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

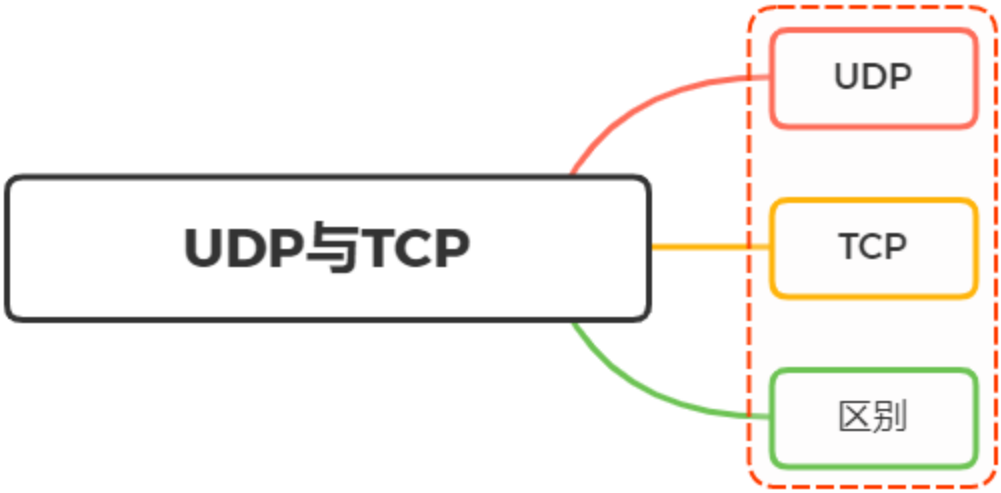
不同点：

- 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、多层交换机 | IP、ICMP、ARP |
| | 网络接口层 | 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：

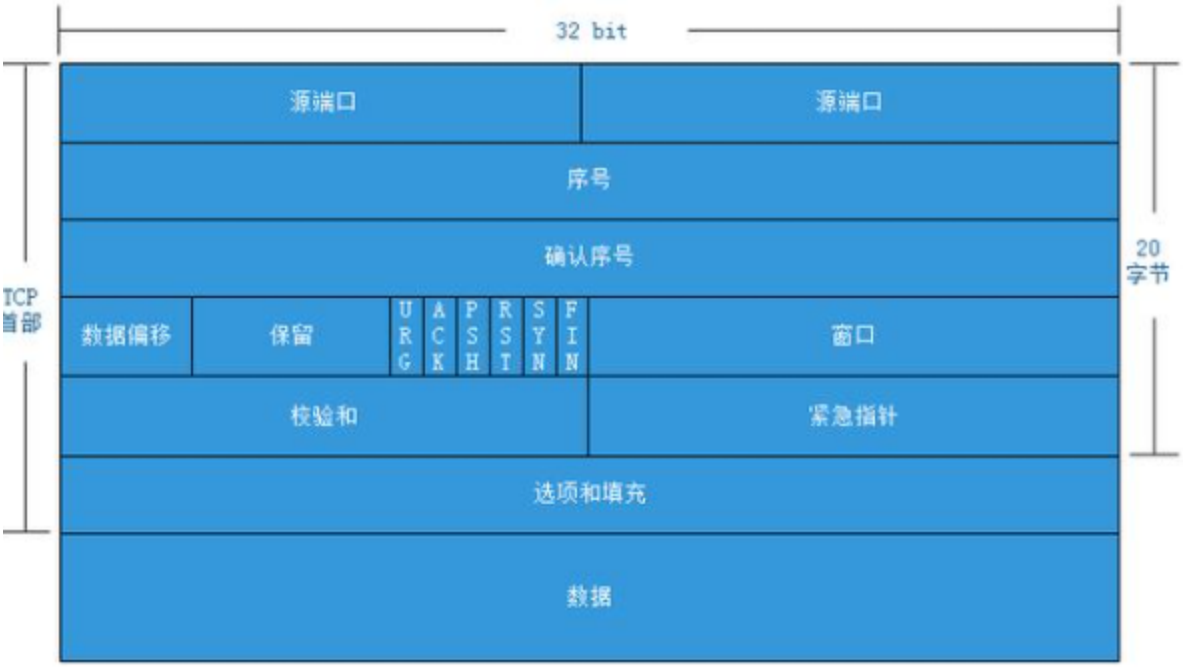
- UDP 不提供复杂的控制机制，利用 IP 提供面向无连接的通信服务
- 传输途中出现丢包，UDP 也不负责重发
- 当包的到达顺序出现乱序时，UDP没有纠正的功能。
- 并且它是将应用程序发来的数据在收到的那一刻，立即按照原样发送到网络上的一种机制。即使是出现网络拥堵的情况，UDP 也无法进行流量控制等避免网络拥塞行为

