计算机网络

参考

moon聊技术

小林coding

程序员库森

帅地玩编程

整体性

- Conception：
  a. OSI七层：应用层、表示层、会话层、传输层、网络层、数据链路层、物理层（大而全，复杂，先有理论模型，没有实际应用）
  b. TCP/IP四层：应用层、传输层、网络层、网络接口层（实际引用发展，实质上TCP/IP只有上面三层，第四层没有什么具体内容）
  c. 五层：应用层、传输层、网络层、数据链路层、物理层（对七层和四层的折中）
- Question：
  a. 每一层的作用？
     - 应用层：专注于为用户提供应用功能，不关心数据如何传输，工作在用户态，其下工作在内核态
     - 表示层：数据格式的转换，加解密压缩解压缩
     - 会话层：负责两点之间建立、维持和终止通信，例如服务器验证用户登录
     - 传输层：为应用层提供数据传输服务，将上层数据分段并提供端到端的，可靠/不可靠的传输；还要负责流量控制等问题
     - 网络层：实际的传输功能，寻址（找到 IP 地址对应的设备，如果找不到设备，就需要路由导航）+ 路由（找到数据应该往哪里发送 方向盘）功能
     - 数据链路层：为网络层提供链路级别的服务，将网络层传下来的包组成帧
     - 物理层：为数据链路层提供二进制传输的服务，传输bit流
  b. 每一层传输的基本数据单元?
     - 应用层：报文
     - 传输层：段
     - 网络层：包
     - 数据链路层：帧
     - 物理层：bit流
  c. 计算机网络7层协议每一层设计的协议?
     - 应用层：HTTP、FTP（文件传输）、SMTP（电子邮件）、SNMTP（网络管理）、TELNET（远程登录协议）、DNS、SSH（安全外壳协议）
     - 表示层：格式转换、数据加密、压缩，
     - 会话层：建立、管理、维护会话

- 传输层：TCP、UDP（建立、管理、维护端到端的链接）
- 网络层：IP、ICMP（Internet控制报文协议，PING协议使用的）、ARP、RARP（属于网络层协议，工作内容是在数据链路层的）
- 数据链路层：PPP、ARQ、路由器
- 物理层：IEEE802、交换机

d. 为什么要分层？

- 各层之间相互独立，并不需要知道下一层是如何实现的，仅需要通过接口联系
- 灵活性好，某一层发生变化时，只要借口不变，其他层不受影响
- 易于设计实现和标准化
- 太少每一层协议太复杂，太多各层功能无法分清

e. **键入网址到网页显示，都发生了什么？**

- 应用层：输入URL，解析URL（协议名称、服务器域名、目录名）；生成HTTP请求信息（GET、POST报文）；DNS查询域名对应的IP地址（如果本地有DNS缓存或者hosts文件直接返回）
- 传输层：发送TCP连接请求，三次握手；发起HTTP请求
- 网络层：生成IP头部
- 数据链路层：加上MAC头部
- 网卡（检查、数字信息转换为电信号）
- 交换机
- 路由器
- 到达服务端（解析MAC、IP、TCP、HTTP）
- 服务器再返回响应，生成HTTP响应，加TCP、IP、MAC、经过网卡、交换机、路由器到达客户端
- 四次挥手；浏览器解析HTML、浏览器显示渲染页面、执行js脚本，相应ajax请求

f. 浏览器输入URL到页面打开的过程如何加速

- 使用http3.0协议（多路复用、tcp连接换成udp连接减少三次握手时间、0-RTT建连，向前纠错机制减少重传）
- 数据压缩，减少传输时间和次数
- CDN服务器，地域分布式缓存静态资源
- 页面先呈现静态资源及框架，动态资源延迟加载，从使用者角度加速
- 每次打开做好缓存，例如DNS缓存、静态资源缓存

g. 服务器收到一个包如何处理

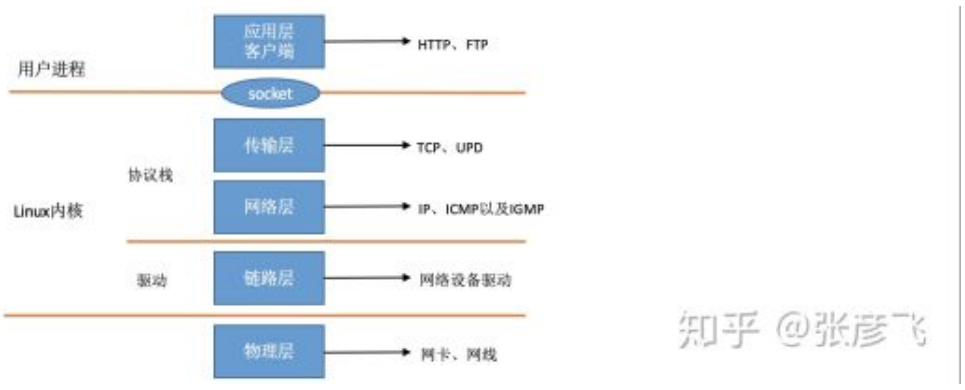
- 查看MAC头部：是否和服务器自己的MAC地址符合
- 查看IP头部：
  - 看IP地址是否符合；
  - 从IP头部中能得到用的是TCP还是UDP协议
- 查看TCP头部：
  - 查看序列号是不是想要的，返回一个ACK，不是则丢弃
  - 查看TCP中的端口号，将包发给HTTP进程
- HTTP进程发现

- 把网页封装到响应报文中，加上TCP、IP、MAC头部打包发给客户端

h. **网卡、交换机、路由器工作层次和基本原理**

- 网卡：
  - 工作在OSI的物理层和数据链路层；
  - 通过硬件支持和网卡驱动程序实现802.3Ethernet、802.11WIFI协议吗，完成物理层的信号收发和数据链路层的帧的封装和解封；
  - 网卡和计算机操作系统之间主要通过I/O中断技术和直接内存访问（DMA）技术完成交互
  - 进行 socket 编程时，OS内部实现的TCP/IP协议栈的相关模块完成TCP报文和IP数据包的封装或解封装，由网卡实现链路层帧的封装或解封装，由网卡实现物理信号的转换和发射。
- **路由器和交换机**：
  - 路由器工作在网络层；交换机工作在数据链路层
  - 路由器根据IP地址寻址；交换机根据MAC地址寻址
  - 路由器将一个IP分给多个主机使用，对外IP也是同一个；交换机对外表现的IP各有不同
  - 路由器提供防火墙的服务；交换机不能提供该功能

i. 内核收包过程？



- 讲解

j. **影响网络带宽的因素？**

- 带宽：在单位时间(一般指的是1秒钟)内能传输的数据量
  - 物理因素，网线是否完好，没有干扰
  - 共享带宽，用的人多不多，能分到多少资源

k. **千兆网理论带宽多大？**

- 网卡速度按1000进位，下载速度按1024进位
- 1000Mbps = 1,000,000,000比特/秒 = 125,000,000字节/秒 = 125MB/s

应用层 HTTP

- Conception：

a. HTTP定义

b. HTTP状态码

c. HTTP请求方式

d. HTTP请求/相应报文

e. HTTP1.0

f. HTTP1.1
g. HTTPS（SST/TLS）
h. HTTP2.0
i. HTTP3.0
j. DNS
k. Cookie/Session
l. 数字签名、数字证书

- Question：

a. 如何理解HTTP协议?
   - 基于TCP协议实现，超文本传输协议，一个简单的请求-响应协议，超文本（不仅是文字，还有图片、视频、链接）、传输（双向协议、两点之间传输数据）、协议（约定和规范）

b. 说一下HTTP常见的请求方法?

GET	查
POST	改
PUT	增 没有验证机制，存在安全问题
DELETE	删
HEAD	获取报文首部，不返回主体
PATCH	对资源进行部分修改
OPTIONS	查询指定的URL支持的方法
CONNECT	要求在与代理服务器通信时建立隧道
TRACE	追踪路径

c. GET/POST区别?
   - GET和POST本质上就是TCP链接，并无区别，由于HTTP规定和浏览器的限制，导致在应用过程中体现出一些不同

区别	**GET**	**POST**
数据传输方式	从服务器获取数据	向服务器新增/提交数据
对数据长度的限制	当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）	无限制
对数据类型的限制	只允许 ASCII 字符，存在中文需要先进行编码（因为写在URL中的规定）	无限制，支持标准字符集
安全性	较差，所发送的数据是 URL 的一部分，会显示在网页上	较好 参数不会被保存在浏览器历史或 WEB 服务器日志中
可见性	显示在 URL 上	不显示
收藏为书签	可以	不可以

历史记录	可以被保留在历史记录当中	不可以被保留
缓存	能被缓存（因为安全且幂等）	不可以被缓存（修改服务器资源，不存）
传参方式	使用URL或者Cookie传参	将数据放在BODY（实体主体）中
幂等(多次访问结果相同)安全(不会破坏服务器资源)	幂等（HEAD、HEAD、PUT、DELETE）且安全（HEAD、OPTINOS）	不幂等不安全（PUT、DELETE）

- 可缓存（如果要对响应进行缓存，需要满足以下条件）：请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数 情况下不可缓存的；响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501； 响应报文的 Cache-Control 首部字段没有指定不进行缓存。

d. GET请求中URL编码的意义?

- GET请求中会对URL中非西文字符进行编码，为了避免歧义
- eg：name1=va&lu=e1（va&lu=e1代表一个值），但可能理解为name1 = va & lu = e1；URL编码后结果：“name1=va%26lu%3D”，服务端会把%后的字节当初普通字符

e. HTTP状态码?

