**7-计算机网络**

**网络的两种分层模式**

**OSI分层**

* **应用层**（FTP、SMTP、HTTP）：服务用户
* **表示层**：数据格式变换、数据加密解密、数据压缩和恢复
* **会话层**：建立、管理、终止会话；恢复通信、建立数据同步
* **传输层**：（TCP、UDP）负责两个进程的通信，即端到端通信。传输单位：报文段、用户数据报。功能：可靠传输、不可靠传输、差错控制、流量控制、复用分用
* **网络层**：（IP、ICMP、IGMP、ARP、OSPF）把分组从源端传到目的端，传输单位是数据报。功能：路由选择、流量控制、差错控制、拥塞控制
* **数据链路层**：（PPP、STP）将网络层传下来的数据组装成帧，传输单位是帧。功能：组帧、差错控制、流量控制、控制对信道访问
* **物理层**：物理媒体上实现比特流的透明传输，传输单位是比特

**TCP/IP分层**

* **应用层**：（FTP、SMTP、HTTP）支持各种网络应用
* **传输层**：（TCP、UDP）进程到进程的数据传输，端到端传输
* **网络层**：（IP、ICMP、OSPF）源主机到目的主机的数据分组路由和转发
* **数据链路层**：（PPP）把网络层传下来的数据组装成帧
* **物理层**：比特传输

**物理层**

主要设备有中继器和集线器，中继器为了对信号进行再生和还原，集线器为了放大和转发信号，不能分割冲突域（数据传输会碰撞），主机平分带宽。

**数据链路层**

**如何组帧**

组帧就是在IP数据报前后分别加上头部和尾部。主要作用是确定帧的界限。

**组帧方法：**

* **字符计数法**：帧首部使用一个计数字段来表明帧内字符数
* **字符填充法**：使用转义字符在帧的尾部来分割数据
* **零比特填充法**：头部分隔符是一个0后面6个1再一个0，只要数据中出现五个连续的1后面就增加一个0，在接收数据的时候再进行还原。
* **违规编码法**：将1编码成高低电频，0编码成底高电频。发送的时候起始是高高，结束是低低。

**差错控制**

* **奇偶校验码：偶校验码是 一种增加二进制传输系统最小距离的简单和广泛采用的方法。是一种通过增加冗余位使得码字中1的个数恒为奇数或偶数的编码方法，它是一种检错码。**
* **CRC循环冗余码：CRC的工作方法是在发送端产生一个冗余码，附加在信息位后面一起发送到接收端，接收端收到的信息按发送端形成循环冗余码同样的算法进行校验，如果发现错误，则通知发送端重发。**
* **海明码：汉明码也利用了奇偶校验位的概念，通过在数据位后面增加一些比特，可以验证数据的有效性。利用一个以上的校验位，汉明码不仅可以验证数据是否有效，还能在数据出错的情况下指明错误位置。**

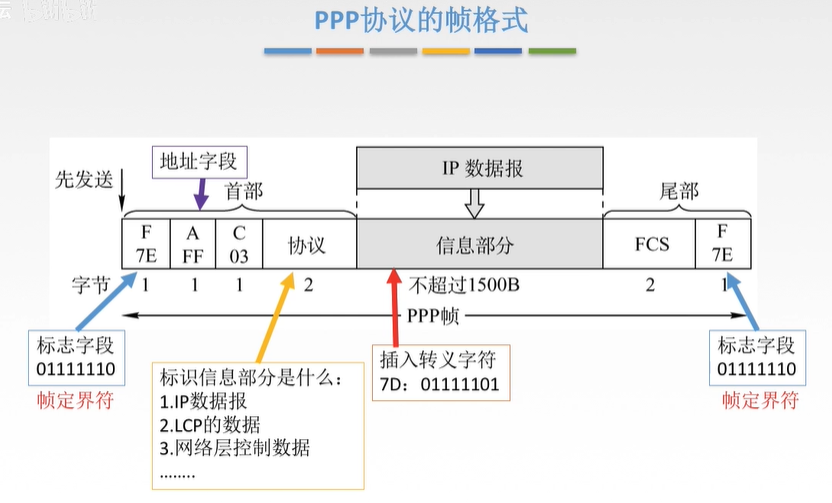
**流量控制**

* **停止等待协议：**发送端发送一帧，接收端就确认一帧，发送端会建立等待时间，超过这个时间接收端还没发送确认，就发送下一帧。如果接收端已经接收到数据，但是确认帧发送端没有收到，接收端每次再收到确认帧会丢弃，然后发送确认帧回去。如果是反过来，发送端第一次的确认帧3s后才收到，第二次的比第一次的提前收到，那么会丢弃第一次的确认帧。
* **回退N帧协议：**发送端不断发送，比如发送1、2、3，然后接收端一个一个，收到1然后确认，收到2然后确认，接收到4然后确认，但是3超过了超时时间还没接收到，就会丢弃3以后收到的帧，重新从3开始计时，接收端来者不拒。
* **选择重传协议：**接收端会建立一个窗口，如果窗口内第一个帧没有收到，会继续接收，直到窗口满了，如果窗口满了且第一个帧还没有收到且超时，则窗口向后面滑动，且将没有收到的帧放到窗口中继续请求没有收到的帧。

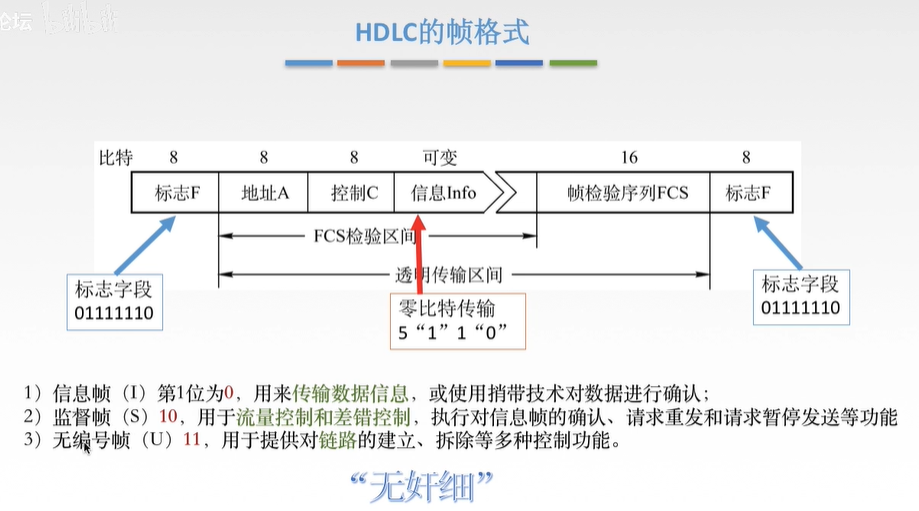
**数据链路层的协议和技术**

**点对点协议**

**PPP协议**



**HDLC协议**



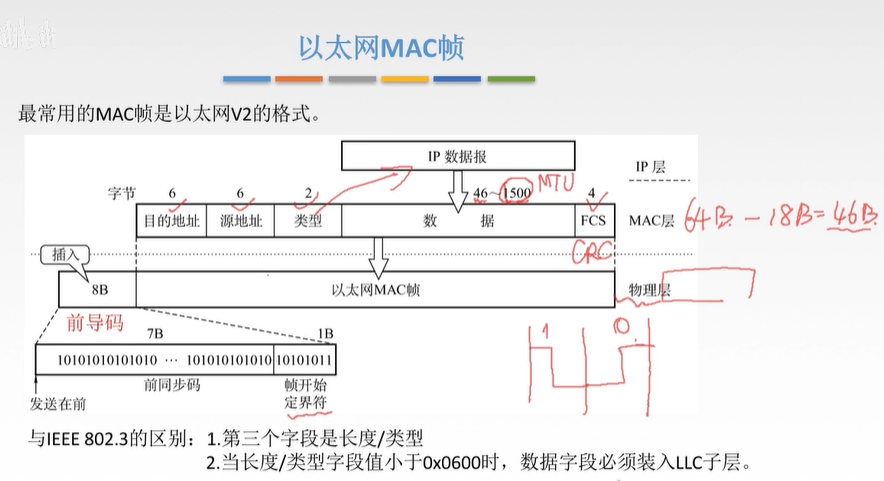
HDLC常常用于同步专线连接的数据封装，可靠传输

**广播协议**

**介质访问控制**

* **静态划分**：频分多路复用、时分多路复用、波分多路复用、码分多路复用
* **动态划分**
* 轮询方式访问：令牌传递协议
* 随机访问：
* ALOHA协议：如有冲突，随机一段时间或者等待一个时间片再进行发送
* CSMA协议：
* 1-坚持：空闲发，忙继续监听
* 非坚持：空闲发，忙等一个时间再发
* p-坚持：空闲p概率发，忙等一个时间再发
* CSMA/CD协议：闲就预约信道，忙则等待再看，通过电压来判断，如果冲突使用二进制退避算法来发送
* CSMA/CA协议：闲就预约信道，忙则等待再看，通过混合检测来判断，如果冲突使用二进制退避算法来发送

**以太网帧**



最小长度为64字节

**链路层设备**

网桥&交换机：对MAC帧目的地址进行转发和过滤，不能隔离广播域，可以隔离冲突域

**网络层**

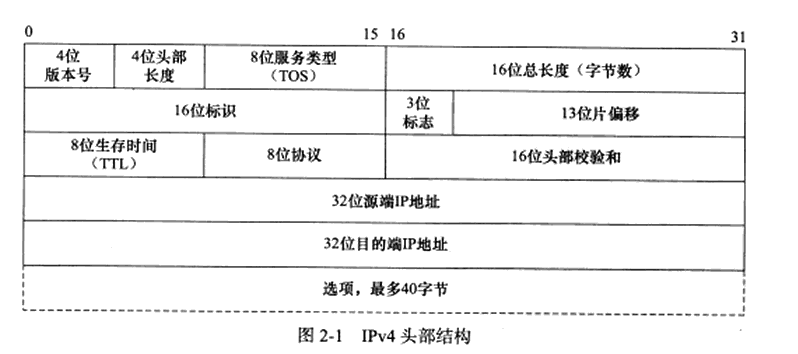
**路由算法**

**距离向量路由算法RIP（内部网关协议）**：每一个路由器都维护从它到其它每一个目的网络的唯一最佳距离记录，最多是15个，16表示网络不可达。RIP仅和相邻路由器交换信息，交换的是自己的路由表，30s交换一次，180s没有收到邻居路由，便更新邻居没了。好详细传得快，坏消息传得慢。

**链路状态路由算法OSPF（内部网关协议）**：最短路径优先，使用了迪杰斯克拉算法的最短路径算法。使用泛洪法向自治系统内所有路由器发送信息。只有当链路变化才会发送信息。

**BGP（外部网关协议）**：通过TCP协议和其它自治系统中的BGP发言人交换路由表。

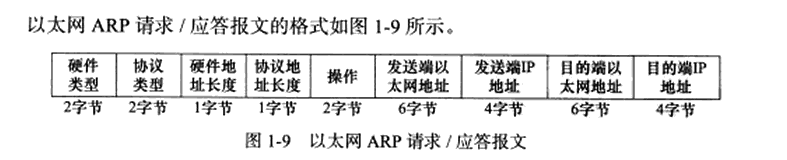
**IP数据报**



IPV4头部长度为20个字节，选项最多40个字节，因此IP头部最多60个字节，以太网帧的MTU（最大传送单元）为1500字节，因此一次最多传送1480个字节的数据，大于1480字节的数据将要被分片

**ARP协议**

ARP报文长度为28字节





ARP（Address Resolution Protocol）是一种用于将IP地址解析为硬件地址（MAC地址）的协议。它在局域网（LAN）中使用，旨在解决将网络层的IP地址映射到数据链路层的硬件地址的问题。

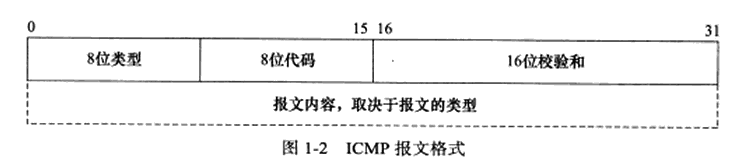
在网络通信中，当主机A需要向主机B发送数据包时，它需要知道目标主机B的硬件地址（MAC地址）。但是，主机A只知道目标主机B的IP地址，不知道其对应的MAC地址。这时，主机A可以使用ARP来解析目标主机的MAC地址。