## 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

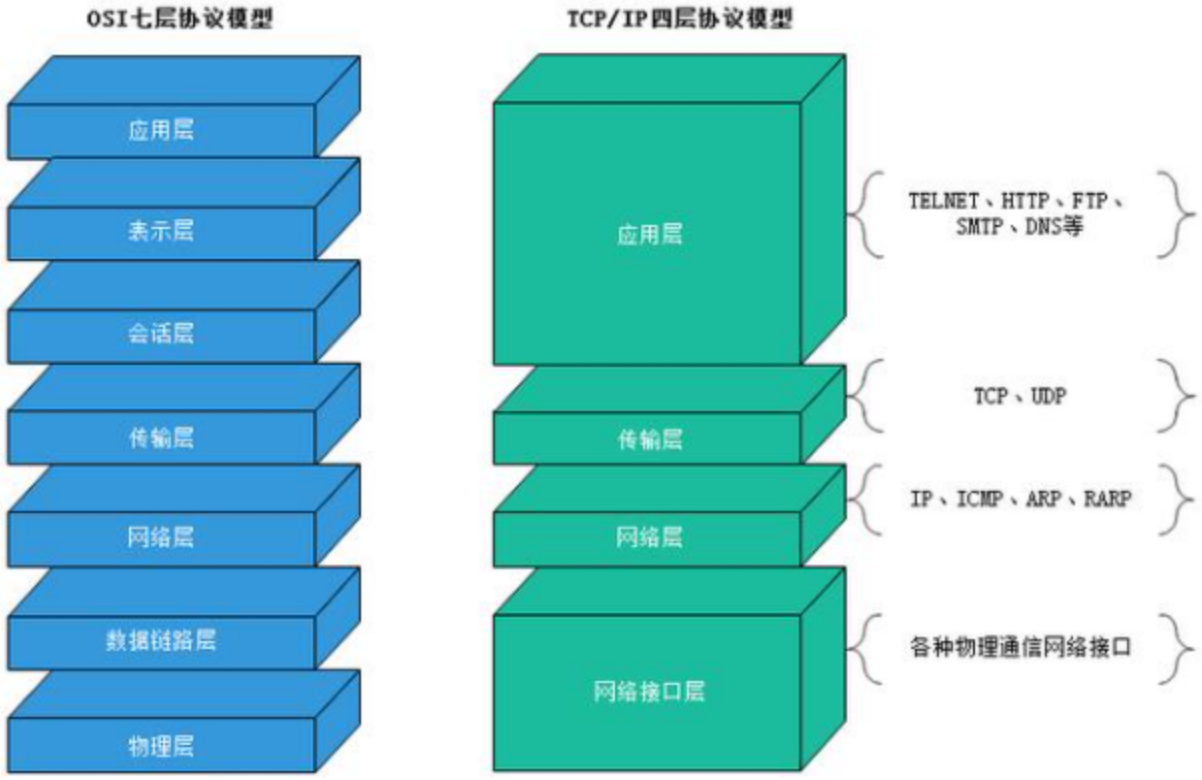



特点如下：

- 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 |
| 双共性 | 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

