计算机基础

计算机网络

(有些同学可能会疑惑, 为什么没有FTP, SMTP, IP, ICMP等协议的内容, 因为这些面试基本不考, 你看了纯纯浪费时间!)

网络基础

OSI 和 TCP/IP 网络分层模型 (高)

OSI: 应用层, 会话层, 表示层, 传输层, 网络层, 数据链路层, 物理层

TCP/IP(4层/5层):

- 应用层: 通过应用进程交互实现一个特定的网络应用
- 传输层: 为网络中的主机提供端到端的数据传输服务(进程之间网络通信)
- 网络层: 在多个网络中, 分组如何路由和寻址
- 数据链路层: 在单个网络的一段链路上如何传输数据帧, 以及帧的差错控制
- 物理层: 在传输介质上用什么信号传输比特流的问题

数据链路层和物理层又被统称为网络接口层.

应用层有哪些常见的协议 (高)

- HTTP 协议（超文本传输协议，网页浏览常用的协议）
- HTTPS协议 (HTTP的安全版)

- DHCP 协议（动态主机配置）
- DNS 系统原理（域名系统）
- FTP 协议（文件传输协议）
- SMTP 协议 (电子邮件协议)

传输层常见协议 (高)

- TCP
- UDP

网络层协议 (中)

- ARP
- IP
- ICMP 协议 (差错控制)

为何要分层 (高)

- 各层相互独立, 只需要关心本层的问题，各层之间不需要关心其他层是如何实现的. 类似于做系统也需要分层
- 提高了整体的灵活性. 类似于高内聚低耦合
- 大问题分解化小. 类似于做系统进行功能分解

TCP和UDP

TCP和UDP的特点, 不同之处? (高)

- TCP是面向连接的(传输数据需要建立连接), UDP是面向无连接的(传输数据不需要建立连接)
- TCP是可靠传输, 传输的数据有序可靠不丢不重. UDP是不可靠传输, 只能是最大努力交付.
- UDP实时性更好, 适合高速传输或者对实时性要求比较高的场景. TCP适合对可靠性要求较高的场景, 如文件传输.
- TCP传输效率比UDP低, 因为TCP传输时需要连接确认重传等机制保证可靠传输. UDP传输效率更高

使用 TCP 的协议有哪些?使用 UDP 的协议有哪些? (低)

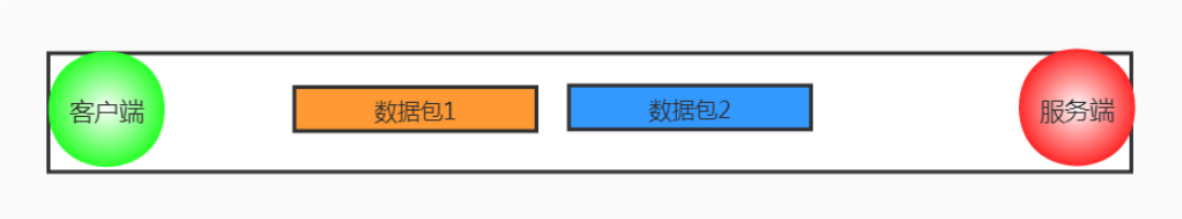
基于TCP: HTTP, HTTPS, FTP, SMTP, SSH(远程登录会话协议)

基于UDP: DNS, DHCP(动态配置 IP 地址)

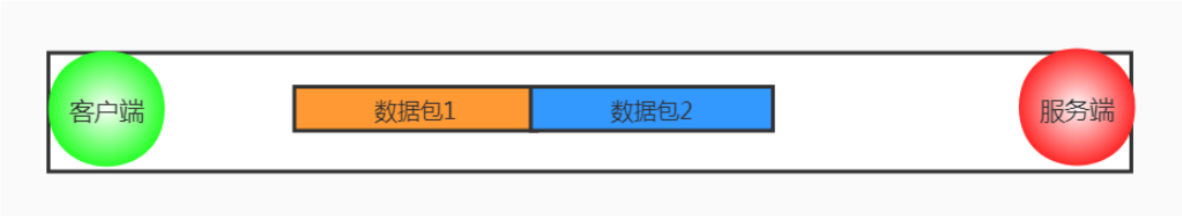
说一下 TCP 粘包是怎么产生的？怎么解决粘包问题的 (中)

