1.为什么要学习网络协议

协议三要素

- **语法**，就是这一段内容要符合一定的规则和格式。例如，括号要成对，结束要使用分号等。
- **语义**，就是这一段内容要代表某种意义。例如数字减去数字是有意义的，数字减去文本一般来说就没有意义。
- **顺序**，就是先干啥，后干啥。例如，可以先加上某个数值，然后再减去某个数值。

常用的网络协议

一个下单过程

应用层

URL

先在浏览器里面输入 https://www.kaola.com ，浏览器只知道名字是“www.kaola.com”，但是不知道具体的地点，所以不知道应该如何访问

DNS

打开地址簿去找，也可以使用另一种更加精准的地址簿查找协议**HTTPDNS**。通过域名来对应最终的IP地址

IP地址

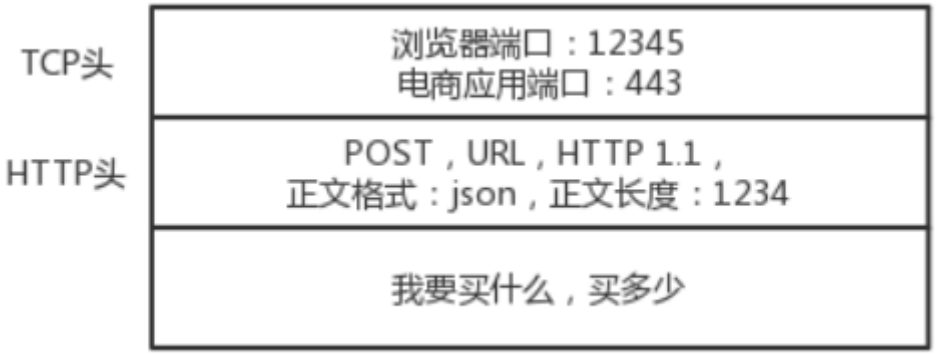
知道了目标地址，浏览器就开始打包请求

- 普通请求，使用HTTP协议
- 购物请求，加密传输使用HTTPS协议
- 

- DNS、HTTP、HTTPS所在的层我们称为**应用层**
- 应用层封装后，浏览器会将应用层的包交给下一层去完成，通过socket编程来实现

传输层

- 下一层是**传输层**。传输层有两种协议，一种是无连接的协议**UDP**，一种是面向连接的协议**TCP**。对于支付来讲，往往使用TCP协议。所谓的面向连接就是，TCP会保证这个包能够到达目的地。如果不能到达，就会重新发送，直至到达。



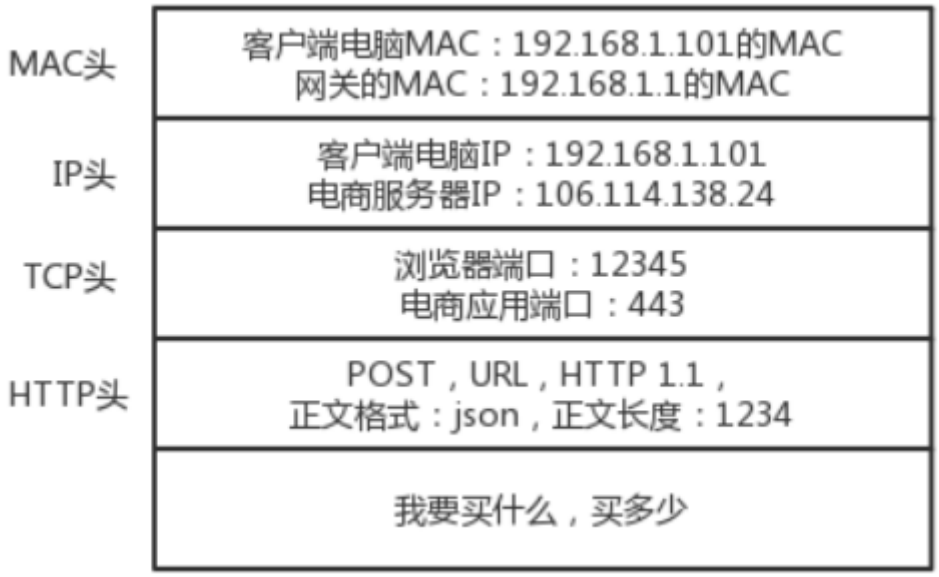
- 通过端口判断得到的包给哪个进程

网络层

传输层封装完毕后，浏览器会将包交给操作系统的**网络层**

在IP协议里面会有源IP地址，即浏览器所在机器的IP地址和目标IP地址，也即电商网站所在服务器的IP地址。

- 操作系统知道了目标IP
  - IP是本地的，从IP地址就可以看出
  - IP是外地的，就需要去找**网关**
- 操作系统启动时，就会被DHCP协议配置IP地址，及默认网关IP地址192.168.1.1
- 操作系统通过**ARP协议**将IP地址传给网关，传送到的地址为MAC地址
-



MAC层

网卡将包发出去，包里有mac地址，可以到达网关

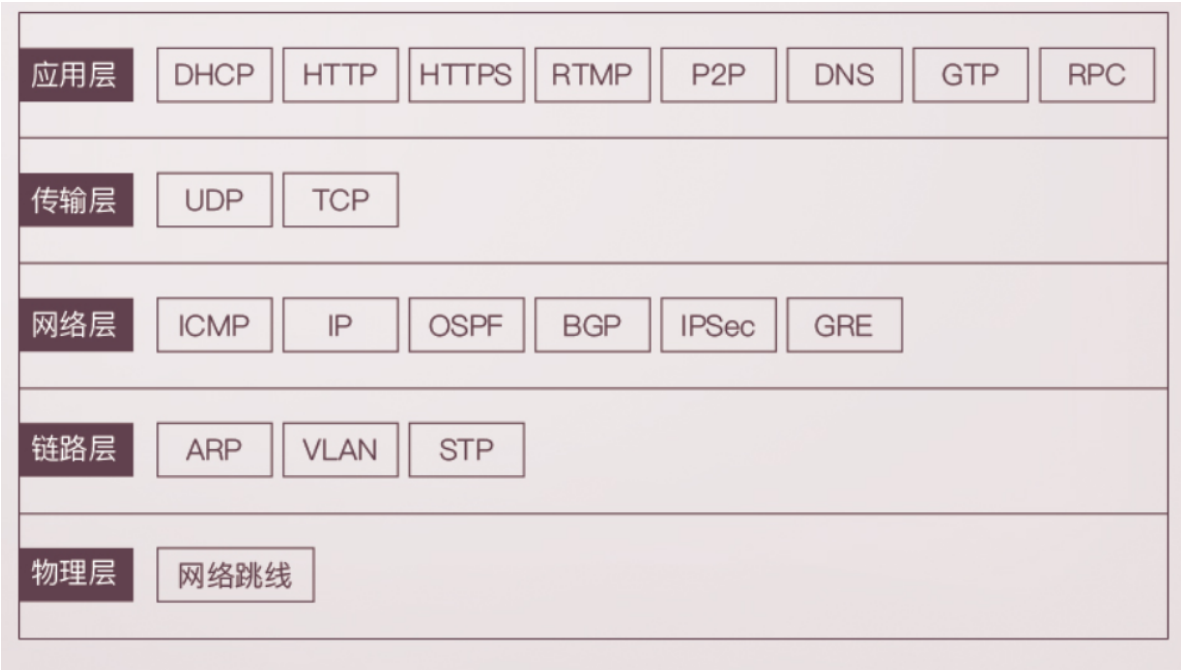
网关收到包后，根据**路由协议**（常用的有**OSPF**和**BGP**），路由到要去的最终MAC地址

最终

最终MAC地址对上了，取下MAC头，发送给操作系统网络层。IP对上，取下IP头。IP头中写了上一层封装的是TCP协议，然后将其交给传输层，即TCP层

TCP头中有目标端口号，找到电商网站的进程，解析HTTP请求内容，知道了需求，告诉相关进程

通过RPC调用来告诉相关进程，当所有对应的部门进程都处理完毕，就回复一个HTTPS包，再通过来的过程，回到个人电脑上



2.网络分层的真实含义

为什么要分层?

复杂的程序和处理过程都要分层，方便让每一层更加专注于做本层的事情

层与层之间的关系

TCP三次握手时，IP和MAC层也在做自己该做的工作，为TCP报文信息加头/解析头

只要是在网络上跑的包，都是完整的。可以有下层没上层，绝对不可能有上层没下层

3.ifconfig：最熟悉又陌生的命令行

IP 地址是一个网卡在网络世界的通讯地址，相当于我们现实世界的门牌号码

分成五类

A、B、C三类地址能包含的主机数量

类别	IP地址范围	最大主机数	私有IP地址范围
A	0.0.0.0-127.255.255.255	16777214	10.0.0.0-10.255.255.255
B	128.0.0.0-191.255.255.255	65534	172.16.0.0-172.31.255.255
C	192.0.0.0-223.255.255.255	254	192.168.0.0-192.168.255.255

无类型域间选路（CIDR）

- 将 32 位的 IP 地址一分为二，前面是**网络号**，后面是**主机号**

10.100.122.2/24

- 这个 IP 地址中有一个斜杠，斜杠后面有个数字 24，后面 24 的意思是，**32 位中，前 24 位是网络号，后 8 位是主机号**

广播地址

10.100.122.255。发送这个地址，所有10.100.122 网络里面的机器都可以收到

子网掩码

255.255.255.0。将子网掩码和IP地址按位计算AND，就能得到网络号，因为后8位全是0，取AND时仍为0，前24为全为1，取AND仍为原值

公有 IP 地址和私有 IP 地址

公有地址与外界交流

私有地址用于内部交流

举例：一个容易“犯错”的 CIDR

16.158.165.91/22

/22 不是 8 的整数倍，不好办，只能先变成二进制来看。16.158 的部分不会动，它占了前 16 位。中间的 165，变为二进制为10100101。除了前面的 16 位，还剩 6 位。所以，这 8 位中前 6 位是网络号，16.158.<101001>，而 <01>.91 是机器号

第一个地址是 16.158.<101001><00>.1，即 16.158.164.1。子网掩码是 255.255.<111111><00>.0，即 255.255.252.0。广播地址为 16.158.<101001><11>.255，即 16.158.167.255

组播地址

使用这一类地址，属于某个组的机器都能收到

ip addr分析

IP地址后的scope

eth0是global，可以对外，可以接收各个地方的包；lo是host，仅仅供本机通信

lo 全称是**loopback**，又称**环回接口**，往往会被分配到 127.0.0.1 这个地址。这个地址用于本机通信，经过内核处理后直接返回，不会在任何网络中出现

MAC地址

IP 地址的上一行是 link/ether fa:16:3e:c7:79:75 brd ff:ff:ff:ff:ff:ff

是一个网卡的物理地址，用十六进制，6 个 byte 表示

为什么不能直接用全局唯一的MAC地址通信?

一个网络包要从一个地方传到另一个地方，除了要有确定的地址，还需要有定位功能。 而有门牌号码属性的 IP 地址，才是有远程定位功能的

但MAC地址也有一定定位功能，但范围很小

所以MAC地址通讯范围较小，局限在一个子网里面

网络设备的状态标识（net_device flags）

<BROADCAST,MULTICAST,UP,LOWER_UP>

- UP 表示网卡处于启动的状态；
- BROADCAST 表示这个网卡有广播地址，可以发送广播包；
- MULTICAST 表示网卡可以发送多播包；
- LOWER_UP 表示 L1 是启动的，也即网线插着呢

MTU1500

最大传输单元 MTU 为 1500，这是以太网的默认值，MTU是二层MAC层的概念。

MAC层有MAC的头，正文（IP头、TCP头、HTTP头等）加MAC头，不允许超过1500字节，超过就要分片传输

qdisc pfifo_fast

qdisc全称**queueing discipline**，中文叫 排队规则，pfifo 先进先出

pfifo_fast 稍微复杂一些，它的队列包括三个波段（band）。在每个波段里面，使用先进先出规则

每个波段有优先级，先处理优先级高的波段

数据包是按照服务类型（**Type of Service，TOS**）被分配到三个波段（band）里面的。TOS 是 IP 头里面的一个字段，代表了当前的包是高优先级的，还是低优先级的

4.DHCP与PXE：IP是怎么来的，又是怎么没的?

如何配置IP地址?

使用 net-tools：

```
$ sudo ifconfig eth1 10.0.0.1/24
$ sudo ifconfig eth1 up
```

使用 iproute2：

```
$ sudo ip addr add 10.0.0.1/24 dev eth1
$ sudo ip link set up eth1
```

为什么同一个交换机上，我地址是16.158.23.6，去ping另一台192.168.1.6，却ping不通?

- 包不完整，MAC层填不了
- Linux默认逻辑，如果是一个跨网段的调用，不会直接将包发送到网络上，而是企图将包发送到网关
- 虽然目标IP是对的，但是MAC地址匹配不上，就无法将包收入

动态主机配置协议（DHCP）

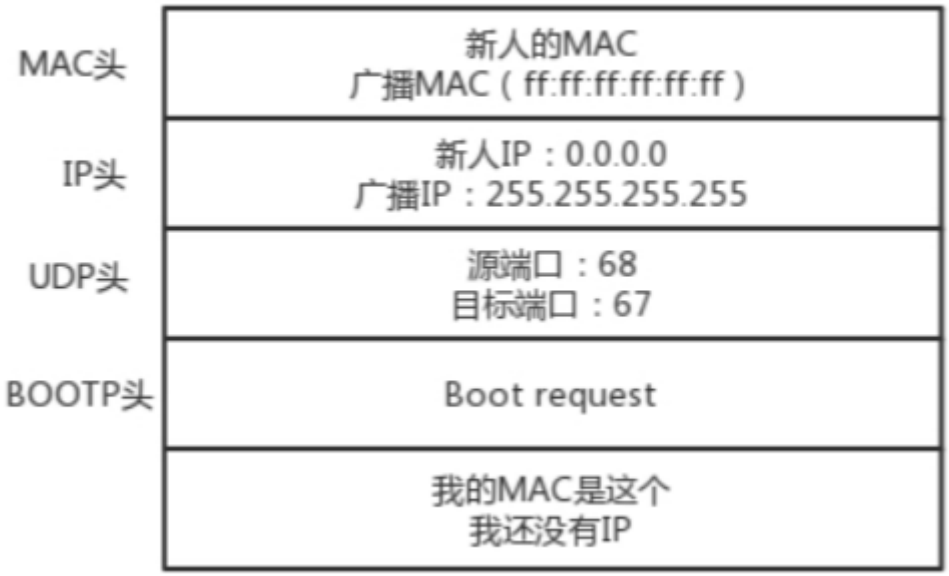
只需要配置一段共享的 IP 地址。每一台新接入的机器都通过 DHCP 协议，来这个共享的 IP 地址里申请，然后自动配置好就可以了

等人走了，或者用完了，还回去，这样其他的机器也能用

解析 DHCP 的工作方式

DHCP Discover

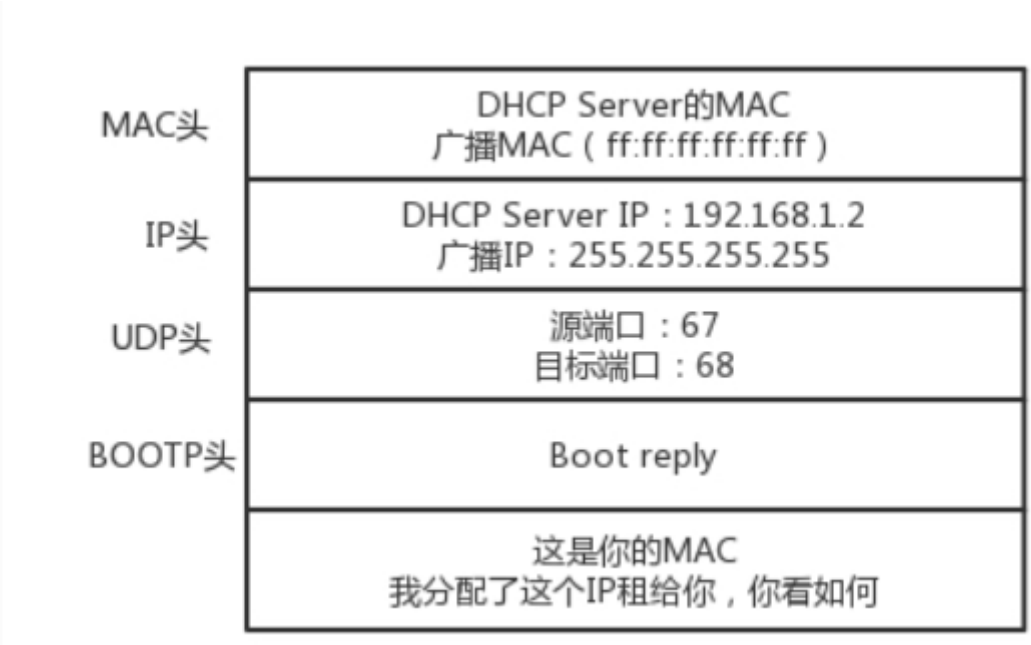
新来的机器使用IP地址0.0.0.0发送一个广播包，目的IP地址为255.255.255.255，广播包封装了UDP，UDP封装了BOOTP