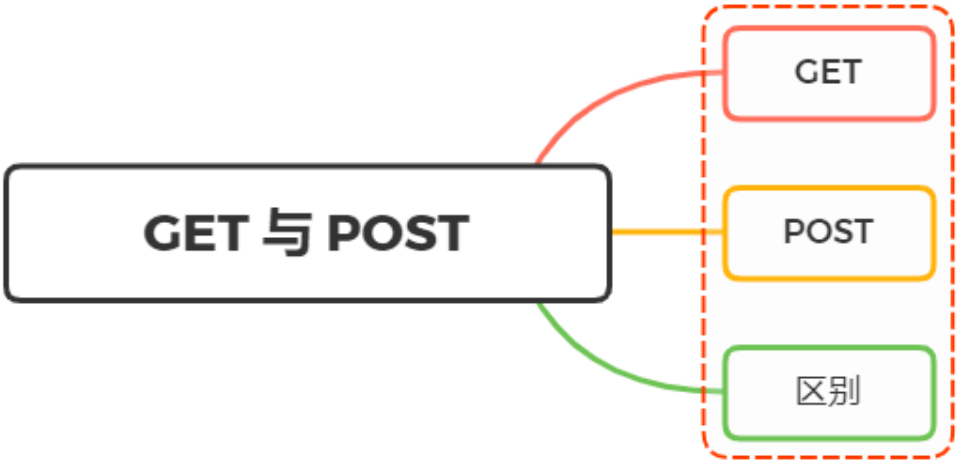
- TCP 面向字节流，将应用层报文看成一串无结构的字节流，分解为多个TCP报文段传输后，在目的站重新装配。UDP协议面向报文，不拆分应用层报文，只保留报文边界，一次发送一个报文，接收方去除报文首部后，原封不动将报文交给上层应用
- TCP 只能点对点全双工通信。UDP 支持一对一、一对多、多对一和多对多的交互通信

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 | TCP |
| TELNET | 远程终端接入 | |
| HTTP | 万维网 | |
| FTP | 文件传输 | |
| DNS | 域名转换 | UDP |
| TFTP | 文件传输 | |
| SNMP | 网络管理 | |
| 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