状态码	**信息**
1xx	信息，中间状态，服务器收到请求，需要请求者继续执行操作
2xx	成功，操作被成功接收并处理
3xx	重定向，需要进一步的操作以完成请求
4xx	客户端错误，请求包含语法错误或无法完成请求
5xx	服务器错误，服务器在处理请求的过程中发生了错误

- 101：切换请求协议，从 HTTP 切换到 WebSocket
- 200 OK：服务器成功处理了请求
- 204 No Content：与200基本相同，响应头没有body数据
- 206 Partial Content：部分数据，和HTTP断点续传有关
- 301（Move permanently）：永久重定向，需要用新的URL访问
- 302（Found）：临时重定向，暂时需要另一个URL访问
- 304（Not Modified）：缓存重定向，协商缓存命中，告诉客户端可以继续使用缓存资源
- 400（Bad Request）：客户端请求报文有错
- 401（Unauthorized）：身份过期，没有通过身份认证
- 403（Forbidden）：通过了身份认证但是没有权限党文
- 404（Not Found）：资源找不到
- 499（client has closed connection）：客户端链接已经关闭
- 500（Internal Server Error）：笼统的服务器内部错误
- 501（Not Implemented）：请求的功能还不支持；即将开业，敬请期待

- 502（Bad Gateway）：错误网关，因此说它是无效的；服务器作为网关或代理时返回的错误码
- 503（Service Unavailable）： 服务器目前无法使用(由于超载或停机维护)。 通常，这只是暂时状态
- 504（Bad Gateway timeout）：网关超时
- 505：不支持HTTP协议版本

f. HTTP缓存机制?
- 强缓存：命中则返回资源
- 协商缓存：和服务器进行协商，满足则返回缓存
- 强缓存：
  - Expire（过期时间HTTP1.0）
  - Cache-Control（HTTP1.1）
- 协商缓存：
  - ETag和If-None-Match：ETag用来判断是否改变，给服务器发的时候ETag在If-None-Match里
  - Last-Modified和If-Modified-Since：Last-Modified是最后一次改变时间，Last-Modified放在If-Modified-Since里
- ETag精度上优，性能上低
- 强缓存和协商缓存共同配合使用
- 参考

g. HTTP请求/相应报文格式、常见字段?
- 请求：请求头（请求方法+URI协议+版本） + 请求首部 + 空行 + 请求主体
- 响应：状态行 （版本+状态码+原因短语）+ 响应首部 + 空行 + 响应主体

h. HTTP请求的完整过程?
- 域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 --> 服务器响应http请求，浏览器得到html代码 --> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户

i. 长连接和短连接？（Keep-live）
- HTTP1.0默认使用短连接，访问的某个HTML类型的web网页中有其他图像、CSS文件，浏览器会建立一个HTTP会话
- HTTP1.1默认使用长连接，响应头会加入Connection : keep_alive，一个网页打开之后TCP连接不会关闭，再访问会重用这个链接，keep-alive有一个保护时间

j. 在交互过程中如果数据传送完了，还不想断开连接怎么办，怎么维持?
- 在 HTTP 中响应体的 Connection 字段指定为 keep-alive
- connection：keep-alive;

k. HTTP1.0问题?
- 短连接无状态，每次三次握手四次挥手
- 第一个请求发出去必须等确认回来，才能发第二个请求

l. HTTP1.1优缺点?
- 优点：简单、灵活易扩展、应用广泛、长连接

- 缺点：明文传输、头部巨大、队头阻塞（管道中一个请求阻塞，后面的请求也被阻塞）、不支持服务器推送、并发连接有限、

m. HTTPS做了什么（三点）？
- 混合加密（解决窃听）
- 摘要算法（解决篡改）
- CA证书（解决冒充）

n. HTTPS优缺点?
- 优点：混合加密解决窃听，摘要算法解决篡改，CA证书解决冒充，虽然不是绝对安全但大幅增加了中间人攻击的成本
- 缺点：费时费电，安全是有范围的，技术门槛高，CA证书收费，需要更多的服务器资源

o. HTTP2.0优缺点?
- 优点：头部压缩、二进制格式、数据流、多路复用、服务器推送
- 缺点：队头阻塞（用的是TCP，stream数据流后续内容需要等待）、握手次数多、连接迁移（4G换WIFI，四元组变化，需要再次三次握手）

p. HTTP3.0优点?
- UDP+QUIC（HTTP2 + TLS + TCP）解决队头阻塞、握手次数减少，解决连接迁移

q. HTTP和HTTPS的区别?
- 安全：HTTP是明文传输，HTTPS在TCP和HTTP中间加入了SSL/TLS安全协议，加密传输
- 握手次数：HTTP三次握手即可，HTTPS在三次握手后还需要SSL/TLS的三次握手过程
- 端口：HTTP端口号为80，HTTPS端口号为443
- 证书：HTTP不需要，HTTPS协议需要向CA申请数字证书

r. HTTP1.0和HTTP1.1的区别?
- 长链接：1.0每一次请求都需要三次握手，并且是串行的请求；1.1新增了长链接
- 管道：1.0发送一个请求必须等待响应才能发送第二个，1.1新增了管道，同一个TCP链接中可以发出多个请求
- 断点续传：HTTP1.0不支持，1.1新增了range字段，指定数据字节位置，开始可以断点续传
- 新增HOST请求头：CSDN
- 缓存处理不同：CSDN
- 错误状态码：1.1新增了24个错误状态响应码，如410表示服务器上的资源被永久性删除

s. HTTP1.1和HTTP2.0的区别?
- 头部压缩：如果发出多个请求，头部相同，2.0会消除重复的部分
- 二进制格式：1.1用纯文本，2.0用二进制格式
- 数据流：2.0数据包不按顺序发送；因此需要数据包标记，指出属于哪个回应，请求回应的数据包称为数据流，每个数据流都做了标记有编号；客户端还可以指定数据流优先级，可以先响应优先级高的
- 多路复用：2.0可以在一个连接中并发多个请求或响应，解决了队头阻塞问题
- 服务器推送：1.1服务器只能被动的响应，2.0可以主动向客户端发送消息

t. HTTP2.0和HTTP3.0的区别?
- 传输层协议不同：2.0用TCP（有队头阻塞），3.0用UDP（无队头阻塞）

- 3.0新增了QUIC（在UDP之上伪TCP+TLS+HTTP2.0的多路复用协议），基于UDP的QUIC可以实现类似TCP的可靠传输
- 握手次数：2.0基于HTTPS实现，三次握手+三次握手，总共6次；3.0用QUIC只需要3次
- 连接迁移（4G换WIFI后IP地址变，QUIC不用四元组，用两个ID标记自己，不需要再次三次握手）

u. HTTP1.0 1.1 2.0 3.0总结

	1.0	1.1	2.0	3.0
优点		长连接 管道	头部压缩 二进制格式 stream数据流 多路复用 服务器推送	udp协议 QUIC握手 连接ID
缺点	短连接 队头阻塞	头部过大 队头阻塞 服务器被动	队头阻塞 握手冗余 连接切换	

v. SST/TLS1.2握手过程 /HTTPS工作过程/HTTPS原理?

- 四次握手：
- 第一次（Client Hello）：客户端向服务端发起连接请求，客户端发一个随机数，发自己支持的加密规则
- 第二次（Server Hello）：服务器把证书（证书里有公钥）发给客户端，服务端发一个随机数，服务器自己保留私钥，选加密算法
- 第三次（Client Key Exchange）：客户端验证证书，客户端用公钥对秘钥（客户端随机数 + 服务端随机数）加密，发给服务端；服务端用私钥解密，得到客户端秘钥
- 第四次（Change Ciper Spec）：服务端生成会话秘钥，用客户端公钥加密；客户端得到之后用私钥解密，得到服务端秘钥
- 之后用协商好的秘钥进行数据传输

- 公钥加密，私钥解密
- TLS1.0用的是RSA秘钥交换算法，TLS1.2目前是主流，使用ECDHE算法（椭圆曲线等等）

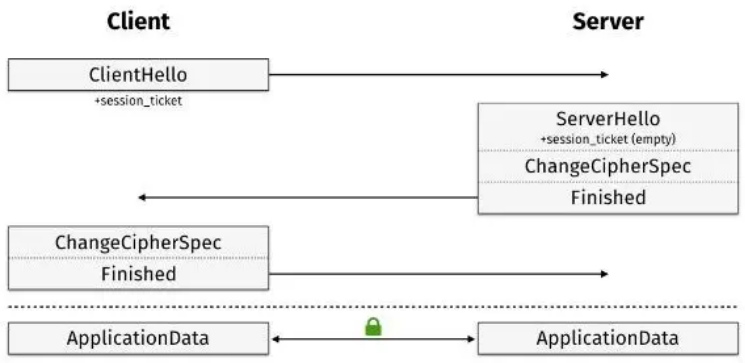
w. **对称加密和非对称加密?**

- 非对称加密：
  - 加密和解密使用不同的秘钥，一把作为公开的公钥，另一把作为私钥。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。
  - 优点：安全性更高，公钥是公开的，秘钥是自己保存的，不需要将私钥给别人。
  - 缺点：加密和解密花费时间长、速度慢，只适合对少量数据进行加密。
- 对称加密：
  - 加密和解密使用同一个秘钥
  - 优点：算法公开、计算量小、加密速度快、加密效率高。

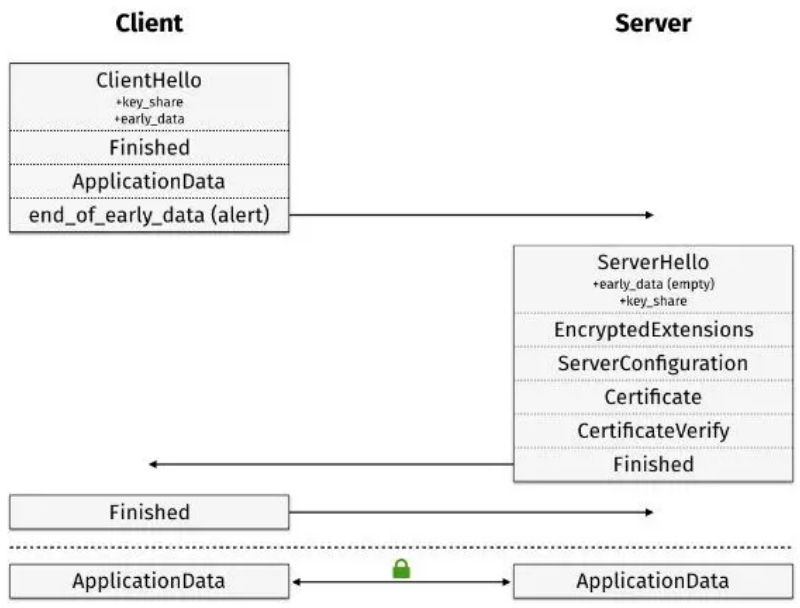
- 缺点：不够安全，如果一方的秘钥被泄露，那么加密信息也就不安全了。另外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一秘钥，这会使得收、发双方所拥有的钥匙数量巨大，密钥管理成为双方的负担。