DHCP Offer（来自DHCP Server）

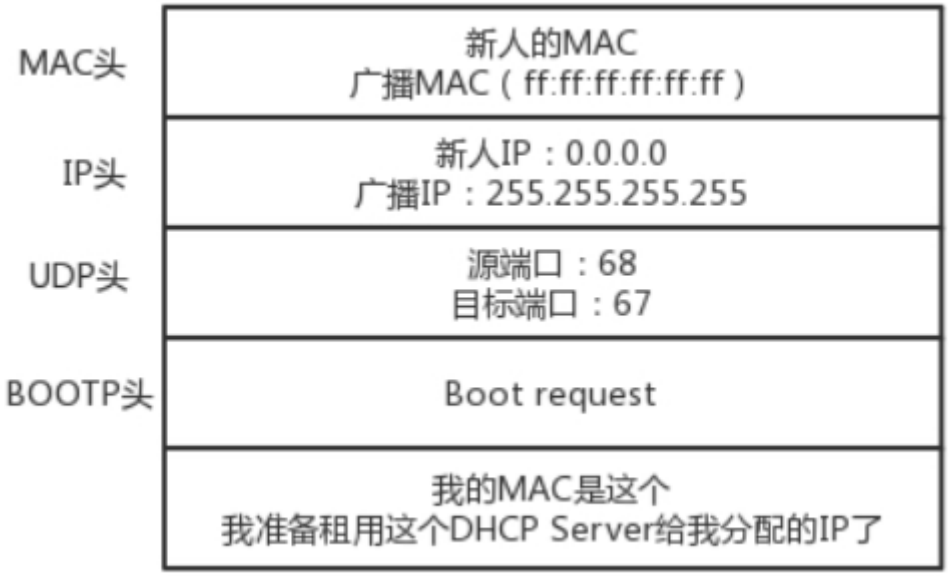
只有 MAC 唯一，IP 管理员才能知道这是一个新人，需要租给它一个 IP 地址，这个过程我们称为**DHCP Offer**，同时，**DHCP Server** 为此客户保留为它提供的 IP 地址，从而不会为其他 DHCP 客户分配此 IP 地址

格式如下



DHCP Server 仍然使用广播地址作为目的地址，因为，此时请求分配 IP 的新人还没有自己的 IP。此外，服务器还发送了子网掩码、网关和IP地址租用期等信息

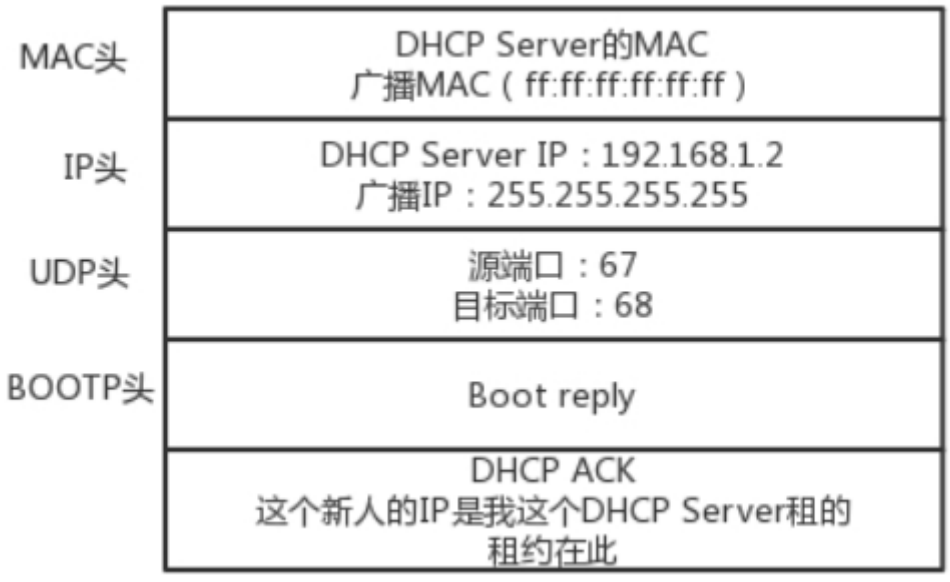
DHCP Request



回复自己的选择，此时DHCP Server还没有确认，它仍然使用0.0.0.0作为源IP地址、255.255.255.255为目标广播。在 BOOTP 里面，接受某个 DHCP Server 的分配的 IP

DHCP ACK

DHCP Server 接收到客户机的 DHCP request 之后，会广播返回给客户机一个 DHCP ACK 消息包，表明已经接受客户机的选择，并将这一 IP 地址的合法租用信息和其他的配置信息都放入该广播包，发给客户机，欢迎它加入网络大家庭



IP 地址的收回和续租

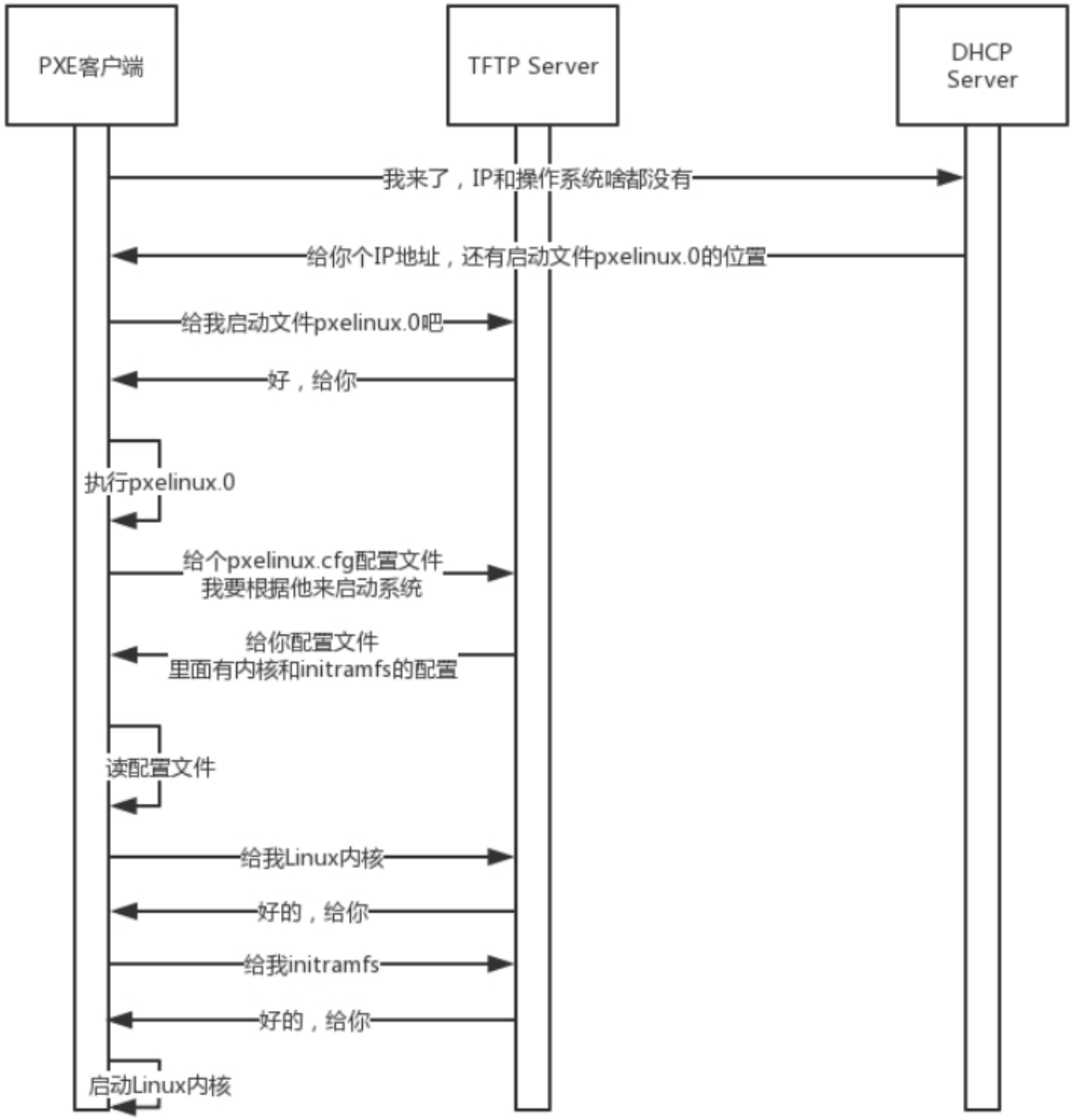
- 客户机会在租期过去 50% 的时候，直接向为其提供 IP 地址的 DHCP Server 发送 DHCP request 消息包
- 客户机接收到该服务器回应的 DHCP ACK 消息包，会根据包中所提供的新的租期以及其他已经更新的 TCP/IP 参数，更新自己的配置
- 这样，IP 租用更新就完成了

预启动执行环境（PXE）

PXE 协议分为客户端和服务器端，由于还没有操作系统，只能先把客户端放在 BIOS 里面。当计算机启动时，BIOS 把 PXE 客户端调入内存里面，就可以连接到服务端做一些操作了。

- PXE客户端自己需要有个IP地址
- 发送DHCP请求，得到一个IP地址，并得知PXE服务器位置，下载文件并初始化系统

PXE工作过程



5.从物理层到MAC层

第一层（物理层）

Hub集线器 将自己收到的每个字节，都复制到其他端口上去

第二层（数据链路层）

Hub是广播模式，需要解决以下几个问题：

1. 这个包是发给谁的？谁应该接收？
2. 大家都在发，会不会产生混乱？有没有谁先发、谁后发的规则？

3. 如果发送的时候出现了错误，怎么办？

MAC的全称是**Medium Access Control**，即**媒体访问控制**，MAC层就是数据链路层要解决的问题

多路访问协议：

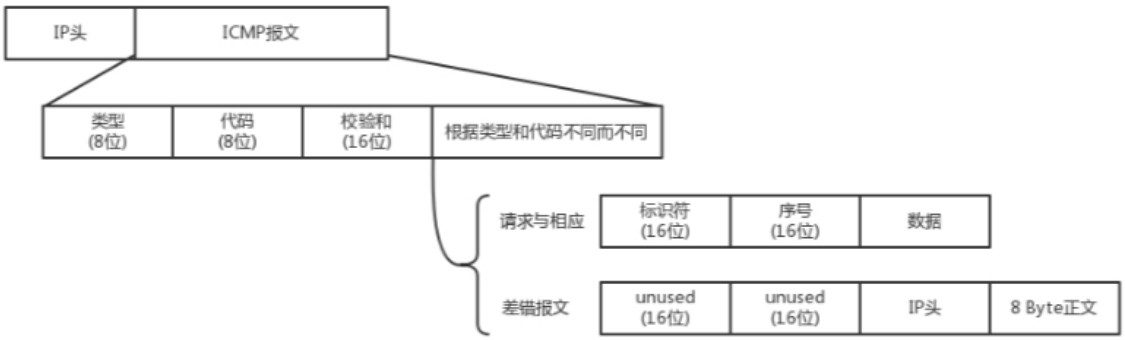
- 信道划分协议
- 轮流协议
- 随机接入协议
-

7.ICMP与ping

ICMP 协议的格式

ping 是基于 ICMP 协议工作的。**ICMP**全称**Internet Control Message Protocol**，就是**互联网控制报文协议**

ICMP 报文是封装在 IP 包里面的。因为传输指令的时候，肯定需要源地址和目标地址。它本身非常简单。因为作为侦查兵，要轻装上阵，不能携带大量的包袱。



ICMP报文有很多的类型，不同的类型有不同的代码。**最常用的类型是主动请求为8，主动请求的应答为0。**

查询报文类型

常用的**ping就是查询报文，是一种主动请求，并且获得主动应答的ICMP协议**，ping发的包符合ICMP协议格式

对ping的主动请求，进行网络抓包，称为**ICMP ECHO REQUEST。**同理主动请求的回复，称为**ICMP ECHO REPLY**

比起原生的ICMP，这里面多了两个字段，一个是**标识符。**这个很好理解，你派出去两队侦查兵，一队是侦查战况的，一队是去查找水源的，要有个标识才能区分。另一个是**序号**，你派出去的侦查兵，都要编个号。如果派出去10个，回来10个，就说明前方战况不错；如果派出去10个，回来2个，说明情况可能不妙

ping可能还有存放发送请求时间值，计算往返时间值等，说明路程长短

差错报文类型

终点不可达为3，源抑制为4，超时为11，重定向为5

第一种是终点不可达。

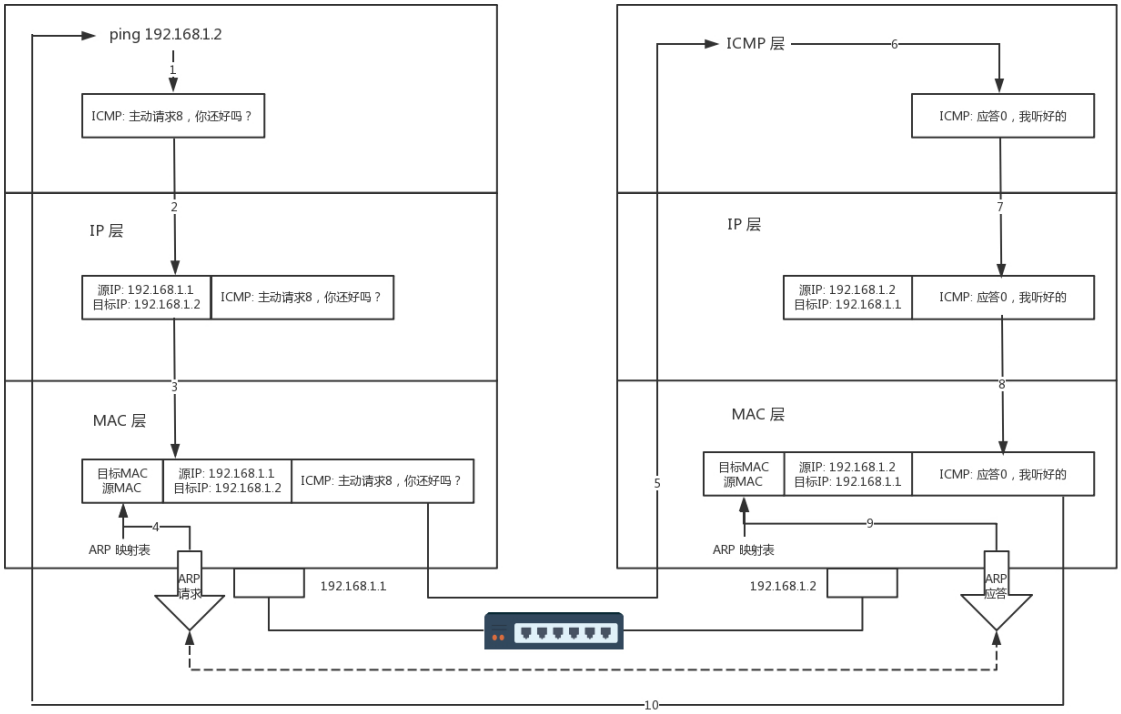
网络不可达代码为0，主机不可达代码为1，协议不可达代码为2，端口不可达代码为3，需要进行分片但设置了不分片位代码为4

第二种是源站抑制，也就是让源站放慢发送速度。

第三种是时间超时，也就是超过网络包的生存时间，网络包还没到达。

第四种是路由重定向，也就是让下次发给另一个路由器。

ping：查询报文类型的使用



10.UDP协议

TCP和UDP的区别

TCP面向连接，UDP面向无连接?