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

`POST` 方法简约版报文是这样的

```
POST /index.html HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

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

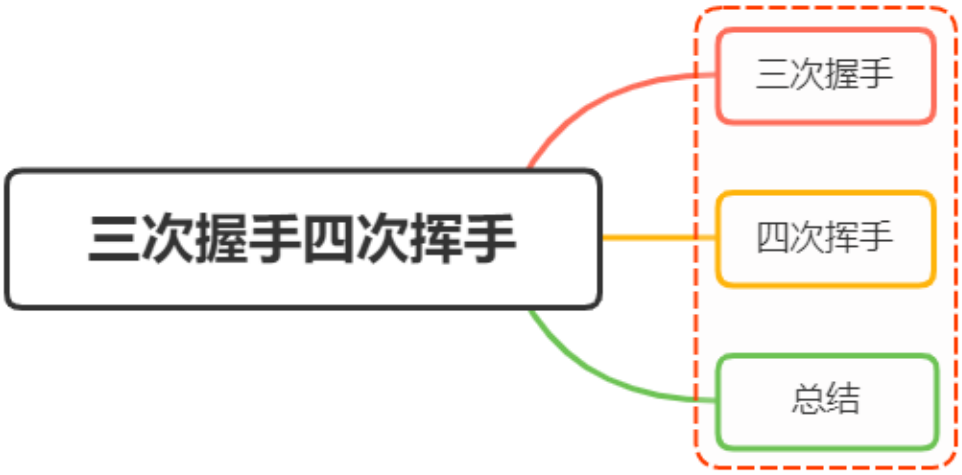
### 4.2.4. 数据包

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?