**ARP的工作原理如下：**

* 主机A检查自己的ARP缓存（ARP Cache），即保存了IP地址和相应MAC地址的本地表。如果目标主机B的IP地址已经存在于ARP缓存中，主机A可以直接获取目标主机的MAC地址，无需进行后续步骤。
* 如果目标主机B的IP地址不在ARP缓存中，主机A会发送一个ARP请求广播（ARP Request Broadcast）。该ARP请求包含主机A的IP地址和MAC地址，并询问局域网中的所有设备，“谁拥有IP地址为X的主机的MAC地址？”
* 所有收到ARP请求的设备都会检查自己的IP地址。如果有设备的IP地址与ARP请求中的目标IP地址匹配，该设备会向主机A发送一个ARP响应（ARP Reply）。该ARP响应包含目标主机B的IP地址和MAC地址。
* 主机A接收到ARP响应后，将目标主机B的IP地址和MAC地址添加到ARP缓存中，并将目标主机B的MAC地址用于发送数据包。

**ICMP协议**

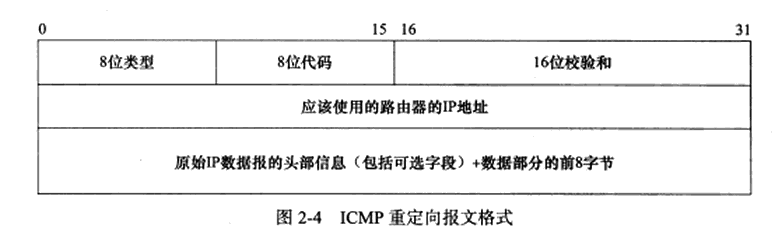
转发IP数据报，提高交付成功的机会。

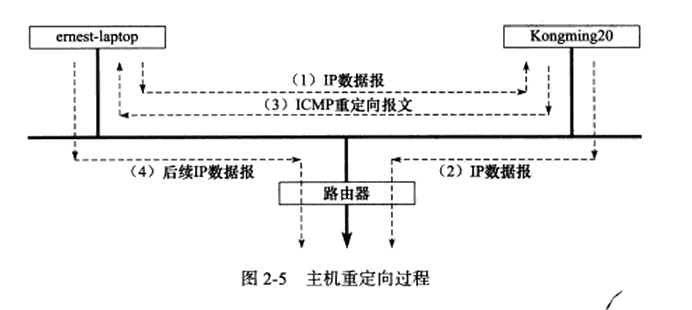


8位类型字段用于区分报文类型，它将ICMP报文分为两大类：一类是差错报文，这类报文用于回应网络错误；另一类是查询报文，用来查询网络状态。

8位代码用来进一步细分报文

16位校验和用的循环校验码





应用：ping 测试两个主机之间的连通性，使用ICMP回送请求和回答报文

traceroute 跟踪一个分组从原点到终点的路径，使用了ICMP时间超过差错报告报文

**IP组播**

**单播：**点对点传输，单独发送到某个目的地。

**广播：**同一个广播域进行传播，是点对多点。

**组播：**某些用户需要特定数据，是点对多点，用户可以返回信息，组播参与使用的是IGMP报文。

**网络层设备**

路由器：可以隔离冲突域和广播域。组成是目的网络IP地址、子网掩码、下一跳IP地址、接口

**传输层**

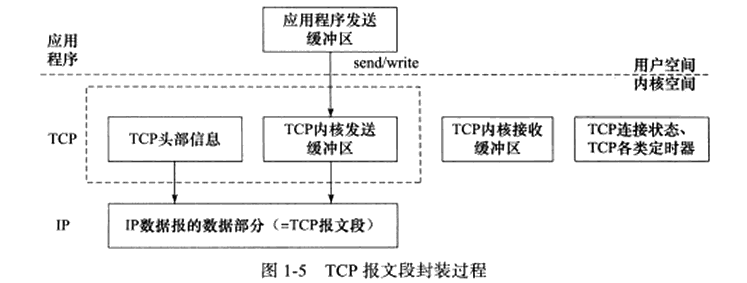
**TCP协议**

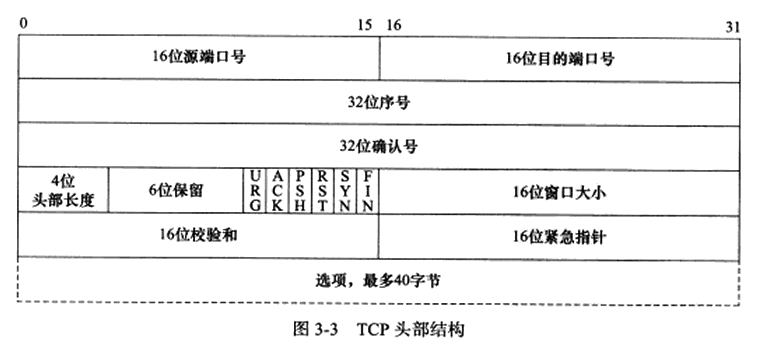
TCP是面向连接的、可靠的进程到进程通信的协议。TCP提供全双工服务，即数据可在同一时间双向传输，每一个TCP都有发送缓存和接收缓存，用来临时存储数据。

**TCP协议特点**

TCP是面向连接的协议；TCP的一个连接有两端（[点对点通信](https://zhida.zhihu.com/search?q=%E7%82%B9%E5%AF%B9%E7%82%B9%E9%80%9A%E4%BF%A1&zhida_source=entity&is_preview=1)）；TCP提供可靠的传输服务；TCP协议提供全双工的通信；TCP是面向[字节流](https://zhida.zhihu.com/search?q=%E5%AD%97%E8%8A%82%E6%B5%81&zhida_source=entity&is_preview=1)的协议。可靠传输的机制：1、校验 2、序号 3、确认 4、重传

**TCP协议字段**



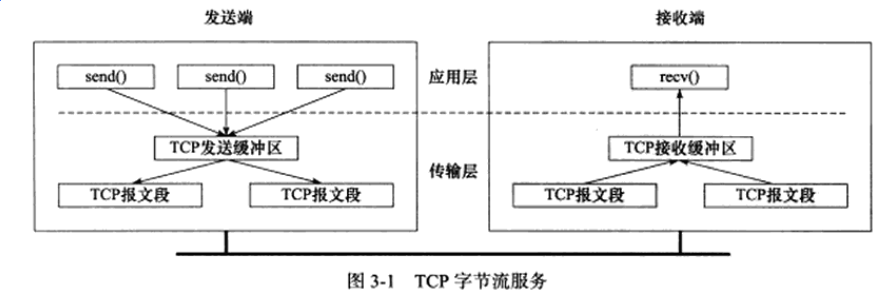


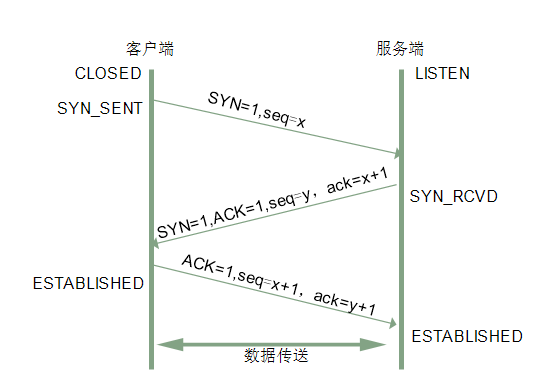
TCP头部最大为40字节，可传送的最大字节数为1460

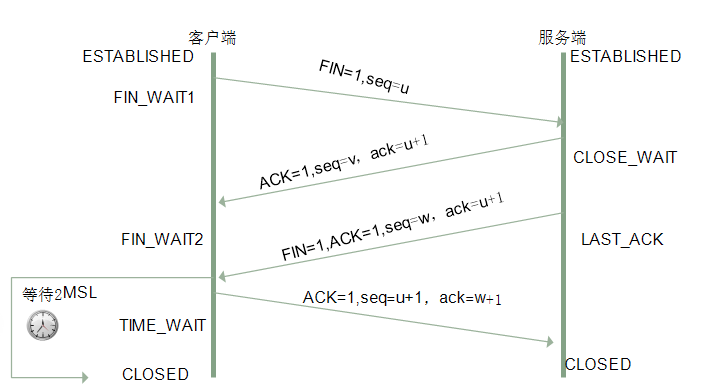
**UGR**：紧急指针 **ACK**：确认号 **PSH**：是否立刻再接收缓冲区读取数据，腾出空间

**RST**：要求对方重新建立连接 **SYN**：请求建立连接  **FIN**：通知对方要关闭连接了

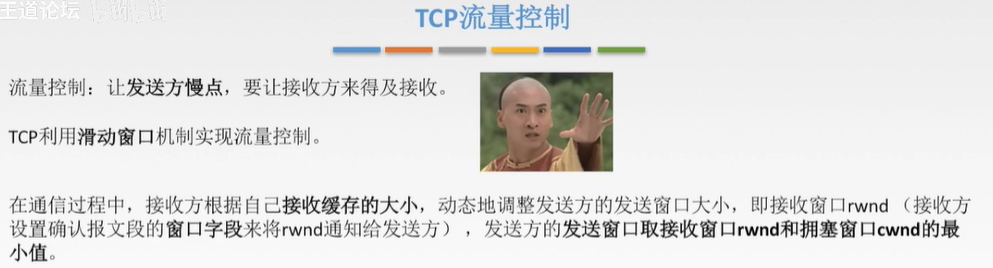
**TCP如何建立和关闭连接**

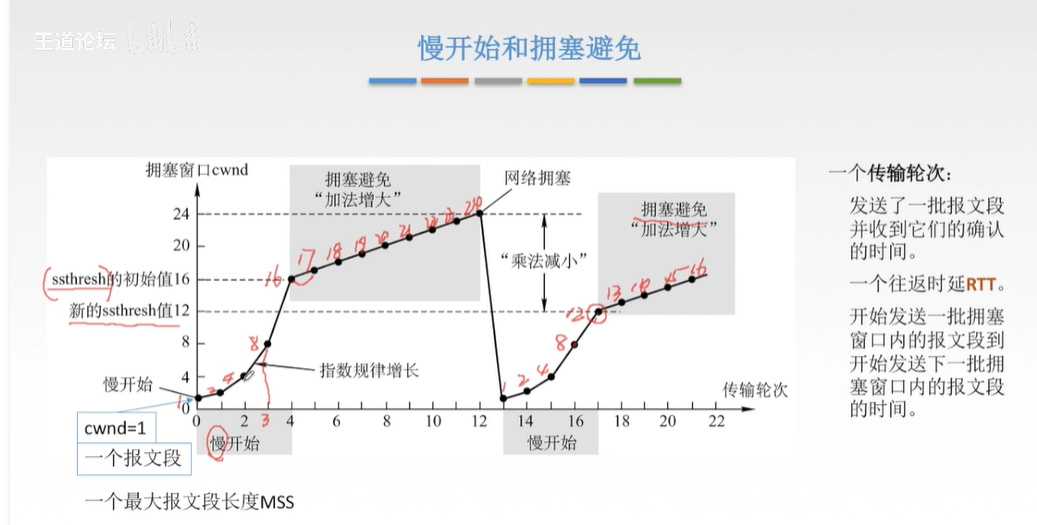






**TCP的流量控制和重传机制**





**保证可靠传输**

* 1.连接管理机制：在传输数据前需要进行建立连接,也就是三次握手，在数据传送完后还需要释放连接，也就是四次挥手。
* 2.数据分段：TCP以报文段为单位进行发送，在建立TCP连接的时候，两端协商TCP报文段中的数据字段(也称为数据包)的最大长度(MSS);其长度加上首部长度就是整个TCP报文段的长度；
* 3.校验和：提供了一种简单的校验，如果收到段的校验和和原来的有差别，那接收方就会丢掉这个报文段；
* 4.序列号：TCP给发送的数据包的编号，如果接收端收到乱序后会进行重新排序，收到重复的也会进行丢弃；
* 5.确认应答：接收方收到报文后会回复确认(累计确认：对所有收到的按序的只确认最后，确认号之前的数据已经全部收到了)；
* 6.重发控制：TCP发出一个报文段后，就会启动一个定时器，等接收方确认这个报文段，如果不能及时收到确认，将重发这个报文段；
* 7.流量控制：通信的双方都有固定大小的缓冲区，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，就把窗口缩小，并把窗口值告诉发送端（提示发送方降低发送的速率，防止包丢失）是利用滑动窗口来实现；
* 8.拥塞控制：当网络产生拥堵时，减少数据的发送；主要是通过拥塞窗口来实现；(慢开始和拥塞避免；快重传和快恢复)；