所谓的建立连接，是为了在客户端和服务端维护连接，而建立一定的数据结构来维护双方交互的状态，用这样的数据结构来保证所谓的面向连接的特性

- **TCP提供可靠交付。**通过TCP连接传输的数据，无差错、不丢失、不重复、按序到达。
- **UDP不保证不丢失，不保证按顺序到达。**
- **TCP面向字节流。**没头没尾，不像IP一样是一个一个的包
- **UDP继承IP特性，基于数据报，一个一个地发，一个一个地收**
- **TCP具有拥塞控制。**包丢了或者网络环境不好了，会调整行为，发送慢点或快点
- **UDP只管发送，不管网络状态或结果如何**
- **TCP是有状态服务**，精确记录发送了没有，收到没有，发送的进度和应该接收哪些
- **UDP是无状态服务**，发出去后不管其他的

UDP的包头

源端口号 (16位)	目的端口号 (16位)
UDP长度 (16位)	UDP校验和 (16位)
数据	

UDP三大特点

- **沟通简单，**默认网络通路容易送达，不容易被丢弃
- **轻信他人，**不会建立连接，监听的端口号可以接收任何人的数据，也可以传给任何人
- **不懂变通，**不知道什么时候该坚持，什么时候该退让。不会根据网络情况进行拥塞控制

UDP的三大使用场景

- 资源少，网络情况好的内网，或丢包不敏感的应用（DHCP，TFTP）
- 不需要一对一沟通，建立连接，可以**广播**的应用（IGMP，组播，VXLAN）
- 需要处理速度快，**时延低，容忍少数丢包**，但要求即使网络拥塞，也不能退避
- 在应用层实现自己的连接策略，又有时延需求

基于UDP的应用层示例

网页/APP的访问

QUIC（全称**Quick UDP Internet Connections**，**快速UDP互联网连接**）

Google提出的，基于UDP改进的通信协议，**目的是降低网络通信延迟，提供更好的用户互动体验**

流媒体协议

视频播放，有的包可丢，有的包不能丢。可以选择性丢帧，但不能连续丢帧

很多直播应用都基于UDP实现了自己的视频传输协议

实时游戏

游戏对实时要求较为严格的情况下，采用自定义的可靠UDP协议，自定义重传策略，能够把丢包产生的延迟降到最低，尽量减少网络问题对游戏性造成的影响

IoT物联网

- 物联网领域终端资源少，很可能只是个内存非常小的嵌入式系统，而维护TCP协议代价太大
- 另一方面，物联网对实时性要求也很高，而TCP还是因为上面的那些原因导致时延大。
- Google旗下的Nest建立Thread Group，推出了物联网通信协议Thread，就是基于UDP协议的

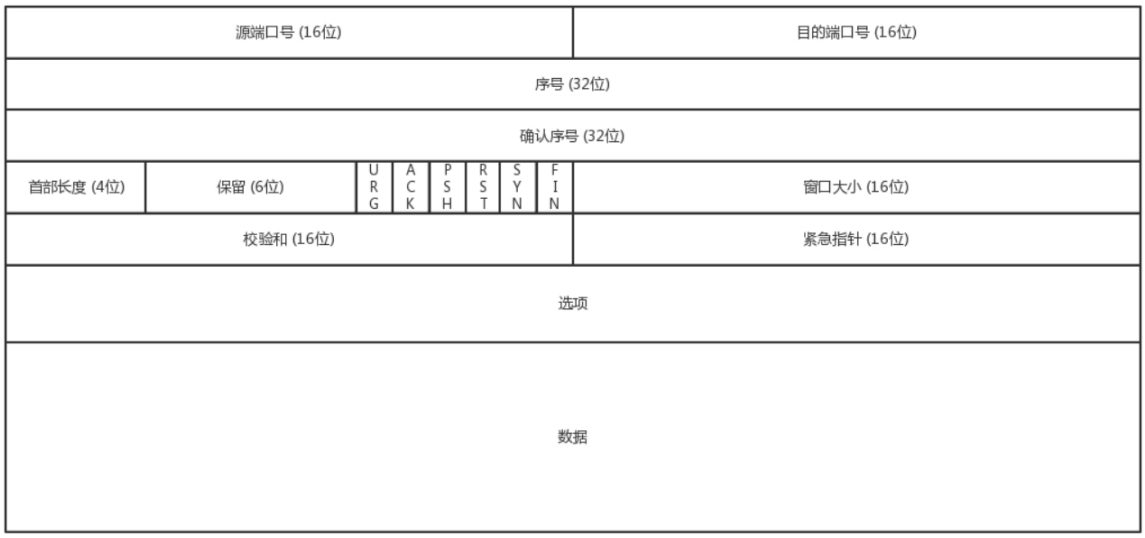
移动通信领域

4G网络里，移动流量上网的数据面对的协议GTP-U是基于UDP的

11.TCP协议（上）

它天然认为网络环境是恶劣的，丢包、乱序、重传，拥塞都是常有的事情，一言不合就可能送达不了，因而要从算法层面来保证可靠性。

TCP包头格式



- 源端口号，目标端口号必不可少
- 包的序号
  - 解决乱序问题，确认哪个先来，哪个后到
- 确认序号
  - 发出去的包要有确认，这样才能知道对方有没有收到
  - 没收到就重新发送，直到送达，解决不丢包的问题
- IP层面没有任何可靠性保证，TCP唯一能做的，就是**不断重传，用各种算法保证**
- 状态位
  - SYN是发起一个连接
  - ACK是回复
  - RST是重新连接
  - FIN是结束连接...
  - TCP是面向连接的，双方要维护连接的状态，这些带状态位的包发送，会引起双方状态变更
- 窗口大小
  - 流量控制，双方各声明一个窗口，标识自己能处理的多少内容，别发太多，月别发太少
- 拥塞控制
- **总结**
  - 有顺序（包序号），不丢包（确认序号），维护连接（不断重传，状态位，算法保证），流量控制（窗口大小），拥塞控制（限制自己发送的快慢）

三次握手

A：您好，我是A。

B：您好A，我是B。

A：您好B。

请求->应答->应答之应答

为什么不是两次？为什么不是四次？

A发送请求连接，有可能收不到回复，有可能饶了很多弯路，很久才到达B那里；

B收到了，如果不愿意建立连接，A重试一阵就放弃，如果愿意，他会发送回应给A；

这时候已经两次握手了，但B也不知道自己的回应能不能到达A，于是他也要求收到回复

只有等到自己发出去的消息收到回复，才算是可靠的连接

最后A应答B的回应，发送到B，成功建立连接

当然四次连接也可以，但是只要双方消息都有去有回，就可以了

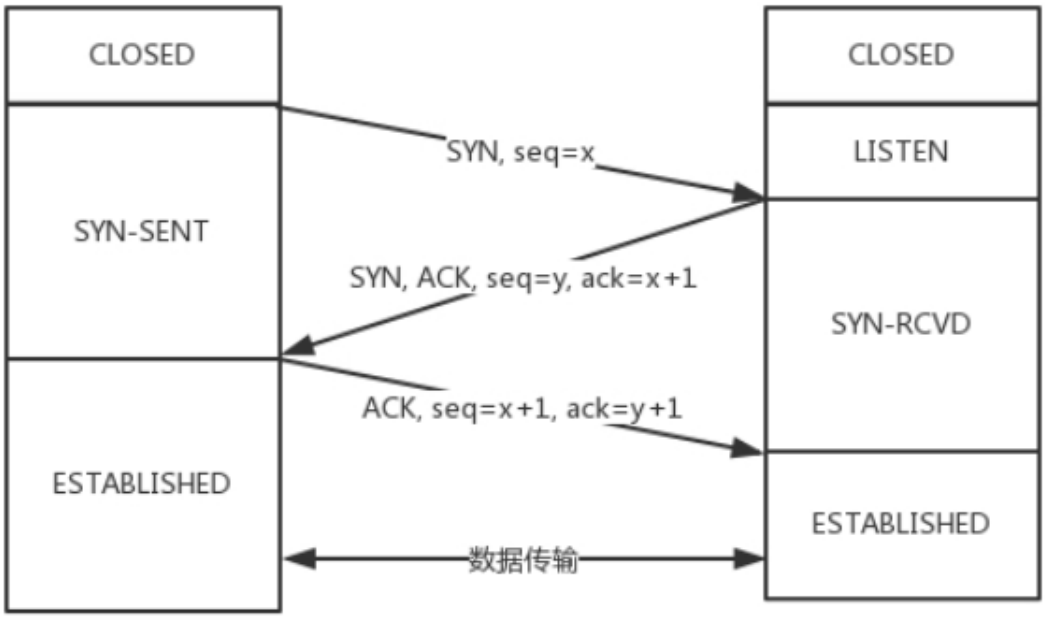
TCP包的序号的问题

A要告诉B，我发送的包序号从哪个开始；B也要告诉A，B发起的包序号从哪个开始，防止网络状况不好造成包的冲突

起始序号随着时间变化，可看作一个32位的计数器，每4ms加一

状态变化时序图

双方为了维护这个连接，都要维护一个状态机，建立过程中，双方的状态变化时序图如下



- 开始都处于CLOSED状态
- 服务端主动监听某个端口，处于LISTEN状态
- 客户端主动发起连接SYN，之后处于SYN-SENT状态
- 服务端收到发起的连接，返回SYN，并且ACK客户端的SYN，之后处于SYN-RCVD
- 客户端收到服务端发来的SYN和ACK，发送ACK的ACK，之后一直处于ESTABLISHED
- 服务端收到ACK的ACK后，处于ESTABLISHED，因为都一发一收了

TCP四次挥手

A：B，我不想玩了。

B：哦，你不想玩了啊，我知道了。

这个时候，还只是A不想玩了，也即A不会再发送数据，但是B不能在ACK的时候，直接关闭，因为很有可能A是发完了最后的数据就准备不玩了，但是B还没做完自己的事情，还是可以发送数据的，所以称为半关闭的状态。

这个时候A可以选择不再接收数据了，也可以选择最后再接收一段数据，等待B也主动关闭。

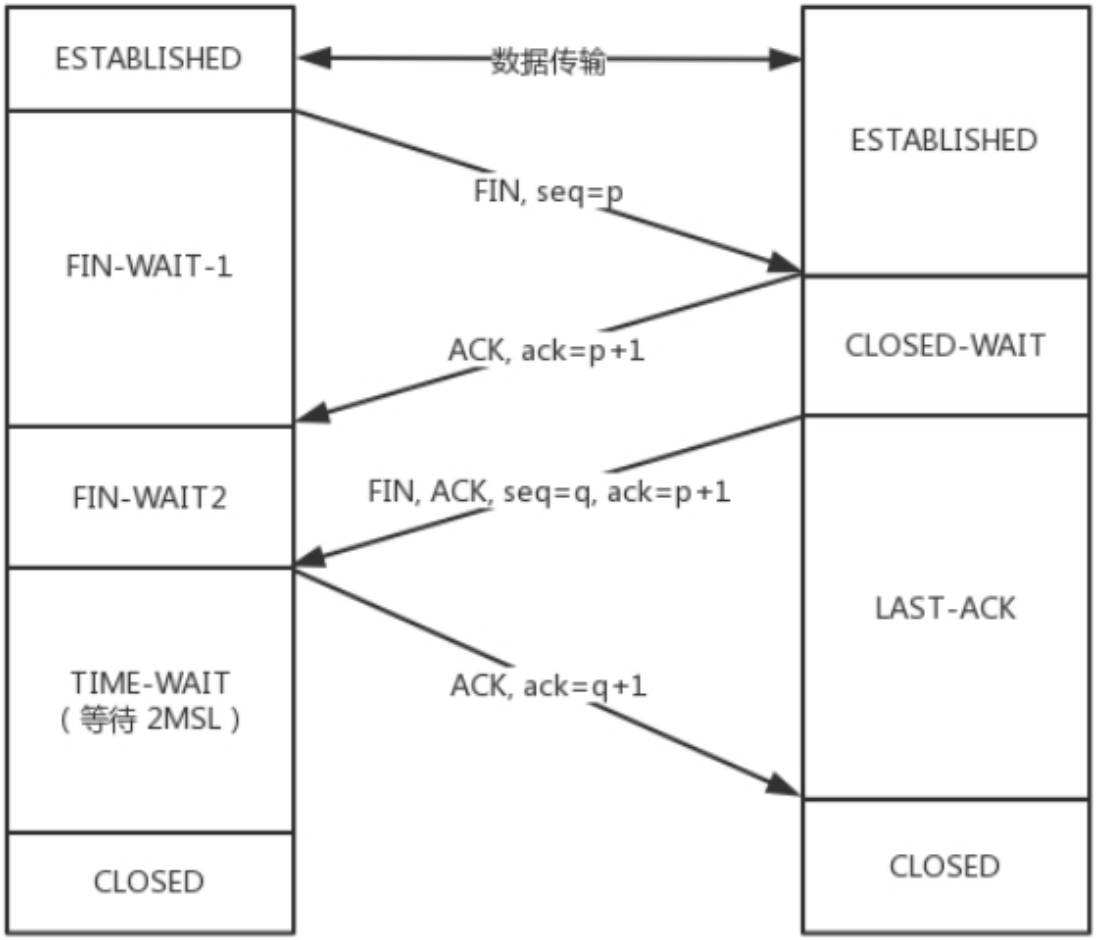
B：A啊，好吧，我也不玩了，拜拜。

A：好的，拜拜。

异常情况

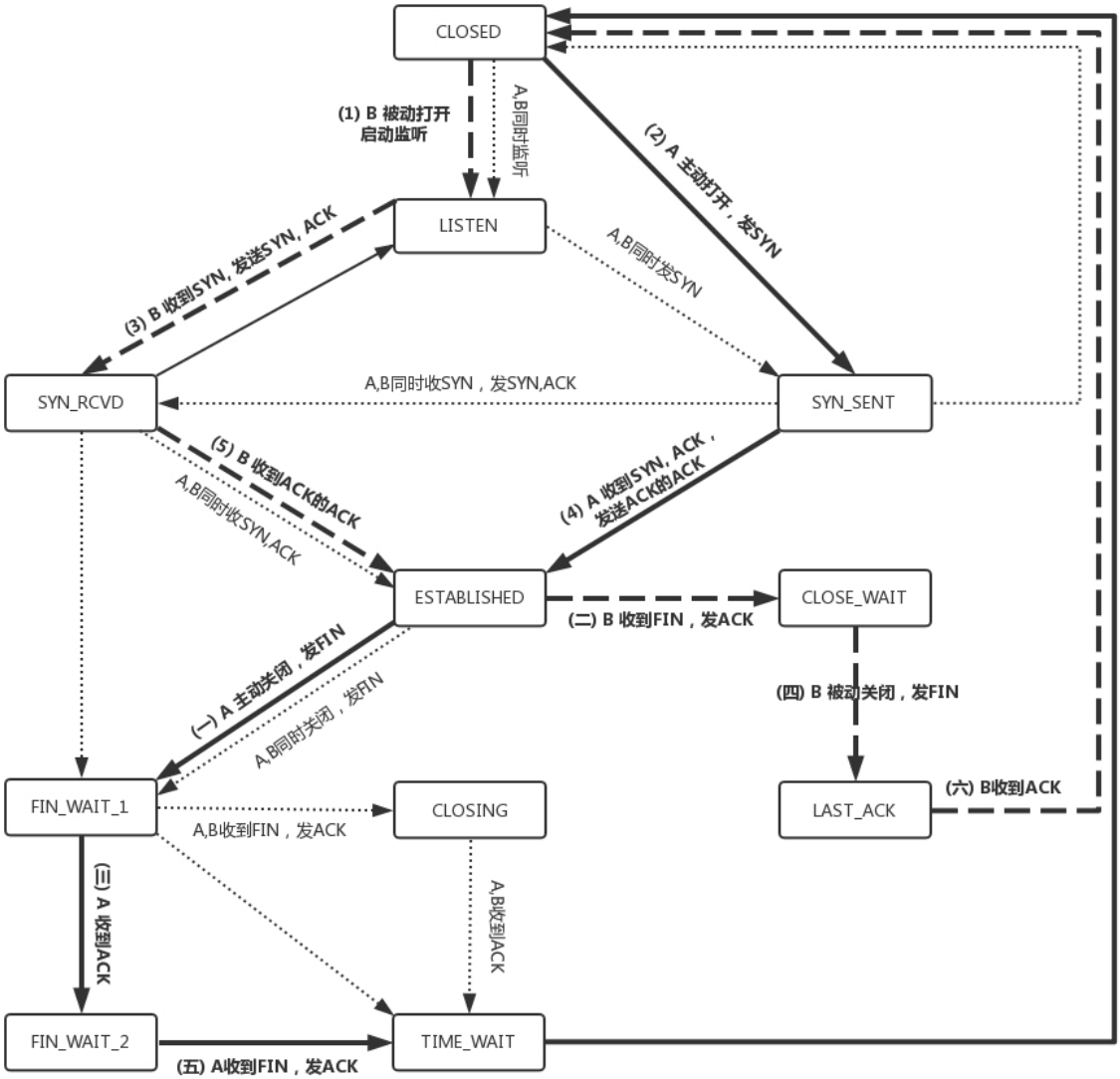
A发完不玩了，直接跑路，B发起结束得不到回答；A发完不玩了，B直接跑路，A不知道B是有事，还是要过一会才发送结束