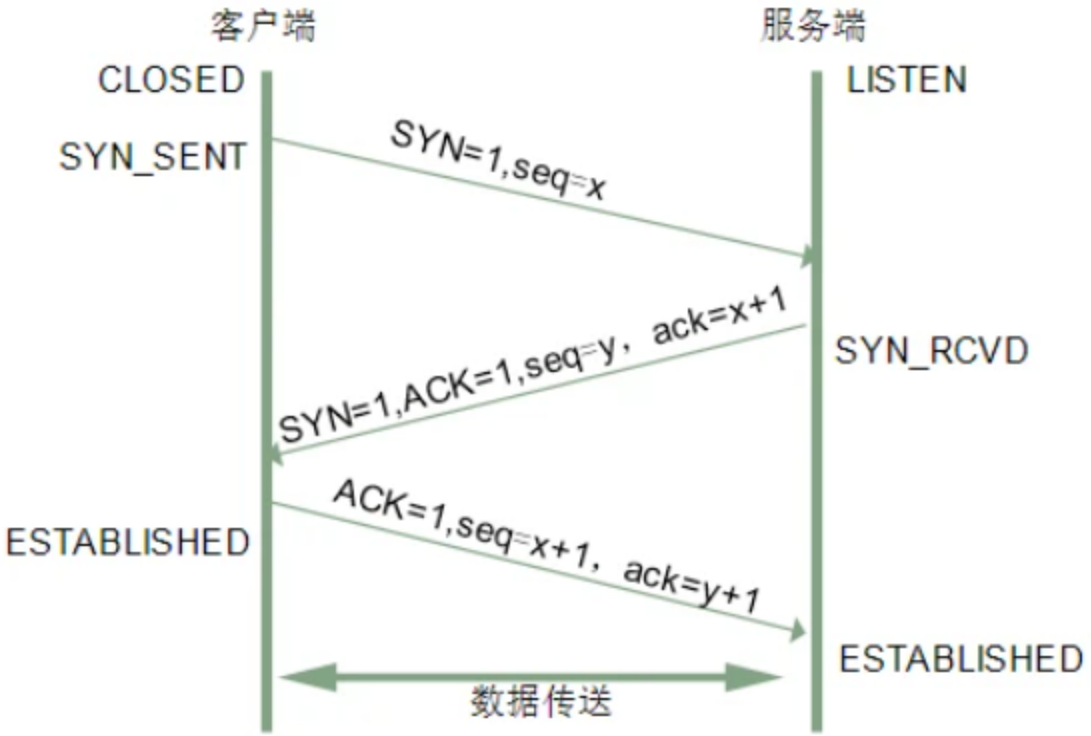

## 5.1. 三次握手

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，双方已建立起了连接




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

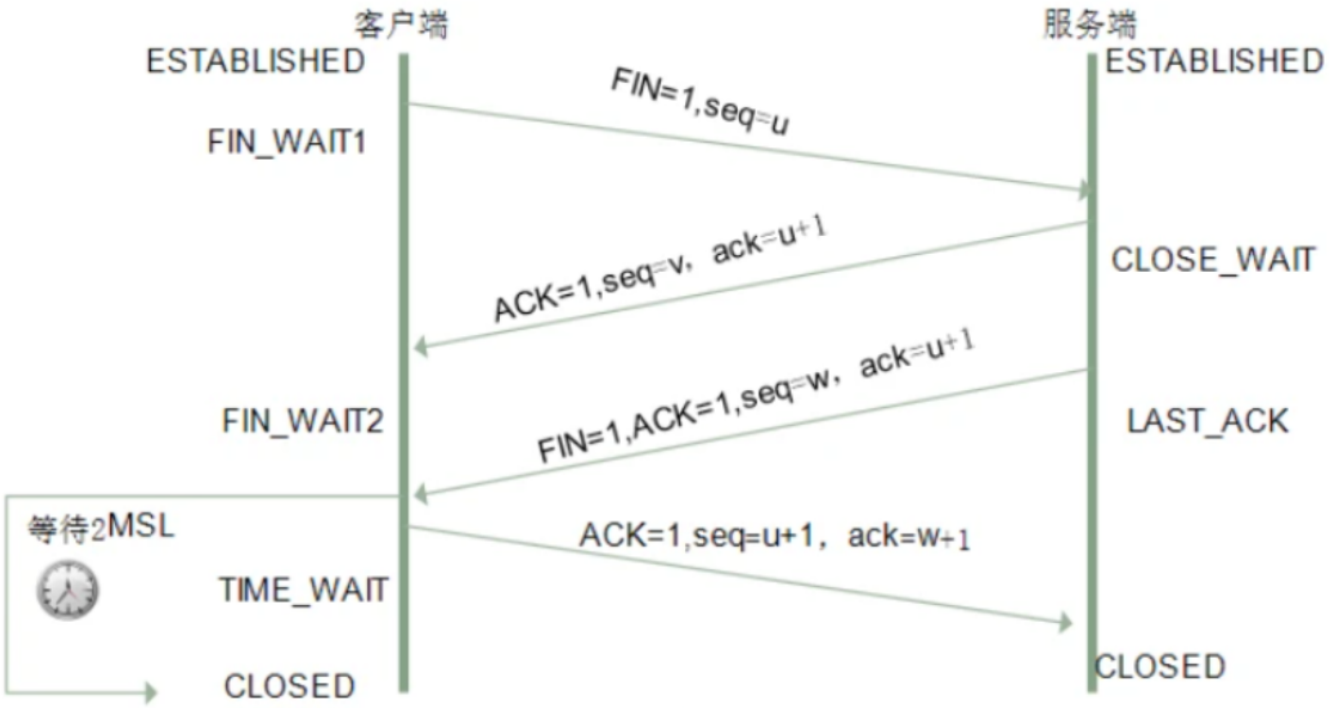
并且两次握手的话, 客户端有可能因为网络阻塞等原因会发送多个请求报文，延时到达的请求又会与服务器建立连接，浪费掉许多服务器的资源

