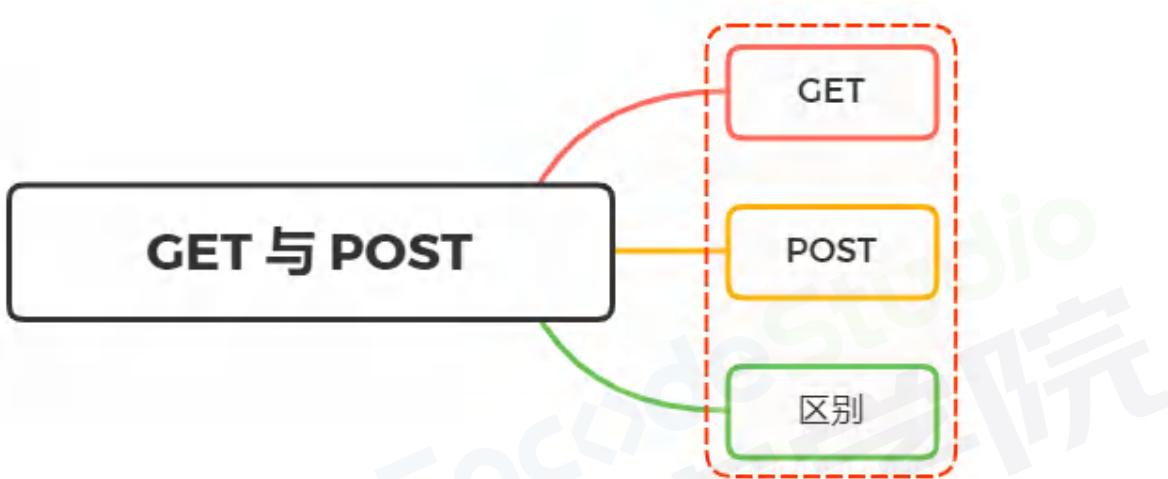
- TCP 面向字节流，将应用层报文看成一串无结构的字节流，分解为多个TCP报文段传输后，在目的站重新装配。UDP协议面向报文，不拆分应用层报文，只保留报文边界，一次发送一个报文，接收方去除报文首部后，原封不动将报文交给上层应用
- TCP 只能点对点全双工通信。UDP 支持一对一、一对多、多对一和多对多的交互通信

两者应用场景如下图：

应用层协议	应用	传输层协议
SMTP	电子邮件	
TELNET	远程终端接入	TCP
HTTP	万维网	
FTP	文件传输	
DNS	域名转换	
TFTP	文件传输	
SNMP	网络管理	UDP
NFS	远程文件服务器	

可以看到，TCP 应用场景适用于对效率要求低，对准确性要求高或者要求有链接的场景，而UDP 适用场景为对效率要求高，对准确性要求低的场景

4. 说一下 GET 和 POST 的区别？



4.1. 是什么

`GET` 和 `POST`，两者是 `HTTP` 协议中发送请求的方法

4.1.1. GET

`GET` 方法请求一个指定资源的表示形式，使用`GET`的请求应该只被用于获取数据

4.1.2. POST

`POST` 方法用于将实体提交到指定的资源，通常导致在服务器上的状态变化或副作用

本质上都是 `TCP` 链接，并无差别

但是由于 `HTTP` 的规定和浏览器/服务器的限制，导致他们在应用过程中会体现出一些区别

4.2. 区别

- `GET`在浏览器回退时是无害的，而`POST`会再次提交请求。
- `GET`产生的URL地址可以被`Bookmark`，而`POST`不可以。
- `GET`请求会被浏览器主动`cache`，而`POST`不会，除非手动设置。
- `GET`请求只能进行`url`编码，而`POST`支持多种编码方式。
- `GET`请求参数会被完整保留在浏览器历史记录里，而`POST`中的参数不会被保留。
- `GET`请求在URL中传送的参数是有长度限制的，而`POST`没有。
- 对参数的数据类型，`GET`只接受`ASCII`字符，而`POST`没有限制。
- `GET`比`POST`更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

- GET参数通过URL传递， POST放在Request body中

4.2.1. 参数位置

貌似从上面看到 GET 与 POST 请求区别非常大，但两者实质并没有区别

无论 GET 还是 POST，用的都是同一个传输层协议，所以在传输上没有区别

当不携带参数的时候，两者最大的区别为第一行方法名不同

```
POST /uri HTTP/1.1 \r\n
```

```
GET /uri HTTP/1.1 \r\n
```

当携带参数的时候，我们都知道 GET 请求是放在 url 中， POST 则放在 body 中

GET 方法简约版报文是这样的

```
1 GET /index.html?name=qiming.c&age=22 HTTP/1.1
2 Host: localhost
```

POST 方法简约版报文是这样的

```
1 POST /index.html HTTP/1.1
2 Host: localhost
3 Content-Type: application/x-www-form-urlencoded
4
5 name=qiming.c&age=22
```

注意：这里只是约定，并不属于 HTTP 规范，相反的，我们可以在 POST 请求中 url 中写入参数，或者 GET 请求中的 body 携带参数

4.2.2. 参数长度

HTTP 协议没有 Body 和 URL 的长度限制，对 URL 限制的大多是浏览器和服务器的原因

IE 对 URL 长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持

这里限制的是整个 URL 长度，而不仅仅是参数值的长度

服务器处理长 URL 要消耗比较多的资源，为了性能和安全考虑，会给 URL 长度加限制

4.2.3. 安全

POST 比 GET 安全，因为数据在地址栏上不可见

然而，从传输的角度来说，他们都是不安全的，因为 HTTP 在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文

只有使用 HTTPS 才能加密安全

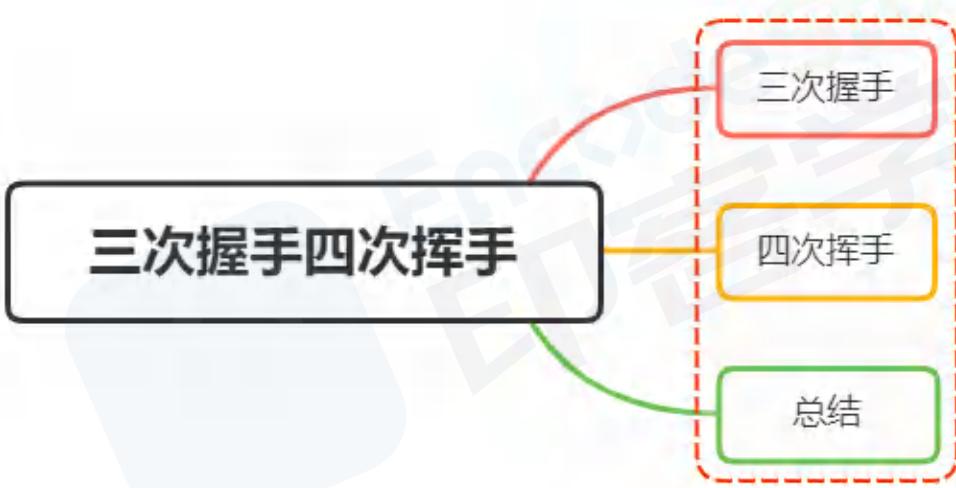
4.2.4. 数据包

对于 GET 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应200（返回数据）

对于 POST，浏览器先发送 header，服务器响应100 continue，浏览器再发送 data，服务器响应200 ok

并不是所有浏览器都会在 POST 中发送两次包，Firefox 就只发送一次

5. 说说TCP为什么需要三次握手和四次挥手？



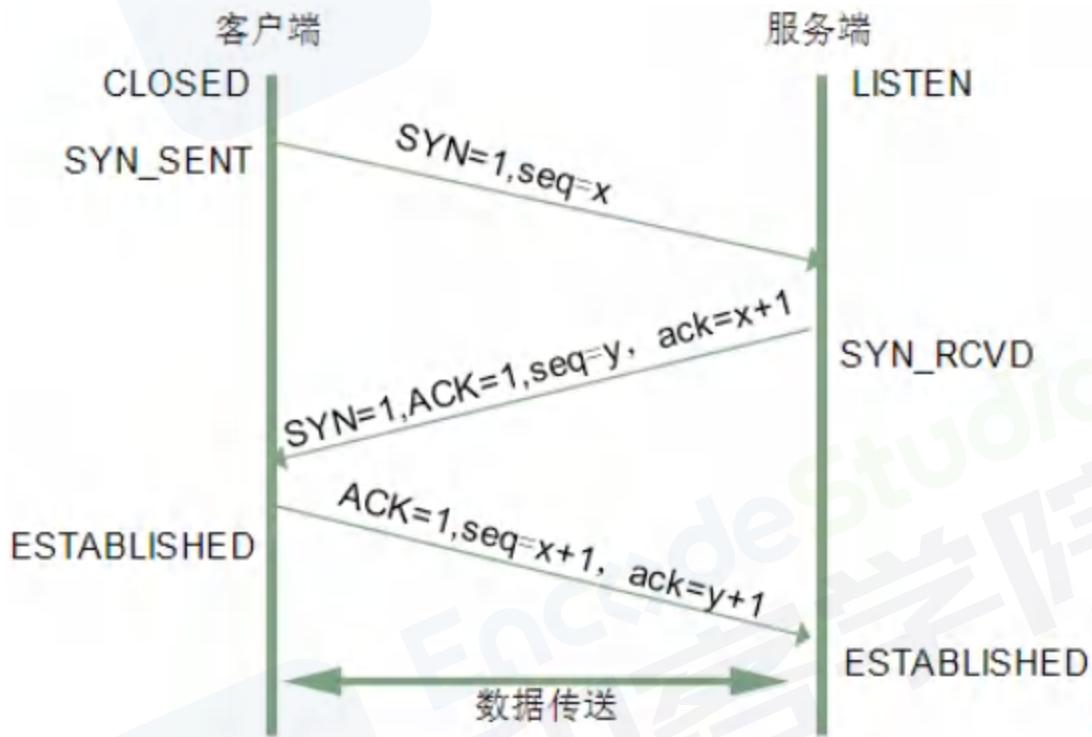
5.1. 三次握手

三次握手 (Three-way Handshake) 其实就是指建立一个TCP连接时，需要客户端和服务器总共发送3个包

主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备

过程如下：

- 第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN(c)，此时客户端处于 SYN_SENT 状态
- 第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，为了确认客户端的 SYN，将客户端的 ISN+1 作为 ACK 的值，此时服务器处于 SYN_RCVD 的状态
- 第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，值为服务器的 ISN+1。此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立了连接



上述每一次握手的作用如下：

- 第一次握手：客户端发送网络包，服务端收到了
这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。
- 第二次握手：服务端发包，客户端收到了
这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常
- 第三次握手：客户端发包，服务端收到了。
这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常

通过三次握手，就能确定双方的接收和发送能力是正常的。之后就可以正常通信了

5.1.1. 为什么不是两次握手？

如果是两次握手，发送端可以确定自己发送的信息能对方能收到，也能确定对方发的包自己能收到，但接收端只能确定对方发的包自己能收到 无法确定自己发的包对方能收到

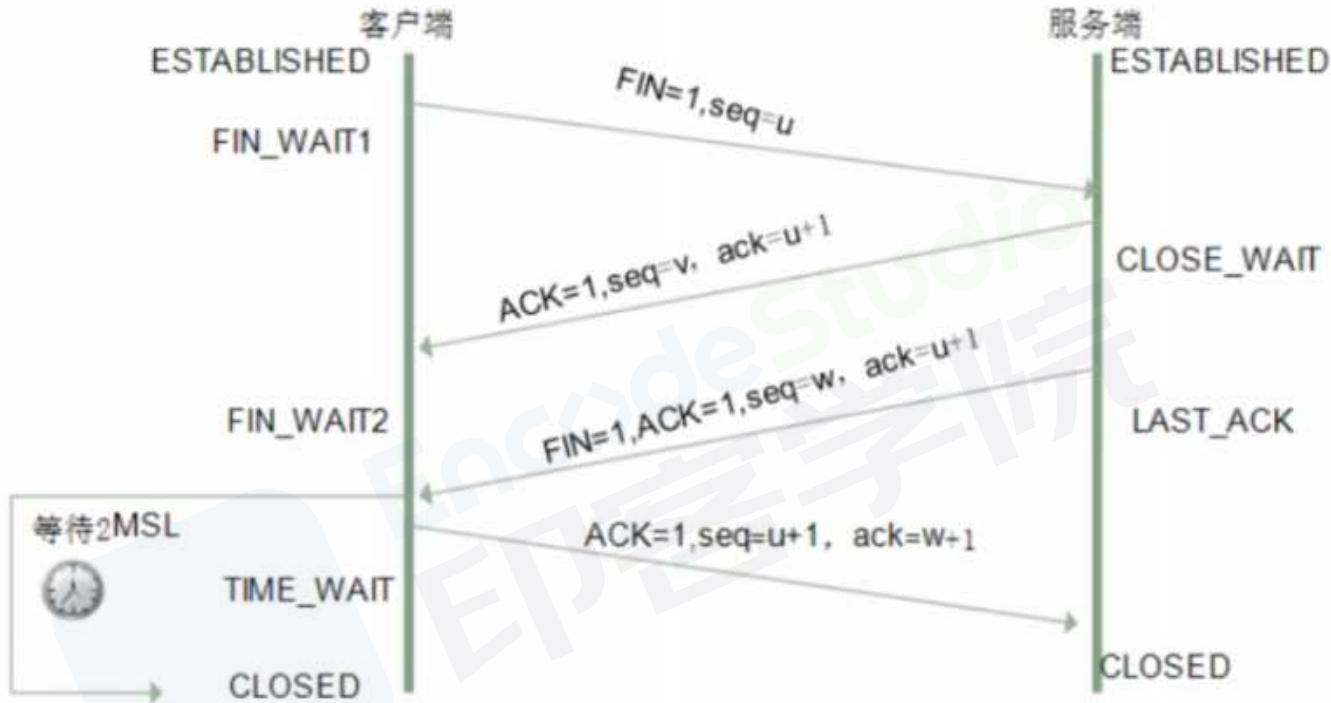
并且两次握手的话，客户端有可能因为网络阻塞等原因会发送多个请求报文，延时到达的请求又会与服务器建立连接，浪费掉许多服务器的资源

5.2. 四次挥手

tcp 终止一个连接，需要经过四次挥手

过程如下：

- 第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 FIN_WAIT1 状态，停止发送数据，等待服务端的确认
- 第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 CLOSE_WAIT 状态
- 第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST_ACK 的状态
- 第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的序列号值，此时客户端处于 TIME_WAIT 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态