x. TLS1.3

- 客户端和服务端互相ping通，提供SSL/TLS证书，交换一个受支持的密码套件就达成一致
- 会话恢复
  - TLS1.2：利用session id



  - TLS1.3：0RTT，直接加密数据进行发送

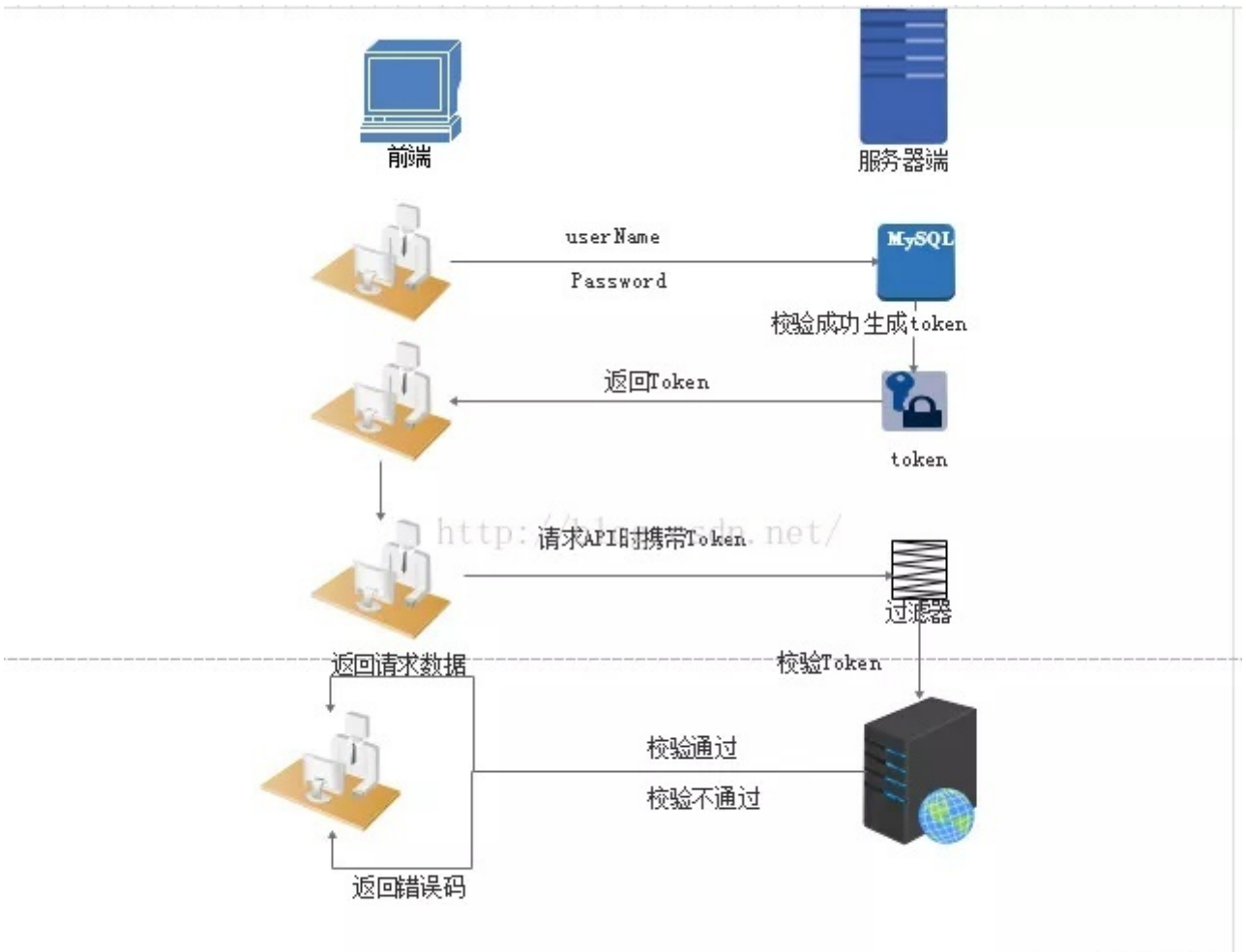


y. Cookie和Session如何配合？

- 问题：http协议无状态
- Cookie过程：
  - cookie由服务器生成，发送给浏览器
  - 客户端向服务端发起http请求
  - 服务端设置一个创建cookie的指令，响应给客户端
  - 客户端收到服务器指令，客户端创建一个cookie
  - 下一次请求时候，客户端发送请求携带这个cookie
- Session过程：
  - 客户端向服务端发起请求
  - 服务端根据设置的 session创建指令，在服务端创建一个编号为sessionid的文件
  - 服务端将sessionid响应给客户端，客户将编号存在cookie中

- 下一次请求，客户端发送请求携带这个sessionid携带在请求中

- Toke n过程：
  - 用户通过用户名和密码发送请求；程序验证；程序返回一个签名的token 给客户端。客户端储存token,并且每次用于每次发送请求；服务端验证token并返回数据。（短期验证）



z. Cookie、Session、token?

- cookie和token比较：cookie需要在服务端存session，有状态；token只需要验证，无状态

	Cookie	Session	token
作用范围	客户端（浏览器）	服务端	
存取方式	只能存ASCII	存任意数据类型	
有效期	可设置为长时间保持（默认登录功能）	较短	
隐私策略	不太安全	安全性较好	
存储大小	单个Cookie保存的数据不能超过4K	远高于Cookie	
保存位置	设置过期时间失效-硬盘；会话结束消失-内存	服务端保存的一个数据结构，跟踪用户状态，可以保存在集群、数据库、文件中	
弊端	**Cookie 中的所有数据在客户端就可以被修改，易伪造，所以产生了Session**	**服务端记录的用户数量太多，占用资源**	

- 还分不清 Cookie、Session、Token、JWT?

aa. session和token区别?

- 其实 token 和 session 的验证机制最大的区别是用“签名验证机制”代替了“白名单验证机制”。
- session 的问题在于**前端传来的 session_id 可以伪造**，所以必须在服务器维护一个 session_id 的白名单来验证 session_id 的合法性。
- token 的改进之处就在这里，token 通过签名机制，只要前端传来的 token 能通过签名认证就是合法的，不需要服务器维护任何东西，所有的需要东西都放在在 token 里面。

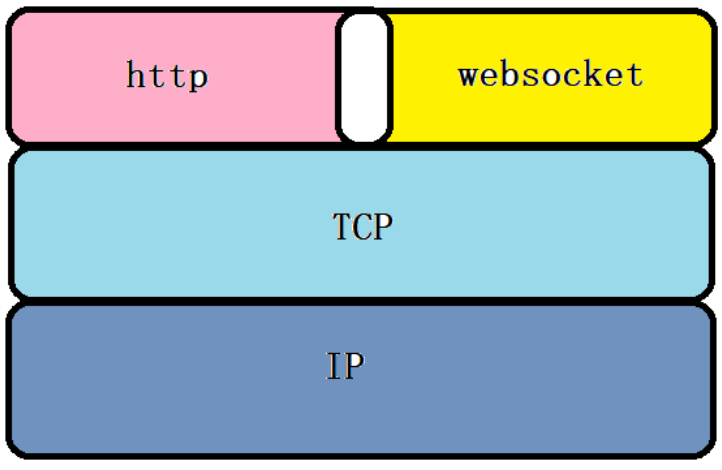
ab. 分布式Session问题?

- 用户多的时候，后端有多台服务器，用户访问不同的服务器会出现登录失效的问题
- 解决方案：
  - 客户端存储（用cookie）；
  - Nginx采用ip_hash的负载策略
  - Session复制；
  - 共享Session（缓存中间件Redis）
  - 文章

ac. Websocket和socket区别

- Socket是TCP/IP网络的API，是为了方便使用TCP或UDP而抽象出来的一层，是位于应用层和传输控制层之间的一组接口；
- WebSocket则是一个典型的应用层协议。

关系图



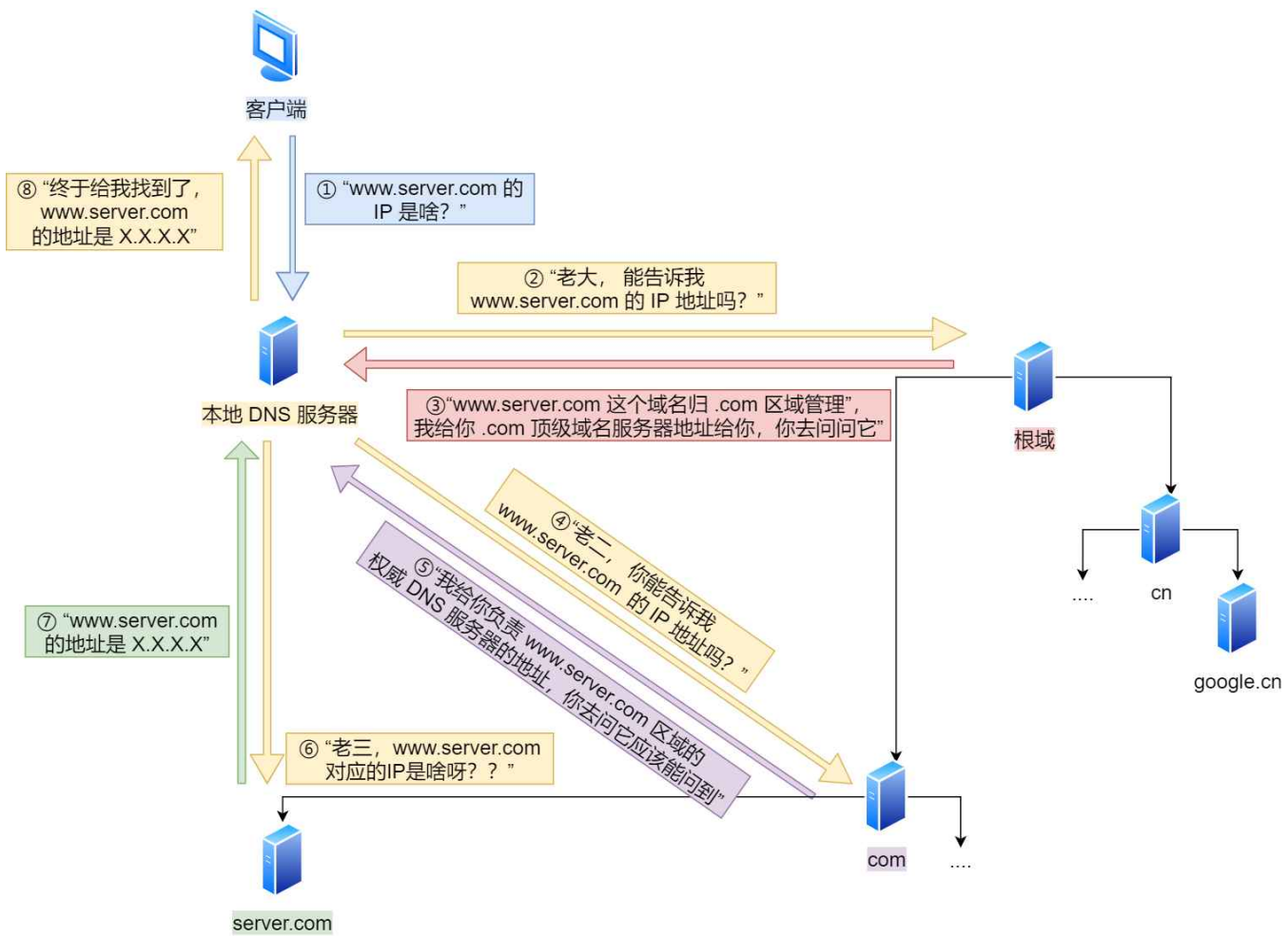
- 相同：WebSocket和HTTP都是应用层协议，都是可靠传输协议，都基于TCP
- 区别：WebSocket是双向协议，HTTP是单向协议
- 联系：WebSocket在建立握手时，数据是通过HTTP传输的。但是建立之后，在真正传输时候是不需要HTTP协议的

ad. 数字签名、数字证书

- 数字证书：如何证明浏览器收到的公钥一定是该网站的公钥?
- 数字签名：如何放**防止数字证书被篡改**?数字签名把证书原本的内容生成一份“签名”，比对证书内容和签名是否一致就能判别是否被篡改
- 文章