TCP断开连接时的状态时序图



- A说不玩了，就进入FIN-WAIT-1状态
- B收到后，发送知道了ACK，进入CLOSE-WAIT状态
- A收到B说知道了，进入FIN-WAIT-2状态，若B直接跑了，A将永远处在这个状态（TCP协议中没有对这个状态的处理，Linux中可以调整tcp_fin_timeout设置一个超时时间）
- B也发送 不玩了，请求到达A后，A发送知道 B不玩了的ACK，从FIN-WAIT-2结束，这时候为了确保B能收到这个ACK，要求A要等待一段时间TIME-WAIT，这个时间要足够长，长到如果B没收到ACK，B重发，A再次重发ACK并足够时间到达B
- A如果直接跑了就还有一个问题，A的端口空出来了，但B不知道，B原来发过的很多包很可能还在路上，所以要等待足够长的时间，让B还没发过来的包都死掉
- 等待时间设置为2MSL，MSL是Maximum Segment Lifetime，报文最大生存时间。是任何报文在网络上存在的最长时间，超过这个时间报文就会被丢弃。TCP报文基于IP协议，IP头中有一个TTL域，是IP数据报可以经过的最大路由数，每经过一个处理它的路由器就减一，到0就被丢弃。同时发送ICMP报文通知源主机。规定MSL为2分钟，实际应用中常用的是30秒，1分钟，2分钟等
- 如果超过了2MSL，B还会重发FIN，但A接收到这个包时，表明已经等待过久，直接发送RST，B就知道A已经不在了

TCP状态机



12.TCP协议（下）

如何实现一个靠谱的协议

- 为了保证顺序性，每个包都有一个ID。在建立连接的时候，会商定起始的ID是什么，然后按照ID一个个发送
- 为了保证不丢包，对于发送的包都要进行应答，但是这个应答也不是一个一个来的，而是会应答某个之前的ID，表示都收到了，这种模式称为**累计确认**或者**累计应答**（**cumulative acknowledgment**）

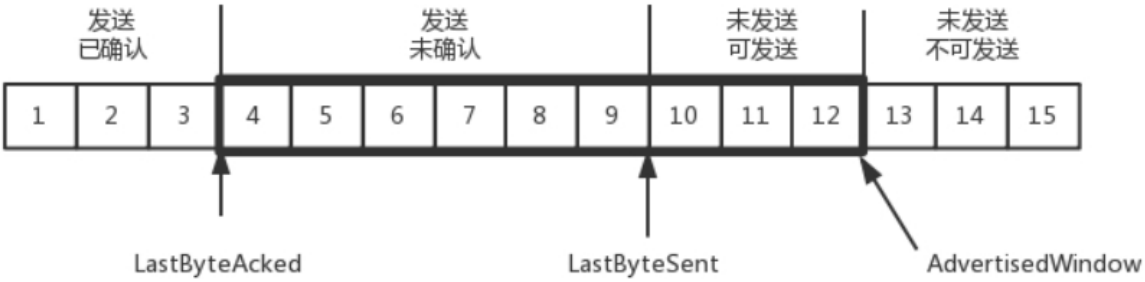
为了记录所有发送和接收的包，TCP需要发送接收端都有缓存来保存这些记录

发送端缓存（按ID一个个排列）

- 发送了并已经确认的
- 发送了并尚未确认的
- 没有发送但等待发送
- 没有发送且暂时不会发送

Advertised window

接收端给发送端报的窗口大小，等于第二部分加第三部分（没做完的和马上要交代的）



- LastByteAcked：第一部分和第二部分的分界线
- LastByteSent：第二部分和第三部分的分界线
- LastByteAcked + AdvertisedWindow：第三部分和第四部分的分界线

接收端缓存

- 接收并确认过的
- 还没接收，马上能接收的（最大工作量）
- 还没接收也没法接收的

- MaxRcvBuffer：最大缓存的量；
- LastByteRead之后是已经接收了，但是还没被应用层读取的；
- NextByteExpected是第一部分和第二部分的分界线。

AdvertisedWindow=MaxRcvBuffer-((NextByteExpected-1)-LastByteRead)

顺序问题与丢包问题

具体场景：

- 发送端来看，1、2、3已经发送并确认；4、5、6、7、8、9都是发送了还没确认；10、11、12是还没发出的；13、14、15是接收方没有空间，不准备发的。
- 接收端来看，1、2、3、4、5是已经完成ACK，但是没读取的；6、7是等待接收的；8、9是已经接收，但是没有ACK的。
- 发送端和接收端当前的状态如下：
  - 1、2、3没有问题，双方达成了一致。
  - 4、5接收方说ACK了，但是发送方还没收到，有可能丢了，有可能在路上。
  - 6、7、8、9肯定都发了，但是8、9已经到了，但是6、7没到，出现了乱序，缓存着但是没办法ACK。

确认与重发的机制

对每个发送了，但是没有ACK的包，都设一个定时器，超过一定时间就重新尝试（必须大于往返时间RTT，也不能过长）

自适应算法

TCP通过采样RTT的时间，然后进行加权平均，算出一个值，而且这个值还是要不断变化的，因为网络状况不断的变化。除了采样RTT，还要采样RTT的波动范围，计算出一个估计的超时时间

超时间隔加倍

每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送

快速重传

当接收方收到一个序号大于下一个所期望的报文段时，就检测到了数据流中的一个间格，于是发送三个冗余的ACK，客户端收到后，就在定时器过期之前，重传丢失的报文段

接收方发现6、8、9都已经接收了，就是7没来，那肯定是丢了，于是发送三个6的ACK，要求下一个是7。客户端收到3个，就会发现7的确又丢了，不等超时，马上重发

Selective Acknowledgment

这种方式需要在TCP头里加一个SACK的东西，可以将缓存的地图发送给发送方。例如可以发送ACK6、SACK8、SACK9，有了地图，发送方一下子就能看出来是7丢了。

流量控制问题

接收方处理比较慢时，导致缓存中没有空间了，可以适当减小发送端的窗口大小，甚至为0

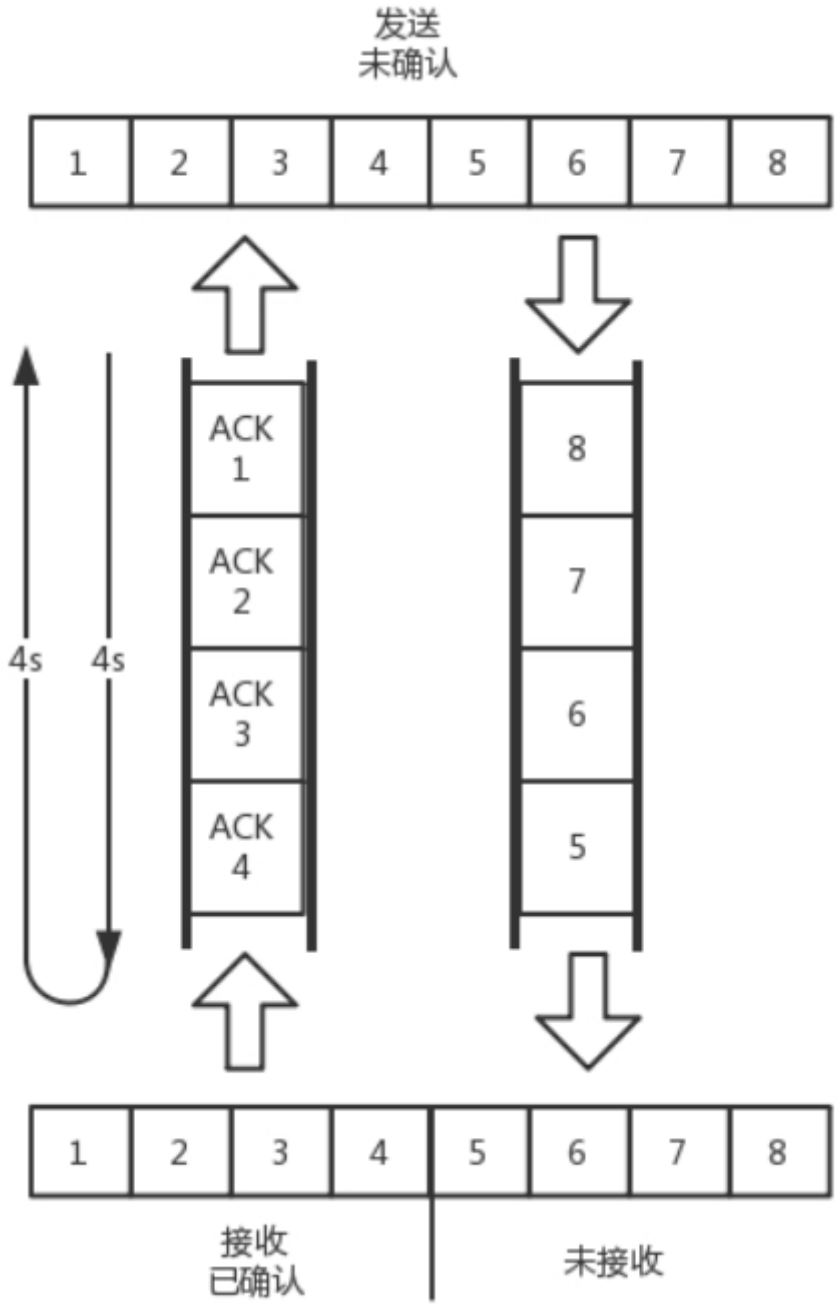
可以当窗口太小的时候，不更新窗口，直到达到一定大小，或者缓冲区一半为空，才更新窗口

拥塞控制问题

前面的滑动窗口rwnd是怕发送方把接收方缓存塞满，而拥塞窗口cwnd，是怕把网络塞满

拥塞窗口和滑动窗口共同控制发送的速度

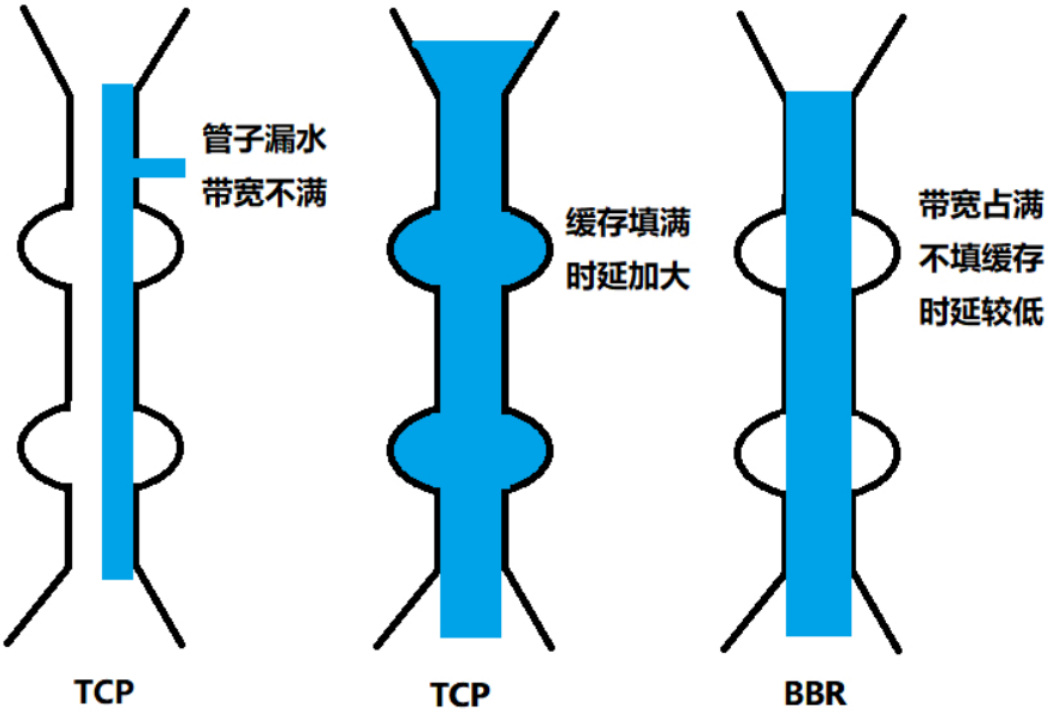
LastByteSent - LastByteAcked <= min {cwnd, rwnd}



上面是一个往返时间为8s，1234已经接收确认，5678已经发送还没有接收的情况

拥塞控制解决 包丢失和超时重传

- 丢包并不代表通道满了，可能本身公网就会丢包，这时认为拥塞了就限制发送是不对的
- TCP拥塞机制等到中间设备都填充满了，才发生丢包，从而降低速度，此时已经晚了
- 优化解决：
    - **TCP BBR拥塞算法**
    - 通过不断加快发送速度，将管道填满，但又不填满中间设备的缓存
    - 达到一个平衡点



13.套接字Socket

Socket编程：基于TCP和UDP。

（TCP协议是基于数据流的，所以设置为SOCK_STREAM，而UDP是基于数据报的，因而设置为SOCK_DGRAM）

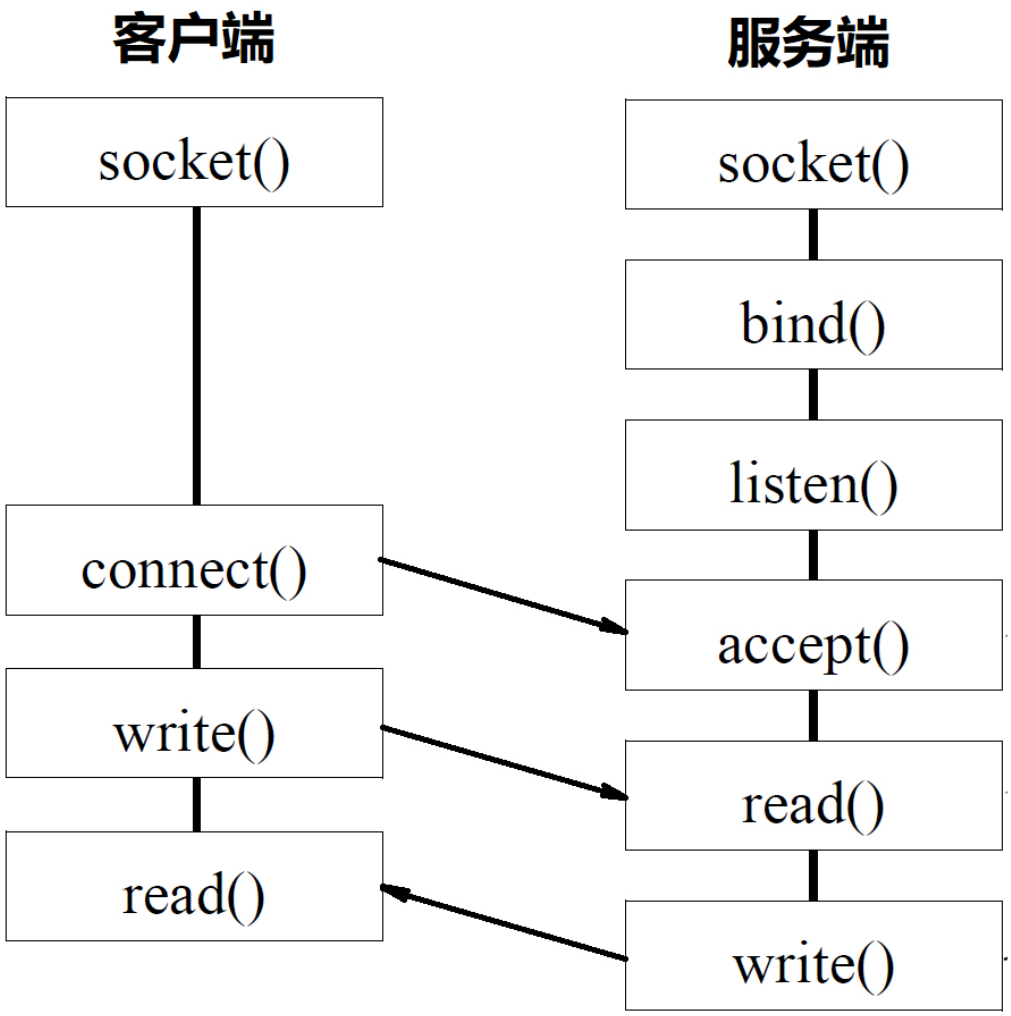
TCP和UDP稍有不同

基于TCP协议的Socket程序函数调用过程

- TCP服务端先监听一个端口（调用bind函数），给socket赋予一个IP地址和端口
- listen函数进行监听，服务端进入listen状态后，客户端就可以发起连接
- 内核中为每个socket维护两个队列。
  - 建立了连接的队列，三次握手已经完毕，处于 established状态
  - 没有完全建立连接的队列，三次握手没有完成，处于 syn_rcvd 状态
- 服务端调用 accept函数，拿出一个已完成的连接进行处理，如果还没完成就要等待
- 服务端等待时，客户端可以通过connect函数发起连接
- **监听的socket和真正用来传数据的socket是两个，即监听socket和已连接socket**