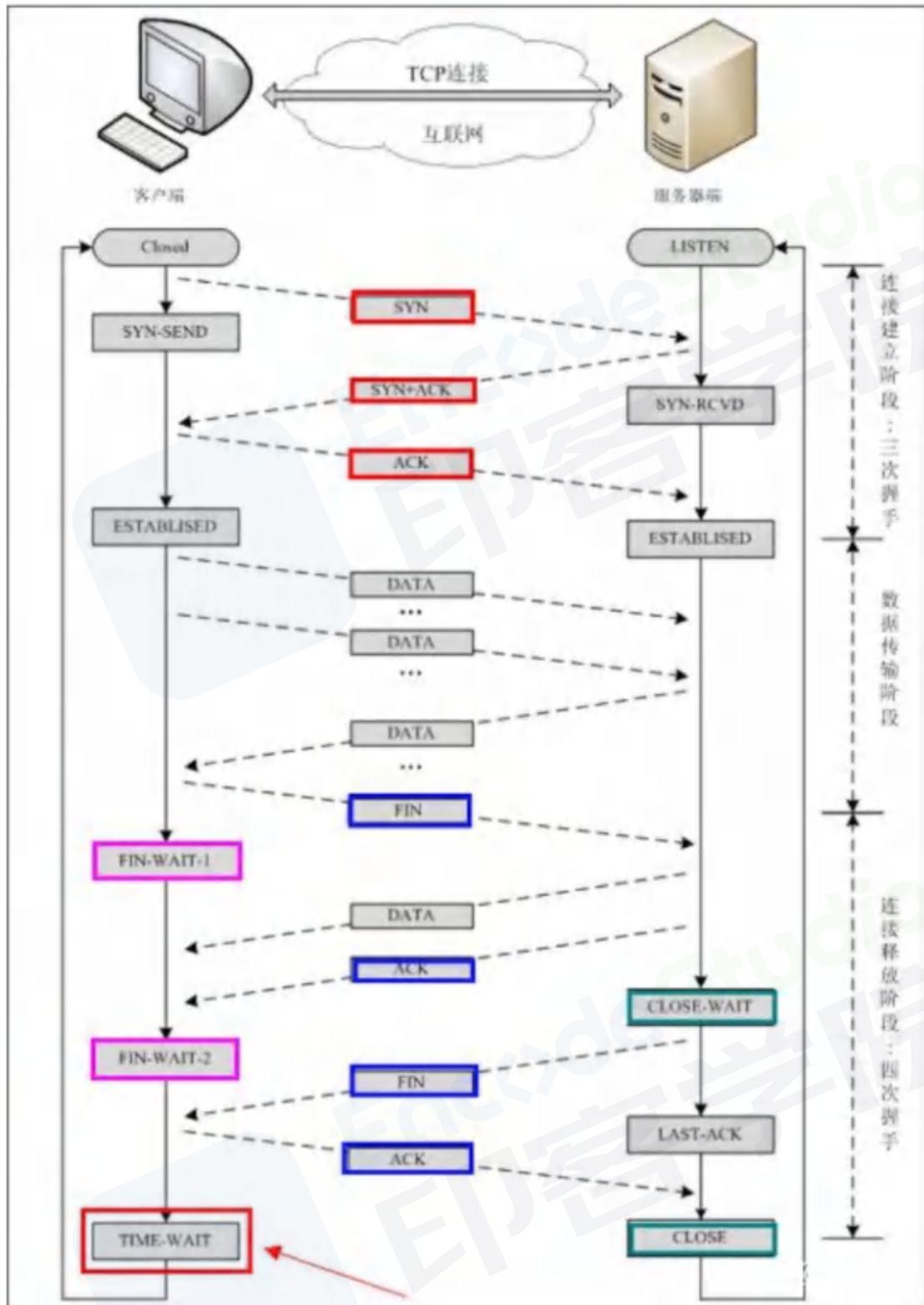


5.2.1. 四次挥手原因

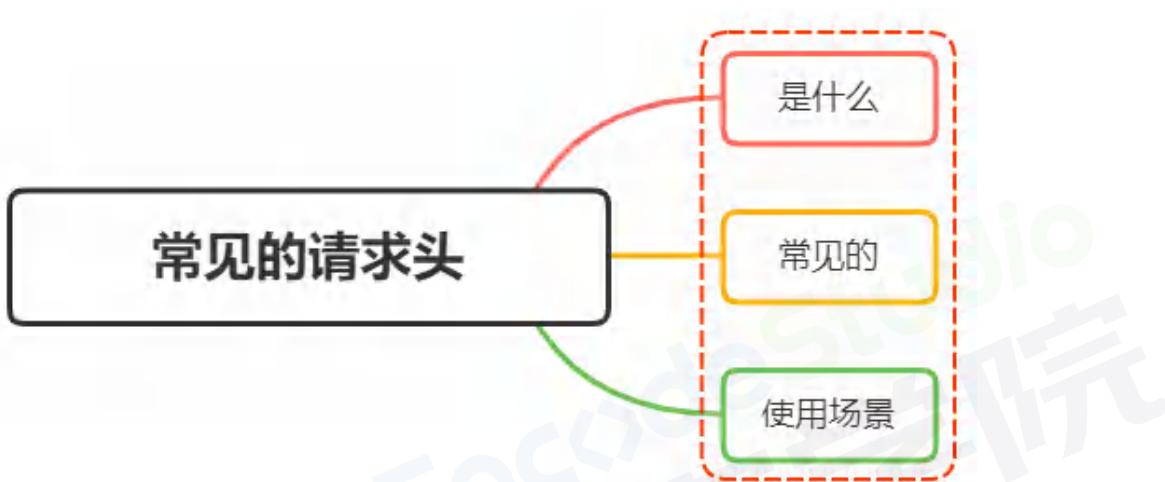
服务端在收到客户端断开连接 `Fin` 报文后，并不会立即关闭连接，而是先发送一个 `ACK` 包先告诉客户端收到关闭连接的请求，只有当服务器的所有报文发送完毕之后，才发送 `FIN` 报文断开连接，因此需要四次挥手

5.3. 总结

一个完整的三次握手四次挥手如下图所示：



6. 说说 HTTP 常见的请求头有哪些？作用？



6.1. 是什么

HTTP头字段 (HTTP header fields) ,是指在超文本传输协议 (HTTP) 的请求和响应消息中的消息头部部分

它们定义了一个超文本传输协议事务中的操作参数

HTTP头部字段可以自己根据需要定义, 因此可能在 Web 服务器和浏览器上发现非标准的头字段

下面是一个 HTTP 请求的请求头:

```
▼ HTTP | ⌂ 复制代码
1 GET /home.html HTTP/1.1
2 Host: developer.mozilla.org
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Referer: https://developer.mozilla.org/testpage.html
8 Connection: keep-alive
9 Upgrade-Insecure-Requests: 1
10 If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT
11 If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"
12 Cache-Control: max-age=0
```

6.2. 分类

常见的请求字段如下表所示:

字段名	说明	示例
Accept	能够接受的回应内容类型 (Content–Types)	Accept: text/plain
Accept–Charset	能够接受的字符集	Accept–Charset: utf–8
Accept–Encoding	能够接受的编码方式列表	Accept–Encoding: gzip, deflate
Accept–Language	能够接受的回应内容的自然语 言列表	Accept–Language: en–US
Authorization	用于超文本传输协议的认证的 认证信息	Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2Ft ZQ==
Cache–Control	用来指定在这次的请求/响应链 中的所有缓存机制 都必须 遵守 的指令	Cache–Control: no–cache
Connection	该浏览器想要优先使用的连接 类型	Connection: keep–alive Connection: Upgrade
Cookie	服务器通过 Set– Cookie (下 文详述) 发送的一个 超文本传 输协议Cookie	Cookie: \$Version=1; Skin=new;
Content–Length	以 八位字节数组 (8位的字 节) 表示的请求体的长度	Content–Length: 348
Content–Type	请求体的 多媒体类型	Content–Type: application/x– www–form–urlencoded
Date	发送该消息的日期和时间	Date: Tue, 15 Nov 1994 08:12:31 GMT
Expect	表明客户端要求服务器做出特 定的行为	Expect: 100–continue
Host	服务器的域名(用于虚拟主机)，以及服务器所监听的传输控 制协议端口号	Host: en.wikipedia.org:80 Host: en.wikipedia.org

If-Match	仅当客户端提供的实体与服务器上对应的实体相匹配时，才进行对应的操作。主要作用时，用作像 PUT 这样的方法中，仅当从用户上次更新某个资源以来，该资源未被修改的情况下，才更新该资源	If-Match: "737060cd8c284d8af7ad308 2f209582d"
If-Modified-Since	允许在对应的内容未被修改的情况下返回304未修改	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
If-None-Match	允许在对应的内容未被修改的情况下返回304未修改	If-None-Match: "737060cd8c284d8af7ad308 2f209582d"
If-Range	如果该实体未被修改过，则向我发送我所缺少的那一个或多个部分；否则，发送整个新的实体	If-Range: "737060cd8c284d8af7ad308 2f209582d"
Range	仅请求某个实体的一部分	Range: bytes=500-999
User-Agent	浏览器的浏览器身份标识字符串	User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/21.0
Origin	发起一个针对 跨来源资源共享的请求	Origin: http://www.example-social-network.com

6.3. 使用场景

通过配合请求头和响应头，可以满足一些场景的功能实现：

6.3.1. 协商缓存

协商缓存是利用的是 【Last-Modified, If-Modified-Since】 和 【ETag、If-None-Match】 这两对请求头响应头来管理的

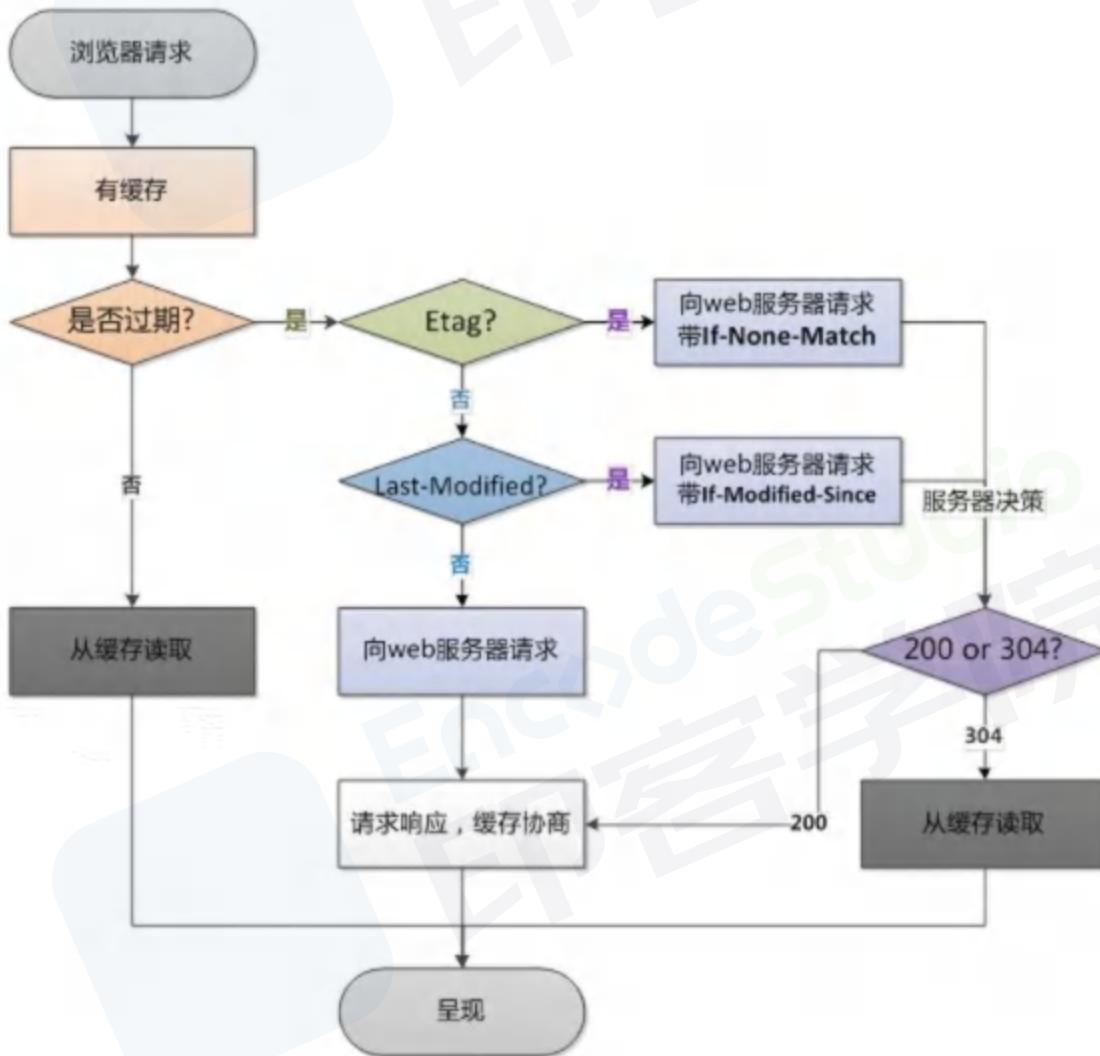
`Last-Modified` 表示本地文件最后修改日期，浏览器会在request header加上 `If-Modified-Since` (上次返回的 `Last-Modified` 的值)，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来

`Etag` 就像一个指纹，资源变化都会导致 `ETag` 变化，跟最后修改时间没有关系，`ETag` 可以保证每一个资源是唯一的

`If-None-Match` 的header会将上次返回的 `Etag` 发送给服务器，询问该资源的 `Etag` 是否有更新，有变动就会发送新的资源回来

而强制缓存不需要发送请求到服务端，根据请求头 `expires` 和 `cache-control` 判断是否命中强缓存

强制缓存与协商缓存的流程图如下所示：



6.3.2. 会话状态

`cookie`，类型为「小型文本文件」，指某些网站为了辨别用户身份而储存在用户本地终端上的数据，通过响应头 `set-cookie` 决定

作为一段一般不超过 4KB 的小型文本数据，它由一个名称（Name）、一个值（Value）和其它几个用于控制 `Cookie` 有效期、安全性、使用范围的可选属性组成

`Cookie` 主要用于以下三个方面：

- 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
- 个性化设置（如用户自定义设置、主题等）
- 浏览器行为跟踪（如跟踪分析用户行为等）

7. 说说HTTP 常见的状态码有哪些，适用场景？



7.1. 是什么

HTTP状态码（英语：HTTP Status Code），用以表示网页服务器超文本传输协议响应状态的3位数字代码

它由 RFC 2616 规范定义的，并得到 `RFC 2518`、`RFC 2817`、`RFC 2295`、`RFC 2774` 与 `RF C 4918` 等规范扩展

简单来讲，`http` 状态码的作用是服务器告诉客户端当前请求响应的状态，通过状态码就能判断和分析服务器的运行状态

7.2. 分类

状态码第一位数字决定了不同的响应状态，有如下：

- 1 表示消息
- 2 表示成功
- 3 表示重定向
- 4 表示请求错误
- 5 表示服务器错误

7.2.1. 1xx

代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束

常见的有：

- 100（客户端继续发送请求，这是临时响应）：这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应
- 101：服务器根据客户端的请求切换协议，主要用于websocket或http2升级

7.2.2. 2xx

代表请求已成功被服务器接收、理解、并接受

常见的有：

- 200（成功）：请求已成功，请求所希望的响应头或数据体将随此响应返回
- 201（已创建）：请求成功并且服务器创建了新的资源
- 202（已创建）：服务器已经接收请求，但尚未处理
- 203（非授权信息）：服务器已成功处理请求，但返回的信息可能来自另一来源
- 204（无内容）：服务器成功处理请求，但没有返回任何内容
- 205（重置内容）：服务器成功处理请求，但没有返回任何内容
- 206（部分内容）：服务器成功处理了部分请求

7.2.3. 3xx

表示要完成请求，需要进一步操作。通常，这些状态代码用来重定向

常见的有：

- 300（多种选择）：针对请求，服务器可执行多种操作。服务器可根据请求者（user agent）选择一项操作，或提供操作列表供请求者选择
- 301（永久移动）：请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置
- 302（临时移动）：服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求
- 303（查看其他位置）：请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码
- 305（使用代理）：请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理
- 307（临时重定向）：服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求

7.2.4. 4xx

代表了客户端看起来可能发生了错误，妨碍了服务器的处理

常见的有：

- 400（错误请求）：服务器不理解请求的语法
- 401（未授权）：请求要求身份验证。对于需要登录的网页，服务器可能返回此响应。
- 403（禁止）：服务器拒绝请求
- 404（未找到）：服务器找不到请求的网页
- 405（方法禁用）：禁用请求中指定的方法
- 406（不接受）：无法使用请求的内容特性响应请求的网页
- 407（需要代理授权）：此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理
- 408（请求超时）：服务器等候请求时发生超时

7.2.5. 5xx

表示服务器无法完成明显有效的请求。这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生

常见的有：

- 500（服务器内部错误）：服务器遇到错误，无法完成请求
- 501（尚未实施）：服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回

此代码

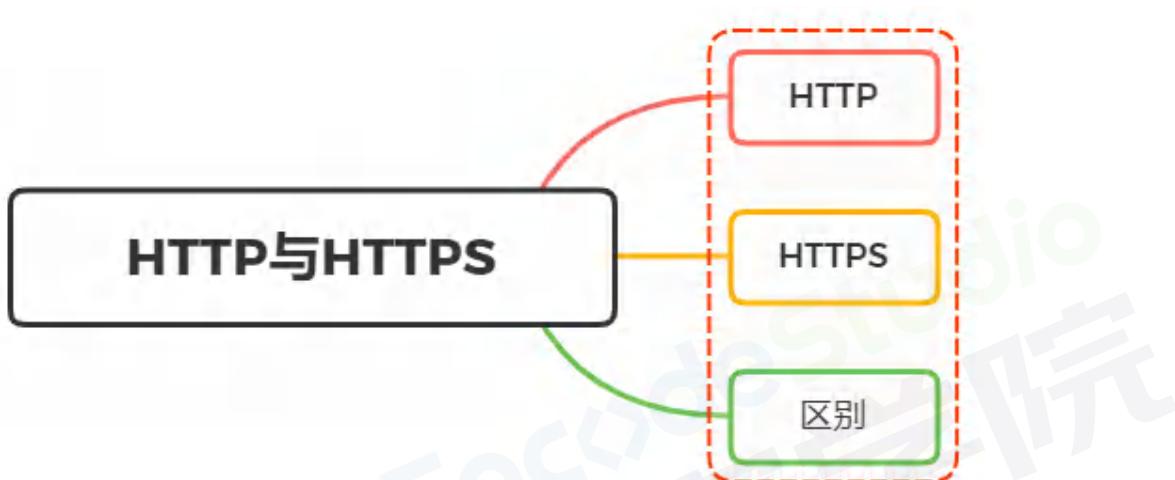
- 502（错误网关）：服务器作为网关或代理，从上游服务器收到无效响应
- 503（服务不可用）：服务器目前无法使用（由于超载或停机维护）
- 504（网关超时）：服务器作为网关或代理，但是没有及时从上游服务器收到请求
- 505（HTTP 版本不受支持）：服务器不支持请求中所用的 HTTP 协议版本

7.3. 适用场景

下面给出一些状态码的适用场景：

- 100：客户端在发送POST数据给服务器前，征询服务器情况，看服务器是否处理POST的数据，如果不处理，客户端则不上传POST数据，如果处理，则POST上传数据。常用于POST大数据传输
- 206：一般用来做断点续传，或者是视频文件等大文件的加载
- 301：永久重定向会缓存。新域名替换旧域名，旧的域名不再使用时，用户访问旧域名时用301就重定向到新的域名
- 302：临时重定向不会缓存，常用于未登陆的用户访问用户中心重定向到登录页面
- 304：协商缓存，告诉客户端有缓存，直接使用缓存中的数据，返回页面的只有头部信息，是没有内容部分
- 400：参数有误，请求无法被服务器识别
- 403：告诉客户端进制访问该站点或者资源，如在外网环境下，然后访问只有内网IP才能访问的时候则返回
- 404：服务器找不到资源时，或者服务器拒绝请求又不想说明理由时
- 503：服务器停机维护时，主动用503响应请求或 nginx 设置限速，超过限速，会返回503
- 504：网关超时

8. 什么是HTTP? HTTP 和 HTTPS 的区别?