应用层 DNS

- Conception：
  a. 递归查询、迭代查询
  b. 负载均衡、负载分配
  c. 根服务器、顶级域服务器、权威服务器
  d. DNS缓存/域名缓存
- Question：
  a. DNS用的是TCP协议还是UDP协议？
     - DNS占用53号端口，同时使用TCP和UDP协议
     - **区域传输**时候用TCP协议
     - **域名解析**时使用UDP协议
  b. DNS解析过程？
     - 先在浏览器中查找域名对应的IP地址缓存，再找硬盘中的hosts文件
     - 如果浏览器缓存和hosts文件中都没有ip地址，浏览器会发一个DNS请求给本地dns服务器（网络接入服务商提供）。
     - 本地DNS服务器先查询缓存记录，有记录直接返回结果。缓存中没有还要向DNS根服务器（包含.com、.cn..）进行查询
     - 本地DNS服务器向域服务器（com）发出请求，域服务器告诉本地DNS服务器，
     - 本地服务器向权威服务器（server.com）发出请求，得到域名和IP地址的对应关系，并且保存在缓存中。



  c. DNS工作原理？
  d. DNS查询方式？
     - 递归查询：主机向本地域名服务器查询；不返回直接向下一步查询

- 迭代查询：本地域名服务器向根域名服务器；每次返回给本地服务器，本地服务器再查询

e. **DNS负载均衡?**
   - 在DNS服务器中为同一个主机名配置多个IP地址，在应答DNS查询时，DNS服务器对每个查询将以DNS文件中主机记录的IP地址按顺序返回不同的解析结果，将客户端的访问引导到不同的机器上去，使得不同的客户端访问不同的服务器

f. DNS域名缓存?
   - 目的：提高查询效率、减少服务器负荷、减少网络上的DNS查询报文数量
   - 服务器回答一个查询请求时，在响应中都指明有效存在的时间；增加这个时间可以减少网络开销，减少时间可以提高域名正确性
   - 不仅本地域名服务器需要高速缓存，主机中也需要高速缓存

g. 清除dns缓存的命令?
   - ipconfig/flushdns

传输层 TCP UDP

- Conception：
  a. TCP/UDP
  b. 三次握手
  c. 四次挥手
  d. TCP保活机制/保活计时器
  e. 超时重传、快速重传
  f. 滑动窗口
  g. 流量控制
  h. 拥塞控制（慢启动、拥塞避免、快速重传、快速恢复）
  i. 半连接、全连接队列
  j. 拆包、粘包
  k. MTU、MSS、RTO、RTT
  l. TCP的keepalive
  m. Socket
- Question：
  a. TCP/UDP区别?

	TCP	**UDP**
是否连接	面向连接（双方传输数据前必须建立一条通道）	面向无连接
是否可靠	可靠（无差错、不丢失、不重复、按序到达）	不可靠
是否有序	有序，接收端可能乱序，TCP会重新排序	无序
传输速度	慢	快
连接对象个数	一对一（只能有两个端点，即两个套接字）	一对一、一对多、多对一、多对多
传输方式	字节流（eg：发送方交给TCP10个，TCP用4个传送，不保留数据边界）	报文（不合并不拆分，保留数据边界

首部开销	最小20字节，最大60字节	8字节
适用场景	要求可靠传输的应用，文件传输	实时应用（视频会议、直播）

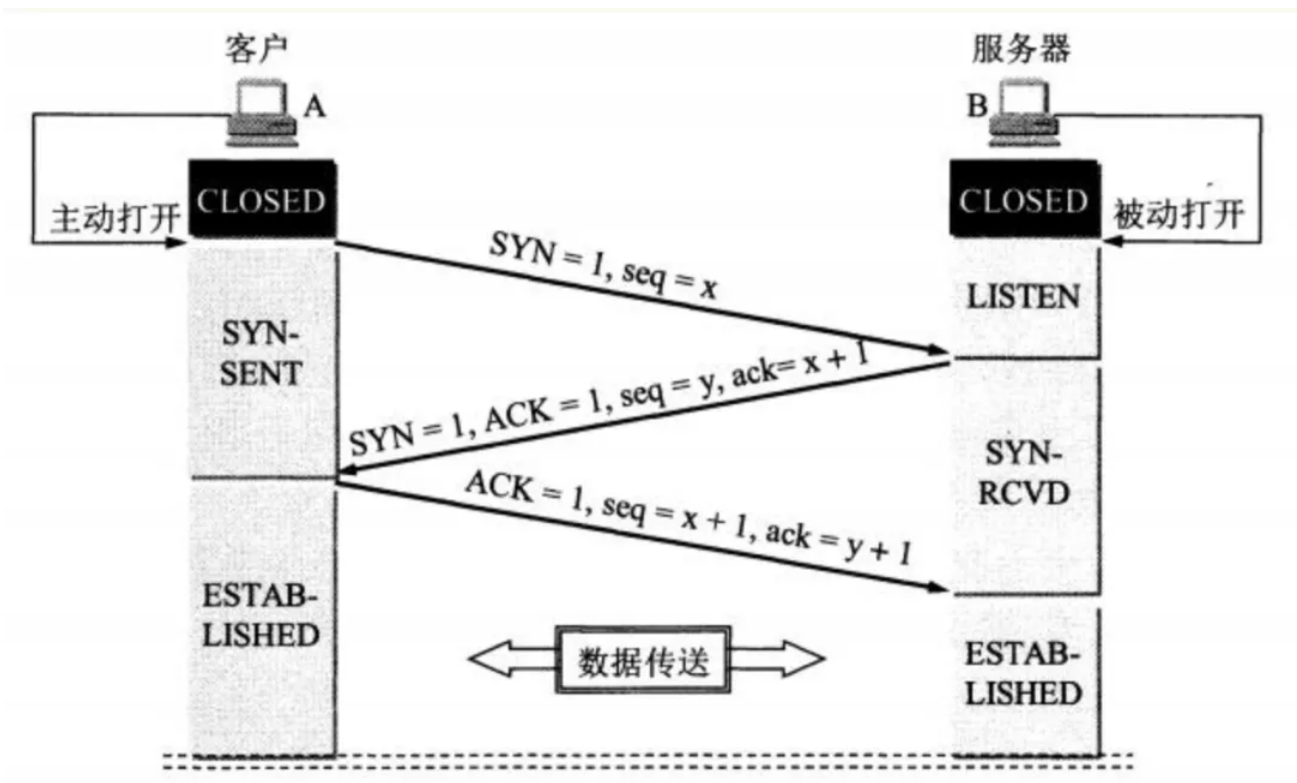
b. TCP特点、UDP特点？
   - TCP：面向连接、点对点、可靠传输、全双工（两方在任何时候都能发送数据）、字节流
   - UDP：无连接、不仅一对一、不可靠、首部开销小、面向报文

c. TCP/UDP对应的场景、对应协议?
   - TCP：面向连接的，可靠的；HTTP/HTTPS、SMTP、HTTP、FTP、TELNET
   - UDP：面向无连接，不可靠的；音视频、直播；广播通信；DNS、SNMP（简单网络管理协议）、TFTP（简单文件传输协议）

d. TCP和UDP可以同时使用吗?
   - 一个进程可以使用同一个端口同时监听TCP、UDP请求
   - 端口的标识是端口号+协议名称，这样在IP数据包中就可以判断是UDP还是TCP协议

e. TCP/UDP头部?
   - 见总结性头部对比

f. 三次握手（发的是什么报文，有哪些信息，发的进入什么状态，对端收之后不回的进入什么状态）
   i. 过程



- 第一次：A的TCP进程创建一个传输控制块TCB，向B发送连接请求报文，同部位SYN=1，选择一个初始序列号seq = x，A进入SYN_SENT状态（客户端的发送能力）
- 第二次：B收到连接请求报文段，同步位SYN=1，确认位ACK=1，选择一个初始序列号seq = y，确认号ack = x + 1，B进入SYN_RCVD状态（服务端接受能力正常、服务端发送能力正常）
- 第三次：A收到B的确认后，再发出确认报文，确认位ACK=1，确认号ack = y + 1，进入ESTABLISHED状态，B收到A的确认报文也进入ESTABLISHED状态（客户端接受能力正常）
- SYN_SENT：发送连接请求后等待匹配的连接请求
- SYN_RECRIVED：收到和发送一个连接请求后等待确认

- ESTABLISHED：代表一个打开的连接，可以发送数据

ii. **ISN（Initial Sequence Number）固定吗?**

- 不固定
  - 固定的话攻击者容易猜出来
  - 而已无法保证久的报文被丢弃
- 范围是0 - 2^32 -1 之间，序号随机生成，每个TCP使用的序号不同

iii. 三次握手能携带数据吗?

- 第三次可以携带
- 第一次如果可以携带的话，攻击者会放入大量数据，重发多次SYN报文，服务器阻塞，避免服务器容易受到攻击

iv. 为什么是三次?

- 防止已经过期的连接到达
  - 旧连接比新连接先到服务端，客户端可以判断是历史连接，回RST报文，两次的话只能默认建立连接
- 双方同步序列号（序列号可以去除重复数据、按序接收）
- 避免资源浪费：三次是最小值

v. 为什么不是两次?

- 不能判断是不是历史连接
- 无法同步双方的序列号

vi. 为什么不是四次?

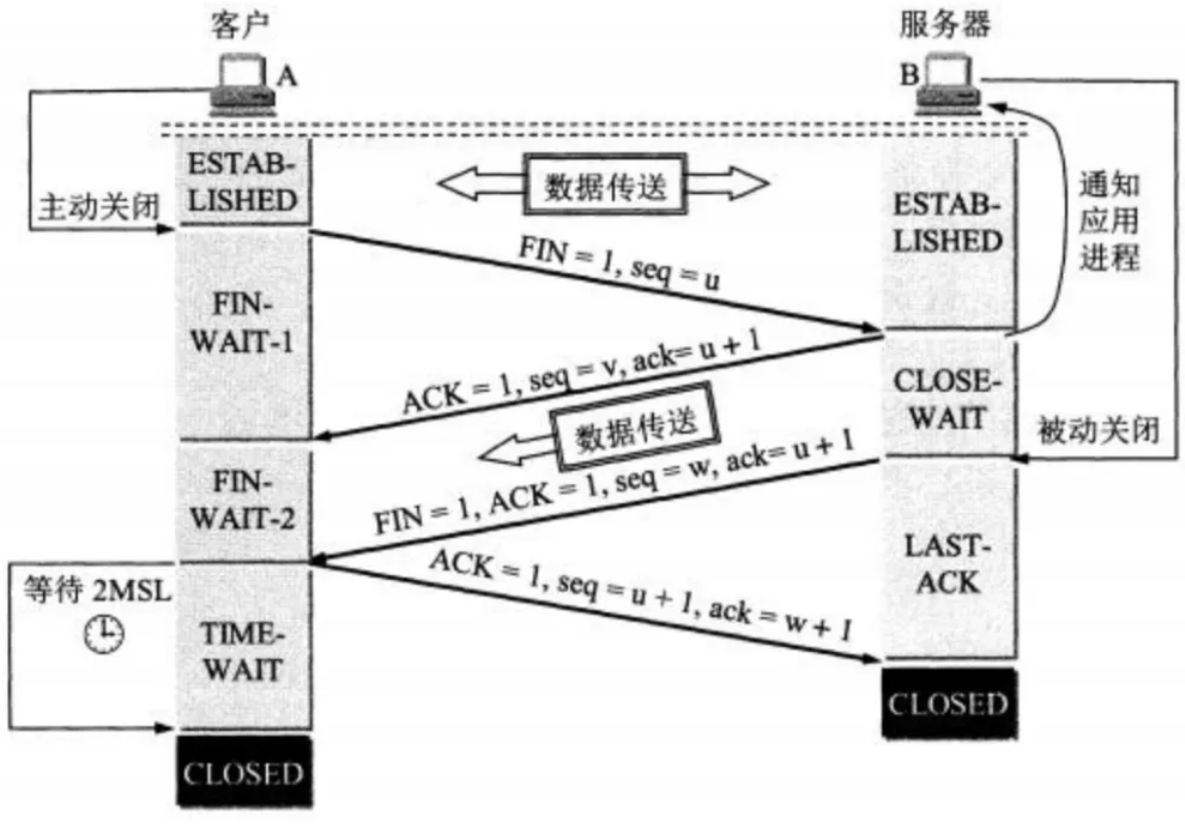
- 资源浪费，三次已经可以建立可靠连接

vii. 第一二三次丢包各会发生什么?