TCP 传输数据基于字节流，从应用层到 TCP 传输层的多个数 据包是一连串的字节流是没有边界的，而且 TCP 首部并没有记录数据包的长度，所以 TCP 传输数 据的时候可能会发送粘包和拆包的问题；而 UDP 是基于数据报传输数据的，UDP 首部也记录了数 据报的长度，可以轻易的区分出不同的数据包的边界

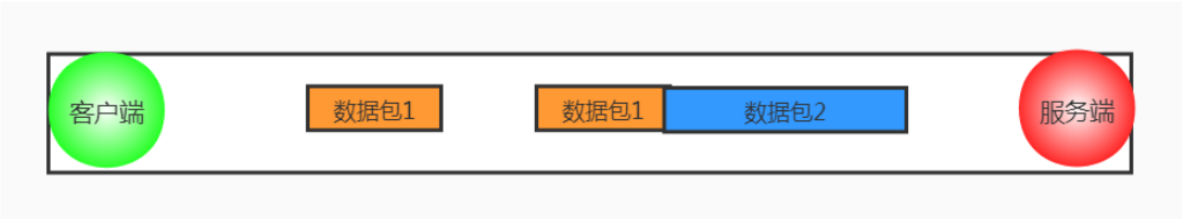
接下来看下 TCP 传输数据的几种情况，首先第一种情况是正常的，既没有发送粘包也没有发生拆包。

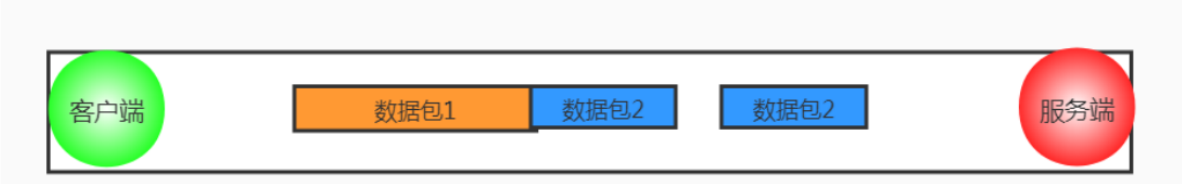


第二种情况发生了明显的粘包现象，这种情况对于数据接收方来说很难处理。



接下来的两种情况发生了粘包和拆包的现象，接收端收到的数据包要么是不完整的要么是多出来一块儿。





造成粘包和拆包现象的原因

- TCP 发送缓冲区剩余空间不足以发送一个完整的数据包，将发生拆包
- 要发送的数据超过了最大报文长度的限制，TCP 传输数据时进行拆包
- 要发送的数据包小于 TCP 发送缓冲区剩余空间, TCP 将多个数据包写满发送缓冲区一次发送出去，将发生粘包
- 接收端没有及时读取 TCP 发送缓冲区中的数据包，将会发生粘包

粘包拆包的解决方法

- 发送端给数据包添加首部，首部中添加数据包的长度属性，这样接收端通过首部中的长度字段 就可以知道数据包的实际长度啦
- 针对发送的数据包小于缓冲区大小的情况，发送端可以将不同的数据包规定成同样的长度，不足这个长度的补充 0，接收端从缓冲区读取固定的长度数据这样就可以区分不同的数据包
- 发送端通过给不同的数据包添加间隔符合确定边界，接收端通过这个间隔符合就可以区分不同的数据包。

TCP 如何保证可靠性（中）

- **校验和:** TCP报文头有检验和字段, 可以用于校验报文是否损坏
- **序列号与确认号:** TCP 发送端发送数据包的时候会选择一个 seq 序列号，接收端收到数据包后会检测数据包的完整性与顺序性, 可以根据序列号进行排序去重，如果检测通过会响应一个 ack 确认号表示收到了数据包
- **确认与超时重传机制:** 接收方收到报文后就会返回一个确认报文, 发送方发送一段时间后没收到确认报文就会进行超时重传.
    - 快速重传: 接收方收到比期望序号大的报文段到达时, 就会发送一个冗余ACK, 指明下一个期待字节的序号，发送端接收到3个以上的重复ACK, TCP就意识到数据发生丢失，需要重传
- **流量控制:** 当接收方来不及处理发送方的数据，能通过滑动窗口，提示发送方降低发送的速率，防止包丢失
- **拥塞控制:** 当网络拥塞时，通过拥塞窗口，减少数据的发送，防止包丢失。