8.1. HTTP

HTTP (HyperText Transfer Protocol), 即超文本运输协议, 是实现网络通信的一种规范



在计算机和网络世界有, 存在不同的协议, 如广播协议、寻址协议、路由协议等等.....

而 HTTP 是一个传输协议, 即将数据由A传到B或将B传输到A, 并且 A 与 B 之间能够存放很多第三方, 如: A<=>X<=>Y<=>Z<=>B

传输的数据并不是计算机底层中的二进制包, 而是完整的、有意义的数据, 如HTML 文件, 图片文件, 查询结果等超文本, 能够被上层应用识别

在实际应用中, HTTP 常被用于在 Web 浏览器和网站服务器之间传递信息, 以明文方式发送内容, 不提供任何方式的数据加密

特点如下:

- 支持客户/服务器模式
- 简单快速: 客户向服务器请求服务时, 只需传送请求方法和路径。由于HTTP协议简单, 使得HTTP服务器的程序规模小, 因而通信速度很快
- 灵活: HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记

- 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间
- 无状态：HTTP协议无法根据之前的状态进行本次的请求处理

8.2. HTTPS

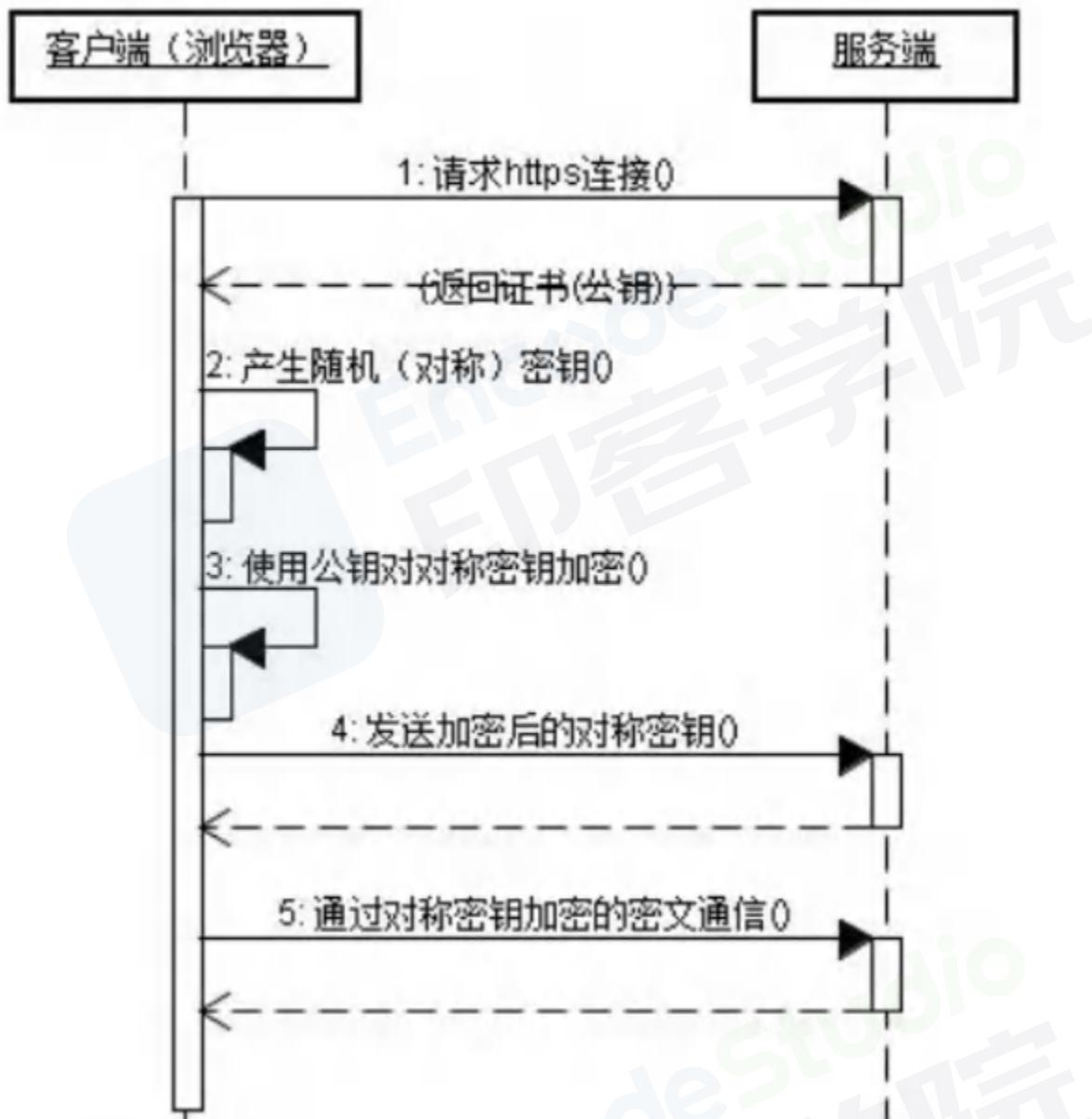
在上述介绍 **HTTP** 中，了解到 **HTTP** 传递信息是以明文的形式发送内容，这并不安全。而 **HTTPS** 出现正是为了解决 **HTTP** 不安全的特性

为了保证这些隐私数据能加密传输，让 **HTTP** 运行安全的 **SSL/TLS** 协议上，即 **HTTPS = HTTP + SSL/TLS**，通过 **SSL** 证书来验证服务器的身份，并为浏览器和服务器之间的通信进行加密

SSL 协议位于 **TCP/IP** 协议与各种应用层协议之间，浏览器和服务器在使用 **SSL** 建立连接时需要选择一组恰当的加密算法来实现安全通信，为数据通讯提供安全支持



流程图如下所示：

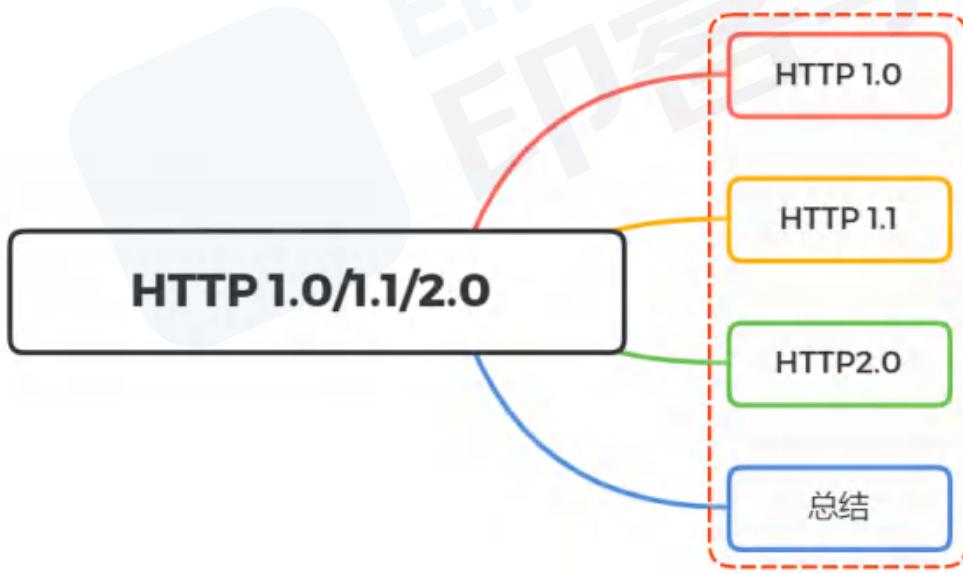


- 首先客户端通过URL访问服务器建立SSL连接
- 服务端收到客户端请求后，会将网站支持的证书信息（证书中包含公钥）传送一份给客户端
- 客户端的服务器开始协商SSL连接的安全等级，也就是信息加密的等级
- 客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站
- 服务器利用自己的私钥解密出会话密钥
- 服务器利用会话密钥加密与客户端之间的通信

8.3. 区别

- HTTPS是HTTP协议的安全版本，HTTP协议的数据传输是明文的，是不安全的，HTTPS使用了SSL/TLS协议进行了加密处理，相对更安全
- HTTP 和 HTTPS 使用连接方式不同，默认端口也不一样，HTTP是80，HTTPS是443
- HTTPS 由于需要设计加密以及多次握手，性能方面不如 HTTP
- HTTPS需要SSL，SSL 证书需要钱，功能越强大的证书费用越高

9. 说说 HTTP1.0/1.1/2.0 的区别？



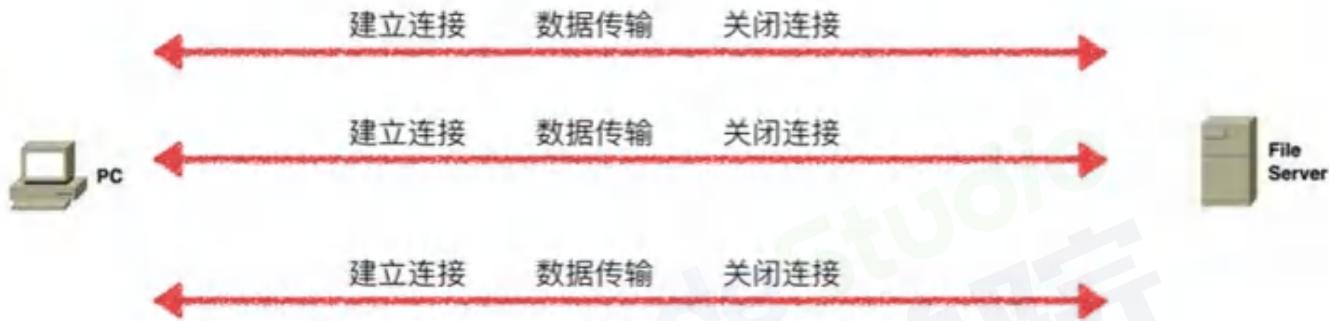
9.1. HTTP1.0

HTTP 协议的第二个版本，第一个在通讯中指定版本号的HTTP协议版本

HTTP 1.0 浏览器与服务器只保持短暂的连接，每次请求都需要与服务器建立一个 TCP 连接

服务器完成请求处理后立即断开 TCP 连接，服务器不跟踪每个客户也不记录过去的请求

简单来讲，每次与服务器交互，都需要新开一个连接

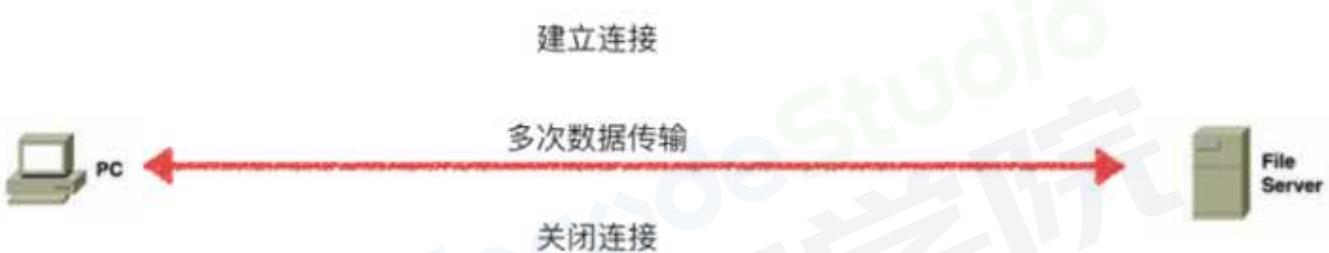


例如，解析 `html` 文件，当发现文件中存在资源文件的时候，这时候又创建单独的链接
最终导致，一个 `html` 文件的访问包含了多次的请求和响应，每次请求都需要创建连接、关系连接
这种形式明显造成了性能上的缺陷
如果需要建立长连接，需要设置一个非标准的Connection字段 `Connection: keep-alive`

9.2. HTTP1.1

在 `HTTP1.1` 中，默认支持长连接（`Connection: keep-alive`），即在一个TCP连接上可以传送多个 `HTTP` 请求和响应，减少了建立和关闭连接的消耗和延迟

建立一次连接，多次请求均由这个连接完成



这样，在加载 `html` 文件的时候，文件中多个请求和响应就可以在一个连接中传输

同时，`HTTP 1.1` 还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间

同时，`HTTP1.1` 在 `HTTP1.0` 的基础上，增加更多的请求头和响应头来完善的功能，如下：

- 引入了更多的缓存控制策略，如`If-Unmodified-Since`, `If-Match`, `If-None-Match`等缓存头来控制缓存策略
- 引入`range`，允许值请求资源某个部分
- 引入`host`，实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多

个虚拟WEB站点

并且还添加了其他的请求方法：`put`、`delete`、`options` ...

9.3. HTTP2.0

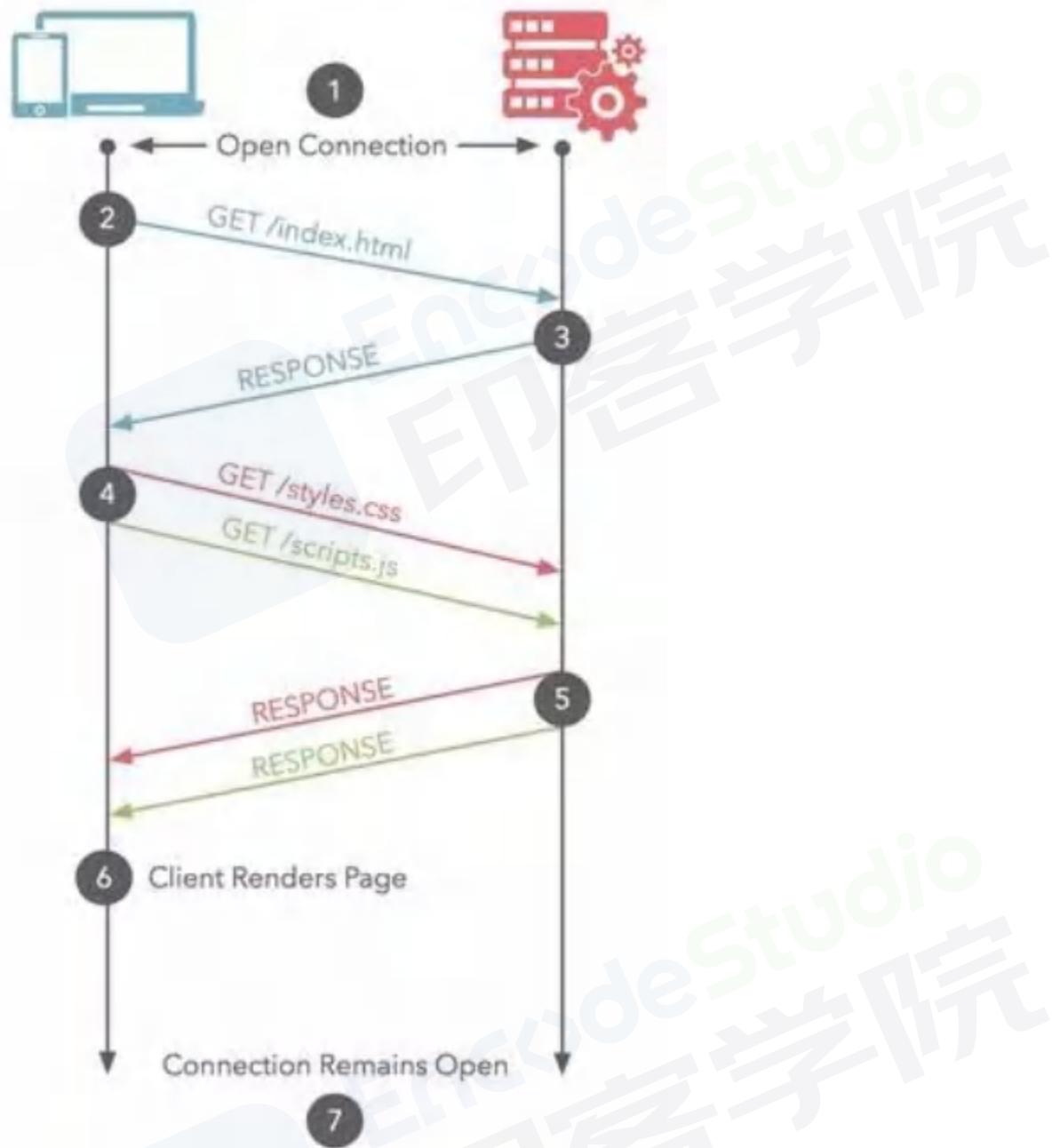
而 `HTTP2.0` 在相比之前版本，性能上有很大的提升，如添加了一个特性：

- 多路复用
- 二进制分帧
- 首部压缩
- 服务器推送

9.3.1. 多路复用

`HTTP/2` 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，而且不用按照顺序一一对应，这样就避免了“队头堵塞”