- 第一次：客户端发送的SYN丢失；客户端没有收到服务器的SYN/ACK包；客户端超时重传n次SYN包之后不再发送（RTO时间指数上涨）（tcp_syn_retries）
- 第二次：服务器的SYN/ACK包丢失；服务器没有收到客户端ACK包，客户端也没有收到服务器的SYN/ACK包；客户端客户端超时重传n次SYN包之后不再发送，服务器超时重传n次SYN/ACK包后不再发送（tcp_synack_retries）
- 第三次：客户端发送的ACK丢失；服务器没有收到客户端ACK包；服务器超时重传n次SYN/ACK包后不再发送断开连接；客户端认为建立了连接，如果发送数据包，始终超时，重发n次（tcp_retries）之后断开，如果不发送数据包，保活机制每隔一段时间发送探测报文，始终未响应则断开（需要2小时11分15秒）

g. 四次挥手

i. 过程?



- 第一次：A发送一个连接释放报文，终止控制位FIN = 1，确认位ACK = 1，序列号seq = x，进入FIN_WAIT_1状态
- 第二次：B发送确认报文段，确认位ACK = 1，ack = x + 1，进入CLOSE-WAIT状态，A收到后进入FIN_WAIT_2状态
- 第三次：B发送连接释放报文（处理完数据之后），终止控制位FIN = 1，确认位ACK = 1，seq = y，ack = x + 1，B进入LAST_ACK状态
- 第四次：A发送确认报文段，确认位ACK = 1，ack = y + 1，A进入TIME_WAIT状态，B收到后进入CLOSED状态
- FIN_WAIT1：等待确认
- CLOSE_WAIT：等待自己发送中断连接请求
- FIN_WAIT2：等待对方的连接中断请求
- LAST_ACK：等待确认
- TIME_WAIT：等待足够时间确认对方收到确认
- CLOSED：没有任何连接状态
- CLOSING状态：比较罕见；正常情况下，FIN->；<-ACK；<-FIN；但是FIN->；<-FIN，主动方没有收到ACK，双方都在关闭，就出现了CLOSING状态；两边都等收到ACK之后进入TIME_WAIT

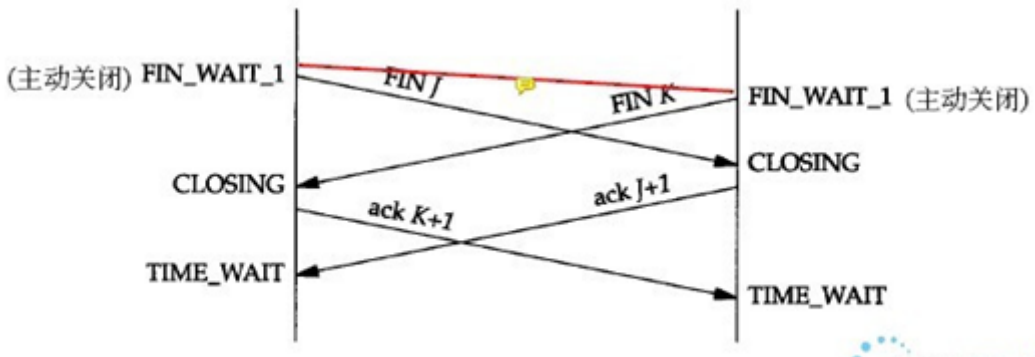


图18-19 同时关闭期间的报文段交换

ii. 为什么四次?

- 客户端发送FIN仅表示客户端不发送了但是还可以接收

- 服务端收到FIN后回一个ACK，但是服务端可能还有数据需要处理和发送，等服务端不再发才发送FIN

iii. TIME_WAIT为什么2MSL？/为什么要等待2MSL?

- 保证ACK能被被动连接方收到（如果被动方没有收到ACK，被动方则会重发FIN，每收到一次服务端的FIN就重启计时器）
- **使得旧连接的数据包得以消失，防止出现在之后的连接中**

iv. TIME_WAIT太多怎么办?

- 危害：客户端端口资源占用（端口就65536个，占满导致无法创建新的连接），服务端内存资源占用（线程池线程被占用，新来的连接没法被处理）
- 解决：
  - **修改内核参数**：端口复用（net.ipv4.tcp_tw_reuse 和 net.ipv4.tcp_tw_recycle，找一个TIME_WAIT状态超过1s的连接给新的连接复用）；
  - **客户端**：短连接变为长连接，减少了关闭次数
  - **服务端**：底层做一个**连接池**，扯字节需求；缩短TIME_WAIT时间
  - 强制关闭（**发送RST包**给对端跳过四次挥手，直接进入CLOSE状态，危险）
  - 使用SO_LINGER

```
struct linger so_linger;
so_linger.l_onoff = 1;
so_linger.l_linger = 0;
setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger,sizeof(so_linger));

l_onoff为非 0， 且l_linger值为 0，那么调用close后，会立该发送一个RST标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了TIME_WAIT状态，直接关闭。
```

- **如果服务端要避免过多的 TIME_WAIT 状态的连接，就永远不要主动断开连接，让客户端去断开，由分布在各处的客户端去承受 TIME_WAIT。**

v. SO_REUSEADDR选项?

- 让端口释放后立即就可以被再次使用，端口重用（正常Linux下需要2分钟）

vi. 使用HTTP的keep-allive选项可以避免出现大量TIME_WAIT

- 短连接：连接->传输数据->关闭连接
- 长连接：连接->传输数据->保持连接 -> 传输数据-> 。。。->关闭连接。

vii. TIME_WAIT是服务器还是客户端的状态?

- 主动断开连接一方进入的状态，通常都是客户端的状态，服务端一般不设置主动断开连接
- 服务器主动断开连接的情况：TIME_WAIT 需要等待 2MSL，在大量短连接的情况下，TIME_WAIT会太多，这也会消耗很多系统资源。对于服务器来说，在 HTTP 协议里指定 KeepAlive（浏览器重用一个 TCP 连接来处理多个 HTTP 请求），由浏览器来主动断开连接，可以一定程度上减少服务器TIME_WAIT太多，不再等2MSL，减少资源消耗

h. TCP如何保证的可靠性?

- TCP头部图
- 序列号/确认应答：根据序列号对数据进行排序，还可以去掉重复的数据；根据序列号可以知道收到了哪些数据，下次从哪里发送

- 校验和/数据包校验：检验接受的数据是否有异常和差错，如果出现差错，则丢弃TCP段，重发
- 超时重传
- 滑动窗口：解决了必须一发一答才能再发的问题提高了报文传输的效率
- 流量控制：B通过告诉A缓冲区的大小，控制A的发送量
- 拥塞控制：通过慢启动、拥塞避免、快速重传、快速恢复避免网络阻塞

i. TCP保活机制/保活计时器？（keeplive）

- 已经建立了连接，对端出现故障，发送探测报文，2小时11分15秒未回复则默认断开连接
- 只要收到对端数据，计时器就复位重新计时

j. TCP 和 HTTP keepalive

- TCP keepalive（保活机制 内核）：没有报文传输的过程中，通过探测报文确定对方是否还存货，否则不断开会一直保持连接
- HTTP keep-alive（长连接 应用程序）：避免三次握手四次挥手的消耗

```
1 Connection: Keep-Alive
2 Connection:close
```

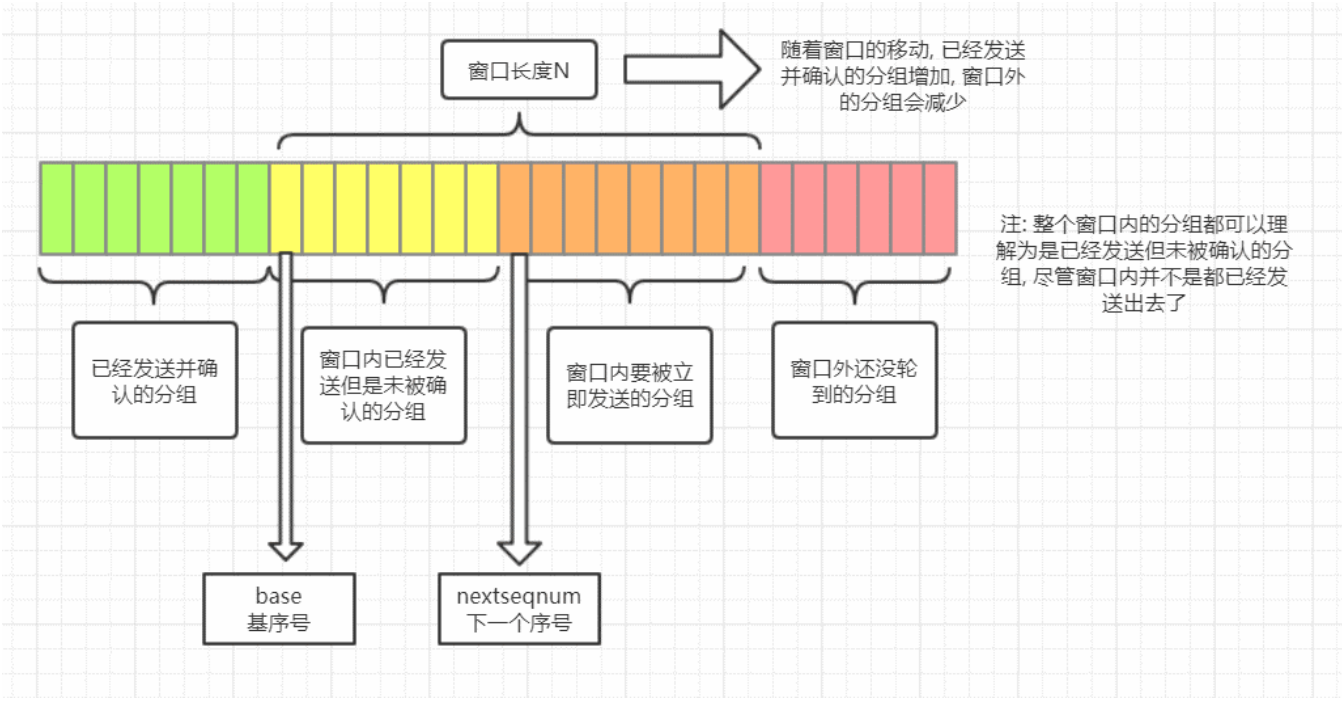
- 小林coding

k. 超时重传、快速重传？

l. 为什么快速重传选择3次ACK？（阿秀80）

m. 滑动窗口？

- 原因：TCP每次发送一个数据(MSS)都要进行一次应答(ACK)，只有上一个收到确认才发送下一个，效率低
- 窗口大小：无需等待确认应答，而可以继续发送数据的最大值
- TCP使用滑动窗口实现对流量控制的机制
- 发送方：
- 已发送并确认 <- 已发送未确认 <- 立即要发送的 <- 还没有进入窗口的（窗口 = 已发送未确认 + 立即要发送的）