## 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

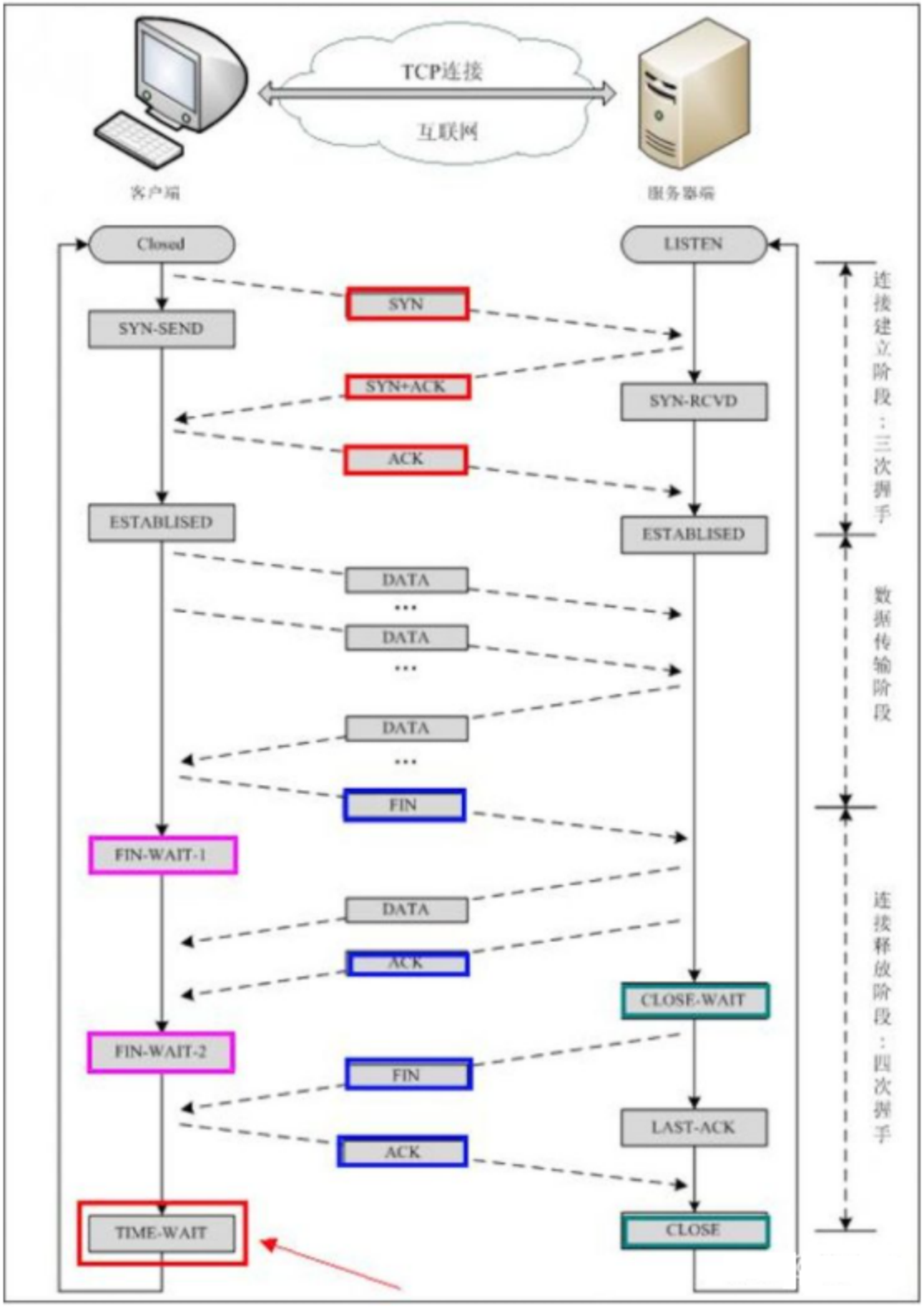



### 5.2.1. 四次挥手原因

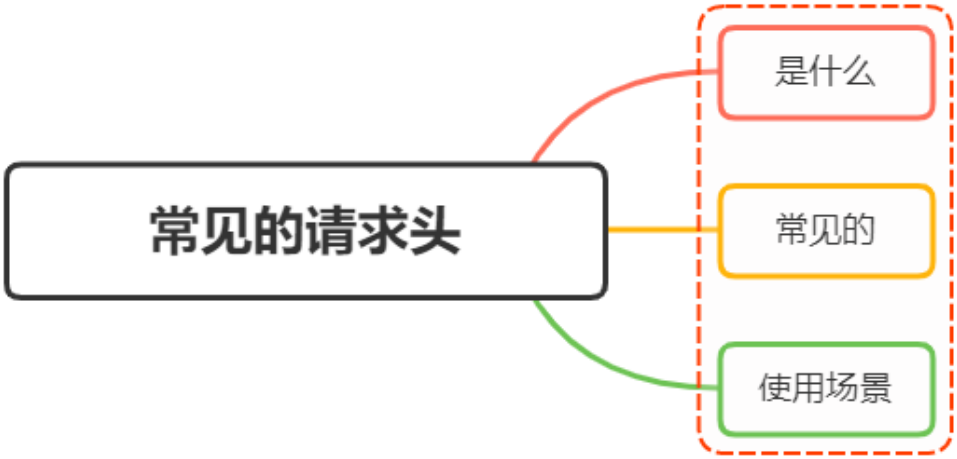
服务端在收到客户端断开连接 `Fin` 报文后，并不会立即关闭连接，而是先发送一个 `ACK` 包先告诉客户端收到关闭连接的请求，只有当服务器的所有报文发送完毕之后，才发送 `FIN` 报文断开连接，因此需要四次挥手

## 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