HTTP/2 Multiplexing



上图中，可以看到第四步中 `css`、`js` 资源是同时发送到服务端

9.3.2. 二进制分帧

帧是 `HTTP2` 通信中最小单位信息

`HTTP/2` 采用二进制格式传输数据，而非 `HTTP 1.x` 的文本格式，解析起来更高效

将请求和响应数据分割为更小的帧，并且它们采用二进制编码

`HTTP2` 中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流

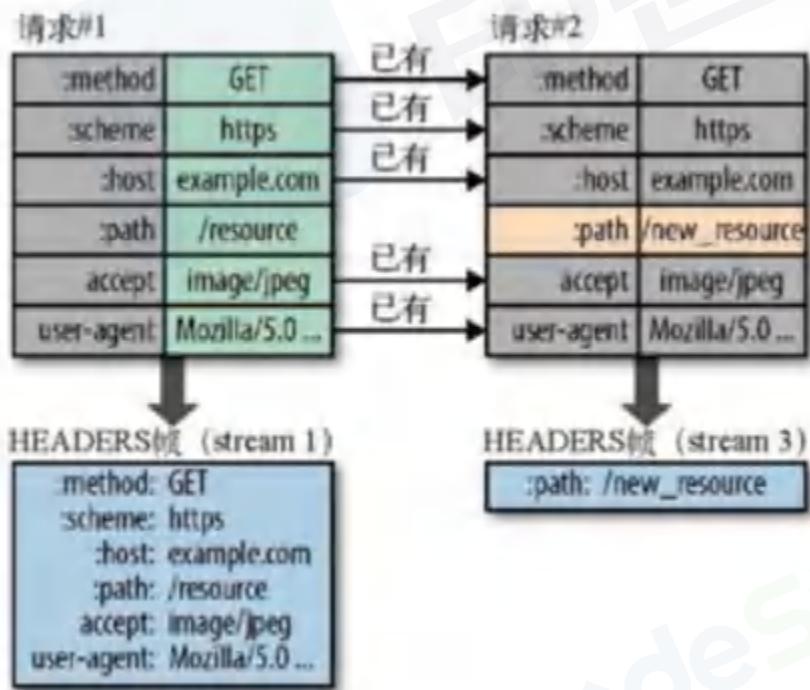
每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装，这也是多路复用同时发送数据的实现条件

9.3.3. 首部压缩

HTTP/2 在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键值对，对于相同的数据，不再通过每次请求和响应发送

首部表在 HTTP/2 的连接存续期内始终存在，由客户端和服务器共同渐进地更新

例如：下图中的两个请求， 请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销



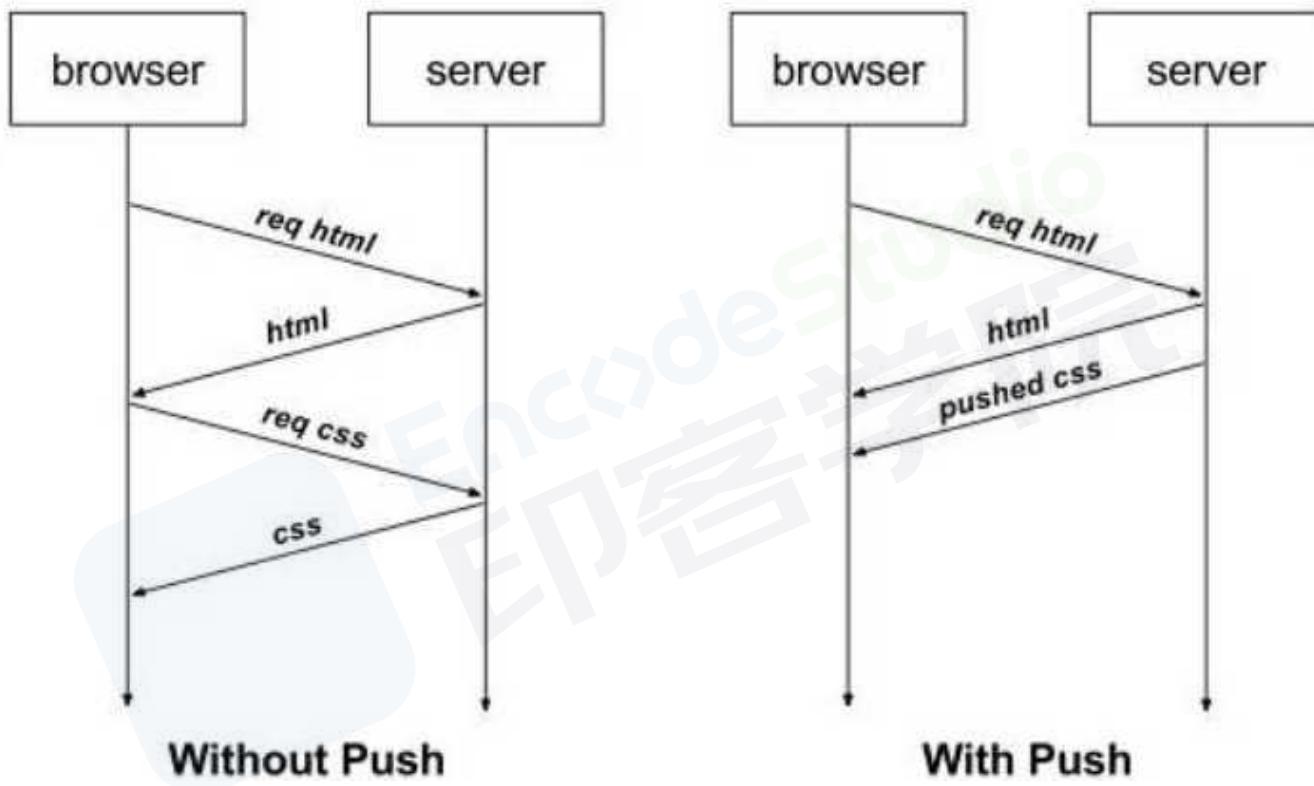
9.3.4. 服务器推送

HTTP2 引入服务器推送，允许服务端推送资源给客户端

服务器会顺便把一些客户端需要的资源一起推送到客户端，如在响应一个页面请求中，就可以随同页面的其它资源

免得客户端再次创建连接发送请求到服务器端获取

这种方式非常合适加载静态资源



9.4. 总结

HTTP1.0:

- 浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接

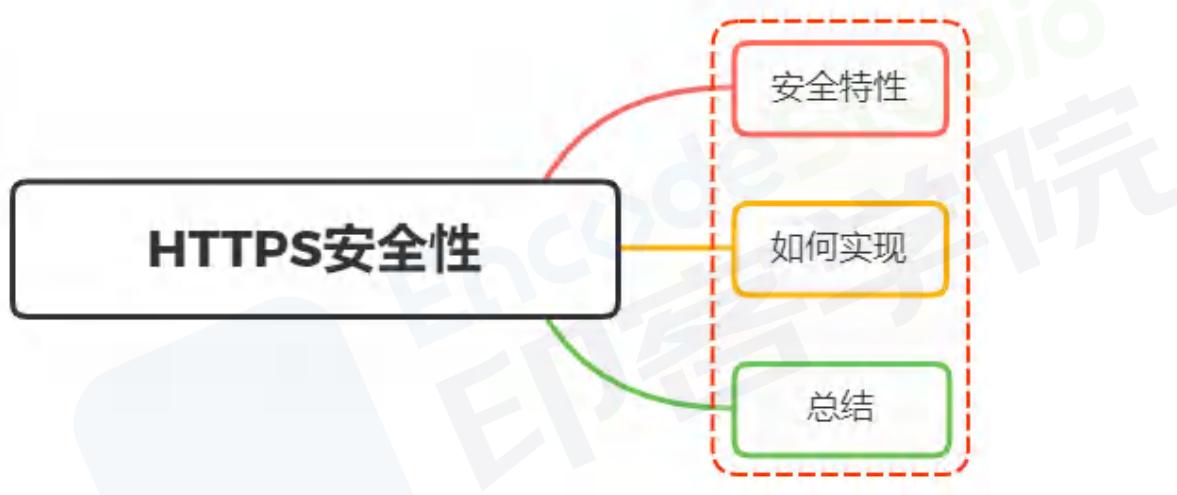
HTTP1.1:

- 引入了持久连接，即TCP连接默认不关闭，可以被多个请求复用
- 在同一个TCP连接里面，客户端可以同时发送多个请求
- 虽然允许复用TCP连接，但是同一个TCP连接里面，所有的数据通信是按次序进行的，服务器只有处理完一个请求，才会接着处理下一个请求。如果前面的处理特别慢，后面就会有许多请求排队等着
- 新增了一些请求方法
- 新增了一些请求头和响应头

HTTP2.0:

- 采用二进制格式而非文本格式
- 完全多路复用，而非有序并阻塞的、只需一个连接即可实现并行
- 使用报头压缩，降低开销
- 服务器推送

10. 为什么说HTTPS比HTTP安全？HTTPS是如何保证安全的？



10.1. 安全特性

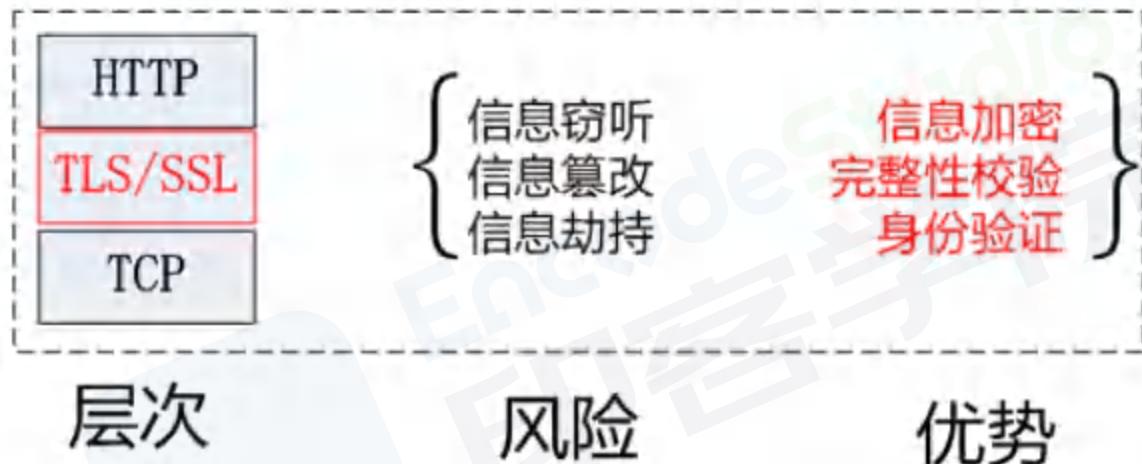
在上篇文章中，我们了解到 **HTTP** 在通信过程中，存在以下问题：

- 通信使用明文（不加密），内容可能被窃听
- 不验证通信方的身份，因此有可能遭遇伪装

而 **HTTPS** 的出现正是解决这些问题，**HTTPS** 是建立在 **SSL** 之上，其安全性由 **SSL** 来保证

在采用 **SSL** 后，**HTTP** 就拥有了 **HTTPS** 的加密、证书和完整性保护这些功能

SSL(Secure Sockets Layer 安全套接字协议),及其继任者传输层安全 (Transport Layer Security, TLS) 是为网络通信提供安全及数据完整性的一种安全协议



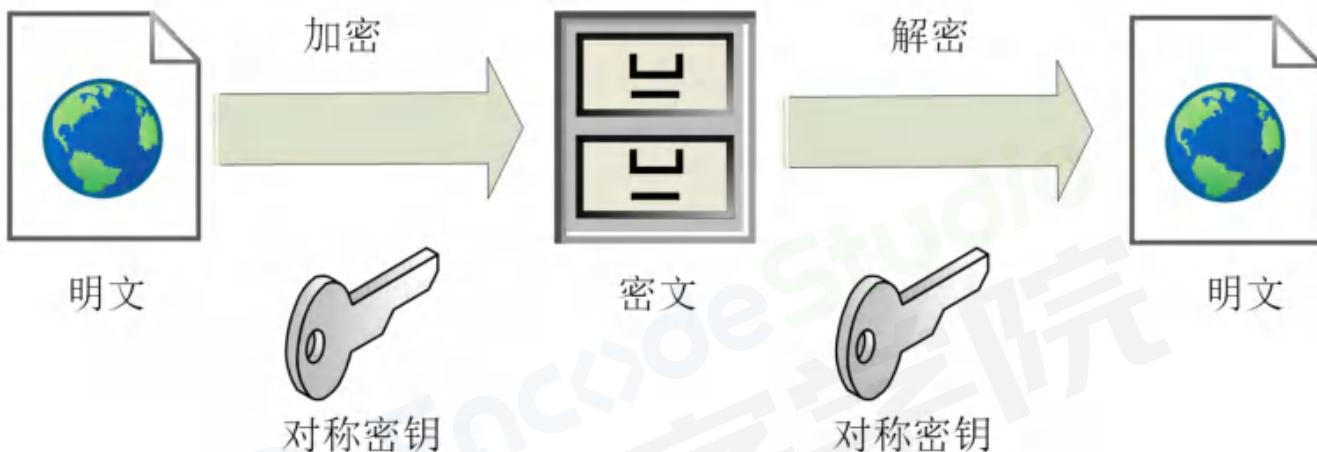
10.2. 如何做

SSL 的实现这些功能主要依赖于三种手段：

- 对称加密：采用协商的密钥对数据加密
- 非对称加密：实现身份认证和密钥协商
- 摘要算法：验证信息的完整性
- 数字签名：身份验证

10.2.1. 对称加密

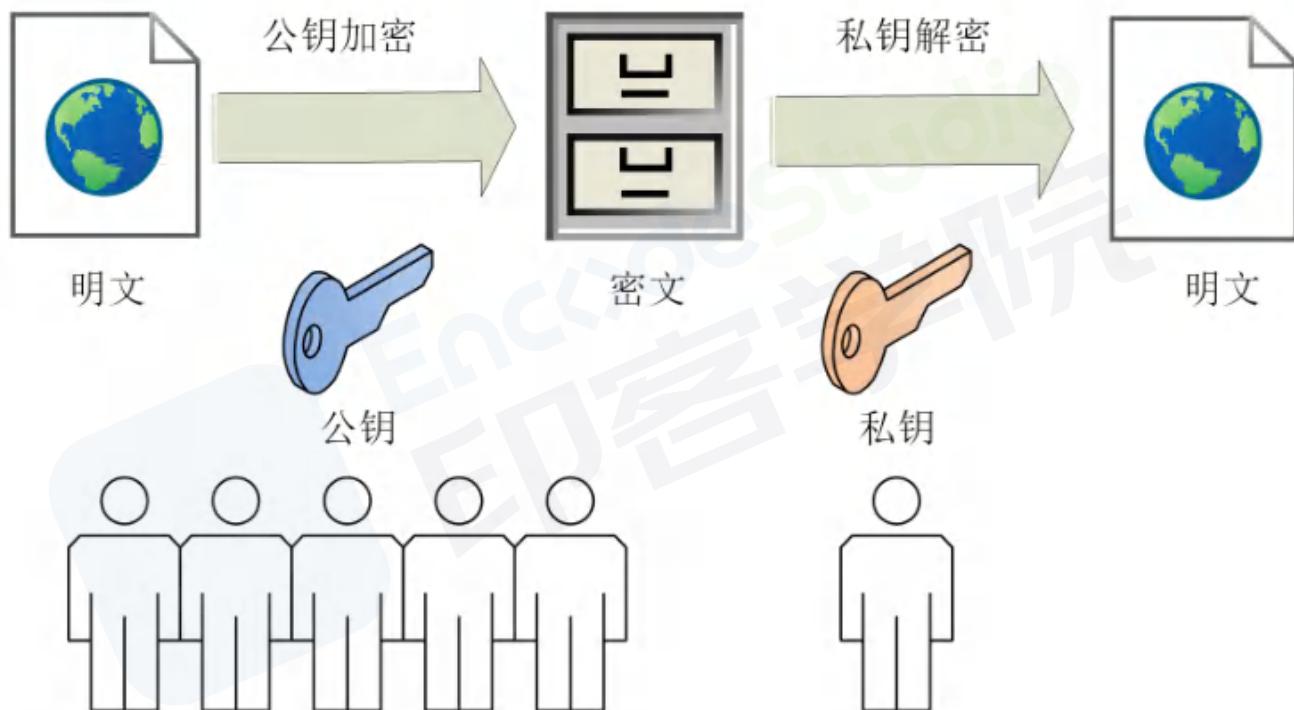
对称加密指的是加密和解密使用的秘钥都是同一个，是对称的。只要保证了密钥的安全，那整个通信过程就可以说具有了机密性