连接建立成功后，双方开始通过read和write函数读写数据，就像往文件流写东西一样

如下图所示：



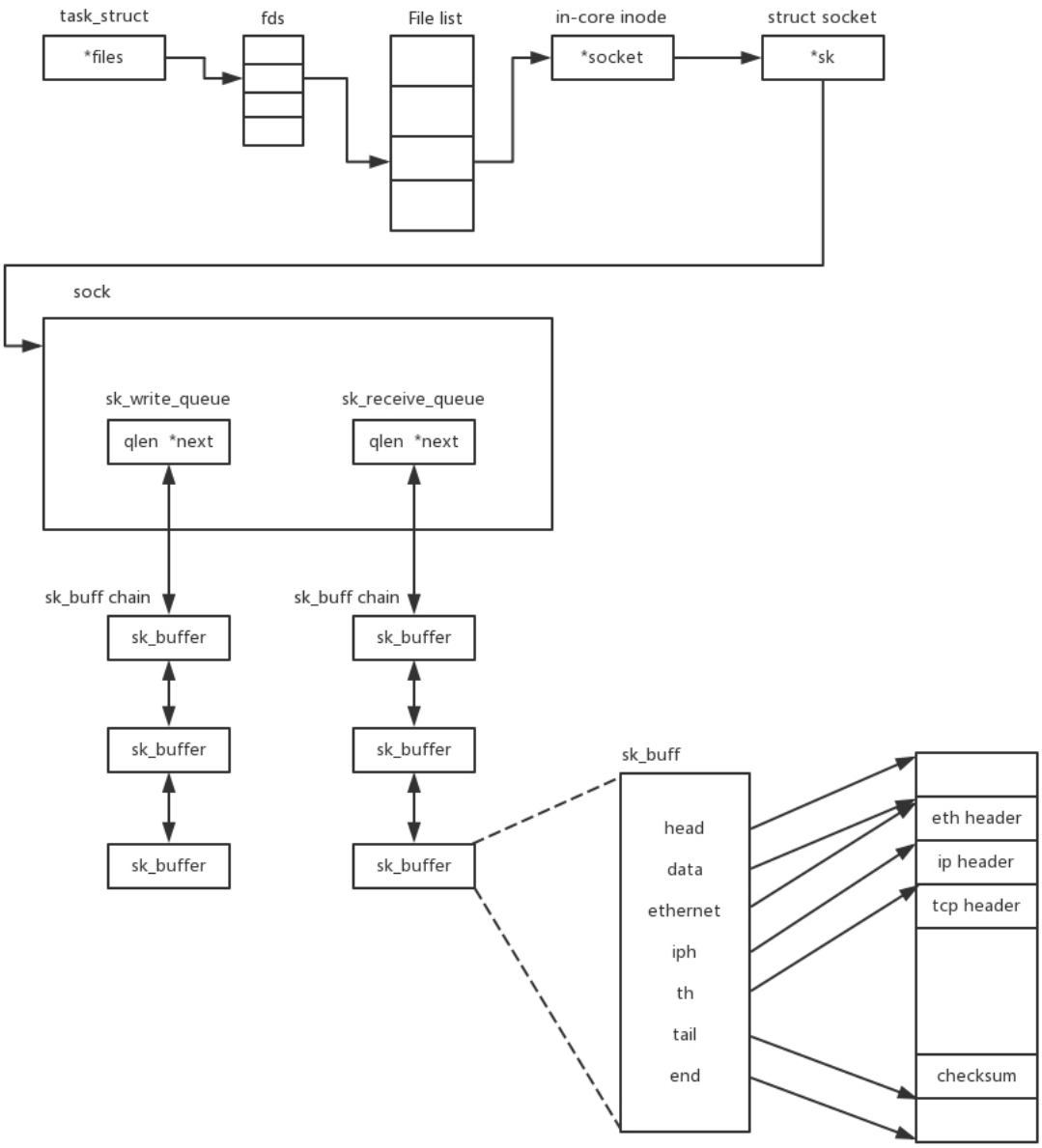
内核中，socket是一个文件，对应相应的文件描述符

每一个进程都有一个数据结构task_struct，里面指向一个文件描述符数组，来列出这个进程打开的所有文件的文件描述符。文件描述符是一个整数，是这个数组的下标。

这个数组中的内容是一个指针，指向内核中所有打开的文件的列表。既然是一个文件，就会有一个inode，只不过Socket对应的inode不像真正的文件系统一样，保存在硬盘上的，而是在内存中的。在这个inode中，指向了Socket在内核中的Socket结构。

在这个结构里面，主要的是两个队列，一个是**发送队列**，一个是**接收队列**。

在这两个队列里面保存的是一个缓存sk_buff。这个缓存里面能够看到完整的包的结构

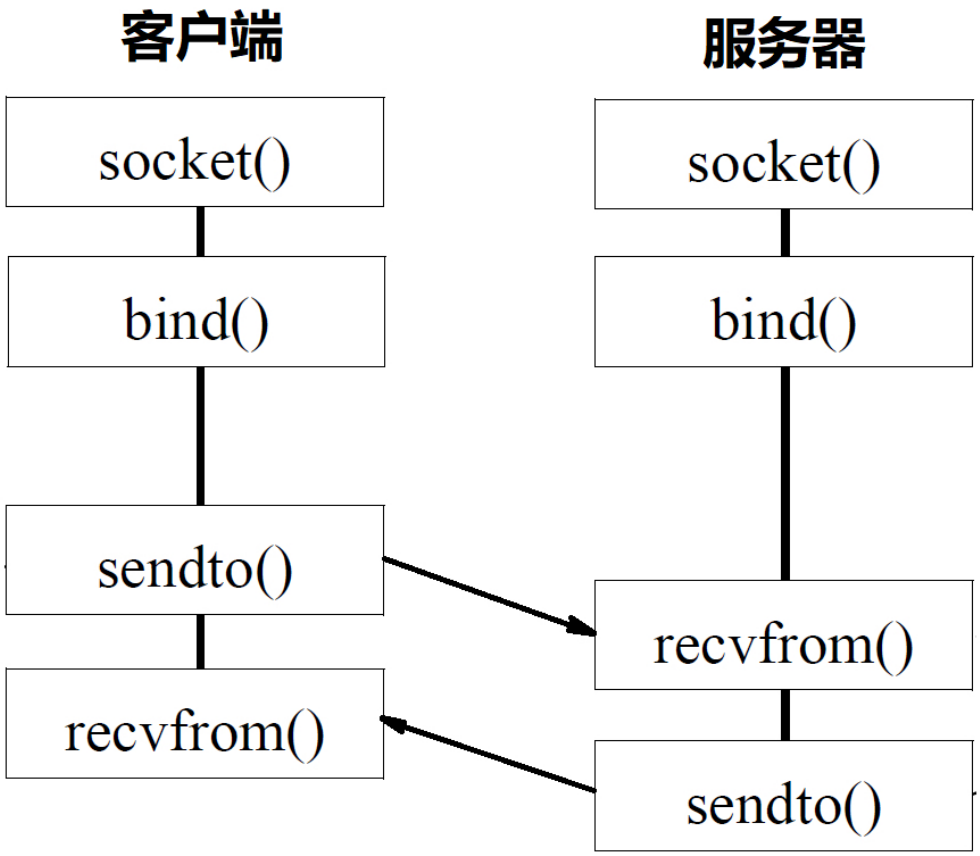


基于UDP协议的Socket程序函数调用过程

对于UDP来讲，过程有些不一样

UDP是没有连接的，所以不需要三次握手，也就不需要调用listen和connect，但是，UDP的的交互仍然需要IP和端口号，因而也需要bind

UDP是没有维护连接状态的，因而不需要每对连接建立一组Socket，而是只要有一个Socket，就能够和多个客户端通信。也正是因为没有连接状态，每次通信的时候，都调用sendto和recvfrom，都可以传入IP地址和端口



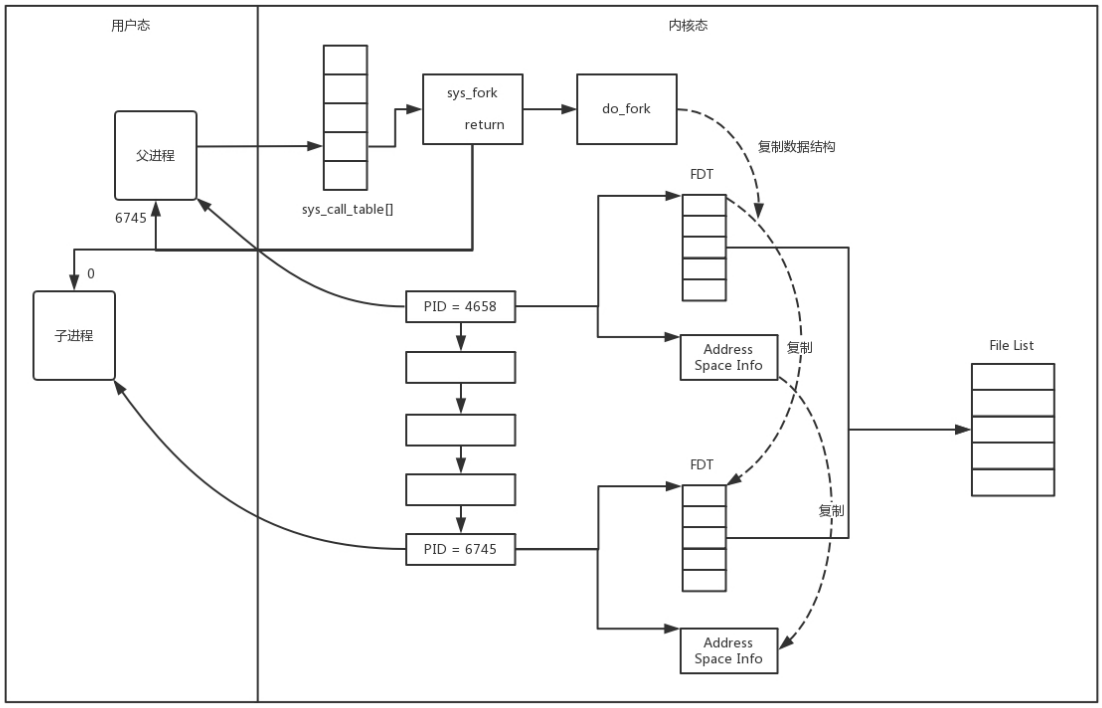
服务器如何接更多的项目?

多进程方式

一旦建立了一个连接，就会有一个已连接Socket，这时候你可以创建一个子进程，然后将基于已连接Socket的交互交给这个新的子进程来做

在Linux下，创建子进程使用fork函数

在Linux内核中，**会复制文件描述符的列表，也会复制内存空间，还会复制一条记录当前执行到了哪一行程序的进程。**显然，复制的时候在调用fork，复制完毕之后，父进程和子进程都会记录当前刚刚执行完fork。这两个进程刚复制完的时候，几乎一模一样，只是根据fork的返回值来区分到底是父进程，还是子进程。**如果返回值是0，则是子进程；如果返回值是其他的整数，就是父进程。**

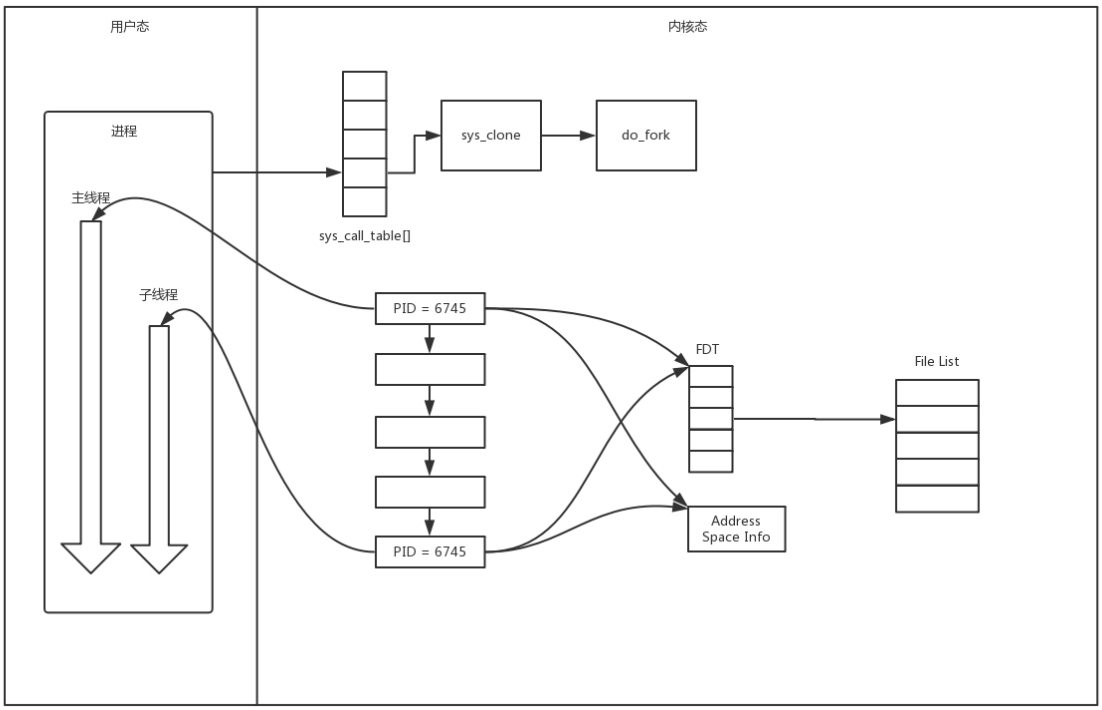


因为复制了文件描述符列表，而文件描述符都是指向整个内核统一的打开文件列表的，因而父进程刚才因为accept创建的已连接Socket也是一个文件描述符，同样也会被子进程获得

父进程可以通过子进程ID查看子进程是否完成项目，是否需要退出

多线程方式

在Linux下，通过pthread_create创建一个线程，也是调用do_fork。不同的是，虽然新的线程在task列表会新创建一项，但是很多资源，例如**文件描述符列表、进程空间，还是共享的，只不过多了一个引用而已**