TCP流量控制 (中)

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的**窗口字段**可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据

TCP拥塞控制 (中)

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。

拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。

相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做到的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP 发送方要维持一个 **拥塞窗口(cwnd)** 的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

慢开始算法 – Slow Start

所谓慢开始/慢启动，也就是TCP连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了网络通道的秩序。

慢启动算法：

1. 连接建好的开始先初始化拥塞窗口cwnd大小为1，表明可以传一个MSS(最大报文段长度)大小的数据。
2. 每当收到ACK确认报文(每经过一个往返时延RTT), cwnd大小直接乘以2, 呈**指数让升**, 也就是一次发送2^n个报文
3. 还有一个ssthresh, 是一个门限值(默认为16), 当cwnd >= ssthresh时，就会进入“拥塞避免算法”

拥塞避免算法 – Congestion Avoidance

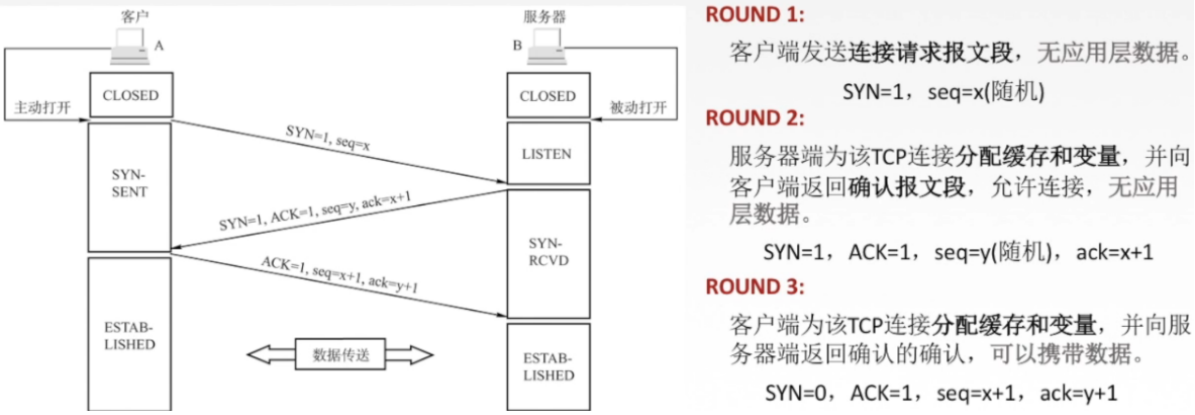
如同前边说的，当拥塞窗口大小cwnd大于等于慢启动阈值ssthresh后，就进入拥塞避免算法。算法如下：

1. 让拥塞窗口 cwnd 缓慢增大，即每经过一个往返时间 RTT 就把发送放的 cwnd 加 1. **加法增大**
2. 过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络的最佳值。

快重传与快恢复

1. cwnd大小缩小为当前的一半
2. ssthresh设置为缩小后的cwnd大小
3. 然后进入拥塞避免算法 (加法增大)

三次握手 (高)



建立一个 TCP 连接需要“三次握手”，缺一不可：

- **一次握手**:客户端发送带有 SYN（SEQ=x） 标志的数据包 -> 服务端，然后客户端进入 **SYN_SEND** 状态，等待服务器的确认；
- **二次握手**:服务端发送带有 SYN+ACK(SEQ=y,ACK=x+1) 标志的数据包 –> 客户端,然后服务端进入 **SYN_RECV** 状态
- **三次握手**:客户端发送带有带有 ACK(ACK=y+1) 标志的数据包 –> 服务端，然后客户端和服务器端都进入**ESTABLISHED** 状态，完成TCP三次握手