10.2.2. 非对称加密

非对称加密，存在两个秘钥，一个叫公钥，一个叫私钥。两个秘钥是不同的，公钥可以公开给任何人使用，私钥则需要保密

公钥和私钥都可以用来加密解密，但公钥加密后只能用私钥解密，反过来，私钥加密后也只能用公钥解密



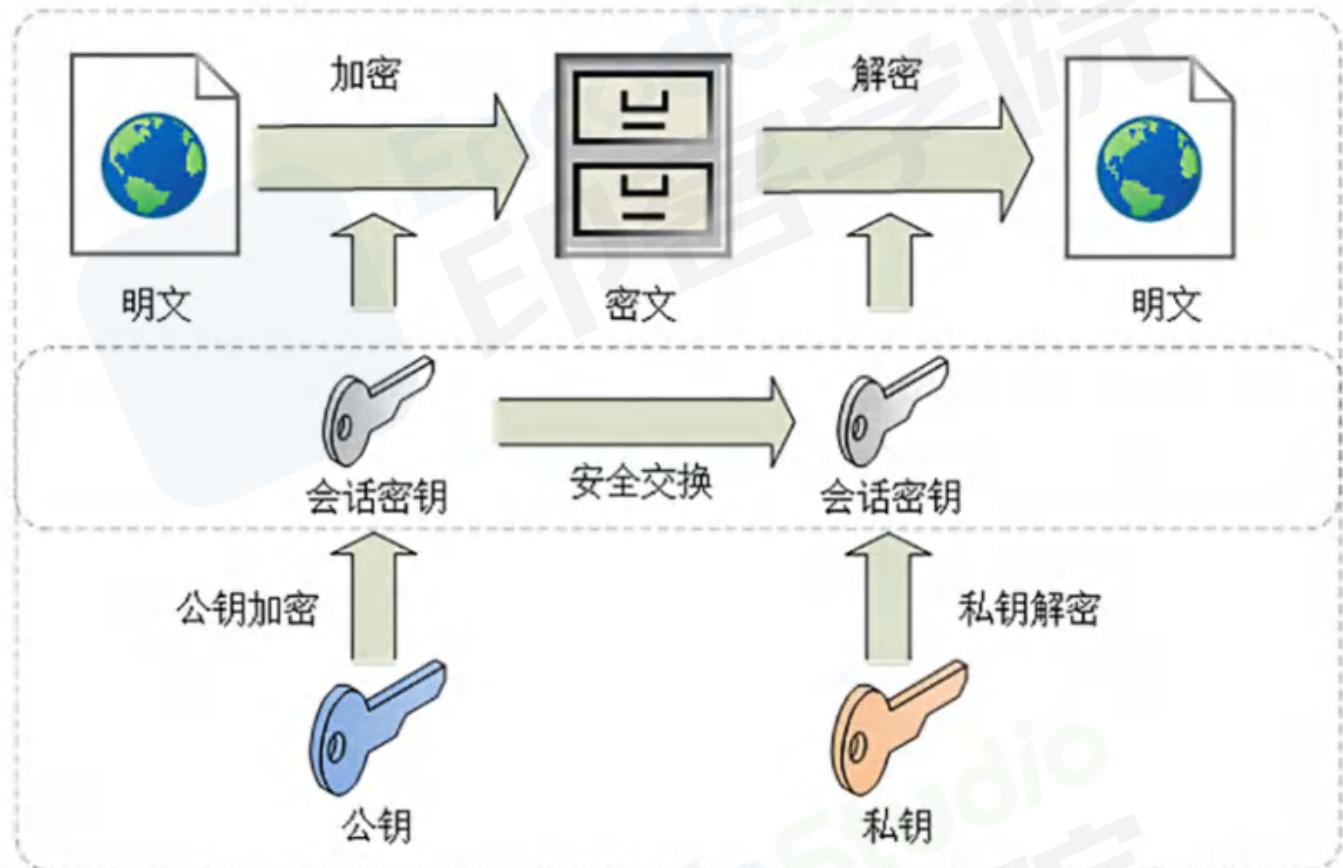
10.2.3. 混合加密

在 **HTTPS** 通信过程中，采用的是对称加密+非对称加密，也就是混合加密。

在对称加密中讲到，如果能够保证了密钥的安全，那整个通信过程就可以说具有了机密性。

而 HTTPS 采用非对称加密解决秘钥交换的问题

具体做法是发送密文的一方使用对方的公钥进行加密处理“对称的密钥”，然后对方用自己的私钥解密拿到“对称的密钥”



这样可以确保交换的密钥是安全的前提下，使用对称加密方式进行通信。

10.2.3.1. 举个例子

网站秘密保管私钥，在网上任意分发公钥，你想要登录网站只要用公钥加密就行了，密文只能由私钥持有者才能解密。而黑客因为没有私钥，所以就无法破解密文

上述的方法解决了数据加密，在网络传输过程中，数据有可能被篡改，并且黑客可以伪造身份发布公钥，如果你获取到假的公钥，那么混合加密也并无多大用处，你的数据仍被黑客解决

因此，在上述加密的基础上仍需加上完整性、身份验证的特性，来实现真正的安全，实现这一功能则是摘要算法

10.2.4. 摘要算法

实现完整性的手段主要是摘要算法，也就是常说的散列函数、哈希函数

可以理解成一种特殊的压缩算法，它能够把任意长度的数据“压缩”成固定长度、而且独一无二的“摘要”字符串，就好像是给这段数据生成了一个数字“指纹”



摘要算法保证了“数字摘要”和原文是完全等价的。所以，我们只要在原文后附上它的摘要，就能够保证数据的完整性

比如，你发了条消息：“转账 1000 元”，然后再加上一个 SHA-2 的摘要。网站收到后也计算一下消息的摘要，把这两份“指纹”做个对比，如果一致，就说明消息是完整可信的，没有被修改

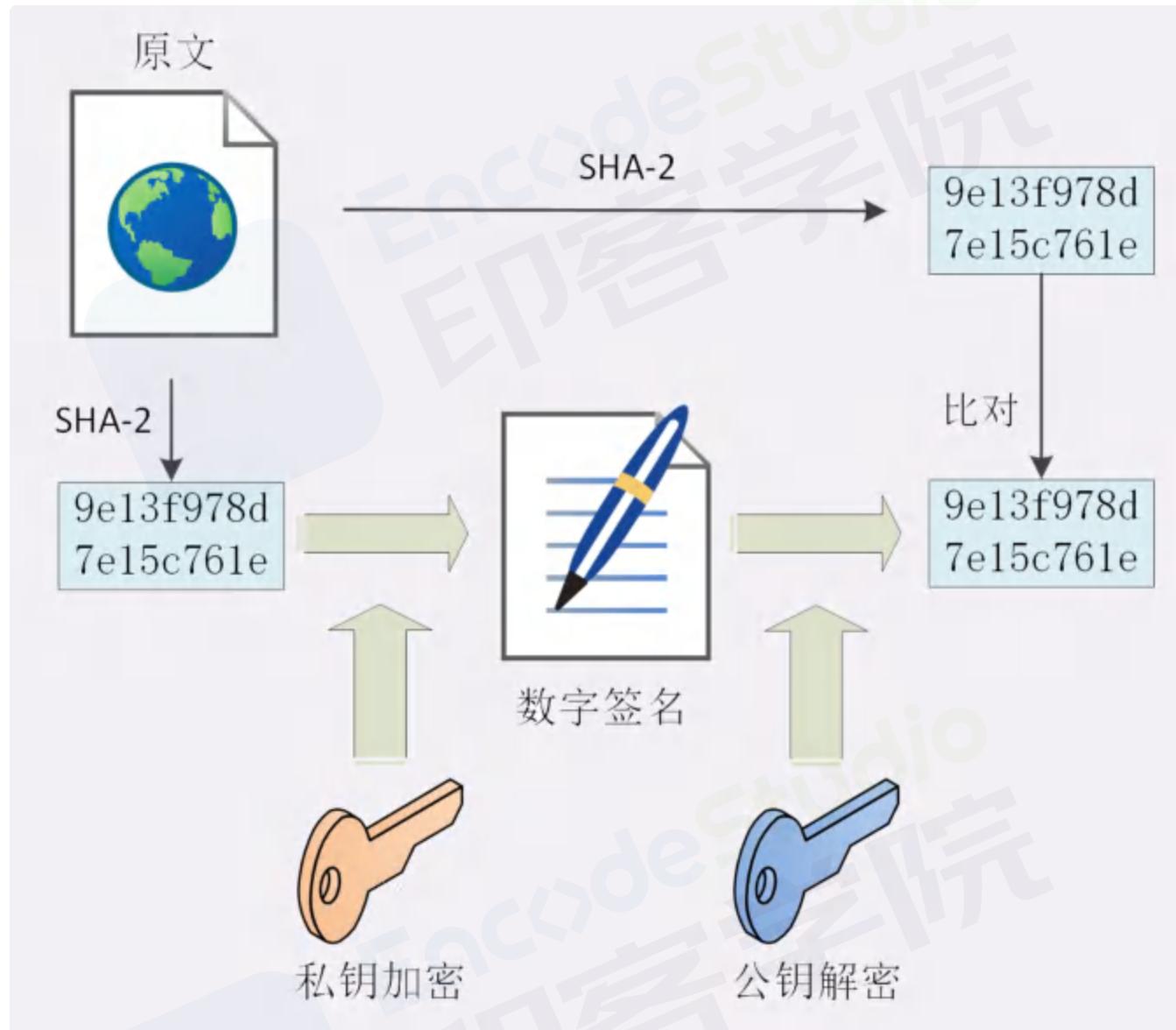


10.2.5. 数字签名

数字签名能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名

原理其实很简单，就是用私钥加密，公钥解密

签名和公钥一样完全公开，任何人都可以获取。但这个签名只有用私钥对应的公钥才能解开，拿到摘要后，再比对原文验证完整性，就可以像签署文件一样证明消息确实是你发的



和消息本身一样，因为谁都可以发布公钥，我们还缺少防止黑客伪造公钥的手段，也就是说，怎么判断这个公钥就是你的公钥

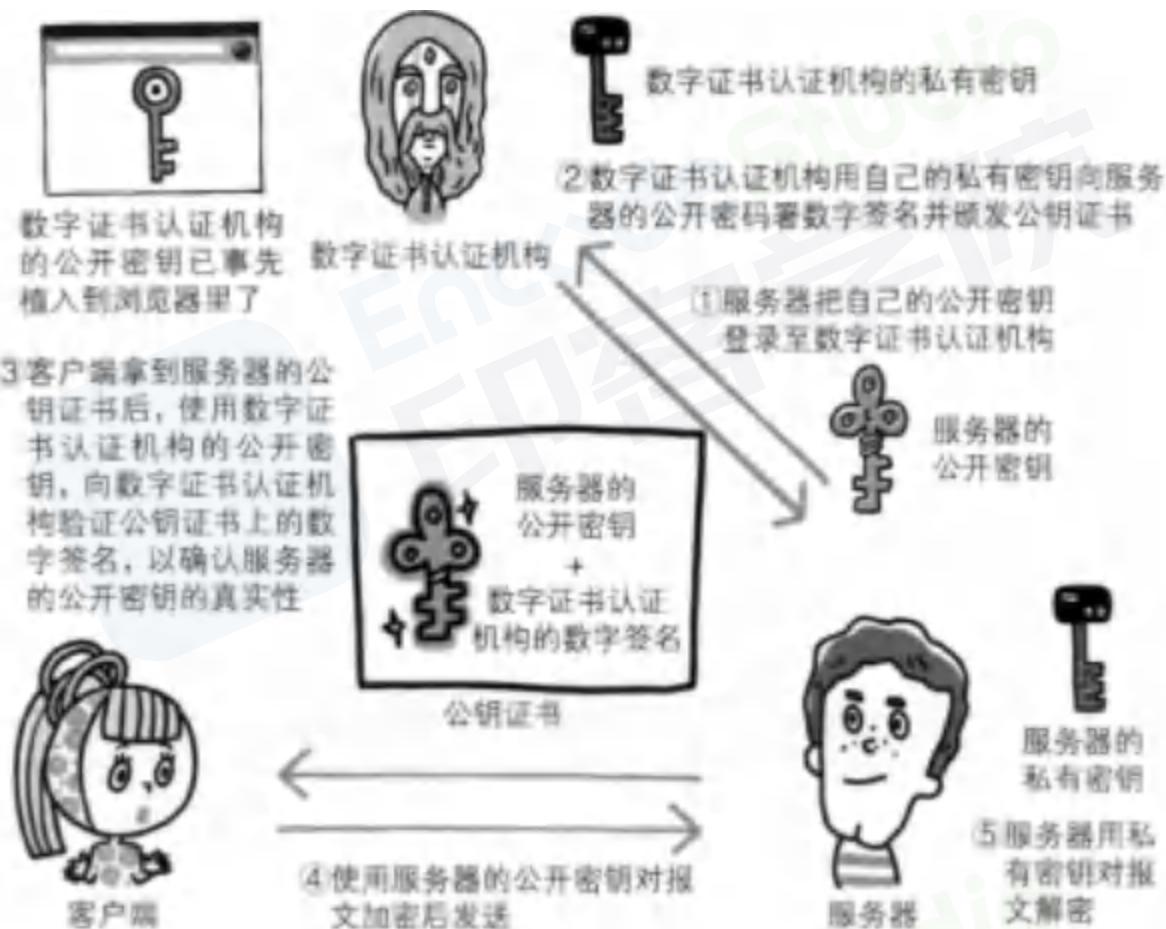
这时候就需要一个第三方，就是证书验证机构

10.2.6. CA验证机构

数字证书认证机构处于客户端与服务器双方都可信赖的第三方机构的立场

CA 对公钥的签名认证要求包括序列号、用途、颁发者、有效时间等等，把这些打成一个包再签名，完整地证明公钥关联的各种信息，形成“数字证书”

流程如下图：



- 服务器的运营人员向数字证书认证机构提出公开密钥的申请
- 数字证书认证机构在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名
- 然后分配这个已签名的公开密钥，并将该公开密钥放入公钥证书后绑定在一起
- 服务器会将这份由数字证书认证机构颁发的数字证书发送给客户端，以进行非对称加密方式通信

接到证书的客户端可使用数字证书认证机构的公开密钥，对那张证书上的数字签名进行验证，一旦验证通过，则证明

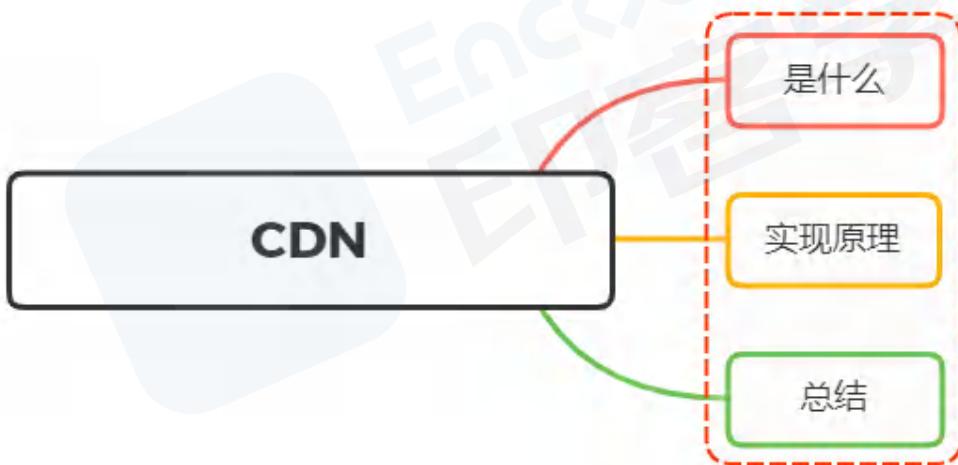
- 认证服务器的公开密钥的是真实有效的数字证书认证机构
- 服务器的公开密钥是值得信赖的

10.3. 总结

可以看到，HTTPS 与 HTTP 虽然只差一个 SSL，但是通信安全得到了大大的保障，通信的四大特性都以解决，解决方式如下：

- 机密性：混合算法
- 完整性：摘要算法
- 身份认证：数字签名
- 不可否定：数字签名

11. 如何理解CDN？说说实现原理？



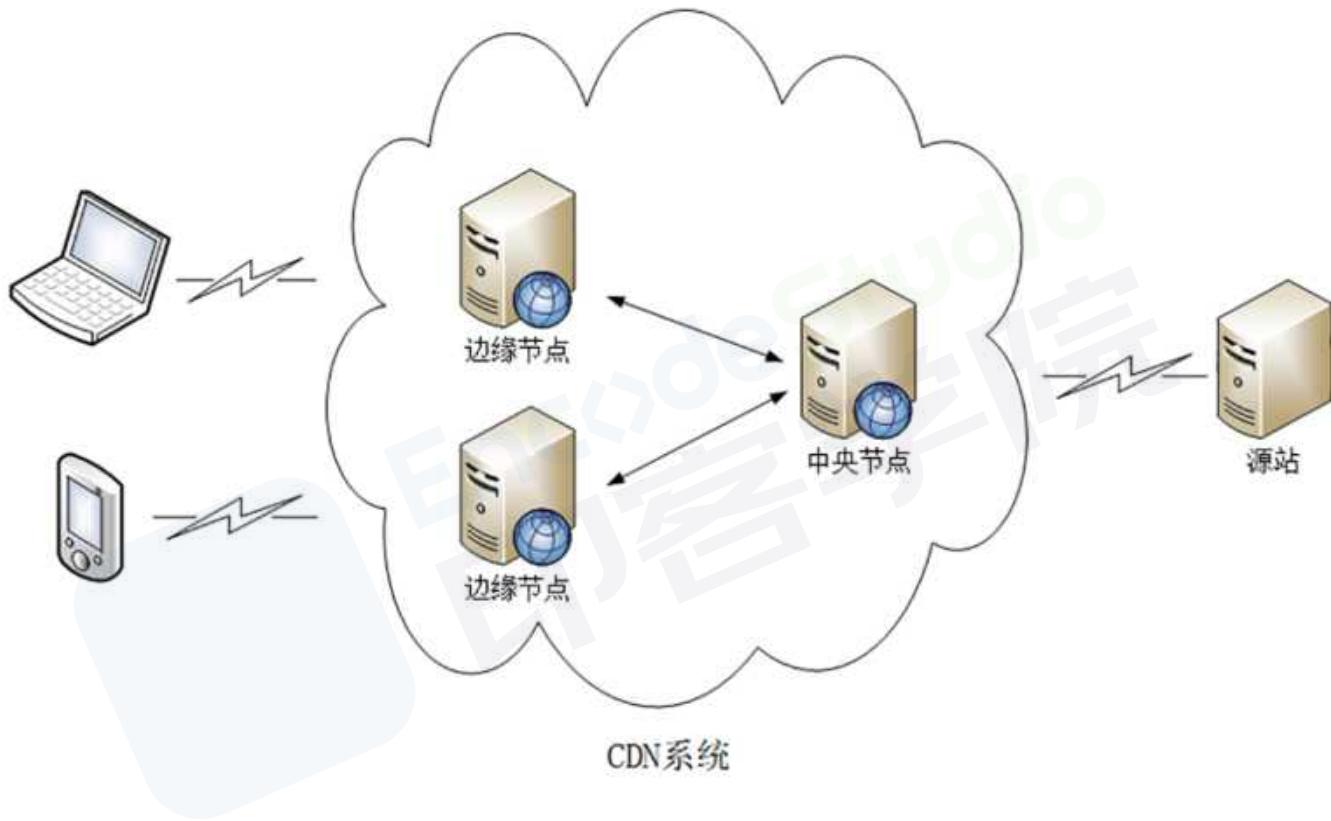
11.1. 是什么

CDN (全称 Content Delivery Network)，即内容分发网络

构建在现有网络基础之上的智能虚拟网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN 的关键技术主要有内容存储和分发技术