**TCP 可靠传输的基本原理**

* [停止等待协议](https://zhida.zhihu.com/search?q=%E5%81%9C%E6%AD%A2%E7%AD%89%E5%BE%85%E5%8D%8F%E8%AE%AE&zhida_source=entity&is_preview=1)：发送方生成新的消息，发送给接收方，并且此时不会产生新的消息，需要收到接受方的确认消息后，才会产生新的消息。[超时重传](https://zhida.zhihu.com/search?q=%E8%B6%85%E6%97%B6%E9%87%8D%E4%BC%A0&zhida_source=entity&is_preview=1)：如果发送方的消息在传输的过程种丢失了，接收方没有收到消息，就会进行超时重传；如果接收方发送的确认消息，在传输的过程中丢失，也会进行超时重传，因此 每发送一个消息，都需要设置一个定时器。停止等待协议是最简单的可靠传输协议，但停止等待协议对信道的利用效率不高。
* [连续ARQ协议](https://zhida.zhihu.com/search?q=%E8%BF%9E%E7%BB%ADARQ%E5%8D%8F%E8%AE%AE&zhida_source=entity&is_preview=1)：ARQ(Automatic Repeat reQuest：[自动重传请求](https://zhida.zhihu.com/search?q=%E8%87%AA%E5%8A%A8%E9%87%8D%E4%BC%A0%E8%AF%B7%E6%B1%82&zhida_source=entity&is_preview=1))，由于单个发送和确认效率低，我们可以通过批量发送和确认来提升效率。TCP的可靠传输基于连续ARQ协议，TCP的滑动窗口以字节为单位，如果接收到的序号没有按序收到确认号，在超时时间内就会进行重新传送，为了避免对整个窗口中的字节进行重传，因此TCP协议使用了选择重传来提高传输效率。选择重传：重传的是一段字节流，而不是某个字节，在TCP选项里存储的是需要重传的字节流的边界。选择重传需要指定需要重传的字节，每一个字节都有唯一的32位序号。

**快重传和快恢复**

在没有快重传时，如果一个数据段丢失了，会等待一定的超时周期才重传(每发送一个数据段之后就会启动一个计时器)，这增加了时延；当发送方连续收到三个重复的确认（ACK）消息时，会立即重传丢失的数据包，并进入快速恢复状态。这种机制可以尽早发现数据包的丢失，并迅速恢复传输，减少传输延迟。

1. 快重传算法：

* 当发送方连续收到三个重复的ACK消息时，会立即重传丢失的数据包。
* 如果只有一个数据包丢失，接收方会重复发送之前的ACK消息，直到数据包被重传。
* 如果多个数据包丢失，接收方只会部分确认已经接收到的数据包，而不是全部确认。

1. 快恢复算法：

* 重传数据包后，发送方的慢启动阈值（ssthresh）会被设置为当前拥塞窗口大小的一半。
* 然后将拥塞窗口大小设置为慢启动阈值，进入拥塞避免状态，逐步增加拥塞窗口大小。

**TCP常见问题**

**为什么要三次握手，两次不行吗？**

第一次握手，服务端收到，服务器得到结论：确定客户端发送能力正常，服务端接收能力正常。

第二次握手，客户端收到，客户端得到结论：确定服务端和客户端接收和发送正常。

第三次握手，服务端收到，服务端得到结论：确定服务端和客户端接收和发送正常。

如果只有两次，服务端无法确定客户端是否接收正常以及服务端发送是否正常，如果客户端发送一个连接，第一个丢失，于是发了第二个，第二个达到后建立然后断开，此时第一个到了建立连接，此时客户端又已拒绝访问，此时服务端就会一直等待客户端发送数据。

**什么是半连接队列？**

指的是二次握手以后，对方还未完全建立连接，服务端所保存的状态。

**什么是泛洪攻击，如何防范？**

指的是短时间大量连接请求发送到服务器，却没有第三次握手，导致正常的SYN因为队列满了而被丢弃而导致网络拥塞，解决办法有：缩短超时时间、增大半连接数、过滤网关防护、SYN cookies技术。

**为什么要挥手四次？**

为了确保双方都发送数据完毕。

**time\_wait为什么要等待2MSL？**

2MSL是一次报文的往返时延，TCP连接的一端会等待2MSL（Maximum Segment Lifetime）的时间，这个时间通常设置为2分钟，因此等待的总时间一般为4分钟

* 确保客户端发送最后一个ACK报文段可以达到服务器
* 防止“已经失效的连接请求报文段”出现在本连接中

**服务器出现大量time\_wait和close\_wait的原因，如何修复？**

time\_wait是由于高并发服务器产生，服务器主动关闭导致，可以修改配置，使用time\_wait的快速回收和重用。

close\_wait是因为服务器还在读，对面就提前关闭socket，服务器只要检测出客户端已经关闭，就关闭socket

1. 判断socket，如果read返回0则断开，如果是负数，则检查errno，如果不是EAGIN，则也断开连接
2. 每一个socket设置时间戳，如果一定时间内没有动静，则关闭socket
3. 设置心跳检测，如果收到了RST则关闭

**什么是粘包，如何处理？**

在TCP协议中，发送方发送若干数据到接收方接收时候，接收方一次将所有数据都读取，导致数据都头尾相连。解决办法有：（1）约定发送数据包的长度，不够则填充。（2）使用分隔符（3）在数据包的头部设定包的长度的字符串

**快重传为什么会选择3次呢？**

首先要明白这个3次指的是3次冗余，是不包括前面的正常确认的。

原因就是因为网络传输是不可靠的，在将不同数据段发送的时候，可能会导致到达接收端的顺序是不一样的，也就是乱序，比如后面的N+1到了，N+2到了，但是N没到，那这时候可能等一等一会就到了，也可能就是丢了，在发3次的时候A丢包的可能性是最高的。所以选择3，其实在实际抓包的时候，快重传都会在大于3后发生；总的原因就是两次可能是因为乱序导致的，3次一定是包丢了

**三次握手的状态信息是保存在**[**TCP协议栈**](https://www.baidu.com/s?sa=re_dqa_generate&wd=TCP%E5%8D%8F%E8%AE%AE%E6%A0%88&rsv_pq=c337803501cfe0cf&oq=%E4%B8%89%E6%AC%A1%E6%8F%A1%E6%89%8B%E7%9A%84%E7%8A%B6%E6%80%81%E4%BF%A1%E6%81%AF%E6%98%AF%E4%BF%9D%E5%AD%98%E5%9C%A8%E4%BB%80%E4%B9%88%E5%9C%B0%E6%96%B9&rsv_t=e0b9qqq9RLX1wKNUqjTtYAVqZss/eloqE7uuEzBReGnZ6D45war4KLMdMws&tn=baidu&ie=utf-8)**中**

**UDP协议**

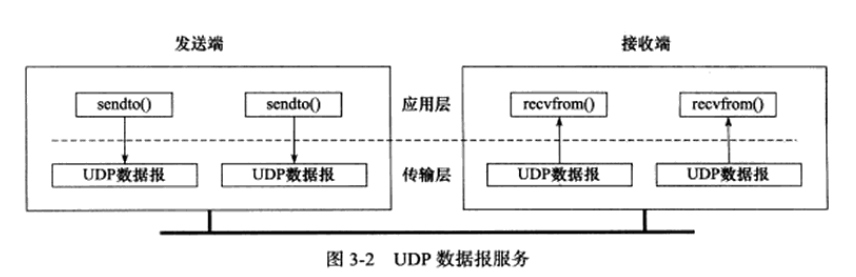
**UDP 概念**

UDP 是一种简单的面向无连接的传输层协议，它提供了在网络上发送数据的功能，但不提供可靠性、流量控制、错误恢复或顺序传输等功能。UDP被设计为尽可能简单和轻量，适用于一些对实时性要求较高、数据量较小、容忍少量丢失的应用场景，比如语音通话、视频流等。

**UDP协议的特点**

UDP是[无连接协议](https://zhida.zhihu.com/search?q=%E6%97%A0%E8%BF%9E%E6%8E%A5%E5%8D%8F%E8%AE%AE&zhida_source=entity&is_preview=1)；UDP不能保证可靠的交付数据，即UDP协议不会[感知网络](https://zhida.zhihu.com/search?q=%E6%84%9F%E7%9F%A5%E7%BD%91%E7%BB%9C&zhida_source=entity&is_preview=1)是否拥塞，UDP协议不管网络是否拥塞，都会把数据交付出去，给这个网络就完了。无法保证数据在网络中是否丢失；UDP是面向报文传输的；UDP没有拥塞控制，而且UDP首部开销很小





**UDP 常见问题**

**UDP出现丢包怎么做，如果接收是乱序，如何保证数据的有序性？**

扩大缓冲区，建立应答机制，建立滑动窗口。

**UDP 和 TCP主要区别**



* 比如TCP是面向连接的，在传输数据之前需要先建立连接，数据发送完毕后也需要关闭连接，而UDP不需要连接。
* 比如TCP是面向字节流的，而UDP是面向报文的。
* 比如TCP需要有确认应答机制，而UDP不需要。

**TCP的面向字节流和UDP的面向报文怎么理解？**

* TCP是面向字节流的协议，也就是说数据是以字节流的形式传递给对方的，没有固定的报文或者说报文边界概念。双方都会有一个缓冲区，如果字节流太长，TCP就会拆分进行发送，如果字节流很短，那也可以等着缓冲区中的字节流变长了以后再构成一个报文段进行发送。
* UDP是面向报文的协议，应用层交给UDP多长的协议，就发送多长的报文，不合并也不拆分，保留报文的边界，一次发送一个。

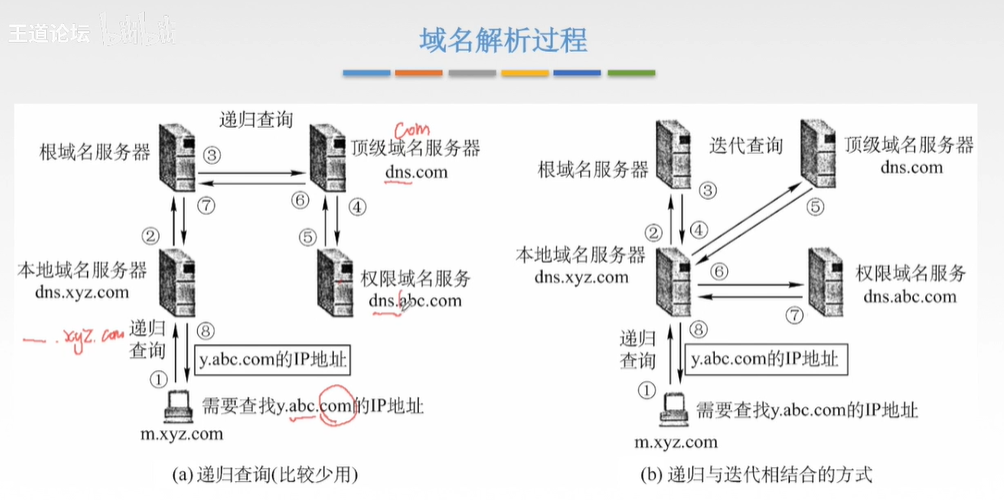
**QQ或者微信聊天是采用什么协议？**

* TCP的应用场景：文件的上传和下载、浏览器上网、绝大多数应用都是TCP；
* UDP的引用场景：音视频传输(qq和微信)、共享屏幕等；

QQ和微信主要是采用UDP的方法，主要是因为效率高、传输快，占用资源少，但是主要是不可靠，所以很多时候会辅助一些算法来保证可靠传输；在qq里，一台服务器要容纳十几万的并发连接，所以都是去进行建立连接这样的过程效率太低，太影响信息传输的效率，所以采用了UDP协议；它们应该是在上层采用了一些算法来实现可靠，比如客户端使用UDP协议发出消息后，服务器收到该包，那就需要使用UDP协议发回一个应答包，用这样的方法实现可靠传输；