新的线程也可以通过已连接Socket处理请求，从而达到并发处理的目的

IO多路复用，一个线程维护多个Socket

由于Socket是文件描述符，，可以用某个线程管理所有的Socket，都放在一个文件描述符集合fd_set中，这就是**项目进度墙**

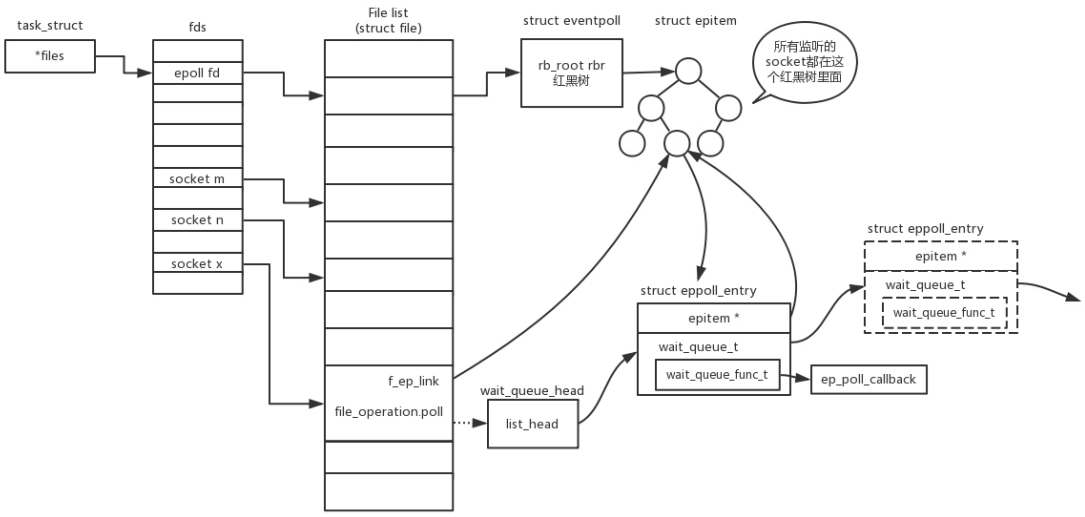
然后调用select函数来监听文件描述符集合是否有变化

那些发生变化的文件描述符在fd_set对应的位都设为1，表示Socket可读或者可写，从而可以进行读写操作，然后再调用select，接着盯着下一轮的变化

IO多路复用，从“派人盯着”到“有事通知”

事件通知不需要挨个轮询这些socket的状态，而是socket变化了，就主动通知

通过函数 epoll来实现，它在内核中的实现不是通过轮询的方式，而是通过**注册callback函数**的方式，当某个文件描述符发送变化的时候，就会主动通知



14.HTTP协议

以访问http://www.163.com 为例

http://www.163.com是个 URL，叫作**统一资源定位符。**之所以叫统一，是因为它是有格式的。HTTP 称为协议，www.163.com 是一个域名，表示互联网上的一个位置。有的 URL 会有更详细的位置标识，例如 http://www.163.com/index.html

HTTP 请求的准备

目前使用的 HTTP 协议大部分都是 1.1。在 1.1 的协议里面，默认是开启了 Keep-Alive 的，这样建立的 TCP 连接，就可以在多次请求中复用

建立了连接以后，浏览器就要发送 HTTP 的请求。

请求的格式就像这样

HTTP 的报文大概分为三大部分。第一部分是**请求行**，第二部分是请求的**首部**，第三部分才是请求的**正文实体**

HTTP 请求的构建

第一部分：请求行

在请求行中，URL 就是 http://www.163.com ，版本为 HTTP 1.1。这里要说一下的，就是方法。方法有几种类型

GET

GET 就是去服务器获取一些资源。对于访问网页来讲，要获取的资源往往是一个页面。其实也有很多其他的格式，比如说返回一个 JSON 字符串，到底要返回什么，是由服务器端的实现决定的

POST

它需要主动告诉服务端一些信息，而非获取。要告诉服务端什么呢？一般会放在正文里面。正文可以有各种各样的格式。常见的格式也是 JSON

PUT

就是向指定资源位置上传最新内容。但是，HTTP 的服务器往往是不允许上传文件的，所以 PUT 和 POST 就都变成了要传给服务器东西的方法

POST 往往是用来创建一个资源的，而 PUT 往往是用来修改一个资源的

DELETE

这个顾名思义就是用来删除资源的。例如，我们要删除一个云主机，就会调用 DELETE 方法

第二部分：首部字段

Accept-Charset

表示**客户端可以接受的字符集**。防止传过来的是另外的字符集，从而导致出现乱码

Content-Type

正文的格式。例如，我们进行 POST 的请求，如果正文是 JSON，那么我们就应该将这个值设置为 JSON

缓存

- **Cache-control**

当客户端发送的请求中包含 max-age 指令时，如果判定缓存层中，资源的缓存时间数值比指定时间的数值小，那么客户端可以接受缓存的资源；当指定 max-age 值为 0，那么缓存层通常需要将请求转发给应用集群

- **If-Modified-Since**

也就是说，如果服务器的资源在某个时间之后更新了，那么客户端就应该下载最新的资源；如果没有更新，服务端会返回“304 Not Modified”的响应，那客户端就不用下载了，也会节省带宽

HTTP请求的发送

HTTP协议是基于TCP协议的，所以它使用面向连接的方式发送请求，通过stream二进制流的方式传给对方。当然，到了TCP层，它会把二进制流变成一个报文段发送给服务器

TCP可能会多次传包，保证可达性

TCP层发送每一个报文的时候，都需要加上自己的地址（即源地址）和它想要去的地方（即目标地址），将这两个信息放到IP头里面，交给IP层进行传输。

IP层需要查看目标地址和自己是否是在同一个局域网。

- 如果是，就发送ARP协议来请求这个目标地址对应的MAC地址，然后将源MAC和目标MAC放入MAC头，发送出去即可；
- 如果不在同一个局域网，就需要发送到网关，还需要发送ARP协议，来获取网关的MAC地址，然后将源MAC和网关MAC放入MAC头，发送出去

网关收到包发现MAC符合，取出目标IP地址，根据路由协议找到下一跳的路由器，获取下一跳路由器的MAC地址，将包发给下一跳路由器

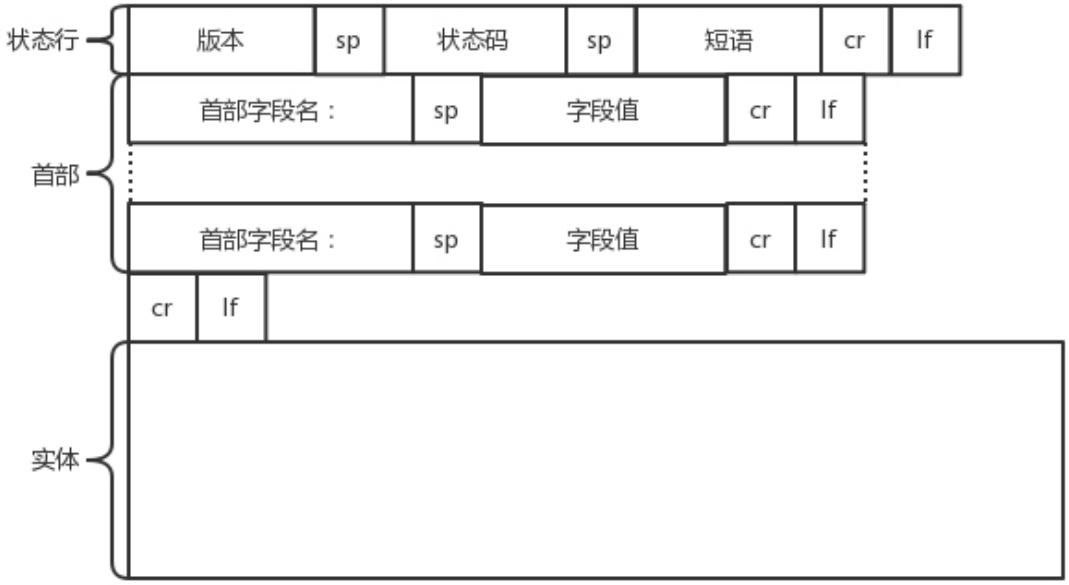
这样路由器一跳一跳终于到达目标的局域网。这个时候，最后一跳的路由器能够发现，目标地址就在自己的某一个出口的局域网上。于是，在这个局域网上发送ARP，获得这个目标地址的MAC地址，将包发出去

目标的机器发现MAC地址符合，就将包收起来；发现IP地址符合，根据IP头中协议项，知道自己上一层是TCP协议，于是解析TCP的头，里面有序列号，需要看一看这个序列包是不是我要的，如果是就放入缓存中然后返回一个ACK，如果不是就丢弃

TCP头里面还有端口号，HTTP的服务器正在监听这个端口号。于是，目标机器自然知道是HTTP服务器这个进程想要这个包，于是将包发给HTTP服务器。HTTP服务器的进程看到，原来这个请求是要访问一个网页，于是就把这个网页发给客户端

HTTP返回的构建

HTTP的返回报文也是有一定格式的。这也是基于HTTP 1.1的。



第一部分：状态行

状态码会反应HTTP请求的结果，短语会大概说一下原因

接下来是返回首部的**key value**

第二部分：首部

- **Retry-After**

告诉客户端应该在多长时间后再次尝试一下。例如503错误：服务暂时不再和这个值配合使用

- **Content-Type**

返回的是HTML还是JSON

构造好了返回的HTTP报文，接下来就是把这个报文发送出去。还是交给Socket去发送，还是交给TCP层，让TCP层将返回的HTML，也分成一个个小的段，并且保证每个段都可靠到达

这些段加上TCP头后会交给IP层，然后把刚才的发送过程反向走一遍。虽然两次不一定走相同的路径，但是逻辑过程是一样的，一直到达客户端

客户端发现MAC地址符合、IP地址符合，于是就会交给TCP层。根据序列号看是不是自己要的报文段，如果是，则会根据TCP头中的端口号，发给相应的进程。这个进程就是浏览器，浏览器作为客户端也在监听某个端口。

当浏览器拿到了HTTP的报文。发现返回“200”，一切正常，于是就从正文中将HTML拿出来。HTML是一个标准的网页格式。浏览器只要根据这个格式，展示出一个绚丽多彩的网页

HTTP 2.0

HTTP 1.1在应用层以纯文本的形式进行通信。每次通信都要带完整的HTTP的头，而且不考虑pipeline模式的话，每次的过程总是像上面描述的那样一去一回。

这样在实时性、并发性上都存在问题

HTTP 2.0的优化

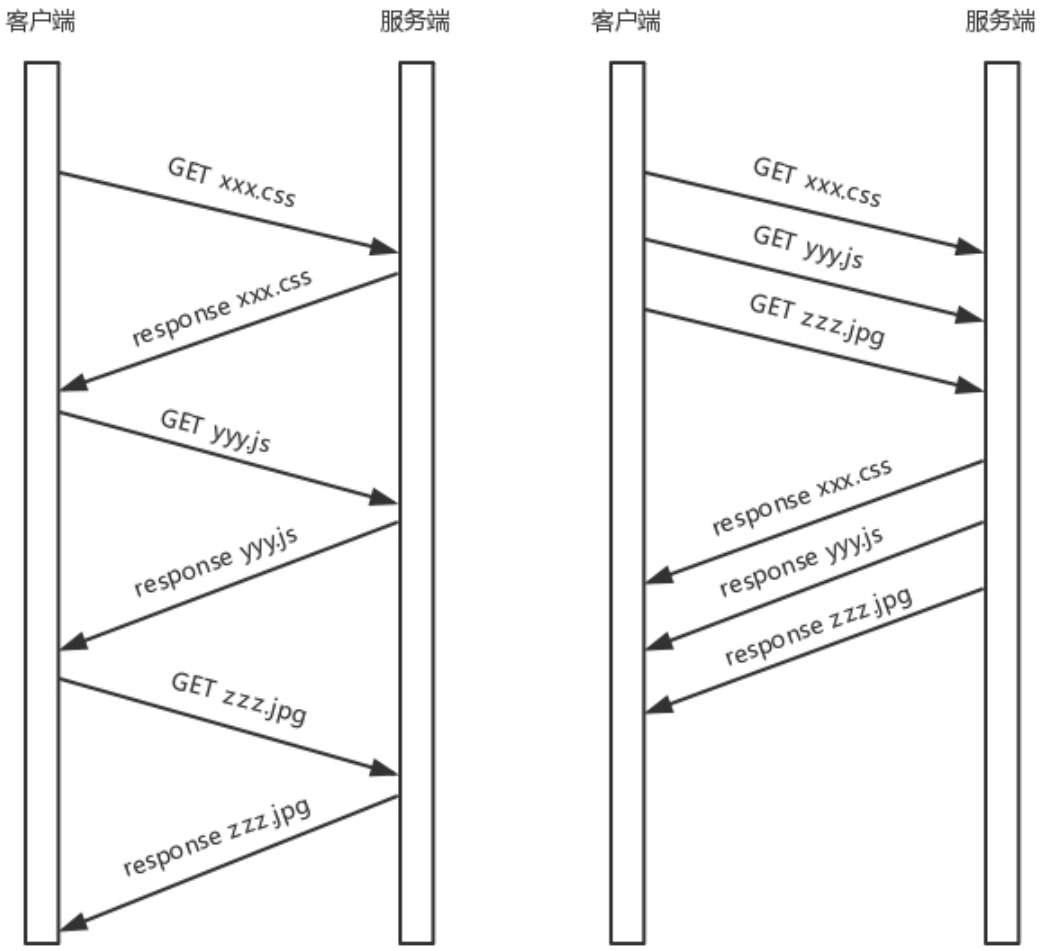
- HTTP 2.0会对**HTTP的头进行一定的压缩**，将原来每次都要携带的大量key value在两端建立一个索引表，对相同的头只发送索引表中的索引

- HTTP 2.0协议将一个TCP的连接中，切分成多个流，每个流都有自己的ID，而且流可以是客户端发往服务端，也可以是服务端发往客户端。它其实只是一个虚拟的通道。流是有优先级的
- HTTP 2.0还将所有的传输信息分割为更小的消息和帧，并对它们采用二进制格式编码。常见的帧有**Header帧**，用于传输Header内容，并且会开启一个新的流。再就是**Data帧**，用来传输正文实体。多个Data帧属于同一个流。

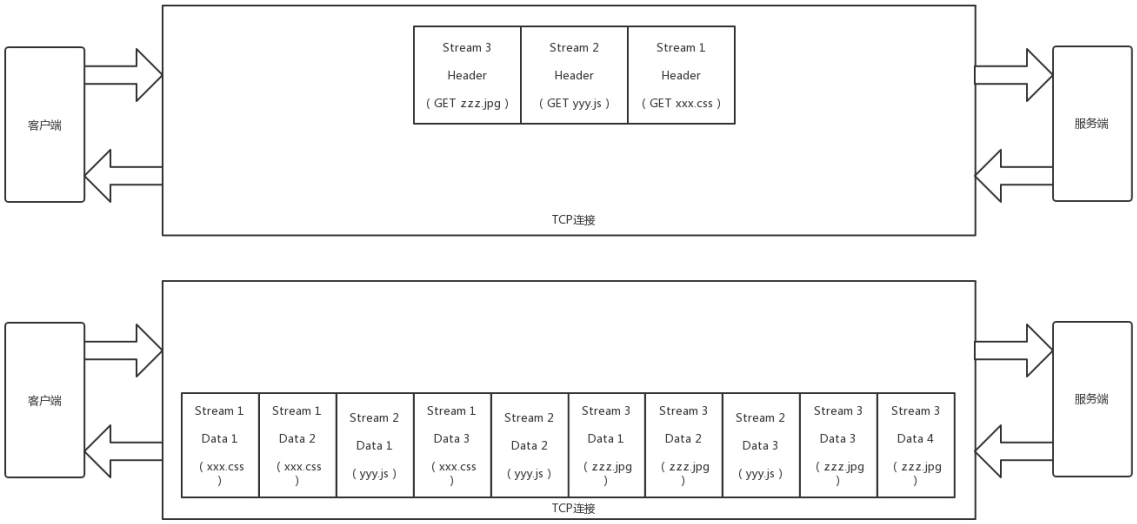
下面是一个例子

假设我们的一个页面要发送三个独立的请求，一个获取css，一个获取js，一个获取图片jpg

如果使用HTTP 1.1就是串行的，但是如果使用HTTP 2.0，就可以在一个连接里，客户端和服务端都可以同时发送多个请求或回应，而且不用按照顺序一对一对应。



HTTP 2.0其实是将三个请求变成三个流，将数据分成帧，乱序发送到一个TCP连接中



HTTP 2.0成功解决了HTTP 1.1的队首阻塞问题，同时，也不需要通过HTTP 1.x的pipeline机制用多条TCP连接来实现并行请求与响应；减少了TCP连接数对服务器性能的影响，同时将页面的多个数据css、js、 jpg等通过一个数据链接进行传输，能够加快页面组件的传输速度’

QUIC协议

机制一：自定义连接机制

一条TCP连接是由四元组标识的，分别是源 IP、源端口、目的 IP、目的端口

一旦一个元素发生变化时，就需要断开重连，重新连接。在移动互联情况下，当手机信号不稳定或者在WIFI和 移动网络切换时，都会导致重连，从而进行再次的三次握手，导致一定的时延20.