简单来讲，CDN 就是根据用户位置分配最近的资源

于是，用户在上网的时候不用直接访问源站，而是访问离他“最近的”一个 CDN 节点，术语叫边缘节点，其实就是缓存了源站内容的代理服务器。如下图：



11.2. 原理分析

在没有应用 **CDN** 时，我们使用域名访问某一个站点时的路径为

用户提交域名 → 浏览器对域名进行解释 → **DNS** 解析得到目的主机的IP地址 → 根据IP地址访问发出请求 → 得到请求数据并回复

应用 **CDN** 后，**DNS** 返回的不再是 **IP** 地址，而是一个 **CNAME** (Canonical Name) 别名记录，指向 **CDN** 的全局负载均衡

CNAME 实际上在域名解析的过程中承担了中间人（或者说代理）的角色，这是 **CDN** 实现的关键

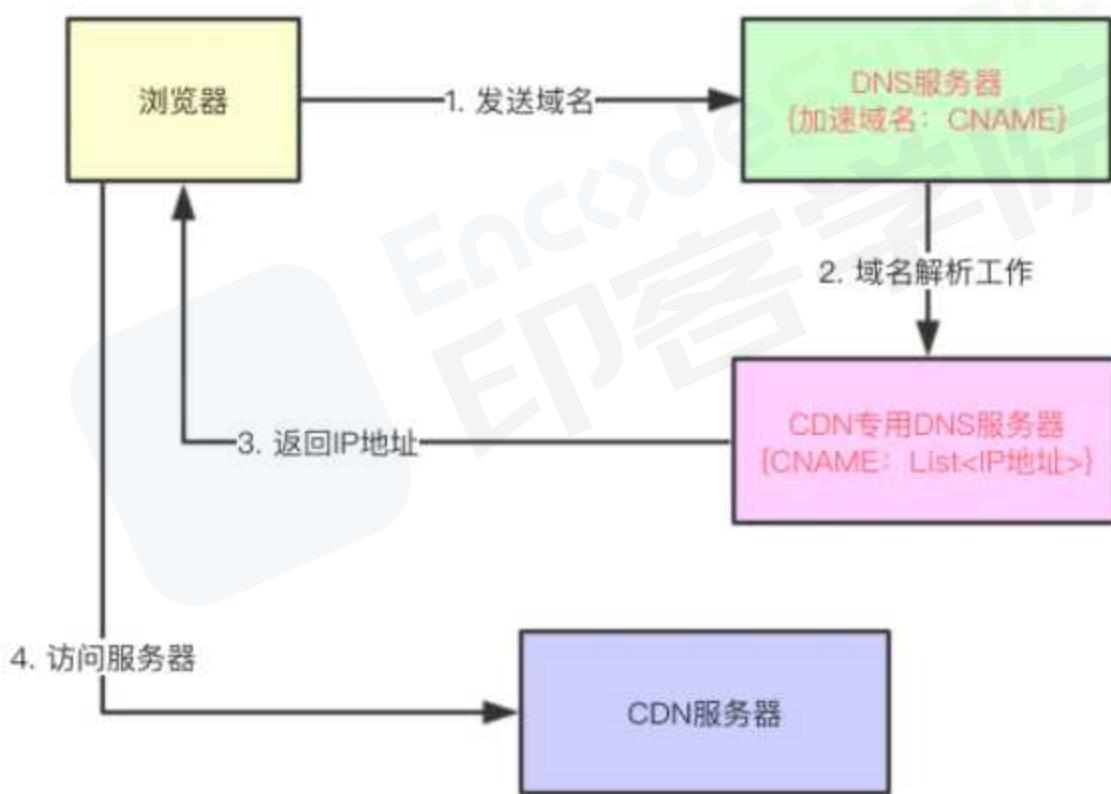
11.2.1.1. 负载均衡系统

由于没有返回 **IP** 地址，于是本地 **DNS** 会向负载均衡系统再发送请求，则进入到 **CDN** 的全局负载均衡系统进行智能调度：

- 看用户的 IP 地址，查表得知地理位置，找相对最近的边缘节点
- 看用户所在的运营商网络，找相同网络的边缘节点
- 检查边缘节点的负载情况，找负载较轻的节点
- 其他，比如节点的“健康状况”、服务能力、带宽、响应时间等

结合上面的因素，得到最合适的边缘节点，然后把这个节点返回给用户，用户就能够就近访问 **CDN** 的缓存代理

整体流程如下图：



11.2.1.2. 缓存代理

缓存系统是 **CDN** 的另一个关键组成部分，缓存系统会有选择地缓存那些最常用的那些资源

其中有两个衡量 **CDN** 服务质量的指标：

- 命中率：用户访问的资源恰好在缓存系统里，可以直接返回给用户，命中次数与所有访问次数之比
- 回源率：缓存里没有，必须用代理的方式回源站取，回源次数与所有访问次数之比

缓存系统也可以划分出层次，分成一级缓存节点和二级缓存节点。一级缓存配置高一些，直连源站，二级缓存配置低一些，直连用户

回源的时候二级缓存只找一级缓存，一级缓存没有才回源站，可以有效地减少真正的回源

现在的商业 **CDN** 命中率都在 90% 以上，相当于把源站的服务能力放大了 10 倍以上

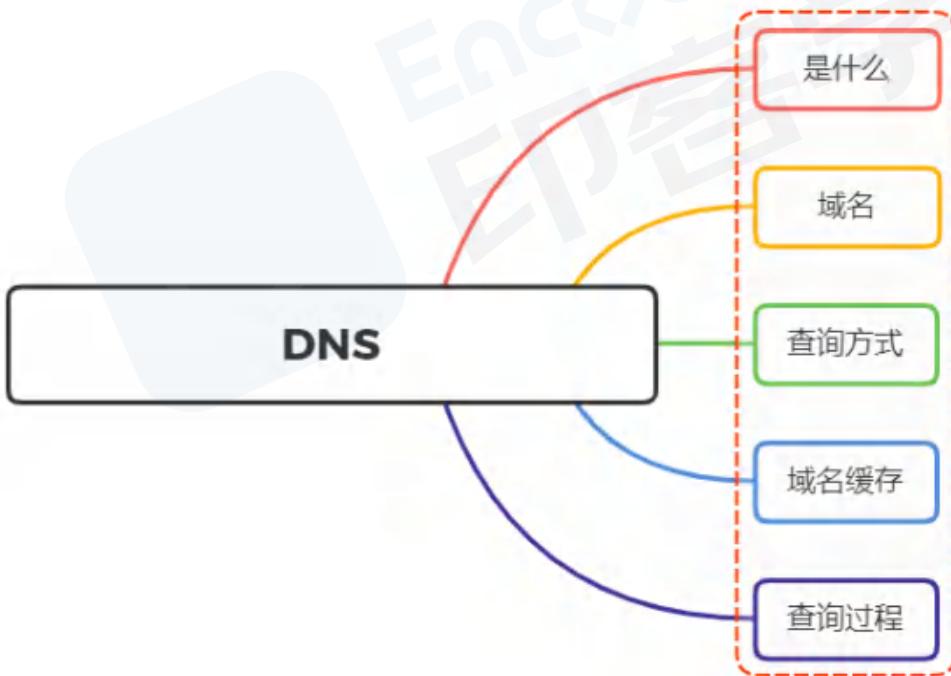
11.3. 总结

CDN 目的是为了改善互联网的服务质量，通俗一点说其实就是提高访问速度

CDN 构建了全国、全球级别的专网，让用户就近访问专网里的边缘节点，降低了传输延迟，实现了网站加速

通过 CDN 的负载均衡系统，智能调度边缘节点提供服务，相当于 CDN 服务的大脑，而缓存系统相当于 CDN 的心脏，缓存命中直接返回给用户，否则回源

12. DNS协议 是什么？说说DNS 完整的查询过程？

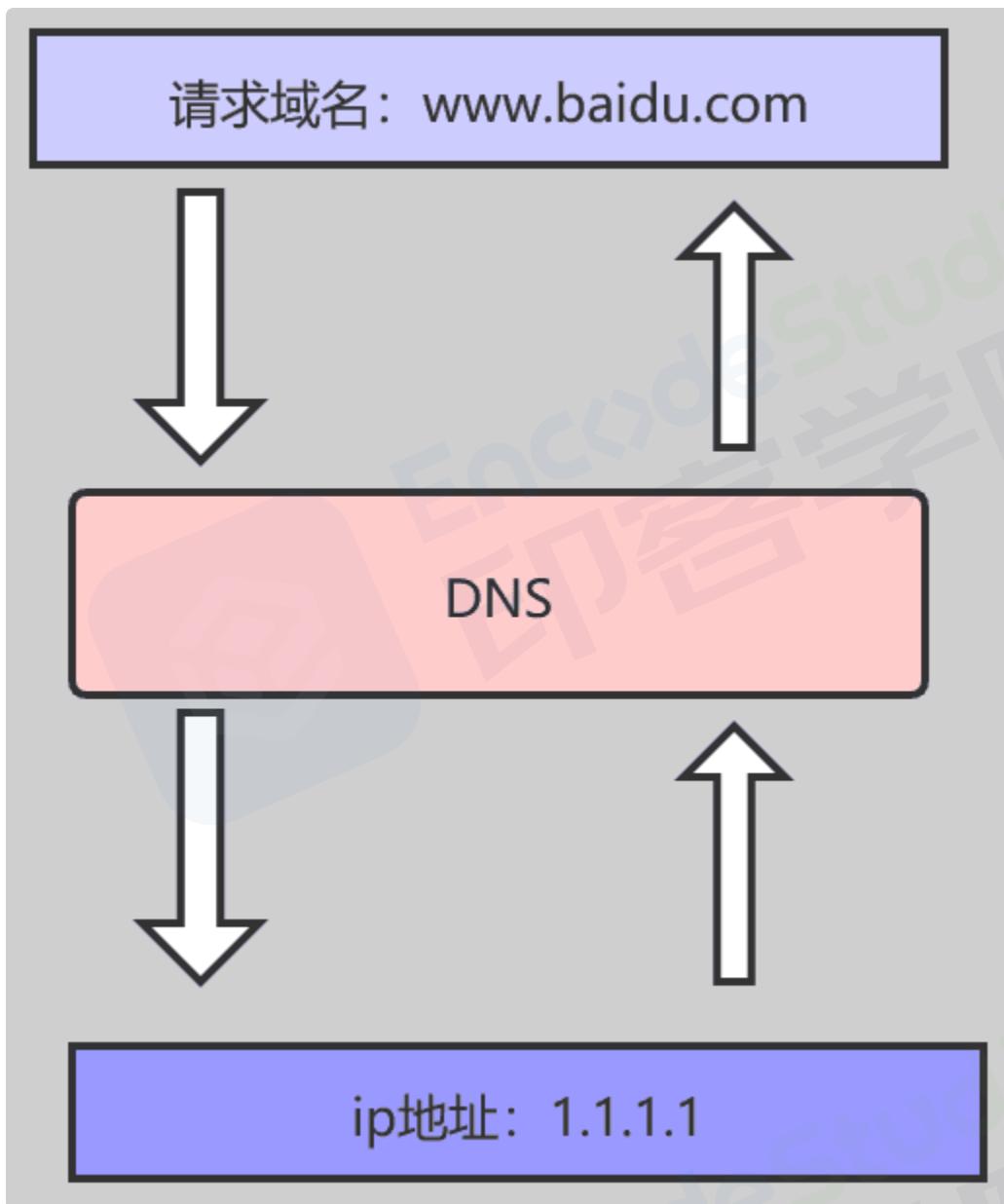


12.1. 是什么

DNS (Domain Names System)，域名系统，是互联网一项服务，是进行域名和与之相对应的 IP 地址进行转换的服务器

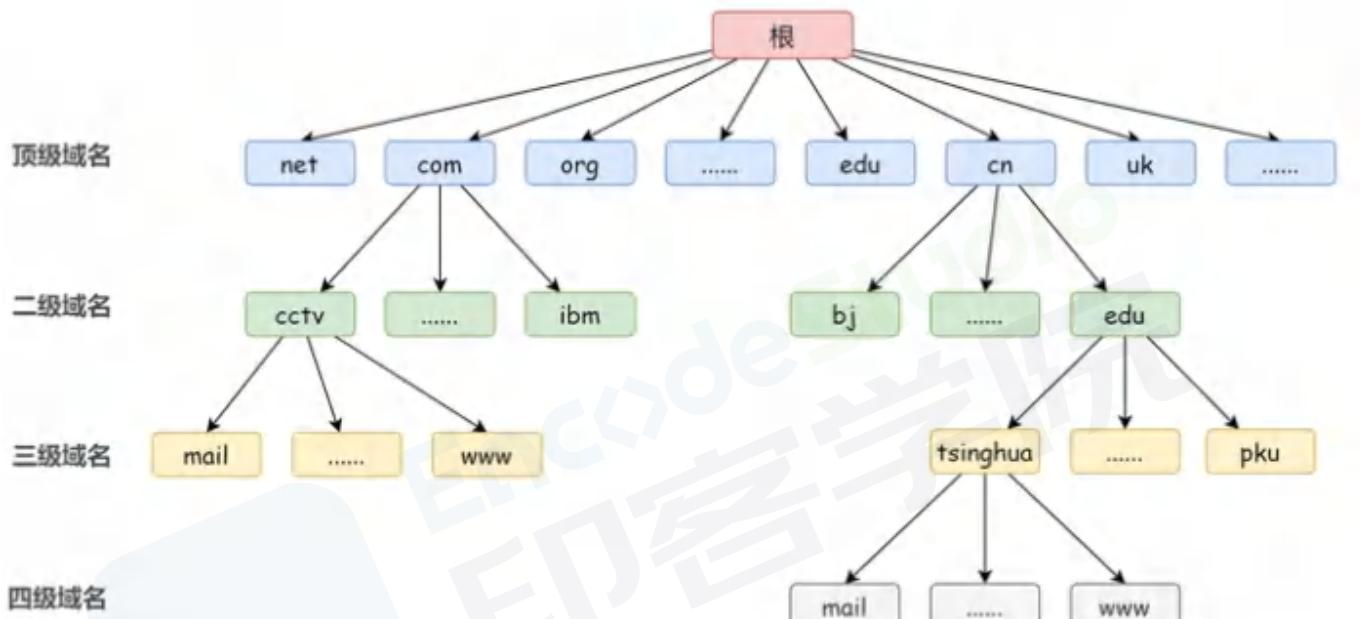
简单来讲，DNS 相当于一个翻译官，负责将域名翻译成 ip 地址

- IP 地址：一长串能够唯一地标记网络上的计算机的数字
- 域名：是由一串用点分隔的名字组成的 Internet 上某一台计算机或计算机组的名称，用于在数据传输时对计算机的定位标识



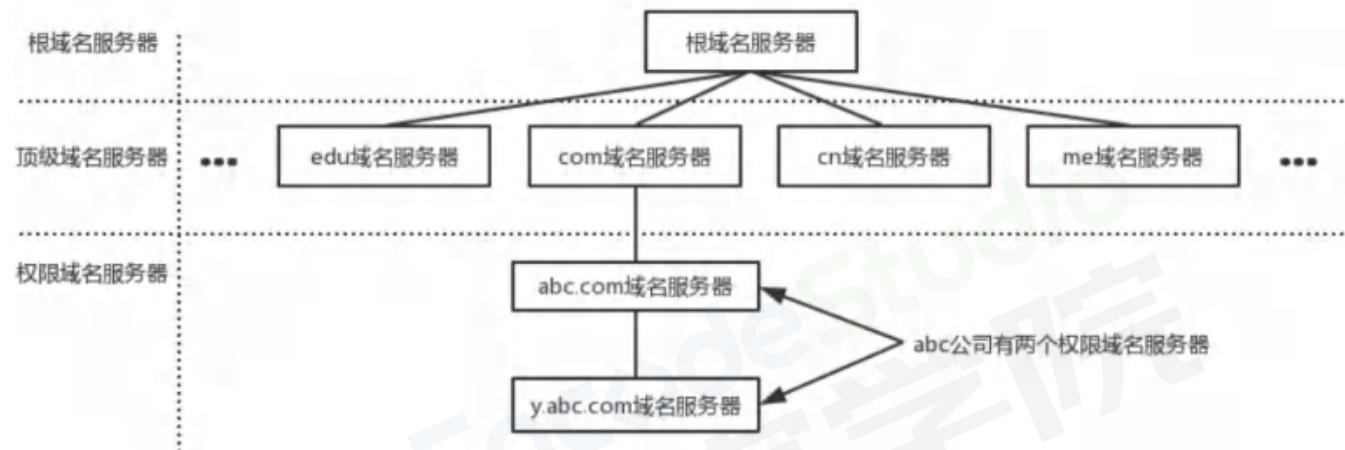
12.2. 二、域名

域名是一个具有层次的结构，从上到下依次为根域名、顶级域名、二级域名、三级域名...