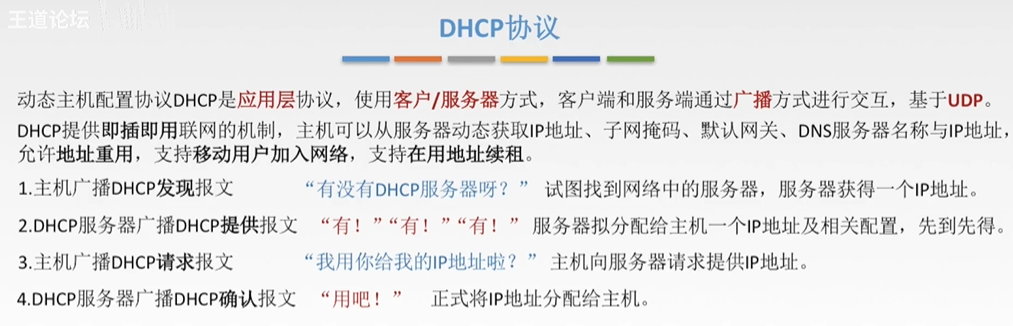
**应用层**

**DNS**

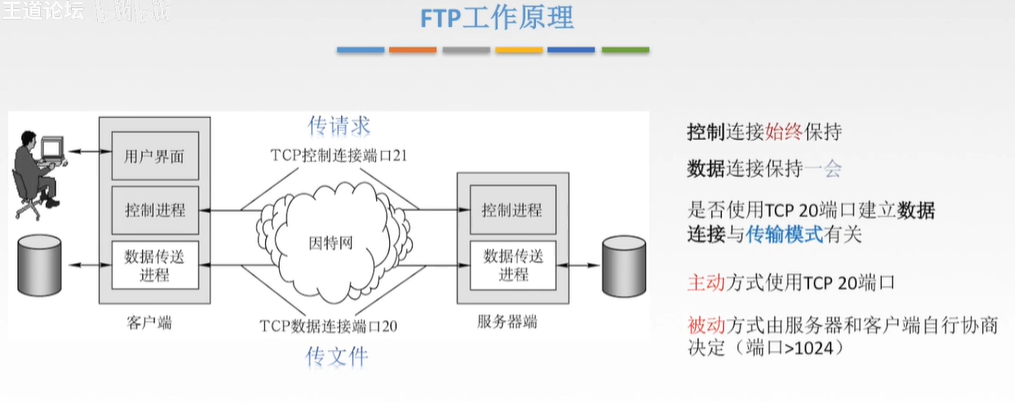


dns协议是基于udp、tcp协议的，端口号为53，用户到服务器采用udp协议，dns服务器通信用tcp交流

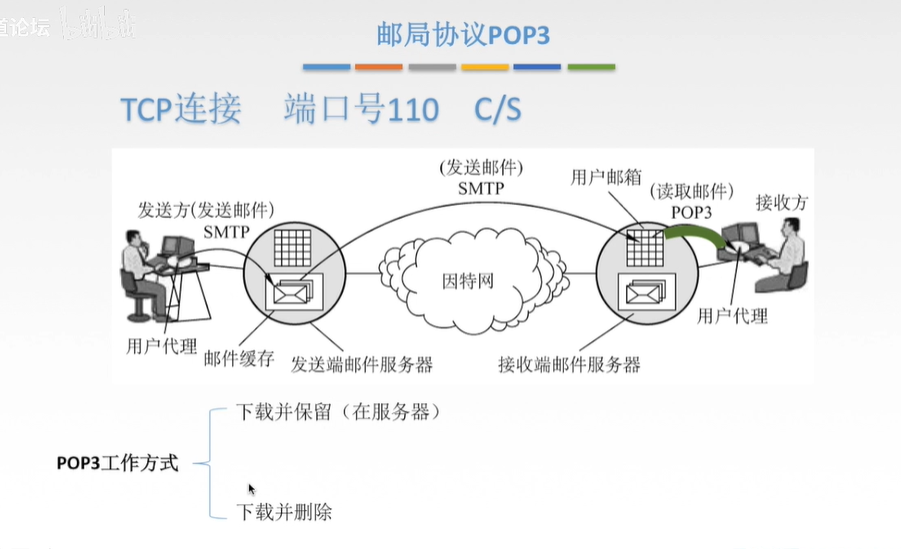
**DHCP协议**



**FTP**



**SMTP、POP3**



**HTTP**

**http请求头**



* 请求方法分为Get和Post
* Get将请求参数追加在url后，请求参数受到url长度限制，请求报文没有请求体，Get方式浏览器会主动缓存
* Post请求将参数放在请求体上，比较安全，支持多种编码方式
* referer：表示这个请求是从哪个url跳转过来
* accept：告诉服务器，该请求可以支持的相应类型
* cookies：客户端通过这个报文属性传递给服务端
* connection：表示和客户端连接的类型，keep-alive表示持久连接，close表示短链接

**http响应报文**



* 状态码：

1\*\*(信息类)：表示接收到请求并且继续处理

100——Continue客户必须继续发出请求

101——客户要求服务器根据请求转换HTTP协议版本

2\*\*(响应成功)：表示请求被成功接收、理解和接受

200——OK表明该请求被成功地完成，所请求的资源发送回客户端

201—— Created 提示知道新文件的URL

202——Accepted接受和处理、但处理未完成

3\*\*(重定向类)：为了完成指定的动作，必须接受进一步处理

300——请求的资源可在多处得到

301——Moved Permanently本网页被永久性转移到另一个URL

302——请求的网页被转移到一个新的地址，但客户访问仍继续通过原始URL地址，重定向，新的URL会在response中的Location中返回，浏览器将会使用新的URL发出新的Request。

304——Not Modified自从上次请求后，请求的网页未修改过，服务器返回此响应时，不会返回网页内容，代表上次的文档已经被缓存了，还可以继续使用

4\*\*(客户端错误类)：请求包含错误语法或不能正确执行

400——客户端请求有语法错误，不能被服务器所理解

403——Forbidden 禁止访问，服务器收到请求，但是拒绝提供服务

404——Not Found一个404错误表明可连接服务器，但服务器无法取得所请求的网页，请求资源不存在。eg：输入了错误的URL

5\*\*(服务端错误类)：服务器不能正确执行一个正确的请求

HTTP 500 Internal Server Error - 服务器遇到错误，无法完成请求

HTTP 502 - 网关错误

* set-cookies：写入cookies

**https**

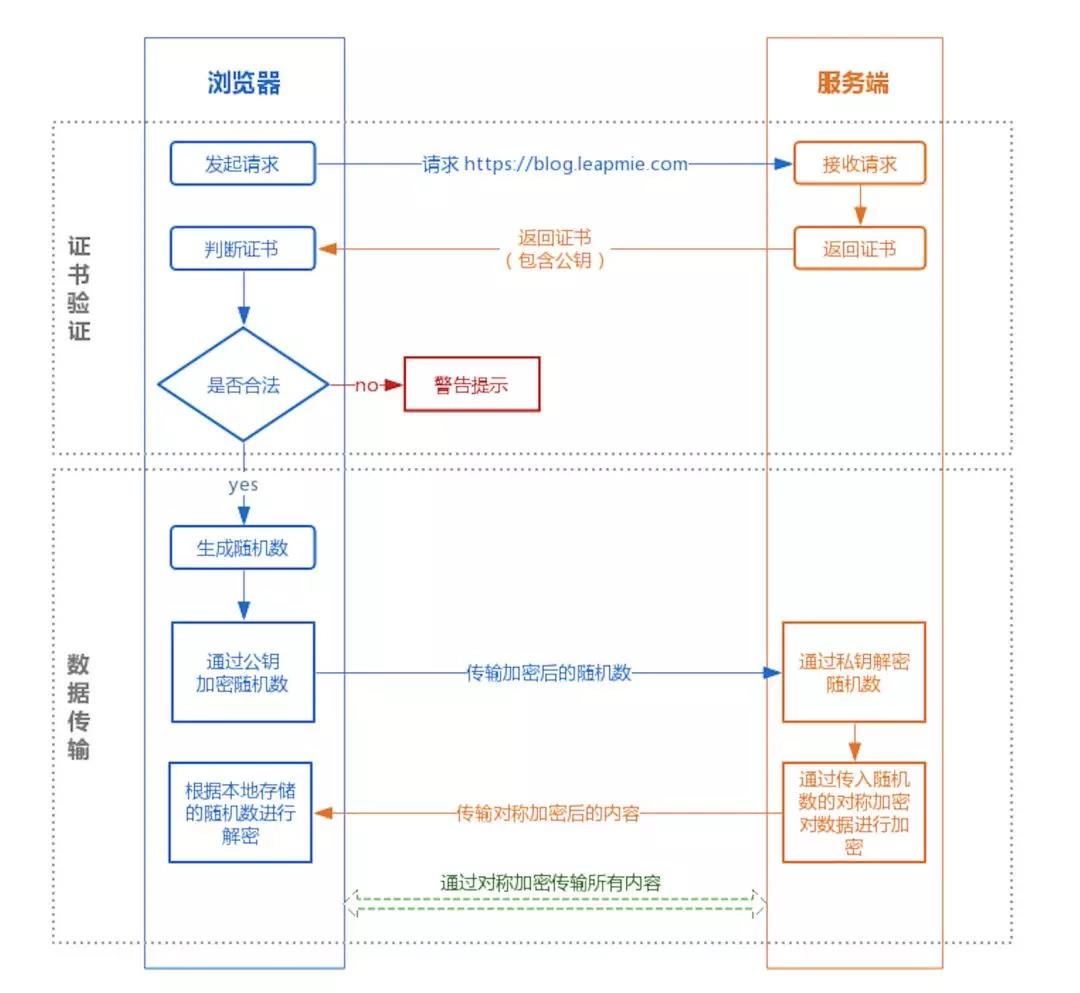
基于HTTP协议，通过SSL或TLS提供加密处理数据、验证对方身份以及数据完整性保护

https做了以下三个事情：（1）加密 （2）数据一致性 （3）身份认证

https比https多了SSL/TLS

TLS(Transport Layer Security) 是 SSL(Secure Socket Layer) 的后续版本，它们是用于在互联网两台计算机之间用于身份验证和加密的一种协议。

TLS 在根本上使用对称加密和 非对称加密 两种形式。



**中间人攻击**

1.客户端请求被劫持（如DNS劫持等），所有的客户端请求均被转发至中间人的服务器；

2.中间人服务器返回中间人伪造的“伪证书”（包含伪公钥）；

3.客户端创建随机数，通过中间人证书的伪公钥对随机数进行加密后传输给中间人，然后凭随机数构造对称加密算法对要进行传输的数据内容进行对称加密后传输；

4.中间人因为拥有客户端生成的随机数，从而能够通过对称加密算法进行数据内容解密；

5.中间人再以“伪客户端”的身份向正规的服务端发起请求；

6.因为中间人与服务器之间的通信过程是合法的，正规服务端通过建立的安全通道返回加密后的数据内容；

7.中间人凭借与正规服务器建立的对称加密算法进行数据内容解密；

8.中间人再通过与客户端建立的对称加密算法对正规服务器返回的数据内容进行加密传输；

9.客户端通过中间人建立的对称加密算法对返回的数据内容进行解密；

由于缺少对证书的真伪性验证，所有客户端即使发起了Https请求，但客户端完全不知道自己发送的请求已经被第三方拦截，导致其中传输的数据内容被中间人窃取。

**1.证书包含的主要信息**：

颁发机构信息

公钥

公司信息

域名

有效期

指纹

等等

**2.证书的合法性依据**

首先，权威机构是需要通过认证的。其次证书的可信性基于信任制，CA认证机构需要对其颁发的证书进行信用担保，只要是CA认证机构颁发的证书，我们就认为是合法的。CA认证机构会对证书申请人的信息进行审核的。

**3.浏览器如何验证证书的合法性？**

浏览器发起https请求时，服务器会返回网站的SSL证书，浏览器需要对证书做以下验证：

**验证域名、有效期等信息是否正确；**

判断证书来源是否合法。每份签发证书都可以根据验证链查找到对应的根证书，操作系统、浏览器会在本地存储权威机构的根证书，利用本地根证书可以对对应机构签发的证书进行来源验证；

判断证书是否被篡改。需要与CA服务器进行对比校验；

判断证书是否已被吊销。通过CRL(Certificate Revocation List 证书注销列表) 和 OCSP（Online Certificate Status Protocol 在线证书状态协议）实现，其中OCSP可用于第3步中以减少与CA服务器的交互，提高验证效率。

以上任意一步都同时满足的情况下，浏览器才认为证书是合法的。

**只有认证机构可以生产证书吗？**

如果需要浏览器不提示安全风险，那只能通过认证机构签发的证书。但浏览器通常只是会提示安全风险，并不会限制网站的访问，所有从技术上来说，谁都可以生产证书，只要有证书就能够完成网站的https传输。

**客户端的本地随机数被窃取了怎么办？**

其实https并不包含对随机数的安全保证，https保证的只是数据传输过程安全，而随机数存储于本地，本地的安全属于另一安全范畴，应对的措施有安装杀毒软件、反木马、浏览器升级修复漏洞等。（这也反映了Https协议并不是绝对的安全的）