- 接收方：

- 已接收并确认<-未收到收据但可以接收<-未收到收据但不能接收
- 窗口为0还可以发送吗？可以发送紧急数据（终止远程登录）、可以发送1字节的数据报来通知接收方希望接受的下一字节以及窗口大小

n. 流量控制？

- 原因：发送方一直发送，接收方处理不过来，发送方会重发，影响性能，所以提出了流量控制
- 流量控制：每次接收到数据后会将剩余可处理数据的大小告诉发送方，控制发送速率，保证来得及接受
- 问题1（小林）：没有被处理的数据存放在缓冲区，操作系统管理，系统繁忙会导致缓冲区减小，会导致丢包，怎么办？
- 解决1：所以TCP不允许同时减少缓存又收缩窗口（先收缩窗口，过段时间在减少缓存，避免了丢包）
- 上述问题的例子： 发送方接收方窗口大小各为200字节，发送方发送100字节的给接收方，此时双方各剩100字节，但是此时操作系统非常忙，将接收方的缓存区减少了50字节，这时接收方就会告诉发送方，我还有50字节可用，但是在接收方发送到达之前，发送方是不知道的，只会看到自己还有100字节可用，那么就继续发送数据，如果发送了80字节数据，那么接收方缓存区大小为50字节，就会丢失30字节的数据，也就是会发生丢包现象。
- 问题2（小林）：接收方窗口大小为0之后告诉发送方，之后窗口大小变大的报文丢失了，发送方一直认为0，进入死锁，怎么办？
- 解决2：零窗口通知之后启动持续计时器，超时就会主动发送窗口探测报文
- 问题3（小林）：糊涂窗口综合症，接收窗口已经很小了，但是发送方还是义无反顾的发送，TCPIP头部本来就大，只传输了几个字节，影响性能
- 解决3：接收方避免通关小窗口，发送方避免发送小数据（Nagle算法）

o. 拥塞控制（慢启动、拥塞避免、快速重传、快速恢复）？

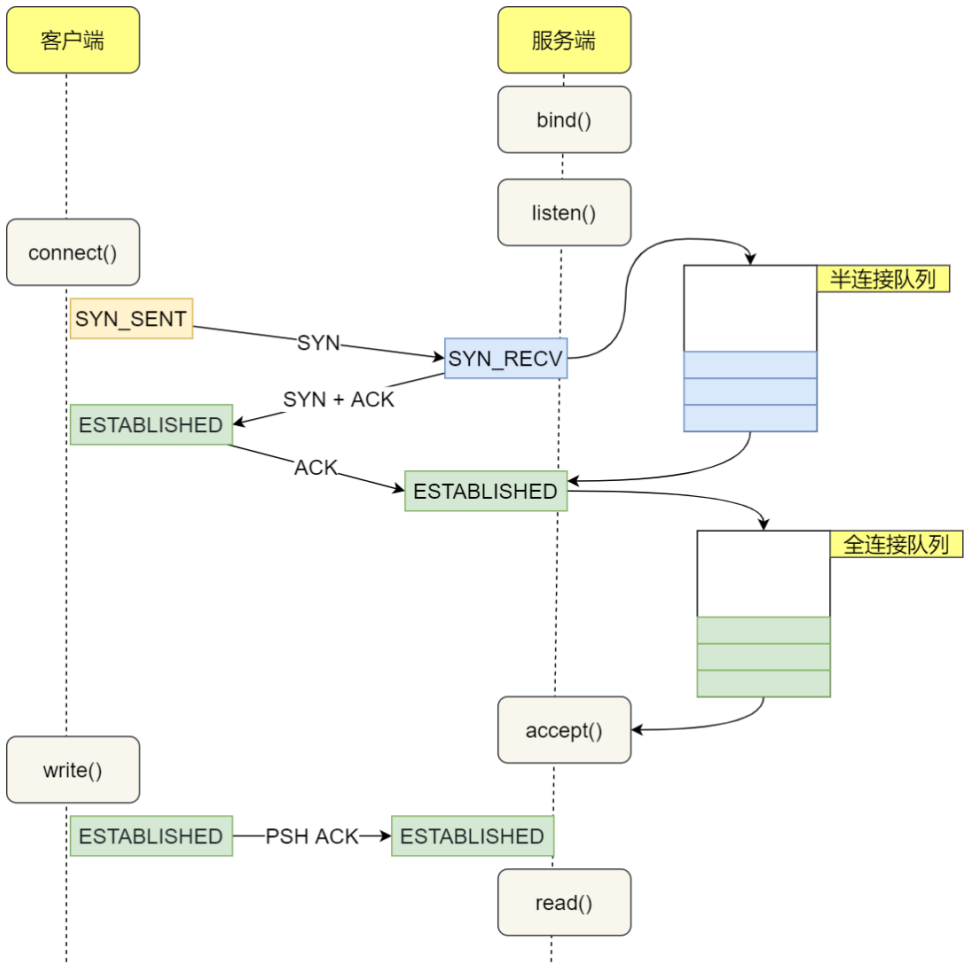
- 原因：网络出现拥堵时，继续发送数据会导致丢失，再重传，反而再次增大了网络负担，死循环
- 拥塞控制：避免发送方的数据填满整个网络
- swnd = min(cwnd, rwnd) ssthresh（65535字节）
- 慢启动：每收到一个ACK，cwnd += 1（发1、ACK1、+1，第二次发2、ACK2、+2，指数增长）
- 拥塞避免：cwnd = ssthresh时候，每收到一个ACK，cwnd += 1/cwnd（发8，ACK8，+1/8 * 8，线性增长）
- 超时重传：cwnd = 1，ssthresh /= 2，再进行慢启动拥塞避免的过程（已经废弃）
- 快速重传：收到三个相同的ack，快速重传丢失的报文，cwnd /= 2，ssthresh = cwnd
- 快速恢复：cnwd = ssthresh + 3（因为收到了三个重复的ACK）；重传丢失的数据包；如果重复的ack，cwnd += 1，如果新的ACK，cwnd = ssthresh，也就是进入了拥塞避免（快速重传 + 只降一半而不是直接重新慢启动）

p. **BBR算法**

- 对丢包不敏感的拥塞控制算法，只根据自己测量的即时rate来调整cwnd的大小，即BBR只负责发送多少数据包，至于发哪些数据包它不管，传统的拥塞控制算法，当碰到丢包时，网络便交给了快速恢复算法（比如PRR）接管，拥塞控制算法是不能插足的，直到重传结束返回到open状态

q. 半连接队列、全连接队列？

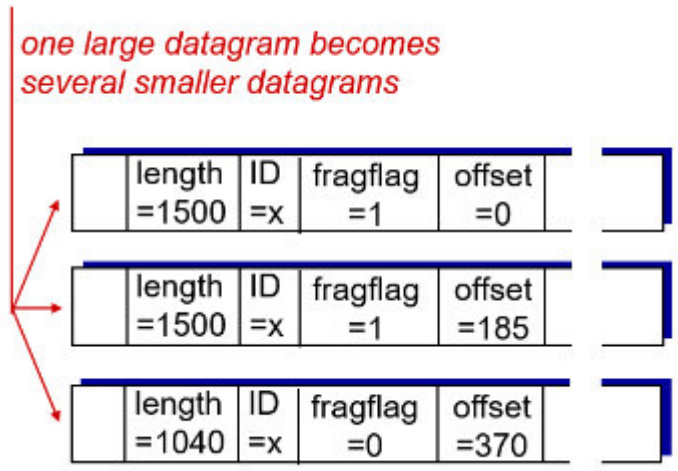
- 被动方收到SYN请求后放入半连接队列（SYN队列）
- 被动方收到ACK报文后放入全连接队列（accept队列）



- 两个队列都有大小限制，超过容量就丢弃连接或者返回RST包

r. 粘包、拆包？TCP如何连接多次传送的报文？

- 原因：TCP是基于字节流的，TCP把应用层传来的大小不等的数据块仅看成一连串无结构的字节流，没有边界；TCP首部也没有表示数据长度的字段
- 粘包：①发送端的几个数据包小于TCP缓冲区大小；**Nagle**算法（没有收到确认前一直在缓冲区进行堆压，会合并）；②接收端 放数据速度 > 应用层拿数据速度；
- 拆包：发送端的一个数据包大于TCP缓冲区大小；大于MSS大小；大于MTU大小
- 问题：接收端处理困难，无法区分一个完整的数据包
- 解决：
  - 在**首部添加**数据包的**长度**（发送端给每个数据包添加首部，首部中包含数据包的长度，接收端可以知道每个数据包的实际长度）；
  - **设置固定长度**（发送端将每个数据包设置固定长度，接收端读取固定长度数据把数据包分开）；
  - **增加特殊字符边界**（数据包之间增加边界，回车、空格等特殊字符，接收端可以通关这个特殊字符区分）；
  - 自定义消息结构：由包头（数据长度）和数据组成
  - netty解决粘包拆包问题
- tips：UDP没有拆包粘包，收到的全是完全正确的包，有丢包乱序
- UDP拆开是因为IP分片，IP分片通过ID()、fragflag(是不是最后一个数据报)、offset(偏移)进行组合