例如 `www.xxx.com`，`www` 为三级域名、`xxx` 为二级域名、`com` 为顶级域名，系统为用户做了兼容，域名末尾的根域名 `.` 一般不需要输入

在域名的每一层都会有一个域名服务器，如下图：

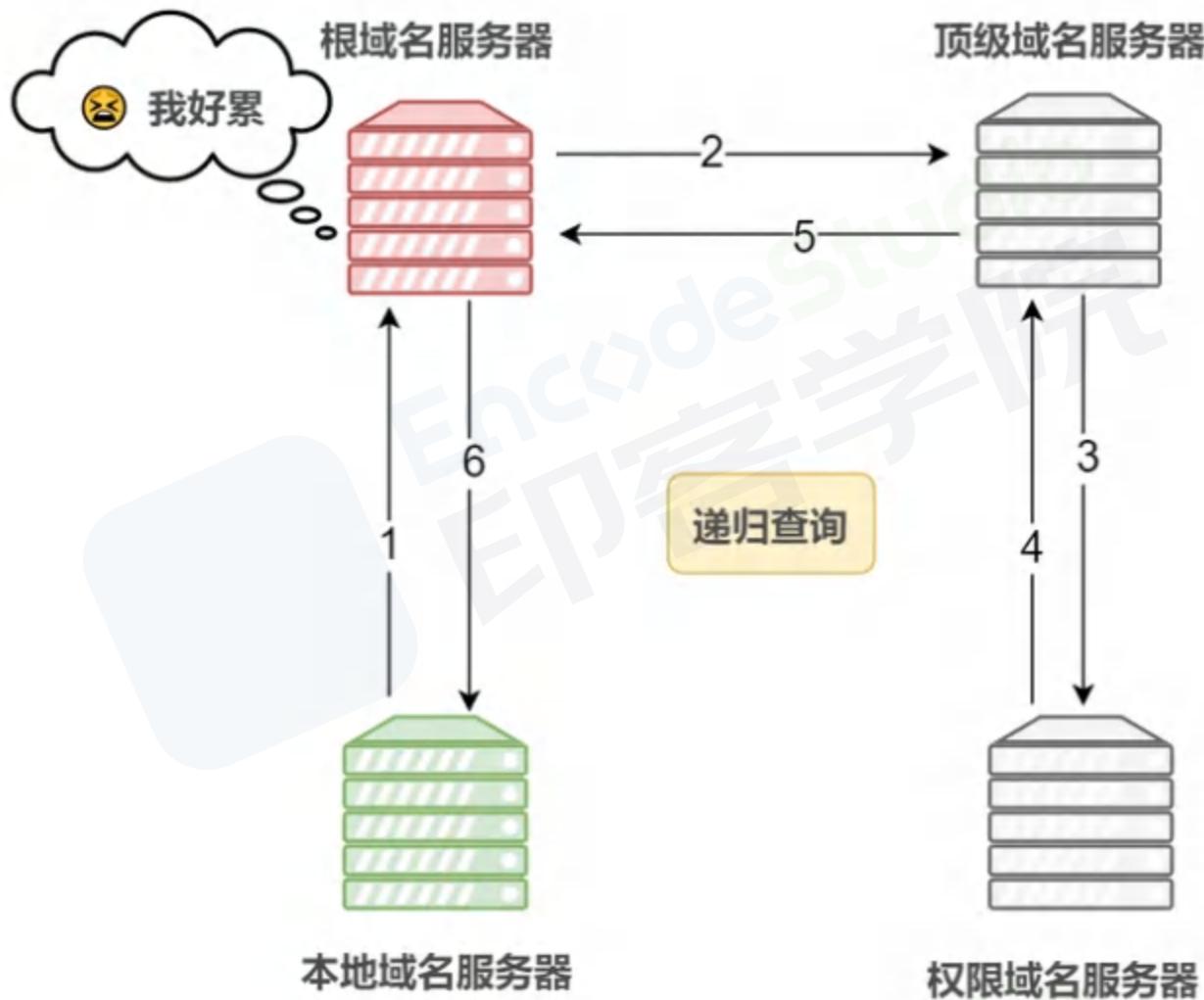


除此之外，还有电脑默认的本地域名服务器

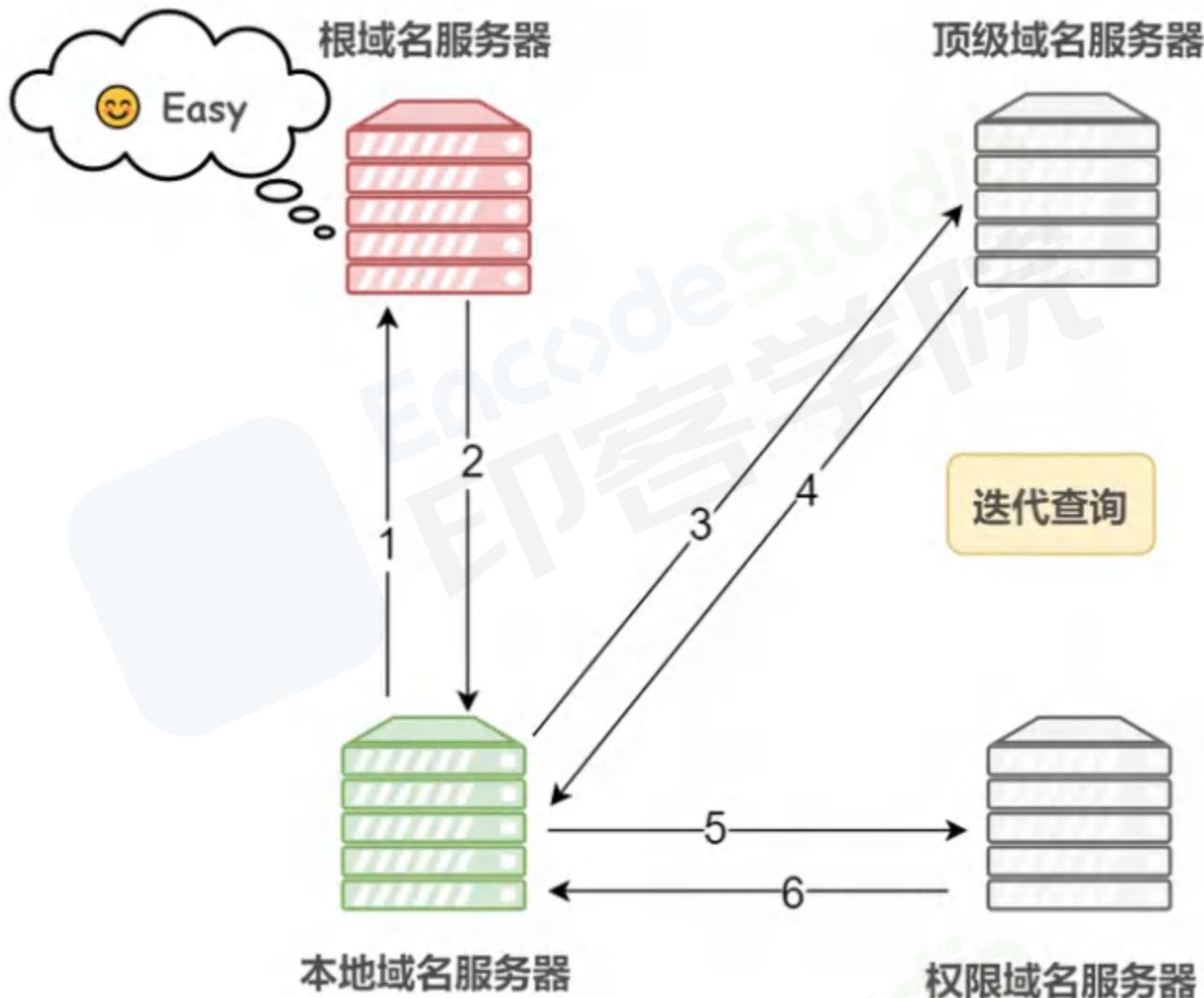
12.3. 查询方式

DNS 查询的方式有两种：

- 递归查询：如果 A 请求 B，那么 B 作为请求的接收者一定要给 A 想要的答案



- 迭代查询：如果接收者 B 没有请求者 A 所需要的准确内容，接收者 B 将告诉请求者 A，如何去获得这个内容，但是自己并不去发出请求



12.4. 域名缓存

在域名服务器解析的时候，使用缓存保存域名和 IP 地址的映射

计算机中 DNS 的记录也分成了两种缓存方式：

- 浏览器缓存：浏览器在获取网站域名的实际 IP 地址后会对其进行缓存，减少网络请求的损耗
- 操作系统缓存：操作系统的缓存其实是用户自己配置的 hosts 文件

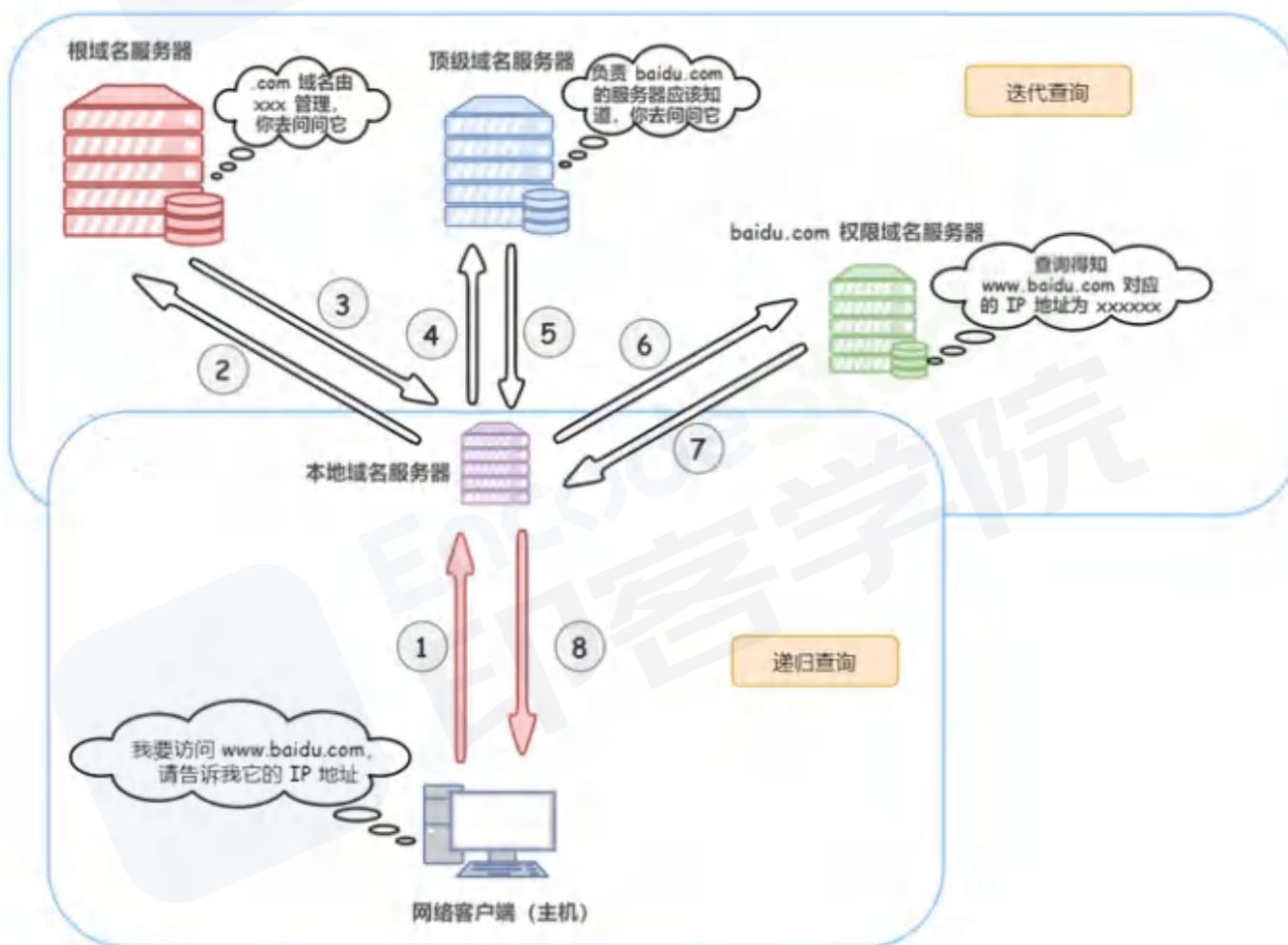
12.5. 查询过程

解析域名的过程如下：

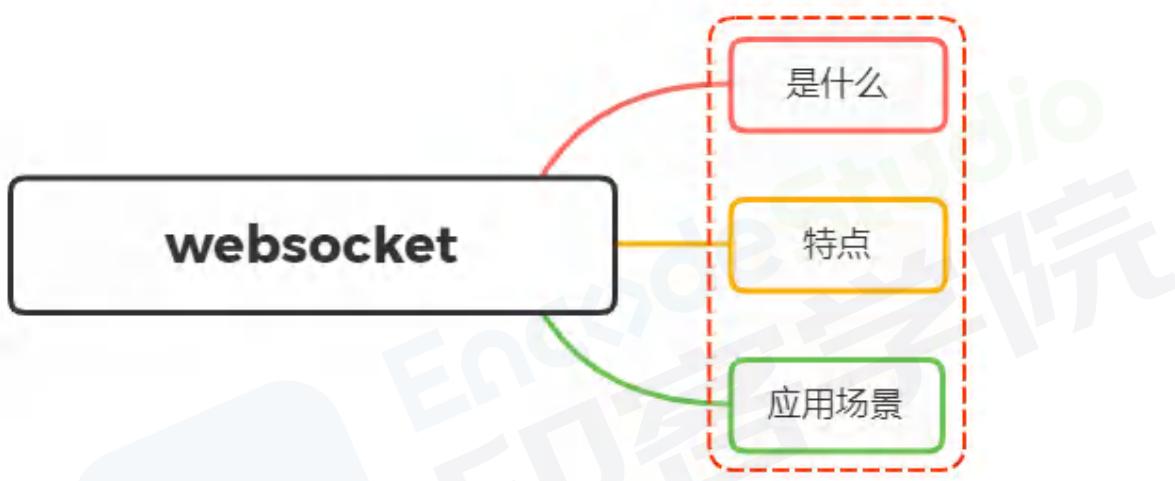
- 首先搜索浏览器的 DNS 缓存，缓存中维护一张域名与 IP 地址的对应表

- 若没有命中，则继续搜索操作系统的 DNS 缓存
- 若仍然没有命中，则操作系统将域名发送至本地域名服务器，本地域名服务器采用递归查询自己的 DNS 缓存，查找成功则返回结果
- 若本地域名服务器的 DNS 缓存没有命中，则本地域名服务器向上级域名服务器进行迭代查询
 - 首先本地域名服务器向根域名服务器发起请求，根域名服务器返回顶级域名服务器的地址给本地服务器
 - 本地域名服务器拿到这个顶级域名服务器的地址后，就向其发起请求，获取权限域名服务器的地址
 - 本地域名服务器根据权限域名服务器的地址向其发起请求，最终得到该域名对应的 IP 地址
- 本地域名服务器将得到的 IP 地址返回给操作系统，同时自己将 IP 地址缓存起来
- 操作系统将 IP 地址返回给浏览器，同时自己也将 IP 地址缓存起来
- 至此，浏览器就得到了域名对应的 IP 地址，并将 IP 地址缓存起来

流程如下图所示：



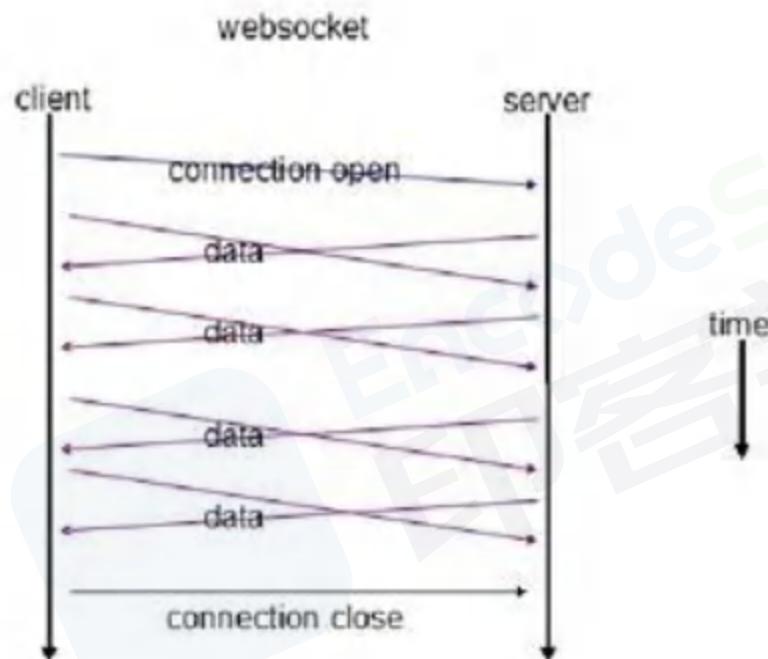
13. 说说对WebSocket的理解？应用场景？



13.1. 是什么

WebSocket，是一种网络传输协议，位于 **OSI** 模型的应用层。可在单个 **TCP** 连接上进行全双工通信，能更好的节省服务器资源和带宽并达到实时通迅