**使用Https被抓包了会怎样？**

由于Https的数据是加密，常规下抓包工具代理请求后抓到的包内容是加密状态的，无法直接查看。

但是，浏览器只会提示安全风险，如果用户授权仍然继续访问网站，完成请求。那么，只有客户端是我们自己的终端，我们授权的情况下，便能够建立中间人网络，而抓包工具作为中间人的代理。

通常， HTTPS 抓包工具的使用方法是会生成一个证书，用户需要手动把证书安装到客户端中，然后终端发起的所有请求通过该证书完成与抓包工具的交互，然后抓包工具再转发请求到服务器，最后把服务器返回的结果在控制台输出后再返回给终端，从而完成整个请求的闭环。

即是，HTTPS 只防止用户在不知情的情况下通信被监听，如果用户主动授信，是可以构建“中间人”网络

**http http2 http3对比**

**http1.1：**

* 高延迟 — 队头阻塞：队头阻塞是指当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一并被阻塞，会导致客户端迟迟收不到数据。解决办法：将资源分配到不同域名上，提高连接上限；减少请求数量；内联一些资源：css，base64图像等；合并小文件；
* 无状态特性：连接状态没有记忆能力，会掉登陆状态
* 不安全性

**http2：**

* 多路复用，但是容易timeout，服务器压力上升
* 数据全部转为二进制分帧传输
* 丢包的时候整个TCP都要等待重传

**http3：**

基于UDP协议，彻底解决了头部阻塞，支持多路复用（是基于UDP写的）

**Http 重定向流程**

1. 客户端向服务器发送HTTP请求，并在请求头中包含URI（Uniform Resource Identifier），也就是要访问的资源地址。
2. 服务器接收到客户端的HTTP请求后，开始处理请求，并根据请求的内容和服务器的配置进行判断。
3. 当服务器判断需要进行重定向时，会在响应头中设置适当的状态码。比如，如果是永久重定向，就会设置状态码301，并在响应头Location字段中包含新的URL地址。
4. 客户端收到服务器的响应后，会根据状态码进行相应的处理。如果是301或者302状态码，客户端会自动跳转到新的URL地址；如果是303状态码，客户端会以GET方法请求新的URL地址；如果是307状态码，客户端会保持原有的请求方法向新的URL地址发送请求。
5. 服务器接收到客户端的重定向请求后，会再次处理请求，并按照重定向后的URL地址返回相应的资源。

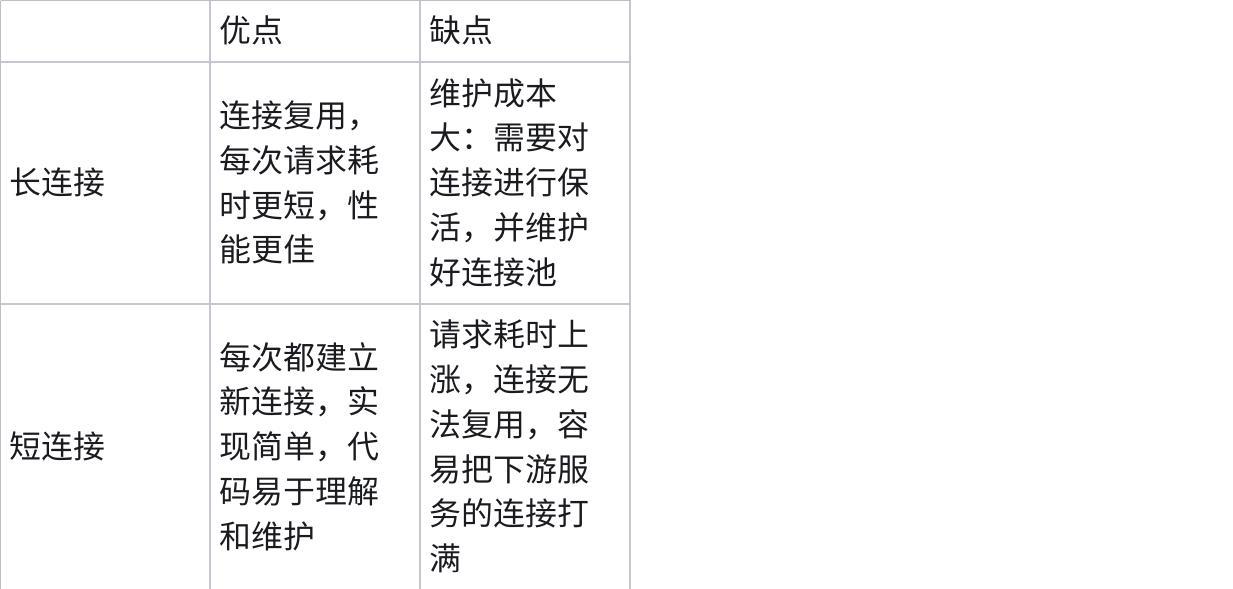
**长短连接的区别**

* 长连接：“长”的字面意思是维持时间长，「连接长期存在，也就是说程序可以复用这条连接，不用每次发起请求都重新建立连接」。这就好比你要过河，然后修建了一座桥，大家都可以使用这座桥，这座桥被大家复用了。
* 如何保活
* 第1种：在运输层开启[tcp协议](https://zhida.zhihu.com/search?q=tcp%E5%8D%8F%E8%AE%AE&zhida_source=entity&is_preview=1)的keep alive特性。
* 第2种：在应用层使用[应用层协议](https://zhida.zhihu.com/search?q=%E5%BA%94%E7%94%A8%E5%B1%82%E5%8D%8F%E8%AE%AE&zhida_source=entity&is_preview=1)，做定期的心跳请求做[连接保活](https://zhida.zhihu.com/search?q=%E8%BF%9E%E6%8E%A5%E4%BF%9D%E6%B4%BB&zhida_source=entity&is_preview=1)，原理和tcp的keep alive特性类似，但是应用层的[心跳保活](https://zhida.zhihu.com/search?q=%E5%BF%83%E8%B7%B3%E4%BF%9D%E6%B4%BB&zhida_source=entity&is_preview=1)可以探测到对端服务服务过载的情况，也更可控，也更及时。
* 第3种：[空闲连接](https://zhida.zhihu.com/search?q=%E7%A9%BA%E9%97%B2%E8%BF%9E%E6%8E%A5&zhida_source=entity&is_preview=1)释放，当发现连接空闲时间超过一定阈值时，则释放连接。
* 短连接：“短”的字面意思是维持时间短，「连接在请求结束之后就被释放了，也就是说程序无法复用这条连接，每次发起请求都要建立新的连接」。这就好比红蓝两军在交战，红军在追击蓝军，蓝军过河时搭建了一座桥，通过之后立马把桥炸毁，红军要过河只能再重新建桥。

**短链接耗时更长**

* 连接建立，需要三次握手，需要的时间成本至少为「1个RTT时间」。
* 连接释放，需要四次挥手，需要的时间成本至少为「1个RTT时间，但是因为close调用默认是立即返回的，后续由操作系统完成剩余的报文交互，故连接释放不会阻塞进程，所以这个成本可以忽略不计」

长短连接的优缺点如下表所示。



**点击图片可查看完整电子表格**

**请求一个url全过程**

1. **DNS查询**
2. 浏览器查看浏览器缓存-->系统缓存-->查找本地host文件-->本地DNS服务器缓存，如果缓存中有，则直接在屏幕中显示页面内容。若没有，则进行DNS解析，获取相应的IP地址，本地域名服务器会向 根域名服务器发送一个请求。如果根域名服务器也不存在该域名时，本地域名会向顶级域名服务器（TLD）发送一个请求，即根域名服务器-->顶级域名服务器-->权威域名服务器依次类推下去。直到最后本地域名服务器得到URL对应的IP地址并把它缓存到本地，供下次查询使用。
3. **TCP连接**
4. **发送HTTP请求**
5. **服务器处理HTTP请求并返回HTTP报文**
6. **浏览器解析并render页面**
7. **HTTP连接断开**

**HTTP解决粘包问题的几种方法**

HTTP协议在处理网络请求时，经常会遇到粘包问题，即多个请求数据包被合并成一个数据包发送。为了解决这个问题，HTTP采用了以下几种方法：

1. 使用分隔符的方法

* 在每个请求的末尾添加一个特定的分隔符，例如换行符或自定义的特殊字符。服务器在接收到请求时，根据分隔符将其分割成单个请求进行处理。

1. 使用固定长度的方法

* 在请求头中指定请求的长度，服务器根据指定的长度读取数据包。如果数据包长度不足，则等待直到接收到完整的请求。

1. 使用头部解析的方法

* HTTP请求中的每个请求都以一个请求头开始。服务器可以解析每个请求头，并根据头部中的信息判断这个请求是否完整。如果请求头中没有指定长度或其他有用的信息，服务器可以根据数据包的大小和请求头的大小来判断是否存在粘包问题。

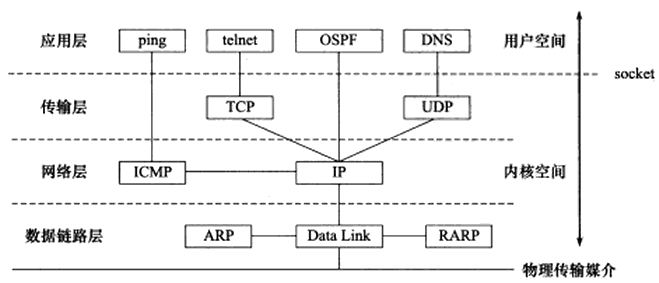
1. 使用[消息边界](https://www.baidu.com/s?wd=%E6%B6%88%E6%81%AF%E8%BE%B9%E7%95%8C&usm=3&ie=utf-8&rsv_pq=da46e50500d7a013&oq=http%E5%A6%82%E4%BD%95%E8%A7%A3%E5%86%B3%E7%B2%98%E5%8C%85&rsv_t=bd4bK9pQI9HAKlM5%2BEplNeN0bzCzp%2FrQtj6GzL8B72vXDpTHR2qEk6oBkUQ&sa=re_dqa_generate)的方法

* HTTP请求的每个请求都是以两个换行符(\r  
  \r  
  )作为结束标记。服务器可以通过判断消息边界来确定一个请求是否完整。如果数据包中存在多个请求，服务器可以通过查找换行符的位置来分割请求。

1. 独立处理每个请求的方法

**网络编程（c++为例）**

**基础协议展示**



**TCP UDP编程**

**大小端检测**

Linux提供了如下4个函数来完成主机字节序和网络字节序之间的转换:

|  |
| --- |
| C++ #include <netinet/in.h> unsigned long int htonl( unsigned long int hostlong ); unsigned short int htons( unsigned short int hostshort ); unsigned long int ntohl( unsigned long int netlong ); unsigned short int ntohs( unsigned short int netshort ); |

它们的含义很明确，比如 htonl 表示“host to network long”，即将长整型(32 bit)的主

机字节序数据转化为网络字节序数据。这 4 个函数中，长整型函数通常用来转换IP地址，短

整型函数用来转换端口号(当然不限于此。任何格式化的数据通过网络传输时，都应该使用

这些函数来转换字节序)。

大端序（Big-Endian）是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中。网络传输使用大端。

|  |
| --- |
| C++  void byteorder() {  union{  short value;  char union\_bytes[sizeof(short)];  }test;   test.value = 0x0102;  if((test.union\_bytes[0] == 1) && (test.union\_bytes[1] == 2))  {  cout<<"big endian\n";  }else{  cout<<"little endian\n";  } } |

**socket结构体**

TCP/IP 协议族有 sockaddr\_in 和 sockaddr\_in6 两个专用 socket 地址结构体，它们分别用

于IPv4和IPv6:

|  |
| --- |
| C++ struct sockaddr\_in {  sa\_family\_t sin\_family;  /\* 地址族:AF INET\*/  u\_intl6\_t sin\_port;  /\*端口号，要用网络字节序表示\*/  struct in\_addr sin\_addr;/\*Ipv4 地址结构体，见下面\*/ }; struct in\_addr  u\_int32\_t s\_addr;  /\*IPv4地址，要用网络字节序表示\*/ }; struct sockaddr\_in6  sa\_family\_t sin6\_family;  /\* 地址族:AF INET6 \*/  u\_intl6\_t sin6\_port;  /\**端口号，要用网络字节序表示\**/  u\_int32\_t sin6\_flowinfo;  /\*流信息，应设置为0\*/  struct in6\_addr sin6\_addr;  /\*IPv6地址结构体，见下面\*/  u\_int32\_t sin6\_scope\_id;  /\*scopeID，尚处于实验阶段\*/ }; struct in6\_addr  unsigned char sa\_addr[16];  /\*IPv6地址，要用网络字节序表示\*/ }; |