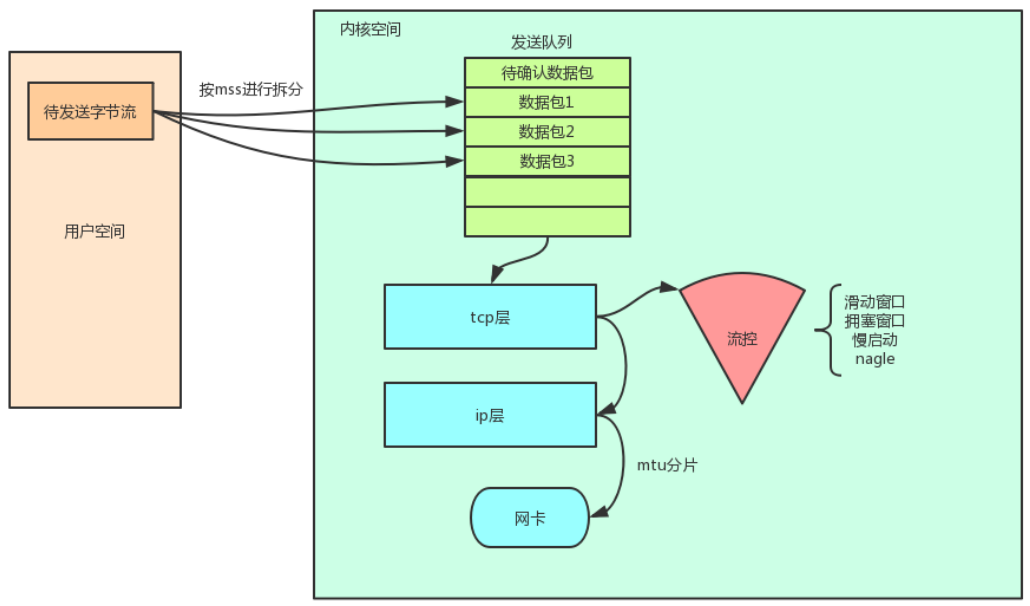


s. 一些名词：MTU、MSS、RTO、RTT？

- Maximum Transmit Unit，最大传输单元：IP头部 + TCP头部 + 数据
- Maximum Segment Size，最大报文段长度：数据
- Retransmission Time Out，重传超时时间
- Round Trip Time，连接的往返时间

t. 为什么会有MSS，没有MSS怎么办？

- 力求在TCP分片，如果在IP分片的话，只有第一个包里有TCP头部，如果丢失其中一个，需要重传整个TCP数据



u. TCP为什么要分包？

- 同上：数据在TCP分段，就是为了在IP层不需要分片，同时发生重传的时候只重传分段后的小份数据

v. TCP连接失败的原因

- 防火墙隔离
- 网络故障，可以通过ping进行调试
- client分配不到端口
- server端连接队列已满

w. 如何解决UDP乱序问题？

- 添加seq/ack机制，确保数据发送到对端
- 添加发送和接收缓冲区，主要是用户超时重传。
- 添加超时重传机制。

x. **服务端挂了，客户端链接还在吗？**
    - 客户端还发送数据：
        - 报文辉超时重传，重传总间隔时长超过tcp_retries2，断开链接
    - 客户端不发送数据：
        - 开启了keepalive机制，会探测是否存在
        - 没开启的话，TCP一直处于ESTABLISHED
y. 如何优化三次握手、四次挥手（小林）？
z. Socket见网络编程

网络层 IP

- Conception：
    a. ABCDE五类地址
    b. 单播、多播、广播
    c. 子网掩码
    d. ARP、RARP、DHCP
    e. PING、ICMP
    f. NAT
- Question：
    a. IP地址分类？
        - A：开头0，网络号8位
        - B：开头10，网络号16位
        - C：开头110，网络号24位
        - D：开头1110，用作多播
        - E：开头1111，保留
    b. 单播、多播、广播？
        - 单播：一对一通讯
        - 广播：同一网络中的主机
        - 多播：跨越不同网络，传送给属于多播组的多个主机
    c. **PING工作原理？**
        - 基于ICMP协议工作

ICMP 类型		
	内容	种类
0	回送应答（Echo Reply）	查询报文类型
3	目标不可达（Destination Unreachable）	差错报文类型
4	原点抑制（Source Quench）	差错报文类型
5	重定向或改变路由（Redirect）	差错报文类型
8	回送请求（Echo Request）	查询报文类型
11	超时（Time Exceeded）	差错报文类型

- 8和0是请求响应消息类型
- 在规定时间内
  - 如果没有收到ICMP应答包，说明目标主机不可达
  - 如果收到的话，会计算时间延迟
- 拓展：
  - traceroute，
  - 作用一：获取IP地址；按从1开始的顺序递增TTL发送请求，每次都在一个路由器失效，返回路由器IP，所有过程加起来可以获得所有路由器的IP
  - 作用二：故意不设置分片，确定MTU；将 IP 包首部的分片禁止标志位设置为 1，途中的路由器不会对大数据包进行分片，而是将包丢弃，通过一个 ICMP 的不可达消息将数据链路上 MTU 的值一起给发送主机，发送主机端每次收到 ICMP 差错报文时就减少包的大小，以此来定位一个合适的 MTU 值，以便能到达目标主机。

d. Ping的作用?

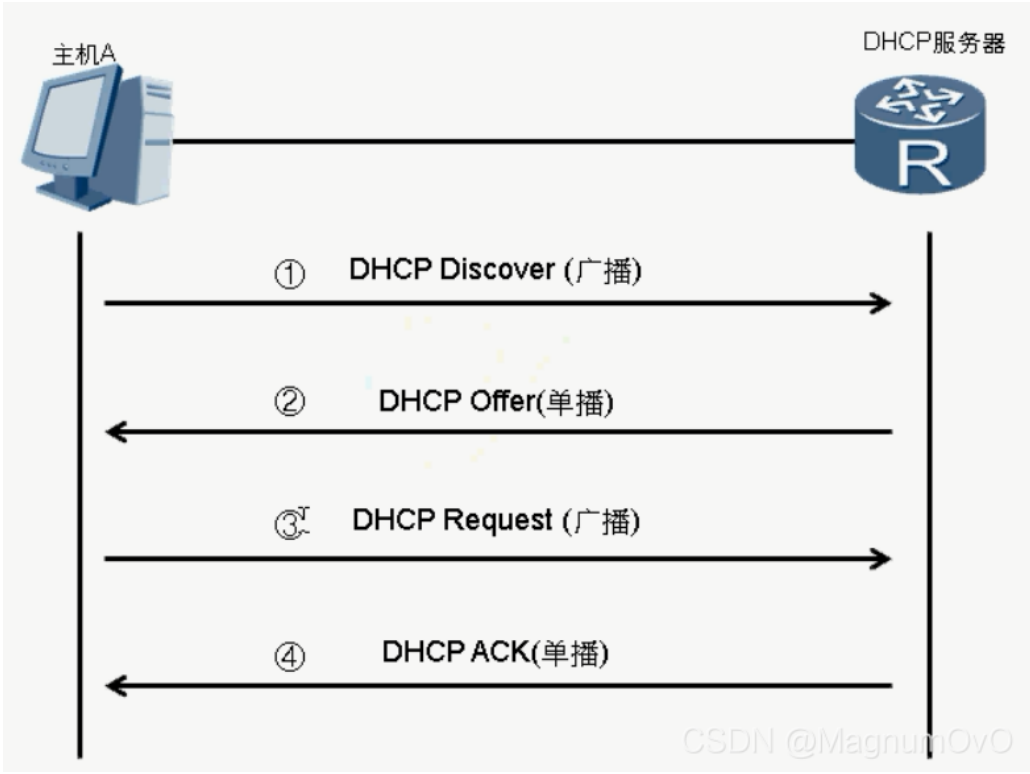
- 测试在两台主机之间能否建立连接，向目的主机发送多个ICMP回送请求报文，没有响应则无法建立链接，有响应可以根据目的主机返回的回送报文的时间和响应的次数估算出数据包往返时间及丢包率

e. **ARP协议的工作原理?**

- 完成了IP地址和MAC地址的转换
- 在ARP缓冲区建立ARP列表，表示IP和MAC的对应关系，如果找到直接返回；如果没找到向本地网段发送一个ARP请求的广播包，所有主机收到这个请求，不相同就忽略，相同该主机先将发送端的IP和MAC地址添加/更新，然后告诉请求方；请求方收到后也进行更新；如果一直没有收到则说明ARP查询失败

f. 局域网内的IP分配策略？是怎么实现的?

- 之前是管理员进行分配
- 现在是**DHCP（动态主机配置协议）**



- DHCP Discover：DHCP客户机向局域网中所有DHCP服务器发送DHCPdiscovery请求。（DHCP客户机向DHCP服务器发动DHCP请求，来索要ip）
- DHCP Offer：局域网中所有的DHCP服务器都会回复DHCPoffer，为客户机提供IP地址。
- DHCP Request：客户机选择第一台DHCP服务器回复的IP地址，并且要发送DHCPrequest通告给局域网内所有的DHCP服务器，他选择了哪个ip和那个DHCP服务器。（通告给所有DHCP服务器，让其他没有被选中的DHCP服务器把未使用的地址进行回收；通告被选中的DHCP服务器，确认此地址被DHCP客户机使用。）

- ABC类地址、子网掩码

g. 跨域问题？

- 定义：浏览器从请求另一个域名的资源时，域名、端口、协议任一不同，都是跨域
- localhost和127.0.0.1虽然都指向本机，但也属于跨域
- 跨域限制：
  - 无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB
  - 无法接触非同源网页的 DOM（Document Object Model、文档对象模型）
  - 无法向非同源地址发送 AJAX 请求（可以发送，但浏览器会拒绝接受响应）
- 解决：
  - JSONP（JSON with Padding：填充式JSON)
  - nginx反向代理
  - PHP端修改header
- 讲解

h. **为什么要有IP地址，为什么要有MAC地址？**

- **只有MAC，没有IP**：路由器需要记住每个MAC地址所在的子网是哪一个，世界上有2^48次方，每个一位，还需要256TB存储
- **只有IP，没有MAC**：IP地址本质上是终点地址，它在跳过路由器（hop）的时候不会改变，而MAC地址则是下一跳的地址，每跳过一次路由器都会改变。
  - IP的
- IP地址本质上是终点地址，它在跳过路由器（hop）的时候不会改变，而MAC地址则是下一跳的地址，每跳过一次路由器都会改变。

i. RARP工作原理?

j. NAT（Network Address Translation）

- 将私有地址转换为共有地址，IPV4公网地址用一个少一个
  - 静态NAT：一对一绑定
  - 动态NAT：用的时候从地址池选一个，用完返回
  - 端口复用：用一个地址的多个port对应多个IP，可以使**多个内部本地地址**同时与**同一个内部全局地址**进行转换
- 讲解

```
1 iptables -t nat -L -n
```

数据链路层

- Conception：

a. 停止等待协议

b. ARQ协议

- Question：

a. 停止等待协议?

- 最简单但也是最基础的数据链路层协议；停止等待就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组。

b. 对ARQ协议的理解?

- Automatic Repeat-reQuest ，自动重传请求协议：超过一段时间没有收到确认，就重发前面发送过的分组
- 连续ARQ协议：缺失后发送该帧和后面的，接收方去重
- 选择ARQ协议：缺失后只发送该帧而不发送后面的，接受方重排

安全性问题

- Conception：

a. SYN洪泛攻击

b. DDos攻击（Distributed Denial of Service）

c. XSS攻击（cross-site scripting）

d. SQL注入

e. CSRF攻击

- Question：

a. SYN洪泛攻击?