客户端和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输



从上图可见，**websocket** 服务器与客户端通过握手连接，连接成功后，两者都能主动的向对方发送或接受数据

而在 **websocket** 出现之前，开发实时 **web** 应用的方式为轮询

不停地向服务器发送 HTTP 请求，问有没有数据，有数据的话服务器就用响应报文回应。如果轮询的频率比较高，那么就可以近似地实现“实时通信”的效果

轮询的缺点也很明显，反复发送无效查询请求耗费了大量的带宽和 CPU 资源

13.2. 特点

13.2.1. 全双工

通信允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合

例如指 A→B 的同时 B→A，是瞬时同步的

13.2.2. 二进制帧

采用了二进制帧结构，语法、语义与 HTTP 完全不兼容，相比 http/2，WebSocket 更侧重于“实时通信”，而 HTTP/2 更侧重于提高传输效率，所以两者的帧结构也有很大的区别

不像 HTTP/2 那样定义流，也就不存在多路复用、优先级等特性

自身就是全双工，也不需要服务器推送

13.2.3. 协议名

引入 ws 和 wss 分别代表明文和密文的 websocket 协议，且默认端口使用80或443，几乎与 http 一致

```
1 ws://www.chrono.com
2 ws://www.chrono.com:8080/srv
3 wss://www.chrono.com:443/im?user_id=xxx
```

13.2.4. 握手

WebSocket 也要有一个握手过程，然后才能正式收发数据

客户端发送数据格式如下：

HTTP | 复制代码

```

1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: dGhlIHNBXBsZSBr25jZQ==
6 Origin: http://example.com
7 Sec-WebSocket-Protocol: chat, superchat
8 Sec-WebSocket-Version: 13

```

- Connection: 必须设置Upgrade, 表示客户端希望连接升级
- Upgrade: 必须设置Websocket, 表示希望升级到Websocket协议
- Sec-WebSocket-Key: 客户端发送的一个 base64 编码的密文, 用于简单的认证秘钥。要求服务端必须返回一个对应加密的“Sec-WebSocket-Accept”应答, 否则客户端会抛出错误, 并关闭连接
- Sec-WebSocket-Version : 表示支持的Websocket版本

服务端返回的数据格式:

HTTP | 复制代码

```

1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZBbK+xOo=Sec-WebSocket-Protocol: c
hat

```

- HTTP/1.1 101 Switching Protocols: 表示服务端接受 WebSocket 协议的客户端连接
- Sec-WebSocket-Accept: 验证客户端请求报文, 同样也是为了防止误连接。具体做法是把请求头里“Sec-WebSocket-Key”的值, 加上一个专用的 UUID, 再计算摘要

13.2.5. 优点

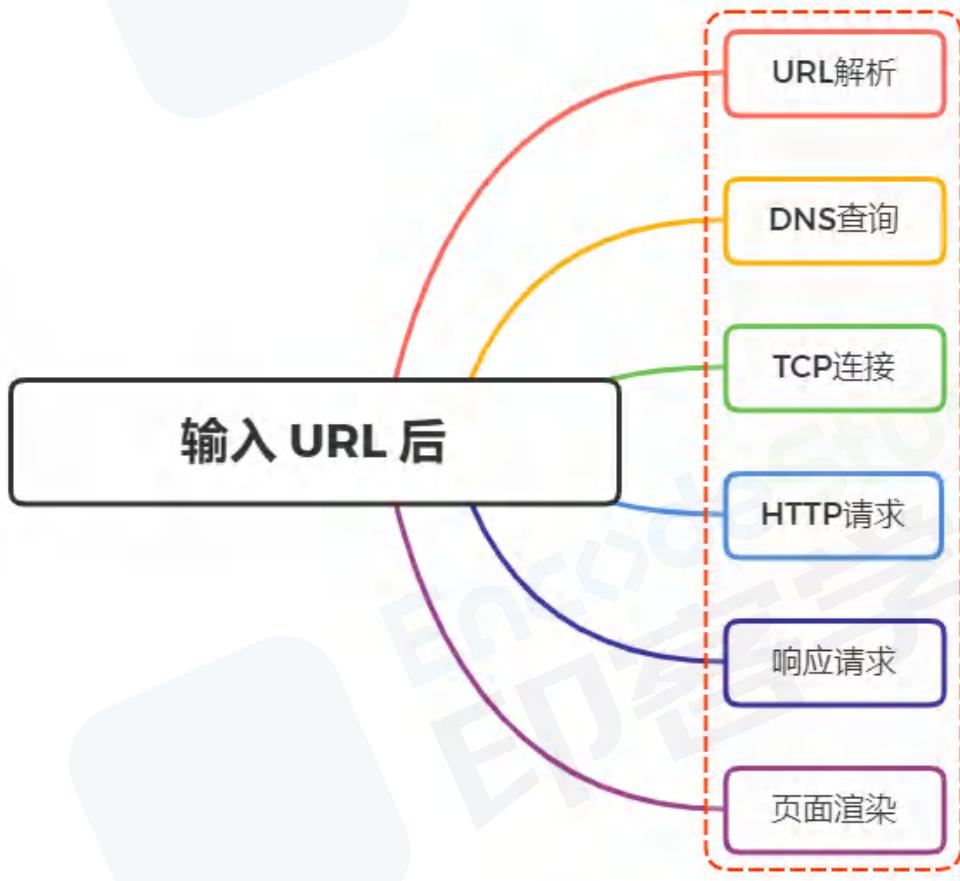
- 较少的控制开销: 数据包头部协议较小, 不同于http每次请求需要携带完整的头部
- 更强的实时性: 相对于HTTP请求需要等待客户端发起请求服务端才能响应, 延迟明显更少
- 保持长连接状态: 创建通信后, 可省略状态信息, 不同于HTTP每次请求需要携带身份验证
- 更好的二进制支持: 定义了二进制帧, 更好处理二进制内容
- 支持扩展: 用户可以扩展websocket协议、实现部分自定义的子协议
- 更好的压缩效果: Websocket在适当的扩展支持下, 可以沿用之前内容的上下文, 在传递类似的数据时, 可以显著地提高压缩率

13.3. 应用场景

基于 `websocket` 的实时通信的特点，其存在的应用场景大概有：

- 弹幕
- 媒体聊天
- 协同编辑
- 基于位置的应用
- 体育实况更新
- 股票基金报价实时更新

14. 说说地址栏输入 URL 敲下回车后发生了什么？



14.1. 简单分析

简单的分析，从输入 `URL` 到回车后发生的行为如下：

- URL解析

- DNS 查询
- TCP 连接
- HTTP 请求
- 响应请求
- 页面渲染

14.2. 详细分析

14.2.1. URL解析

首先判断你输入的是一个合法的 `URL` 还是一个待搜索的关键词，并且根据你输入的内容进行对应操作
`URL` 的解析第过程中的第一步，一个 `url` 的结构解析如下：

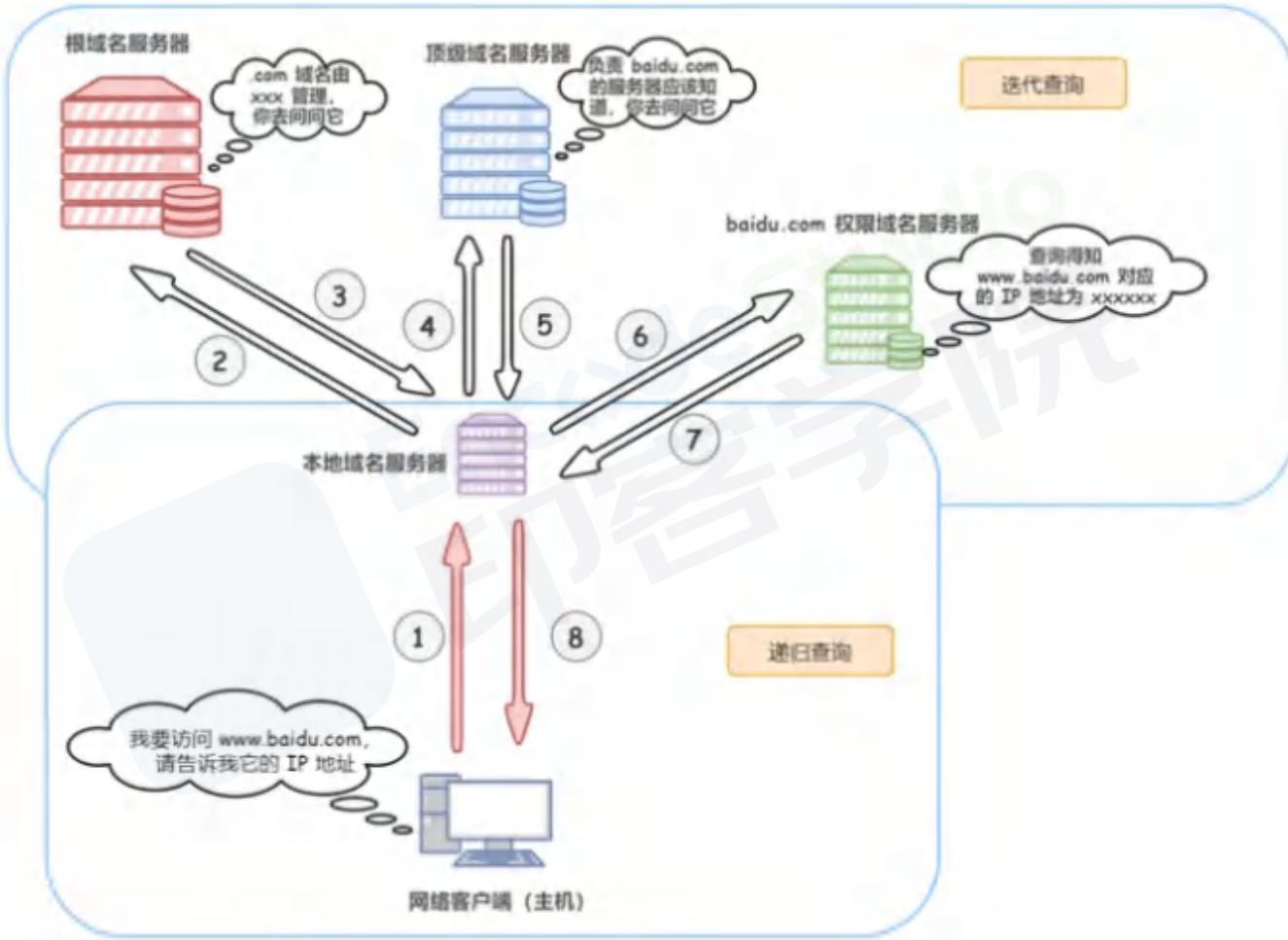


14.2.2. DNS查询

在之前文章中讲过 `DNS` 的查询，这里就不再讲述了

整个查询过程如下图所示：



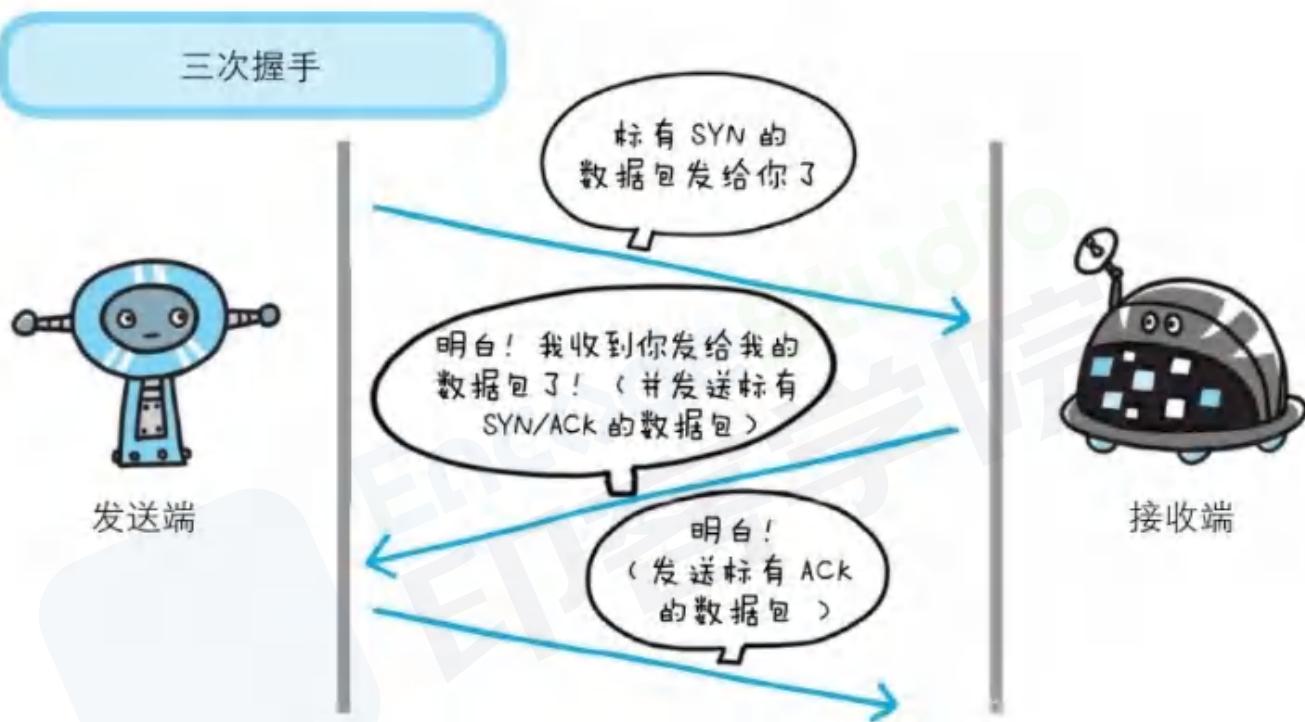


最终，获取到了域名对应的目标服务器 IP 地址

14.2.3. TCP连接

在之前文章中，了解到 `tcp` 是一种面向有连接的传输层协议

在确定目标服务器服务器的 IP 地址后，则经历三次握手建立 TCP 连接，流程如下：



14.2.4. 发送 http 请求

当建立 `tcp` 连接之后，就可以在这基础上进行通信，浏览器发送 `http` 请求到目标服务器
请求的内容包括：

- 请求行
- 请求头
- 请求主体

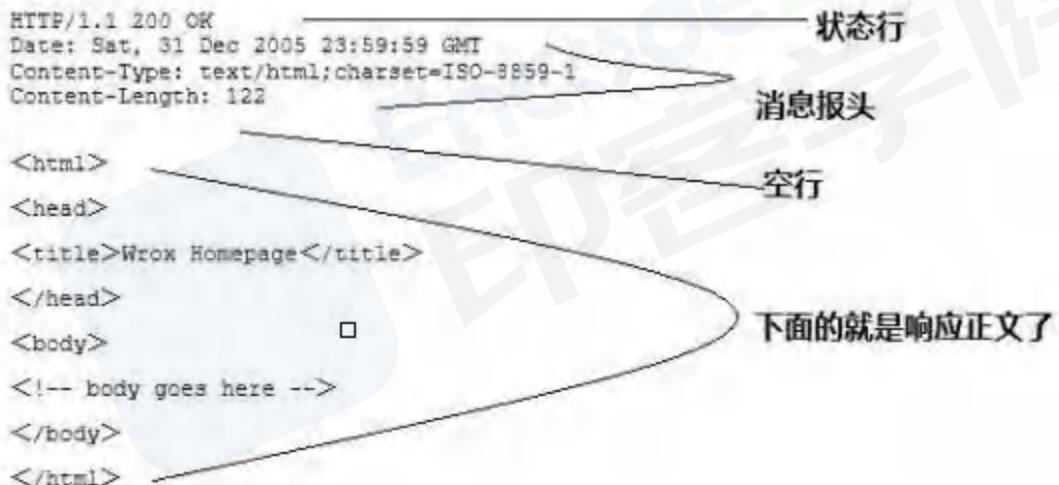
①请求方法 ②请求URL ③HTTP协议及版本

④ 报文头
 POST /chapter17/user.html HTTP/1.1
 Accept: image/jpeg, application/x-ms-application, ..., */*
 Referer: http://localhost:8088/chapter17/user/register.html?
 code=100&time=123123
 Accept-Language: zh-CN
 User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1;
 Content-Type: application/x-www-form-urlencoded
 Host: localhost:8088
 ⑤ 报文体
 Content-Length: 112
 Connection: Keep-Alive
 Cache-Control: no-cache
 Cookie: JSESSIONID=24DF2688E37EE4F66D9669D2542AC17B
 name=tom&password=1234&realName=tomson

14.2.5. 响应请求

当服务器接收到浏览器的请求之后，就会进行逻辑操作，处理完成之后返回一个 **HTTP** 响应消息，包括：

- 状态行
- 响应头
- 响应正文



```

HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 122

<html>
<head>
<title>Wrox Homepage</title>
</head>
<body>
<!-- body goes here -->
</body>
</html>
  
```

状态行

消息报头

空行

下面的就是响应正文了

在服务器响应之后，由于现在 **http** 默认开始长连接 **keep-alive**，当页面关闭之后，**tcp** 链接则会经过四次挥手完成断开

14.2.6. 页面渲染

当浏览器接收到服务器响应的资源后，首先会对资源进行解析：

- 查看响应头的信息，根据不同的指示做对应处理，比如重定向，存储cookie，解压gzip，缓存资源等等
- 查看响应头的 Content-Type的值，根据不同的资源类型采用不同的解析方式

关于页面的渲染过程如下：

- 解析HTML，构建 DOM 树
- 解析 CSS ，生成 CSS 规则树
- 合并 DOM 树和 CSS 规则，生成 render 树
- 布局 render 树（ Layout / reflow ），负责各元素尺寸、位置的计算
- 绘制 render 树（ paint ），绘制页面像素信息
- 浏览器会将各层的信息发送给 GPU，GPU 会将各层合成（ composite ），显示在屏幕上