**IP地址转换函数**

|  |
| --- |
| C++ #include <arpa/inet.h> in\_addr\_t inet\_addr( const char\* strptr ); int inet\_aton( const char\* cp, struct in\_addr\* inp ); char\* inet ntoa( struct in addr in ); |

**inet\_addr** 函数将用点分十进制字符串表示的IPv4地址转化为用网络字节序整数表示的 IPv4 地址。它失败时返回 INADDR NONE。

**inet\_aton** 函数完成和 **inet\_addr** 同样的功能，但是将转化结果存储于参数 inp指向的地址结构中。它成功时返回 1，失败则返回 0。

**inet\_ntoa** 函数将用网络字节序整数表示的IPv4地址转化为用点分十进制字符串表示的 IPv4 地址。但需要注意的是，该函数内部用一个静态变量存储转化结果，函数的返回值指向该静态内存，因此inet\_ntoa 是不可重入的。

下面这对更新的函数也能完成和前面3个函数同样的功能，并且它们同时适用于IPv4

地址和IPv6 地址:

|  |
| --- |
| C++ #include <arpa/inet.h> int inet\_pton( int af, const char\* src, void\* dst ); const char\* inet\_ntop( int af, const void\* src, char\* dst, socklen\_t cnt ); |

**inet\_pton** 函数将用字符串表示的IP地址 src(用点分十进制字符串表示的IPv4 地址或用十六进制字符串表示的IPv6 地址)转换成用网络字节序整数表示的IP地址，并把转换结果存储于 dst 指向的内存中。其中，a f参数指定地址族，可以是 AF\_INET 或者 AF\_INET6。**inet\_pton** 成功时返回 1，失败则返回 0 并设置 errno。

**inet\_ ntop** 函数进行相反的转换，前三个参数的含义与 inet\_pton 的参数相同，最后一个参数cnt 指定目标存储单元的大小。下面的两个宏能帮助我们指定这个大小(分别用于 IPv4 和 IPv6):

|  |
| --- |
| C++ #include <netinet/in.h> #define INET\_ADDRSTRLEN 16 #define`INET6\_ADDRSTRLEN 46 |

**inet\_ntop** 成功时返回目标存储单元的地址，失败则返回 NULL 并设置 errno。

**socket 创建**

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> int socket( int domain, int type, int protocol ); |

**domain** 参数告诉系统使用哪个底层协议族。对TCP/IP协议族而言，该参数应该设置为 PF\_INET(ProtocolFamily of Internet，用于IPv4) 或 PF\_INET6(用于IPv6); 对于 UNIX 本地域协议族而言，该参数应该设置为 PF\_UNIX。关于 socket 系统调用支持的所有协议族，请读者自己参考其 man手册。

**type** 参数指定服务类型。服务类型主要有 SOCK\_STREAM 服务(流服务)和 SOCK\_UGRAM(数据报)服务。对TCP/IP协议族而言，其值取 SOCK\_STREAM 表示传输层使用 TCP 协议，取 SOCK\_DGRAM 表示传输层使用 UDP 协议。

|  |
| --- |
| 值得指出的是，自 Linux 内核版本 2.6.17 起，type 参数可以接受上述服务类型与下面两个重要的标志相与的值:SOCK\_NONBLOCK 和 SOCK\_CLOEXEC。它们分别表示将新创建的 socket 设为非阻塞的，以及用 fork 调用创建子进程时在子进程中关闭该 socket。在内核版本2.6.17 之前的Linux 中，文件描述符的这两个属性都需要使用额外的系统调用(比如 fcntl )来设置。 |

**protocol** 参数是在前两个参数构成的协议集合下，再选择一个具体的协议。不过这个值通常都是唯一的(前两个参数已经完全决定了它的值)。几乎在所有情况下，我们都应该把它设置为 0，表示使用默认协议。

**命名 socket（bind）**

socket 系统调用成功时返回一个 socket 文件描述符，失败则返回-1 并设置 errno。创建 socket时，我们给它指定了地址族，但是并未指定使用该地址族中的哪个具体 socket 地址。将一个 socket 与 socket 地址绑定称为给 socket 命名。在服务器程序中，我们通常要命名 socket，因为只有命名后客户端才能知道该如何连接它。客户端则通常不需要命名 socket，而是采用匿名方式，即使用操作系统自动分配的 socket 地址。命名 socket 的系统调用是 bind，其定义如下:

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> int bind( int sockfd, const struct sockaddr\* my\_addr, socklen\_t addrlen ); |

**bind** 将 my\_addr 所指的 socket 地址分配给未命名的 sockfd 文件描述符，addrlen 参数指出该socket 地址的长度。**bind** 成功时返回 0，失败则返回-1 并设置errno。其中两种常见的 errno 是EACCES 和 EADDRINUSE，它们的含义分别是:

* EACCES，被绑定的地址是受保护的地址，仅超级用户能够访问。比如普通用户将 socket 绑定到知名服务端口(端口号为 0~1023)上时，bind 将返回EACCES 错误。
* EADDRINUSE，被绑定的地址正在使用中。比如将 socket 绑定到一个处于 TIME\_WAIT 状态的 socket 地址。

**被动监听 socket （listen）**

socket 被命名之后，还不能马上接受客户连接，我们需要使用如下系统调用来创建一个监听队列以存放待处理的客户连接:

|  |
| --- |
| C++ #include <sys/socket.h> int listen( int sockfd, int backlog ); |

sockfd 参数指定被监听的 socket。backlog参数提示内核监听队列的最大长度。监听队列的长度如果超过backlog，服务器将不受理新的客户连接，客户端也将收到 ECONNREFUSED 错误信息。在内核版本 2.2 之前的 Linux 中，backlog 参数是指所有处于半连接状态(SYNRCVD)和完全连接状态(ESTABLISHED)的 socket 的上限。但自内核版

本2.2之后，它只表示处于完全连接状态的 socket 的上限，处于半连接状态的 socket的上限则由 /proc/sys/net/ipv4/tcp\_max\_syn\_backlog 内核参数定义。backlog 参数的典型值是 5。

**listen** 成功时返回 0，失败则返回-1 并设置errno。

**服务器接受连接（listen）**

下面的系统调用从listen监听队列中接受一个连接:

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> int accept( int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen ); |

sockfd 参数是执行过 listen 系统调用的监听 socket。addr 参数用来获取被接受连接的远端 socket 地址，该 socket 地址的长度由 addrlen参数指出。accept 成功时返回一个新的连接 socket，该 socket 唯一地标识了被接受的这个连接，服务器可通过读写该 socket 来与被接受连接对应的客户端通信。**accept**失败时返回-1并设置errno。

由此可见，**accept** 只是从监听队列中取出连接，而不论连接处于何种状态(如上面的 ESTABLISHED状态和 CLOSE\_WAIT 状态)，更不关心任何网络状况的变化。

**客户端主动连接（connect）**

如果说服务器通过Iisten调用来被动接受连接，那么客户端需要通过如下系统调用来主动与服务器建立连接:

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> int connect( int sockfd, const struct sockaddr \*serv\_addr, socklen\_t addrlen ); |

sockfd 参数由 socket 系统调用返回一个 socket。serv\_addr 参数是服务器监听的 socket 地

址，addrlen参数则指定这个地址的长度。

**connect** 成功时返回0。一旦成功建立连接，sockfd就唯一地标识了这个连接，客户端就可以通过读写 sockfd 来与服务器通信。connect 失败则返回-1并设置errno。其中两种常见的 errno是 ECONNREFUSED 和 ETIMEDOUT，它们的含义如下:

* ECONNREFUSED，目标端口不存在，连接被拒绝。
* ETIMEDOUT，连接超时。

**关闭连接**

关闭一个连接实际上就是关闭该连接对应的 socket，这可以通过如下关闭普通文件描述符的系统调用来完成:

|  |
| --- |
| C++ #include <unistd.h> int close( int fd ); |

fd 参数是待关闭的 socket。不过，close 系统调用并非总是立即关闭一个连接，而是将 fd 的引用计数减1。只有当fd 的引用计数为0时，才真正关闭连接。多进程程序中，一次 fork 系统调用默认将使父进程中打开的socket的引用计数加1，因此我们必须在父进程和子进程 中都对该 socket 执行 close 调用才能将连接关闭。如果无论如何都要立即终止连接(而不是将socket 的引用计数减1)，可以使用如下的 shutdown 系统调用(相对于close来说，它是专门为网络编程设计的):

|  |
| --- |
| C++ #include <sys/socket.h> int shutdown( int sockfd, int howto ); |

sockfd 参数是待关闭的 socket。howto 参数决定了 shutdown 的行为，它可取表 5-3 中的某个值。

**SHUT \_RD**

关闭sockfd上读的这一半。应用程序不能再针对socket文件描述符执行读操作，并且该socket接收缓冲区中的数据都被丢弃

**SHUT\_WR**

关闭sockfd上写的这一半.sockfd的发送缓冲区中的数据会在真正关闭连接之前全部发送出去，应用程序不可再对该 socket文件描述符执行写操作。这种情况下，连接处于半关闭状态

**SHUT RDWR**

同时关闭 sockfd上的读和写

由此可见，shutdown 能够分别关闭 socket上的读或写，或者都关闭。而 close 在关闭连接时只能将 socket上的读和写同时关闭。shutdown 成功时返回0，失败则返回-1并设置 errno。

**TCP 数据读写**

对文件的读写操作 read 和 write 同样适用于 socket。但是 socket 编程接口提供了几个专门用于 socket 数据读写的系统调用，它们增加了对数据读写的控制。其中用于TCP流数据读写的系统调用是:

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> ssize\_t recv( int sockfd, void \*buf, size\_t len, int flags ); ssize\_t send( int sockfd, const void +buf, size\_t len, int flags ); |

recv 读取 sockfd上 的数据，buf 和 lcn 参数分别指定读缓冲区的位置和大小，flags参数通常设置为0即可。recv 成功时返回实际读取到的数据的长度，它可能小于我们期望的长度len。因此我们可能要多次调用recv，才能读取到完整的数据。recv 可能返回 0，这意味着通信对方已经关闭连接了。recv出错时返回-1并设置errno。send往 sockfd上写入数据，buf和len参数分别指定写缓冲区的位置和大小。send 成功时返回实际写人的数据的长度，失败则返回 -1 并设置 ermo.

**UDP 数据读写**

socket编程接口中用于UDP数据报读写的系统调用是:

|  |
| --- |
| C++ #include <sys/types.h> #include <sys/socket.h> ssize\_t recvfrom( int sockfd, void\* buf, size\_t len, int flags, struct sockaddr\* src\_addr, socklen\_t\* addrlen); ssize\_t sendto( int sockfd, const void\* buf, size\_t len, int flags, const struct sockaddr\* dest\_addr, socklen\_t addrlen ); |

recvfrom 读取 sockfd 上的数据，buf 和len 参数分别指定读缓冲区的位置和大小。因为 UDP 通信没有连接的概念，所以我们每次读取数据都需要获取发送端的 socket 地址，即参数 src\_addr 所指的内容，addrlen 参数则指定该地址的长度。sendto 往 sockfd 上写人数据，buf 和 len 参数分别指定写缓冲区的位置和大小。dest\_addr

参数指定接收端的 socket 地址，addrlen 参数则指定该地址的长度。这两个系统调用的flags参数以及返回值的含义均与 send/recv系统调用的flags参数及返回值相同。

值得一提的是，recvfrom/sendto 系统调用也可以用于面向连接(STREAM)的 socket 的数据读写，只需要把最后两个参数都设置为NULL以忽略发送端/接收端的 socket地址(因为我们已经和对方建立了连接，所以已经知道其socket 地址了)。

**地址信息函数**

在某些情况下，我们想知道一个连接 socket 的本端 socket 地址，以及远端的 socket 地址。下面这两个函数正是用于解决这个问题:

|  |
| --- |
| C++ #include <sys/socket.h> int getsockname( int sockfd, struct sockaddr\* address, socklen\_t\* address\_len ); int getpeername( int sockfd, struct sockaddr\* address, socklen\_t\* address\_len ); |

**getsockname** 获取 sockfd 对应的本端 socket 地址，并将其存储于address参数指定的内存中，该 socket 地址的长度则存储于 address\_len 参数指向的变量中。如果实际 socket 地址的长度大于 address 所指内存区的大小，那么该 socket 地址将被截断。getsockname 成功时返回 0，失败返回-1并设置errmo。