```HTTP
GET /home.html HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/testpage.html
Connection: keep-alive
Upgrade-Insecure-Requests: 1
If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT
If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"
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 QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| 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<br>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<br>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: "737060cd8c284d8af7ad3082f209582d" |
| 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: "737060cd8c284d8af7ad3082f209582d" |
| 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: "737060cd8c284d8af7ad3082f209582d" |
| 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

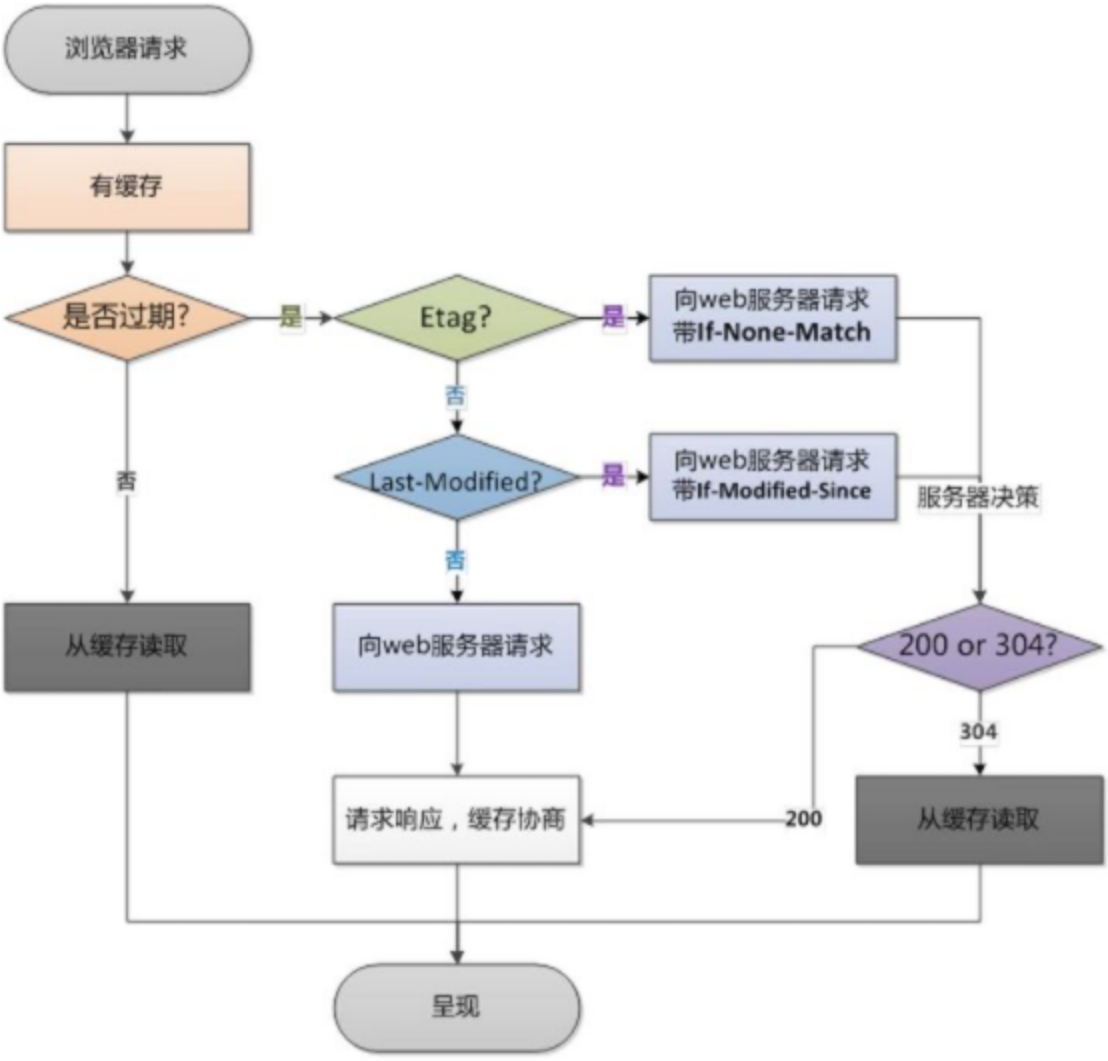
`Last-Modified` 表示本地文件最后修改日期，浏览器会在request header加上 `If-Modified-Since`（上次返回的 `Last-Modified` 的值），询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来