这在TCP是没有办法的，但是基于UDP，就可以在QUIC自己的逻辑里面维护连接的机制，不再以四元组标识，而是以一个64位的随机数作为ID来标识，而且UDP是无连接的，所以当IP或者端口变化的时候，只要ID不变，就不需要重新建立连接

机制二：自定义重传机制

TCP为了保证可靠性，通过使用**序号**和**应答**机制，来解决顺序问题和丢包问题

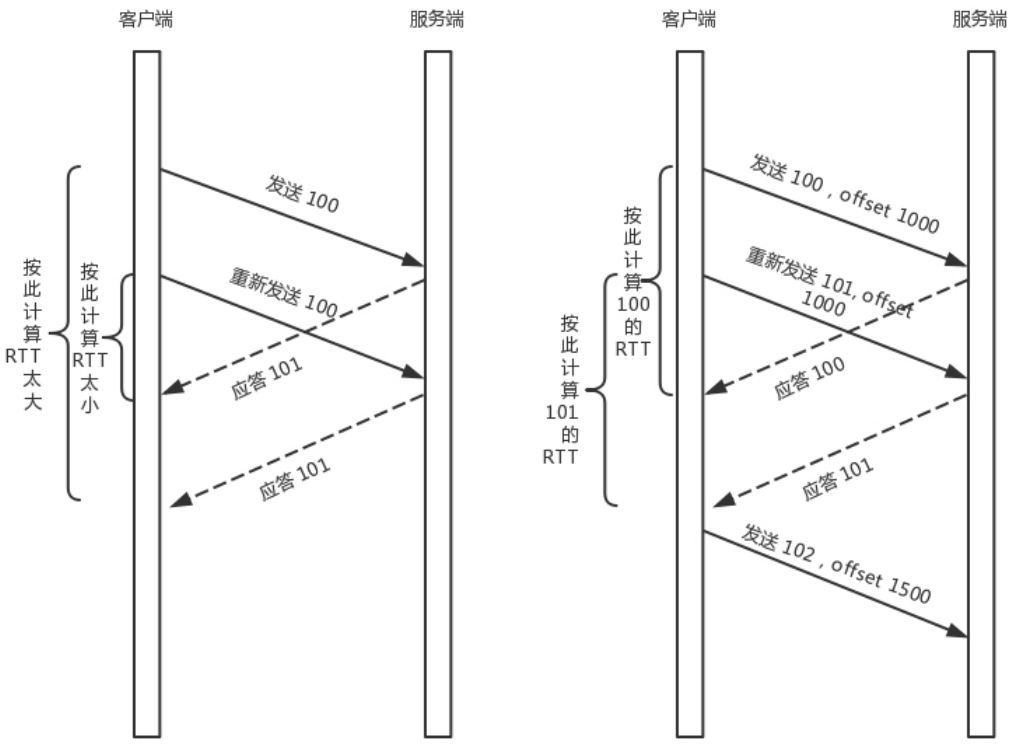
- 超时的判定
- **自适应重传算法**，这个超时是通过**采样往返时间RTT**不断调整的

QUIC也有个序列号，是递增的。任何一个序列号的包只发送一次，下次就要加一了。例如，发送一个包，序号是100，发现没有返回；再次发送的时候，序号就是101了；如果返回的ACK 100，就是对第一个包的响应。如果返回ACK 101就是对第二个包的响应，RTT计算相对准确。

判定包100和包101发送的是不是同样的内容

QUIC定义了一个offset概念。QUIC既然是面向连接的，也就像TCP一样，是一个数据流，发送的数据在这个数据流里面有个偏移量offset，可以通过offset查看数据发送到了哪里，这样只要这个offset的包没有来，就要重发；如果来了，按照offset拼接，还是能够拼成一个流

这解决了RTT采样时间不准确的问题



机制三：无阻塞的多路复用

同HTTP 2.0一样，同一条QUIC连接上可以创建多个stream，来发送多个 HTTP 请求。但是，QUIC是基于UDP的，一个连接上的多个stream之间没有依赖。这样，假如stream2丢了一个UDP包，后面跟着stream3的一个UDP包，虽然stream2的那个包需要重传，但是stream3的包无需等待，就可以发给用户

机制四：自定义流量控制

TCP的流量控制是通过**滑动窗口协议**。QUIC的流量控制也是通过window_update，来告诉对端它可以接受的字节数。但是QUIC的窗口是适应自己的多路复用机制的，不但在一个连接上控制窗口，还在一个连接中的每个stream控制窗口

在TCP协议中，接收端的窗口的起始点是下一个要接收并且ACK的包，即便后来的包都到了，放在缓存里面，窗口也不能右移，因为TCP的ACK机制是基于序列号的累计应答，一旦ACK了一个系列号，就说明前面的都到了，所以只要前面的没到，后面的到了也不能ACK，就会导致后面的到了，也有可能超时重传，浪费带宽

QUIC的ACK是基于offset的，每个offset的包来了，进了缓存，就可以应答，应答后就不会重发，中间的空挡会等待到来或者重发即可，而窗口的起始位置为当前收到的最大offset，从这个offset到当前的stream所能容纳的最大缓存，是真正的窗口大小。显然，这样更加准确

另外，还有整个连接的窗口，需要对于所有的stream的窗口做一个统计

15.HTTPS协议

加密分为两种方式一种是**对称加密**，一种是**非对称加密**

在对称加密算法中，加密和解密使用的密钥是相同的。也就是说，加密和解密使用的是同一个密钥。因此，对称加密算法要保证安全性的话，密钥要做好保密。只能让使用的人知道，不能对外公开

在非对称加密算法中，加密使用的密钥和解密使用的密钥是不相同的。一把是作为公开的公钥，另一把是作为谁都不能给的私钥。公钥加密的信息，只有私钥才能解密。私钥加密的信息，只有公钥才能解密

因为对称加密算法相比非对称加密算法来说，效率要高得多，性能也好，所以交互的场景下多用对称加密

对称加密

假设你和外卖网站约定了一个密钥，你发送请求的时候用这个密钥进行加密，外卖网站用同样的密钥进行解密。这样就算中间的黑客截获了你的请求，但是它没有密钥，还是破解不了。

这看起来很完美，但是中间有个问题，你们两个怎么来约定这个密钥呢？如果这个密钥在互联网上传输，也是很有可能让黑客截获的。黑客一旦截获这个秘钥，它可以佯作不知，静静地等着你们两个交互。这时候你们之间互通的任何消息，它都能截获并且查看，就等你把银行卡账号和密码发出来

我们在谍战剧里面经常看到这样的场景，就是特工破译的密码会有个密码本，截获无线电台，通过密码本就能将原文破解出来。怎么把密码本给对方呢？只能通过**线下传输**

非对称加密

非对称加密的私钥放在外卖网站这里，不会在互联网上传输，这样就能保证这个秘钥的私密性。但是，对应私钥的公钥，是可以在互联网上随意传播的，只要外卖网站把这个公钥给你，你们就可以愉快地互通了。

比如说你用公钥加密，说“我要定外卖”，黑客在中间就算截获了这个报文，因为它没有私钥也是解不开的，所以这个报文可以顺利到达外卖网站，外卖网站用私钥把这个报文解出来，然后回复，“那给我银行卡和支付密码吧”。

先别太乐观，这里还是有问题的。回复的这句话，是外卖网站拿私钥加密的，互联网上人人都可以把它打开，当然包括黑客。那外卖网站可以拿公钥加密吗？当然不能，因为它自己的私钥只有它自己知道，谁也解不开。

另外，这个过程还有一个问题，黑客也可以模拟发送“我要定外卖”这个过程的，因为它也有外卖网站的公钥。

为了解决这个问题，看来一对公钥私钥是不够的，**客户端也需要有自己的公钥和私钥，并且客户端要把自己的公钥，给外卖网站**

这样，客户端给外卖网站发送的时候，用外卖网站的公钥加密。而外卖网站给客户端发送消息的时候，使用客户端的公钥。这样就算有黑客企图模拟客户端获取一些信息，或者半路截获回复信息，但是由于它没有私钥，这些信息它还是打不开

数字证书

不对称加密也会有同样的问题，如何将不对称加密的公钥给对方呢？一种是放在一个公网的地址上，让对方下载；另一种就是在建立连接的时候，传给对方

这两种方法有相同的问题，那就是，作为一个普通网民，你怎么鉴别别人给你的公钥是对的。会不会有人冒充外卖网站，发给你一个它的公钥。接下来，你和它所有的互通，看起来都是没有任何问题的。毕竟每个人都可以创建自己的公钥和私钥。

例如，我自己搭建了一个网站cliu8site，可以通过这个命令先创建私钥。

```
openssl genrsa -out cliu8siteprivate.key 1024
```

然后，再根据这个私钥，创建对应的公钥。

```
openssl rsa -in cliu8siteprivate.key -pubout -outcliu8sitepublic.pem
```

这个由权威部门颁发的称为**证书**（**Certificate**）

证书中包括

- **公钥**
- 证书的**所有者**
- 证书的**发布机构**和证书的**有效期**

生成证书需要发起一个证书请求，然后将这个请求发给一个权威机构去认证，这个权威机构我们称为**CA（Certificate Authority）**

证书请求可以通过这个命令生成。

```
openssl req -key cliu8siteprivate.key -new -out cliu8sitecertificate.req
```

只有用只掌握在权威机构手里的东西签名了才行，这就是CA的私钥

签名算法大概是这样工作的：一般是对信息做一个Hash计算，得到一个Hash值，这个过程是不可逆的，也就是说无法通过Hash值得出原来的信息内容。在把信息发送出去时，把这个Hash值加密后，作为一个签名和信息一起发出去

权威机构给证书签名的命令是这样的。

```
openssl x509 -req -in cliu8sitecertificate.req -CA cacertificate.pem -CAkey
caprivate.key -out cliu8sitecertificate.pem
```

这个命令会返回Signature ok，而cliu8sitecertificate.pem就是签过名的证书。CA用自己的私钥给外卖网站的公钥签名，就相当于给外卖网站背书，形成了外卖网站的证书

我们来查看这个证书的内容。

```
openssl x509 -in cliu8sitecertificate.pem -noout -text
```

Issuer，也即证书是谁颁发的；

Subject，就是证书颁发给谁；

Validity是证书期限；

Public-key是公钥内容；

Signature Algorithm是签名算法。

你不会从外卖网站上得到一个公钥，而是会得到一个证书，这个证书有个发布机构CA，你只要得到这个发布机构CA的公钥，去解密外卖网站证书的签名，如果解密成功了，Hash也对的上，就说明这个外卖网站的公钥没有啥问题

要想验证证书，需要CA的公钥，问题是，你怎么确定CA的公钥就是对的呢？

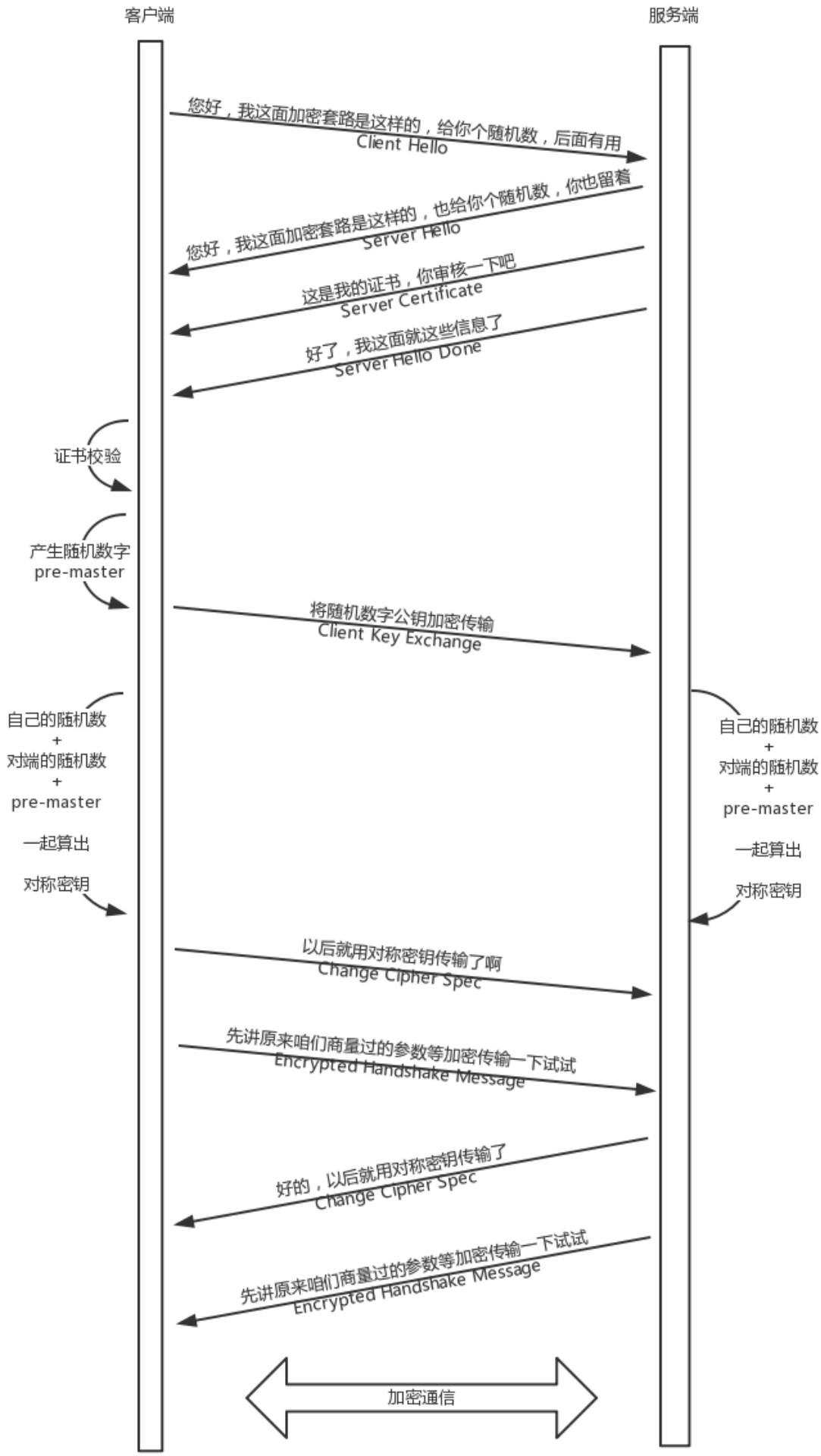
所以，CA的公钥也需要更牛的CA给它签名，然后形成CA的证书。要想知道某个CA的证书是否可靠，要看CA的上级证书的公钥，能不能解开这个CA的签名。就像你不相信区公安局，可以打电话问市公安局，让市公安局确认区公安局的合法性。这样层层上去，直到全球皆知的几个著名大CA，称为**root CA**，做最后的背书。通过这种**层层授信背书**的方式，从而保证了非对称加密模式的正常运转。

除此之外，还有一种证书，称为**Self-Signed Certificate**，就是自己给自己签名。这个给人一种“我就是我，你爱信不信”的感觉。这里我就不多说了。

HTTPS的工作模式

我们可以知道，非对称加密在性能上不如对称加密，那是否能将两者结合起来呢？**例如，公钥私钥主要用于传输对称加密的秘钥**，而真正的双方大数据量的通信都是通过**对称加密**进行的

这就是HTTPS协议的总体思路


客户端
服务端
您好，我这面加密套路是这样的，给你个随机数，后面有用
Client Hello
您好，我这面加密套路是这样的，也给你个随机数，你也留着
Server Hello
这是我的证书，你审核一下吧
Server Certificate
好了，我这面就这些信息了
Server Hello Done
证书校验
产生随机数字
pre-master
将随机数字公钥加密传输
Client Key Exchange
自己的随机数
+
对端的随机数
+
pre-master
一起算出
对称密钥
自己的随机数
+
对端的随机数
+
pre-master
一起算出
对称密钥
以后就用对称密钥传输了啊
Change Cipher Spec
先讲原来咱们商量过的参数等加密传输一下试试
Encrypted Handshake Message
好的，以后就用对称密钥传输了
Change Cipher Spec
先讲原来咱们商量过的参数等加密传输一下试试
Encrypted Handshake Message
加密通信