**getpeermame** 获取 sockfd 对应的远端 socket 地址，其参数及返回值的含义与 getsockname 的参数及返回值相同。

**获得主机完整信息**

gethostbyname 函数根据主机名称获取主机的完整信息，gethostbyaddr 函数根据 ip 地址获取主机的完整信息。gethostbyname 函数通常先在本地的/etc/hosts配置文件中查找主机，如果没有找到，再去访问 DNS 服务器。这些这两个函数的定义如下:

|  |
| --- |
| C++ #include <netdb.h> struct hostent\* gethostbyname( const char\*name ); struct hostent\* gethostbyaddr( const void\* addr, size\_t len, int type ); |

name 参数指定目标主机的主机名，addr 参数指定目标主机的IP地址，len 参数指定 addr 所指IP 地址的长度，type 参数指定 addr 所指 IP 地址的类型，其合法取值包括 AF\_INET(用于IPv4地址)和 AF\_INET6(用于IPv6地址)。

这两个函数返回的都是 hostent 结构体类型的指针，hostent 结构体的定义如下:

|  |
| --- |
| C++ #include <netdb.h> struct hostent{  char\* h\_name;  /\*主机名\*/  char\*\* h aliases;  /\*  主机别名列表，可能有多个\*/  int h\_addrtype;  /\*地址类型(地址族)\*/  int h \_length;  /\*地址长度\*/  char\*\* h\_addr\_list  /\*按网络字节序列出的主机IP地址列表\*/ }; |

**获得服务信息**

getservbyname 函数根据名称获取某个服务的完整信息，getservbyport 函数根据端口号获取某个服务的完整信息。它们实际上都是通过读取/etc/services 文件来获取服务的信息的。

这两个函数的定义如下:

|  |
| --- |
| C++ #include <netdb.h> struct servent\* getservbyname( const char\* name, const char\* proto ); struct servent\* getservbyport( int port, const char\* proto ); |

name 参数指定目标服务的名字，port参数指定目标服务对应的端口号。proto 参数指定服务类型，给它传递“tcp”表示获取流服务，给它传递“udp”表示获取数据报服务，给它传递 NULL则表示获取所有类型的服务。这两个函数返回的都是 servent 结构体类型的指针，结构体 servent的定义如下:

|  |
| --- |
| C++ #include <netdb.h> struct servent{  char\* s\_name;  /\*服务名称\*/  char\*\* s\_aliases;  /\*服务的别名列表，可能有多个\*/  int s\_port;  /\*端口号\*/  char\* s\_proto;  /\*服务类型，通常是tcp或者udp\*/ }; |

**获得主机名和服务名**

getnameinfo 函数能通过socket地址同时获得以字符串表示的主机名(内部使用的是 gethostbyaddr 函数)和服务名(内部使用的是 getservbyport 函数)。它是否可重人取决于其内部调用的 gethostbyaddr 和 getservbyport 函数是否是它们的可重入版本。该函数的定义如下:

|  |
| --- |
| C++ #include <netdb.h> int getnameinfo( const struct sockaddr\* sockaddr, socklen\_t addrlen, char\* host, socklen\_t hostlen, char\* serv, socklen\_t servlen, int flags ); |

getnameinfo 将返回的主机名存储在host参数指向的缓存中，将服务名存储在 serv 参数指向的缓存中，hostlen和 servle n参数分别指定这两块缓存的长度。flags 参数控制 getnameinfo 的行为

**高级 IO 函数**

**pipe 函数**

pipe 函数可用于创建一个管道，以实现进程间通信。我们将在 13.4节讨论如何使用管道来实现进程间通信，本章只介绍其基本使用方式。pipe 函数的定义如下:

|  |
| --- |
| C++ #include <unistd.h> int pipe( int fdl2] ); |

pipe 函数的参数是一个包含两个int 型整数的数组指针。该函数成功时返回 0，并将一对打开的文件描述符值填入其参数指向的数组。如果失败，则返回-1并设置errno。通过 pipe 函数创建的这两个文件描述符 fd[0]和 fd[1]分别构成管道的两端，往 fd[1]写人的数据可以从fd[0] 读出。并且，fd[0]只能用于从管道读出数据，fd[1]则只能用于往管道写入数据，而不能反过来使用。如果要实现双向的数据传输，就应该使用两个管道。默认情况下，这一对文件描述符都是阻塞的。此时如果我们用read系统调用来读取一个空的管道,则 read 将被阻塞，直到管道内有数据可读;如果我们用write系统调用来往一个满的管道(见后文)中写人数据，则write 亦将被阻塞，直到管道有足够多的空闲空间可用。但如果应用程序将fd[0] 和 fd[I1]都设置为非阻塞的，则 read 和write 会有不同的行为。关于阻塞和非阻道读走的话，该管道最多能被写人多少字节的数据。自Linux 2.6.11内核起，管道容量的大小默认是65536字节。我们可以使用fcntl 函数来修改管道容量(见后文)。此外，socket 的基础 API 中有一个 socketpair 函数。它能够方便地创建双向管道。其定义如下:

|  |
| --- |
| C++ #include<sys/types.h> #include<sys/socket.h> int socketpair(int domain, int type, int protocol, int fdl2] ); |

socketpair 前三个参数的含义与 socket 系统调用的三个参数完全相同，但 domain只能使用UNIX本地域协议族AF\_UNIX，因为我们仅能在本地使用这个双向管道。最后一个参数则和 pipe 系统调用的参数一样，只不过 socketpair 创建的这对文件描述符都是既可读又可写的。socketpair 成功时返回 0，失败时返回-1 并设置errno。

**dup 和 dup2 函数**

有时我们希望把标准输人重定向到一个文件，或者把标准输出重定向到一个网络连接(比如CGI 编程)。这可以通过下面的用于复制文件描述符的 dup 或 dup2函数来实现:

|  |
| --- |
| C++ #include <unistd.h> int dup( int file\_descriptor ); int dup2( int file\_descriptor\_one, int file\_descriptor\_two ); |

dup 函数创建一个新的文件描述符，该新文件描述符和原有文件描述符 file\_descriptor 指向相同的文件、管道或者网络连接。并且dup 返回的文件描述符总是取系统当前可用的最小整数值。dup2 和 dup 类似，不过它将返回第一个不小于file\_descriptor\_two 的整数值。dup 和 dup2 系统调用失败时返回-1 并设置errno。

**readv 函数和 writev 函数**

readv 函数将数据从文件描述符读到分散的内存块中，即分散读;writcv 函数则将多块分散的内存数据一并写人文件描述符中，即集中写。它们的定义如下:

|  |
| --- |
| C++ include <sys/uio.h> ssize\_t readv( int fd, const struct iovec\* vector, int count): ssize\_t writev( int fd, const struct iovec\* vector, int count ); |

fd参数是被操作的目标文件描述符。vector 参数的类型是 iovec 结构数组。我们在第 5 章讨论过结构体 iovec，该结构体描述一块内存区。count 参数是 vector 数组的长度，即有多少块内存数据需要从fd 读出或写到 fd。readv 和 writev 在成功时返回读出/写人fd 的字节数，失败则返回-1并设置 errno。它们相当于简化版的 recvmsg 和 sendmsg 函数。考虑第 4 章讨论过的 Web 服务器。当Web服务器解析完一个 HTTP 请求之后，如果目标文档存在且客户具有读取该文档的权限，那么它就需要发送一个 HTTP 应答来传输该文档。这个HTTP应答包含1个状态行、多个头部字段、1个空行和文档的内容。其中，前3部分的内容可能被Wcb服务器放置在一块内存中，而文档的内容则通常被读人到另外一块单独的内存中(通过 read 函数或mmap 函数)。我们并不需要把这两部分内容拼接到一起再发送，而是可以使用writev 函数将它们同时写出，如代码清单6-2所示。

**sendfile 函数**

sendfle 函数在两个文件描述符之间直接传递数据(完全在内核中操作)，从而避免了内核缓冲区和用户缓冲区之间的数据拷贝，效率很高，这被称为零拷贝。sendfle 函数的定义如下:

|  |
| --- |
| C++ #include <sys/sendfile.h> ssize\_t sendfilelint out\_fd, int in\_fd, off\_t\* offset, size\_t count ); |

in\_fd参数是待读出内容的文件描述符，out\_fd参数是待写入内容的文件描述符。offset 参数指定从读人文件流的哪个位置开始读，如果为空，则使用读人文件流默认的起始位置。count参数指定在文件描述符in\_fd和 out\_fd之间传输的字节数。sendfile 成功时返回传输的字节数，失败则返回-1并设置errno。该函数的 man 手册明确指出，in\_fd必须是一个支持类似 mmap 函数的文件描述符，即它必须指向真实的文件，不能是 socket 和管道;而 out\_fd 则必须是一个 socket。由此可见，sendfile几乎是专门为在网络上传输文件而设计的。

**mmap 函数和 munmap 函数**

mmap 函数用于申请一段内存空间。我们可以将这段内存作为进程间通信的共享内存，也可以将文件直接映射到其中。munmap 函数则释放由 mmap创建的这段内存空间。它们的定义如下:

|  |
| --- |
| C++ #include <sys/mman.h> void\* mmap( void \*start, size\_t length, int prot, int flags, int fd, off\_t offset ); int munmap( void \*start, size\_t length ) |

start 参数允许用户使用某个特定的地址作为这段内存的起始地址。如果它被设置成 NULL，则系统自动分配一个地址。length参数指定内存段的长度。prot 参数用来设置内存段的访问权限。它可以取以下几个值的按位或:

* PROT\_READ，内存段可读。
* PROT\_WRITE，内存段可写。
* PROT\_EXEC，内存段可执行。
* PROT\_NONE，内存段不能被访问。

fags参数控制内存段内容被修改后程序的行为。它可以被设置为表6-1中的某些值(这里仅列出了常用的值)的按位或(其中MAP\_SHARED和MAP\_PRIVATE是互斥的，不能同时指定)。

fd参数是被映射文件对应的文件描述符。它一般通过open 系统调用获得。offset参数设置从文件的何处开始映射(对于不需要读入整个文件的情况)。

mmap函数成功时返回指向目标内存区域的指针，失败则返回 MAP\_FAILED((void\*)-1)并设置errno。munmap 函数成功时返回 0，失败则返回-1并设置ermo。

**splice 函数**

splice 函数用于在两个文件描述符之间移动数据，也是零拷贝操作。splice 函数的定义如下:

|  |
| --- |
| C++ #include <fcntl.h> ssize\_t splice( int fd\_in, loff\_t\* off\_in, int fd\_out, loff\_t\* off\_out, size\_t len, unsigned int flags ); |

fd\_in 参数是待输人数据的文件描述符。如果fd\_in 是一个管道文件描述符，那么off\_in参数必须被设置为NULL。如果fd\_in不是一个管道文件描述符(比如socket)，那么offin表示从输入数据流的何处开始读取数据。此时，若offin 被设置为 NULL，则表示从输入数据流的当前偏移位置读人;若off\_in不为 NULL，则它将指出具体的偏移位置。fd\_out/off\_out参数的含义与fd\_in/off\_in相同，不过用于输出数据流。len参数指定移动数据的长度:flags参数则控制数据如何移动，它可以被设置为表6-2 中的某些值的按位或。

使用 splice 函数时，fd\_in 和 fd\_out 必须至少有一个是管道文件描述符。splice函数调用成功时返回移动字节的数量。它可能返回0，表示没有数据需要移动，这发生在从管道中读取数据(fd\_in是管道文件描述符)而该管道没有被写入任何数据时。splice 函数失败时返回-1并设置crmo.