`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

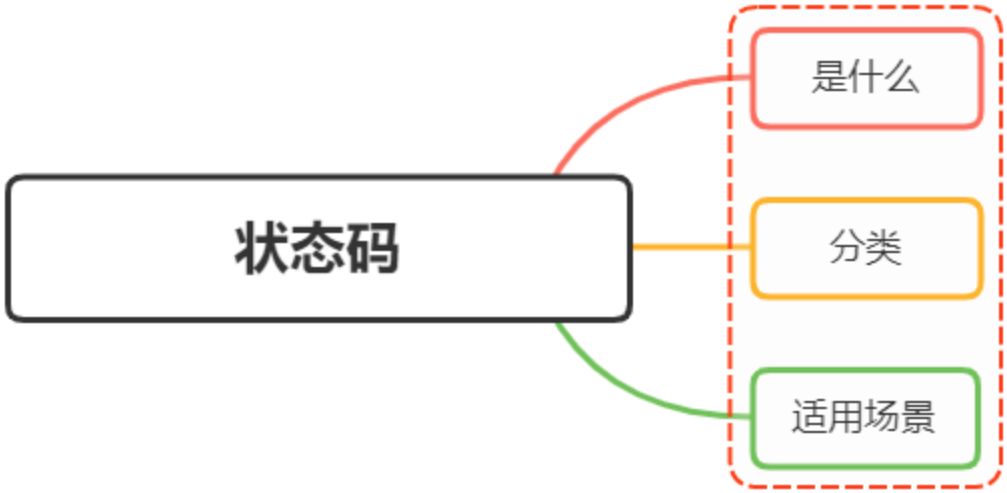
`cookie`，类型为「小型文本文件」，指某些网站为了辨别用户身份而储存在用户本地终端上的数据，通过响应头 `set-cookie` 决定

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 `RFC 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

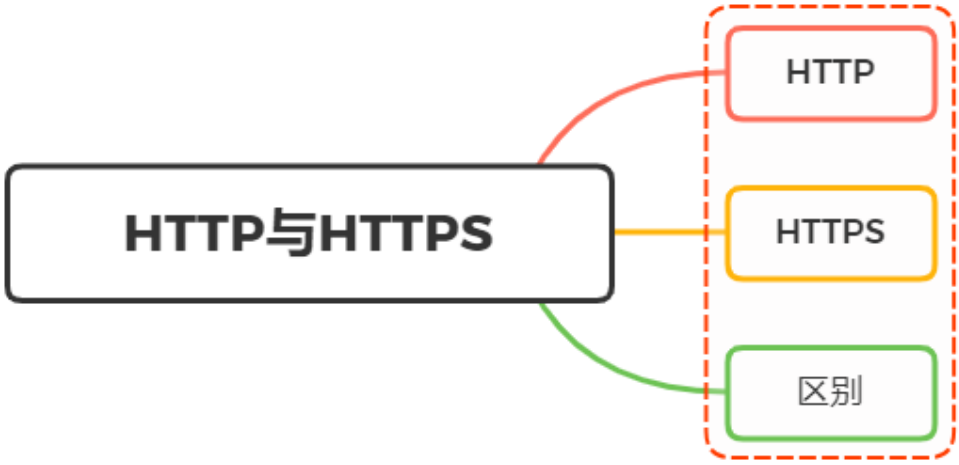
- 502（错误网关）： 服务器作为网关或代理，从上游服务器收到无效响应
- 503（服务不可用）： 服务器目前无法使用（由于超载或停机维护）
- 504（网关超时）： 服务器作为网关或代理，但是没有及时从上游服务器收到请求
- 505（HTTP 版本不受支持）： 服务器不支持请求中所用的 HTTP 协议版本

## 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：