- 当你登录一个外卖网站的时候，由于是HTTPS，客户端会发送Client Hello消息到服务器，以明文传输TLS版本信息、加密套件候选列表、压缩算法候选列表等信息。另外，还会有一个随机数，在协商对称密钥的时候使用
- 然后，外卖网站返回Server Hello消息, 告诉客户端，服务器选择使用的协议版本、加密套件、压缩算法等，还有一个随机数，用于后续的密钥协商
- 然后，外卖网站会给你一个服务器端的证书，然后说：“Server Hello Done，我这里就这些信息了。
- 你当然不相信这个证书，于是你从自己信任的CA仓库中，拿CA的证书里面的公钥去解密外卖网站的证书。如果能够成功，则说明外卖网站是可信的。这个过程中，你可能会不断往上追溯CA、CA的CA、CA的CA的CA，反正直到一个授信的CA，就可以了。
- 证书验证完毕之后，觉得这个外卖网站可信，于是客户端计算产生随机数字Pre-master，发送Client Key Exchange，用证书中的公钥加密，再发送给服务器，服务器可以通过私钥解密出来。
- 到目前为止，无论是客户端还是服务器，都有了三个随机数，分别是：**自己的、对端的，以及刚生成的Pre-Master随机数。**通过这三个随机数，可以在客户端和服务器产生相同的对称密钥。
- 然后发送一个Encrypted Handshake Message，将已经商定好的参数等，采用协商密钥进行加密，发送给服务器用于数据与握手验证。当双方握手结束之后，就可以通过对称密钥进行加密传输了

上面的过程只包含了HTTPS的单向认证，也即客户端验证服务端的证书，是大部分的场景，也可以在更加严格安全要求的情况下，启用双向认证，双方互相验证证书。

重放与篡改

有了加密和解密，黑客截获了包也打不开了，但是它可以发送N次。

这个往往通过 `Timestamp` 和 `Nonce` 随机数联合起来，然后做一个不可逆的签名来保证。

Nonce随机数保证唯一，或者Timestamp和Nonce合起来保证唯一，同样的，请求只接受一次，于是服务器多次受到相同的Timestamp和Nonce，则视为无效即可

如果有人想篡改Timestamp和Nonce，还有签名保证不可篡改性，如果改了用签名算法解出来，就对不上了，可以丢弃了。

16.流媒体协议

三个名词系列

- **名词系列一**：AVI、MPEG、RMVB、MP4、MOV、FLV、WebM、WMV、ASF、MKV。例如RMVB和MP4
- **名词系列二**：H.261、 H.262、H.263、H.264、H.265。重点关注H.264。
- **名词系列**三：MPEG-1、MPEG-2、MPEG-4、MPEG-7。

视频其实就是快速播放一连串连续的图片

每一张图片，我们称为一**帧**。只要每秒钟帧的数据足够多，也即播放得足够快。比如每秒30帧，以人的眼睛的敏感程度，是看不出这是一张张独立的图片的，这就是我们常说的**帧率**（**FPS**）

每一张图片，都是由**像素**组成的，假设为1024*768（这个像素数不算多）。每个像素由RGB组成，每个8位，共24位

我们来算一下，每秒钟的视频有多大?

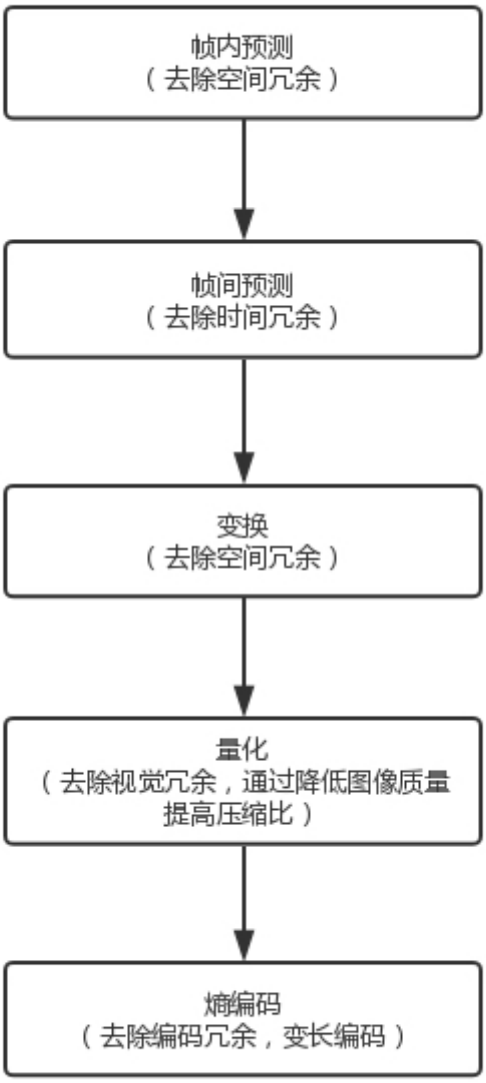
30帧 × 1024 × 768 × 24 = 566,231,040Bits = 70,778,880Bytes

针对数据量这么大的问题，人们想到了**编码**，就是看如何用尽量少的Bit数保存视频，使播放的时候画面看起来仍然很精美。**编码是一个压缩的过程**

视频和图片的压缩过程有什么特点?

视频和图片有这样一些特点

1. **空间冗余**：图像的相邻像素之间有较强的相关性，一张图片相邻像素往往是渐变的，不是突变的，没必要每个像素都完整地保存，可以隔几个保存一个，中间的用算法计算出来。
2. **时间冗余**：视频序列的相邻图像之间内容相似。一个视频中连续出现的图片也不是突变的，可以根据已有的图片进行预测和推断。
3. **视觉冗余**：人的视觉系统对某些细节不敏感，因此不会每一个细节都注意到，可以允许丢失一些数据。
4. **编码冗余**：不同像素值出现的概率不同，概率高的用的字节少，概率低的用的字节多，类似霍夫曼编码（Huffman Coding）的思路。



视频编码的两大流派

- 流派一：ITU（International Telecommunications Union）的VCEG（Video Coding Experts Group），这个称为**国际电联下的VCEG。**既然是电信，可想而知，他们最初做视频编码，主要侧重传输。名词系列二，就是这个组织制定的标准。

- 流派二：ISO（International Standards Organization）的MPEG（Moving Picture Experts Group），这个是**ISO旗下的MPEG**，本来是做视频存储的。例如，编码后保存在VCD和DVD中。当然后来也慢慢侧重视频传输了。名词系列三，就是这个组织制定的标准。

后来，ITU-T（国际电信联盟电信标准化部门，ITU Telecommunication Standardization Sector）与MPEG联合制定了**H.264/MPEG-4 AVC**，这才是我们这一节要重点关注的

经过编码之后，生动活泼的一帧一帧的图像，就变成了一串串让人看不懂的二进制，这个二进制可以放在一个文件里面，按照一定的格式保存起来，这就是名词系列一

直播

编码后的二进制数据也可以通过某种网络协议进行封装，放在互联网上传输，这就是直播

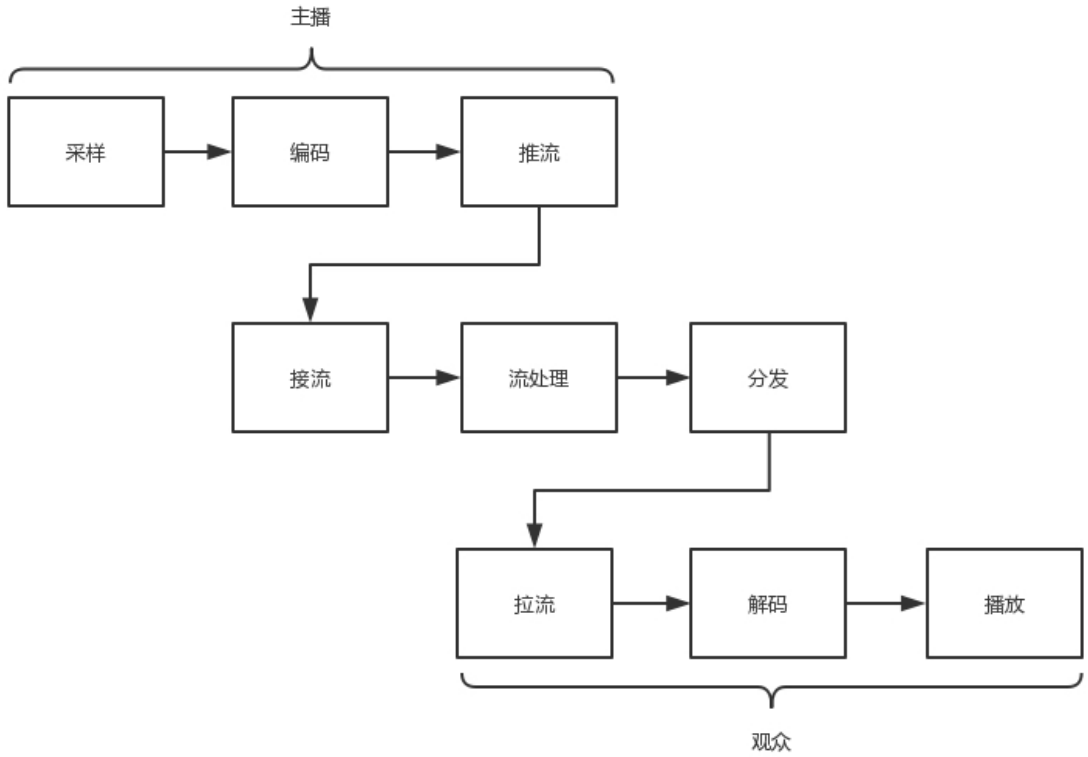
网络协议将**编码**好的视频流，从主播端推送到服务器，在服务器上有个运行了同样协议的服务端来接收这些网络包，从而得到里面的视频流，这个过程称为**接流**

服务端接到视频流之后，可以对视频流进行一定的处理，例如**转码**，也即从一个编码格式，转成另一种格式。因为观众使用的客户端千差万别，要保证他们都能看到直播

流处理完毕之后，就可以等待观众的客户端来请求这些视频流。观众的客户端请求的过程称为**拉流**

如果有非常多的观众，同时看一个视频直播，那都从一个服务器上**拉流**，压力太大了，因而需要一个视频的**分发**网络，将视频预先加载到就近的边缘节点，这样大部分观众看的视频，是从边缘节点拉取的，就能降低服务器的压力

当观众的客户端将视频流拉下来之后，就需要进行**解码**，也即通过上述过程的逆过程，将一串串看不懂的二进制，再转变成一帧帧生动的图片，在客户端**播放**出来



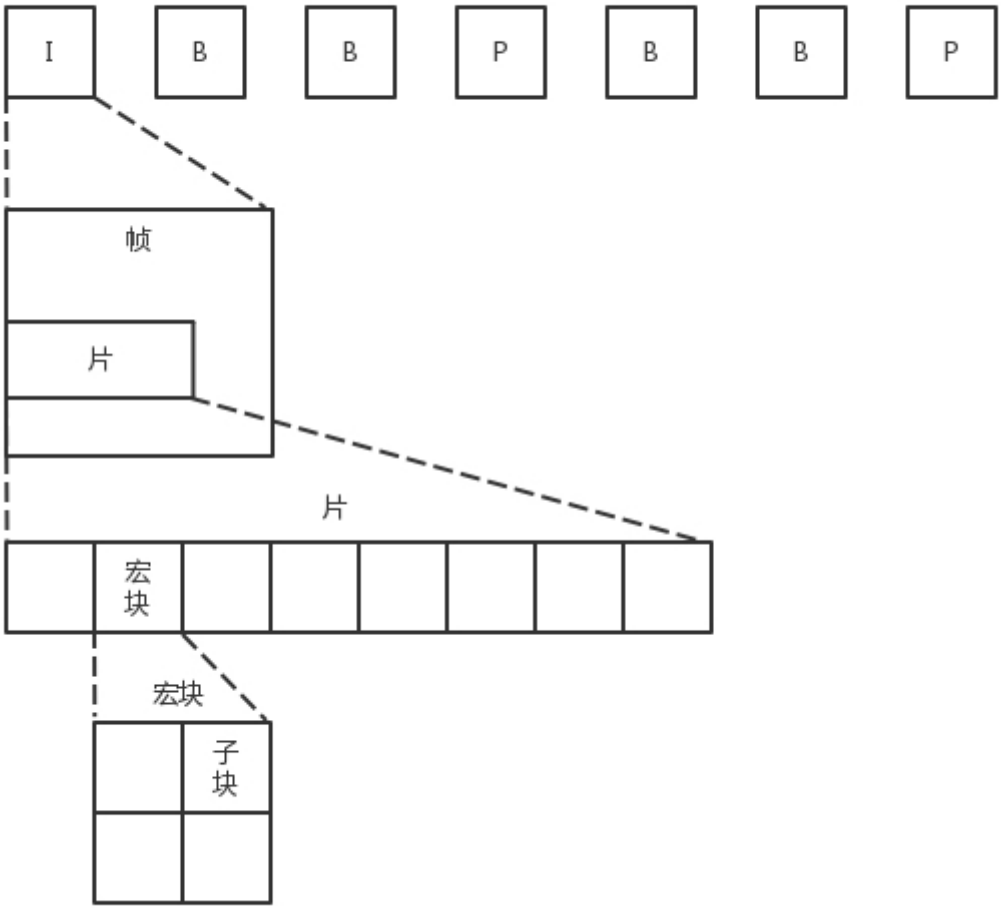
编码：如何将丰富多彩的图片变成二进制流？

视频序列分成三种帧

- **I帧**，也称关键帧。里面是完整的图片，只需要本帧数据，就可以完成解码。

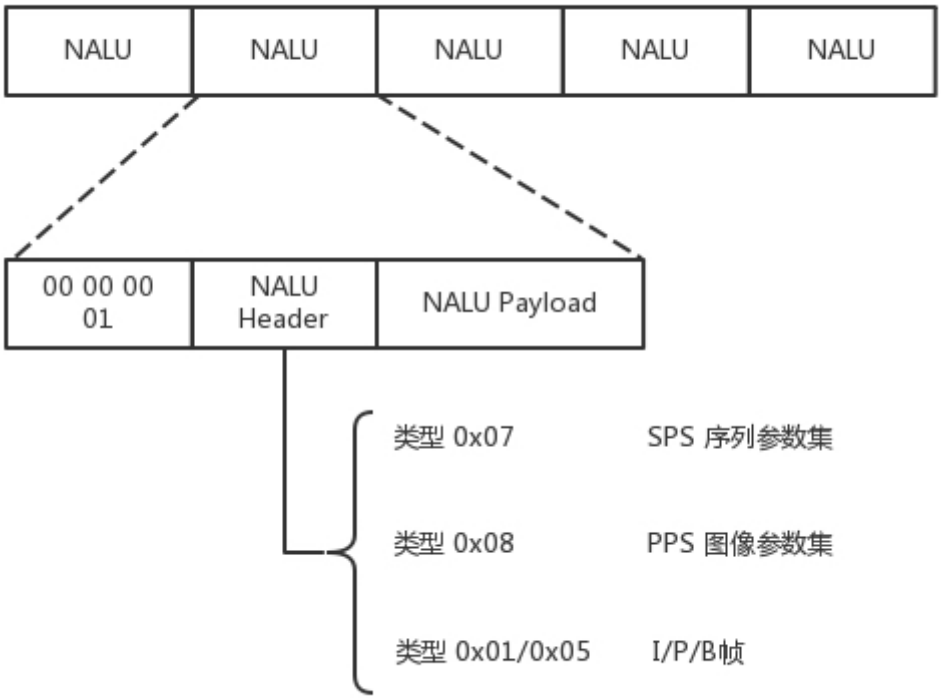
- **P帧**，前向预测编码帧。P帧表示的是这一帧跟之前的一个关键帧（或P帧）的差别，解码时需要用之前缓存的画面，叠加上和本帧定义的差别，生成最终画面。
- **B帧**，双向预测内插编码帧。B帧记录的是本帧与前后帧的差别。要解码B帧，不仅要取得之前的缓存画面，还要解码之后的画面，通过前后画面的数据与本帧数据的叠加，取得最终的画面。

可以看出，I帧最完整，B帧压缩率最高，而压缩后帧的序列，应该是在IBBP的间隔出现的。这就是**通过时序进行编码**



在一帧中，分成多个片，每个片中分成多个宏块，每个宏块分成多个子块，这样将一张大的图分解成一个个小块，可以方便进行**空间上的编码**

这个二进制流是有结构的，是一个个的**网络提取层单元**（**NALU**，**Network Abstraction Layer Unit**）
变成这种格式就是为了传输，因为网络上的传输，默认的是一个个的包，因而这里也就分成了一个个的单元



每一个NALU首先是一个起始标识符，用于标识NALU之间的间隔；然后是NALU的头，里面主要配置了NALU的类型；最终Payload里面是NALU承载的数据