**tee 函数**

tee 函数在两个管道文件描述符之间复制数据，也是零拷贝操作。它不消耗数据，因此源文件描述符上的数据仍然可以用于后续的读操作。tee 函数的原型如下:

|  |
| --- |
| C++ #include <fcntl.h> ssize\_t tee( int fd\_in, int fd\_out, size\_t len, unsigned int lags ); |

该函数的参数的含义与splice相同(但fd\_in和fd\_out 必须都是管道文件描述符)。tee 函数成功时返回在两个文件描述符之间复制的数据数量(字节数)。返回0表示没有复制任何数据.tee 失败时返回-1并设置errno。

**fcntl 函数**

fcntl 函数，正如其名字(file control)描述的那样，提供了对文件描述符的各种控制操作。另外一个常见的控制文件描述符属性和行为的系统调用是 ioctl，而且 ioctl 比 fcntl 能够执行更多的控制。但是，对于控制文件描述符常用的属性和行为，fcntl 函数是由 POSIX 规范指定的首选方法。所以本书仅讨论 fcntl 函数。fcntl 函数的定义如下:

|  |
| --- |
| C++ #include <fcntl.h2> int fcntl( int fd, int cmd, ... ); |

fd 参数是被操作的文件描述符，cmd 参数指定执行何种类型的操作。根据操作类型的不同，该函数可能还需要第三个可选参数arg。

**I/O 复用**

**select系统调用**

select 系统调用的用途是:在一段指定时间内，监听用户感兴趣的文件描述符上的可读、可写和异常等事件。本节先介绍 select 系统调用的 API，然后讨论 select判断文件描述符就绪的条件，最后给出它在处理带外数据中的实际应用。

**select** 系统调用的原型如下:

|  |
| --- |
| C++ #include <sys/select.h> int select( int nfds, fd\_set\* readfds, fd\_set\* writefds, fd\_set\* exceptfds, struct timeval\* timeout ); |

* nfds参数指定被监听的文件描述符的总数。它通常被设置为 select监听的所有文件描述符中的最大值加1，因为文件描述符是从0开始计数的。
* readfds、writefds和 exceptfds参数分别指向可读、可写和异常等事件对应的文件描述符集合。应用程序调用select函数时，通过这3 个参数传人自己感兴趣的文件描述符。

**select** 调用返回时，内核将修改它们来通知应用程序哪些文件描述符已经就绪。

**poll系统调用**

poll系统调用和 select 类似，也是在指定时间内轮询一定数量的文件描述符，以测试其中是否有就绪者。poll的原型如下:

|  |
| --- |
| C++ #include <po11.h> int poll( struct pollfd\* fds, nfds\_t nfds, int timeout ); |

fds参数是一个 pollfd 结构类型的数组，它指定所有我们感兴趣的文件描述符上发生的可读、可写和异常等事件。pollfd 结构体的定义如下:

|  |
| --- |
| C++ struct poiifd{  int fd;  /\*文件描述符\*/  short events;  /\*注册的事件\*/  short revents;  /\*实际发生的事件，由内核填充\*/ }; |

其中，fd 成员指定文件描述符:events 成员告诉 poll监听fd上的哪些事件，它是一系列事件的按位或:revents成员则由内核修改，以通知应用程序fd上实际发生了哪些事件。

**epoll系统调用**

epoll是 Linux 特有的 I/O 复用函数。它在实现和使用上与 select、poll有很大差异。首先，epoll 使用一组函数来完成任务，而不是单个函数。其次，epoll把用户关心的文件描述符上的事件放在内核里的一个事件表中，从而无须像 select和 poll那样每次调用都要重复传入文件描述符集或事件集。但epoll 需要使用一个额外的文件描述符，来唯一标识内核中的这个事件表。这个文件描述符使用如下epoll\_create 函数来创建:

|  |
| --- |
| C++ #include <sys/epoll.h> int epoll\_create( int size ) |

size 参数现在并不起作用，只是给内核一个提示，告诉它事件表需要多大。该函数返回的文件描述符将用作其他所有cpoll系统调用的第一个参数，以指定要访问的内核事件表。

|  |
| --- |
| C++ #include <sys/epoll.h> int epoll\_ctl( int epfd, int op, int fd, struct epoll\_event \*event ) |

fd参数是要操作的文件描述符，op参数则指定操作类型。操作类型有如下 3 种:

* EPOLL\_CTL\_ADD，往事件表中注册 fd上的事件。
* EPOLL\_CTL\_MOD，修改 fd上的注册事件。
* EPOLL\_CTL\_DEL，删除 fd上的注册事件。

event 参数指定事件，它是 epoll\_event 结构指针类型。epoll\_event 的定义如下:

|  |
| --- |
| C++ struct epoll\_event{  uint32\_t events;  /\*epoll事件 \*/  epoll\_data\_t data;/\*用户数据\*/ }; |

其中 events 成员描述事件类型。epoll支持的事件类型和 poll基本相同。表示 epoll事件类型的宏是在 poll 对应的宏前加上“E”，比如 epoll 的数据可读事件是 EPOLLIN。但 epoll有两个额外的事件类型 EPOLLET 和 EPOLLONESHOT。它们对于epoll的高效运作非常关键。

epoll 对文件描述符的操作有两种模式:LT(LevelTrigger，电平触发)模式和 ET(Edge Trigger，边沿触发)模式。LT 模式是默认的工作模式，这种模式下 cpoll 相当于一个效率较高的 poll。当往 epoll 内核事件表中注册一个文件描述符上的 EPOLLET 事件时，epoll将以 ET 模式来操作该文件描述符。ET 模式是 epoll 的高效工作模式。对于采用LT工作模式的文件描述符，当epoll\_wait 检测到其上有事件发生并将此事件通知应用程序后，应用程序可以不立即处理该事件。这样，当应用程序下一次调用 epoll\_wait时，epoll\_wait还会再次向应用程序通告此事件，直到该事件被处理。而对于采用 ET 工作模式的文件描述符，当epoll\_wait 检测到其上有事件发生并将此事件通知应用程序后，应用程序必须立即处理该事件，因为后续的 epoll\_wait 调用将不再向应用程序通知这一事件。可见，ET模式在很大程度上降低了同一个epolI事件被重复触发的次数，因此效率要比LT模式高。

即使我们使用ET模式,

一个 socket上的某个事件还是可能被触发多次。这在并发程序中就会引起一个问题。比如一个线程(或进程，下同)在读取完某个socket上的数据后开始处理这些数据，而在数据的处理过程中该 socket上又有新数据可读(EPOLLIN再次被触发)，此时另外一个线程被唤醒来读取这些新的数据。于是就出现了两个线程同时操作一个 socket 的局面。这当然不是我们期望的。我们期望的是一个 socket 连接在任一时刻都只被一个线程处理。这一点可以使用 epoll 的 EPOLLONESHOT 事件实现。

对于注册了EPOLLONESHOT 事件的文件描述符，操作系统最多触发其上注册的一个可读、可写或者异常事件，且只触发一次，除非我们使用epoll\_ctl函数重置该文件描述符上注册的 EPOLLONESHOT事件。这样，当一个线程在处理某个 socket 时，其他线程是不可能有机会操作该 socket 的。但反过来思考，注册了EPOLLONESHOT事件的 socket一旦被某个线程处理完毕，该线程就应该立即重置这个socket上的EPOLLONESHOT 事件，以确保这个 socket下一次可读时，其EPOLLIN事件能被触发，进而让其他工作线程有机会继续处理这个 socket。

epoll 系列系统调用的主要接口是**epoll\_wait**函数。它在一段超时时间内等待一组文件描述符上的事件，其原型如下:

|  |
| --- |
| C++ #include <sys/epoll.h> int epoll\_wait( int epfd, struct epoll\_event\* events, int maxevents, int timeout ); |

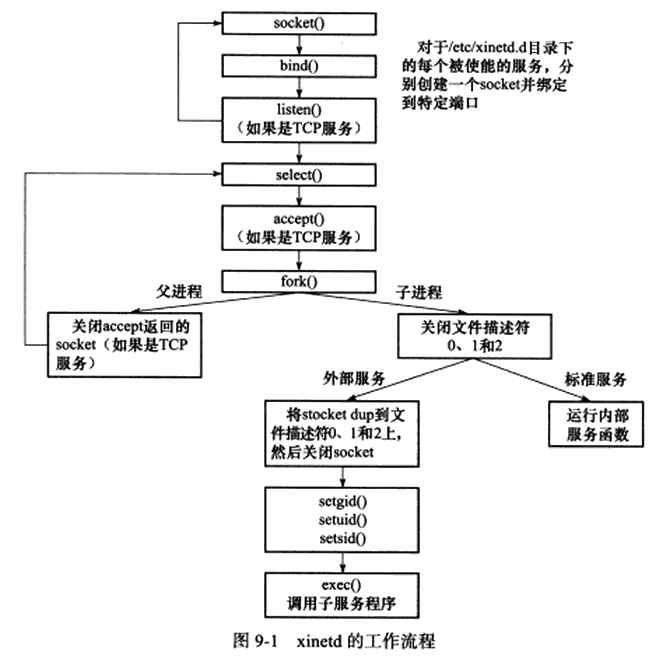
该函数成功时返回就绪的文件描述符的个数，失败时返回-1并设置errno。关于该函数的参数，我们从后往前讨论。timeout 参数的含义与 poll 接口的 timeout 参数相同。maxevents参数指定最多监听多少个事件，它必须大于 0。

epoll\_wait 函数如果检测到事件，就将所有就绪的事件从内核事件表(由epfd参数指定)中复制到它的第二个参数cvents 指向的数组中。这个数组只用于输出 epoll\_wait 检测到的就绪事件，而不像 select和 poll 的数组参数那样既用于传入用户注册的事件，又用于输出内核检测到的就绪事件。这就极大地提高了应用程序索引就绪文件描述符的效率。

**select、poll、epoll 对比**

从实现原理上来说，select和 poll采用的都是轮询的方式，即每次调用都要扫描整个注册文件描述符集合，并将其中就绪的文件描述符返回给用户程序，因此它们检测就绪事件的算法的时间复杂度是O(n)。epoll\_wait则不同，它采用的是回调的方式。内核检测到就绪的文件描述符时，将触发回调函数，回调函数就将该文件描述符上对应的事件插入内核就绪事件队列。内核最后在适当的时机将该就绪事件队列中的内容拷贝到用户空间。因此epoll\_wait 无须轮询整个文件描述符集合来检测哪些事件已经就绪，其算法时间复杂度是O(1)。但是，当活动连接比较多的时候，epoll\_wait 的效率未必比 select 和 poll 高，因为此时回调函数被触发得过于频繁。所以epoll\_wait 适用于连接数量多，但活动连接较少的情况。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | select | poll | epoll |
| 事件集合 | 用户通过三个参数分别传入感兴趣的可读，可写及异常等事件，内核通过对这些参数的现在修改来反馈其中的就绪事件，这使得用户每次调用 select 都要重置这三个参数 | 统一处理所有事件类型，因此只需一个事件集参数。用户通过pollfd.events传入感兴趣的事件，内核通过修改pollfd.events反馈其中就绪的事件 | 内核通过一个事件表直  接管理用户感兴趣的所有事件。因此每次调用epoll\_wait时，无需反复传入用户感兴趣的事件。epoll\_wait 系统调用的参数 events 仅用来反馈就绪的事件 |
| 应用程序索引就绪文件描述符的时间复杂度 | O(n) | O(n) | O(1) |
| 最大支持文件描述符数 | 一般有最大值限制 | 65535 | 65535 |
| 工作模式 | LT | LT | ET |
| 内核实现和工作效率 | 采用轮询方式检测就绪事件，复杂度为O(n) | 采用轮询方式检测就绪事件，复杂度为O(n) | 采用回调方式检测就绪事件，复杂度为O(1) |



**其他问题**

**dns对应多个ip时，如果其中一个ip对应的机器挂了**

当DNS对应多个IP地址时，如果其中一个IP地址对应的机器挂了，浏览器或其他客户端在尝试访问该域名时，会按照顺序依次尝试连接这些IP地址，直到找到一个可用的IP地址或者所有IP地址都连接失败。这个过程被称为“健康检查”或“故障切换”，旨在保证用户能够访问到可用的网站。

**Token、Session和Cookie的主要区别**

* 存储位置：
* Token：存储在客户端，通常以加密的方式存储在localStorage或sessionStorage中，或者作为请求头的一部分发送到服务器。
* Session：存储在服务器端，服务器为每个会话生成一个唯一的Session ID，并通过Cookie传递给客户端。
* Cookie：存储在客户端的浏览器中，作为文本信息保存，每次请求时客户端都会带上Cookie。
* 安全性：
* Token：通常使用加密算法生成，有效期较短且单向不可逆，可以提供较高的安全性。Token的自包含性和签名机制减少了被篡改的风险。
* Session：通过Session ID与客户端建立关联，数据存储在服务器上相对安全，但Session ID的传输存在被截获的风险。
* Cookie：可以设置为HttpOnly以增加安全性，但本身可通过不安全的HTTP传输，存在被截取的风险。
* 资源消耗：
* Token：服务器不需要存储用户信息，只需验证Token，资源消耗较低。
* Session：随着在线用户数增加，服务器需要维护大量Session数据，可能增加服务器负担。
* Cookie：不占用服务器资源，但每次请求都需要传输，可能增加网络负担。
* 跨域支持：
* Token：可以轻松实现跨域认证，因为Token是自我说明的，不受域名限制。
* Session：通常通过Cookie保存Session ID，因此不支持跨域传输。
* Cookie：不支持跨域传输，不同域名下的Cookie不能相互访问。
* 使用场景：
* Token：适用于需要高安全性和跨域支持的场景，如单点登录（SSO）系统。
* Session：适用于需要保存用户会话状态和跨请求共享数据的场景。
* Cookie：适用于需要识别用户和跟踪会话状态的场景，但需要注意安全性和容量限制。