- TCP协议缺陷：第二次握手发送SYN/ACK包之后，收到第三次握手ACK包之前状态为半连接，未收到ACK则会不断重发请求
- 属于DDos攻击的一种，伪造大量不存在的IP地址，攻击者向服务器发送SYN请求，服务器发出SYN- ACK之后等待客户端ACK，不断重发直至超时，这些未收到ACK的连接会占用半连接队列，导致网络阻塞

- 检测：在服务器上看到大量半连接状态，而且IP地址是随机的，基本可以断定是SYN攻击
- 防范：防火墙、路由器等过滤网关回复；TCP/IP改进，增加最大半连接数，缩短超时时间；SYN cookie技术，专门防范SYN洪泛攻击

b. DDos攻击？
- Distributed Denial of Service，分布式拒绝攻击
- 第三次握手客户端不向服务端发送ACK，服务端一直等待ACK
- 预防：限制同时打开的半连接数目、缩短半连接的超时时间

c. Xss攻击？（帅地7补充）
- cross-site scripting 跨站脚本
- 原因：过于信任客户端提交的数据，攻击者在数据中插入js代码
- 分类：
- 攻击者篡改网页，嵌入脚本，控制用户浏览器进行恶意攻击
- 预防：前端：对输入数据进行过滤；后端

d. SQL注入？
- 在用户输入的字符串中加入SQL语句，攻击者可以执行计划外的命令或访问未被授权的语句
- 原理：恶意拼接查询；利用注释执行非法命令；传入非法参数；添加额外条件
- 例子：

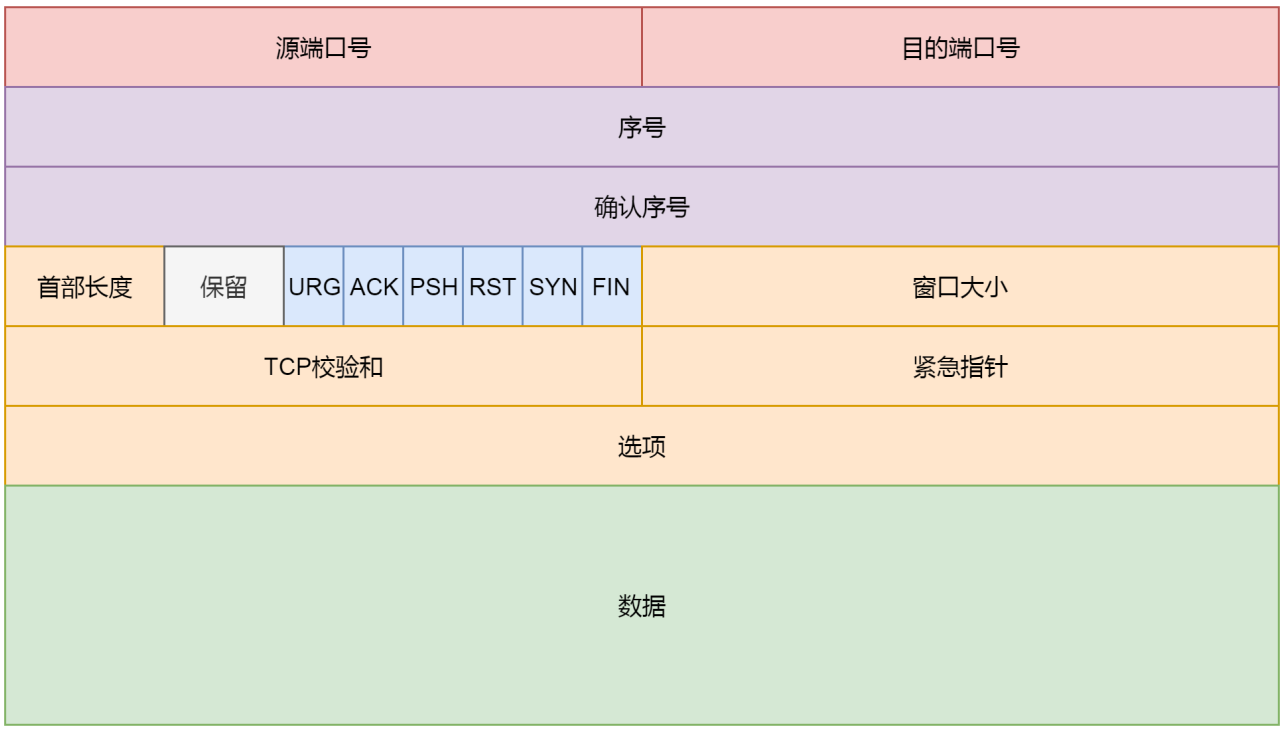
```
1 用户名：‘or 1 = 1 --
2 密 码:
3
4 String sql = “select * from user_table where username=’ “+userName+” ’ and
5 “+password+” ‘”;
6
7 SELECT * FROM user_table WHERE username=’’or 1 = 1 -- and password=’’
8 #username=‘ or 1=1 这个语句一定会成功,--表示注释
```

- 避免方法：提供参数化查询接口；避免原生SQL；限制用户的数据库权限；正则表达式过滤参数

e. CSRF攻击？

总结性

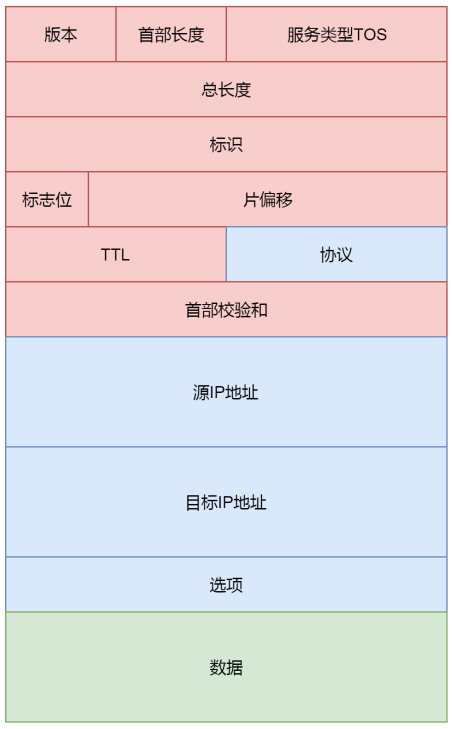
- Conception：
  a. 各种头部
  b. URI、URL
- Question：
  a. TCP头部



- SYN：建立连接
- ACK：确认号合法
- FIN：断开释放连接
- RST：发送错误用于复位
- URG：紧急指针
- PSH：PUSH标志的数据位，不必等待缓冲区满时才发送

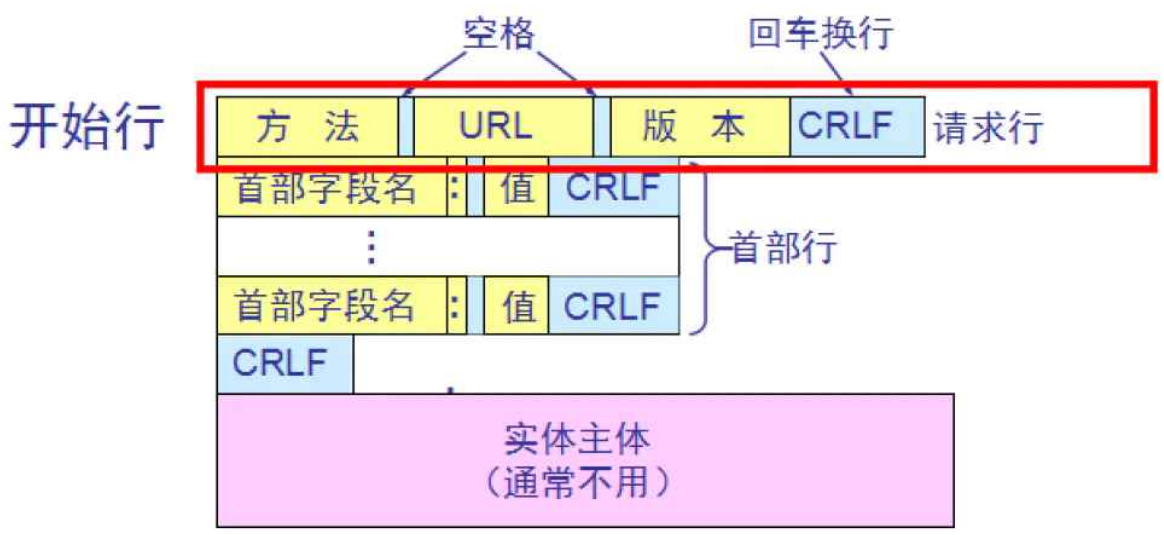
b. UDP头部

c. IP头部

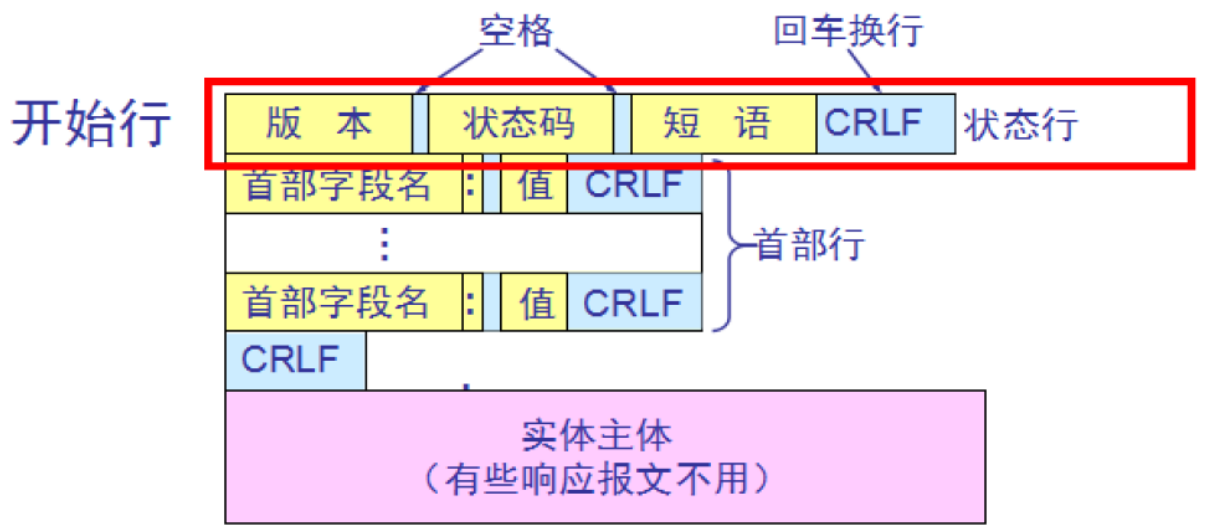


a. HTTP头部

- 头部和请求体之间用空行隔开



- HTTP/1.1 200 OK



1. URI、URL
    - URI：Uniform Resource Identifier，统一资源标识符；（唯一表示资源）
    - URL：Uniform Resource Locator，统一资源定位符；（使用地址来标识资源）
        - schema://host:port/path
    - URN：Uniform Resource Name，统一资源名称；（使用名称来标识资源）
    - URL + URN = URI
    - URL 是 URI 的一个子集

遗留问题

- 服务器缓存？（阿秀9）
- forward和rediret 帅地30

- 停止等待协议