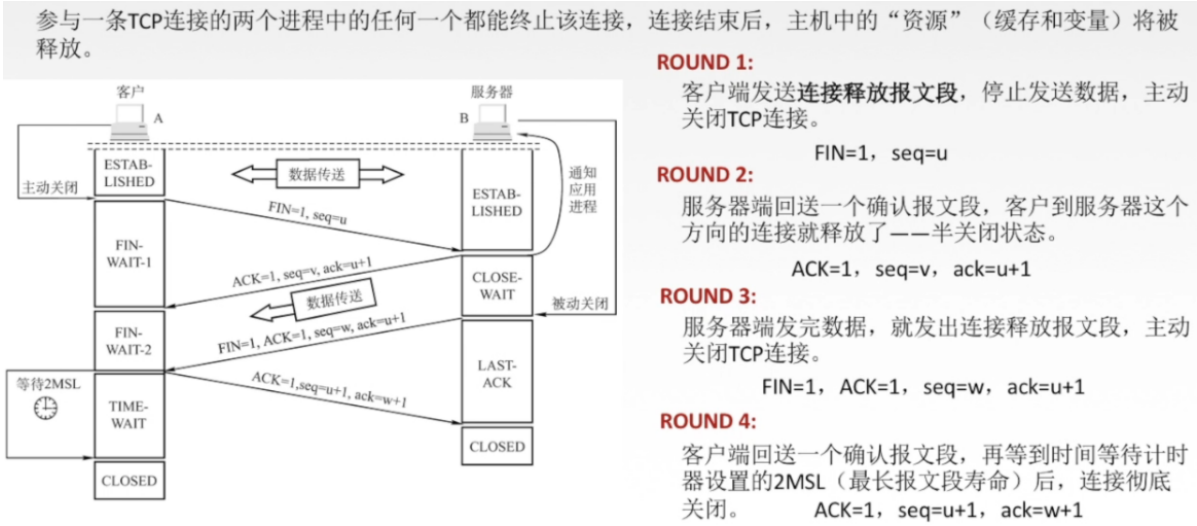
为什么一定要三次握手, 两次不行吗 (高)

三次握手的目标是要确认客户端和服务端双方的收发能力没有问题, 都能发送接收数据. 就好比两个人交流前需要确认, 双方都不是聋子, 都不是哑巴. 如果有一方聋或者哑那就无法交流.

- 第一次握手: 客服端发消息给服务端.
    - 服务端收到消息, 此时服务端确认了自己接收能力没问题, 客户端的发送能力没问题.
- 第二次握手: 服务端发给客户端
    - 客户端收到消息. 此时客户端能收到消息, 说明自己之前发消息一定发出去了, 说明客户端的发送能力没问题.
    - 而现在收到了服务端发的数据, 那么说明服务端一定收到了数据. 故服务端的接收能力没问题.
    - 那此时收到了服务端发送的数据, 说明服务端的发送能没问题, 自己(客户端)的接收能没问题.
    - 此时, 客户端确认了双发的收发能力都没问题.
    - 这也是为什么第三次握手的时间, 客户端能够携带数据.
- 第三次握手: 客户端发消息给服务端. (客户端可以携带数据)

- 为什么需要第三次握手? 因为第二次握手完成后, 客户端确认了双发的收发能力都没问题. 但是服务端只知道自己能收, 对方能发, 无法确认双方的收发能力都没问题.
- 当第三次握手完毕, 服务端才能确双发的收发能力都没问题.

四次挥手 (高)



为什么要四次挥手?

举个例子：A 和 B 打电话，通话即将结束后。

1. **第一次挥手**： A 说“我没啥要说的了”
2. **第二次挥手**： B 回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话
3. **第三次挥手**： 于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”
4. **第四次挥手**： A 回答“知道了”，这样通话才算结束

为什么不能把服务器发送的 ACK 和 FIN 合并起来，变成三次挥手?

因为服务器收到客户端断开连接的请求时，可能还有一些数据没有发完，这时先回复 ACK，表示接收到了断开连接的请求。等到数据发完之后再发 FIN，断开服务器到客户端的数据传送

如果第二次挥手时服务器的 ACK 没有送达客户端，会怎样?

客户端没有收到 ACK 确认，会重新发送 FIN 请求。

为什么要等待 2MSL?

因为如果第四次挥手客户端发送的确定报文丢失, 服务器就收不到确认报文, 会触发超时重传机制, 服务器会重传第三次挥手的连接释放报文段. 若等待2MSL之后, 就能保证服务端收到了确认报文段, 关闭了连接.

为什么是两倍的最长报文寿命?

因为客户端第三次挥手发送确认报文段, 该报文段到服务器端最长时间是1MSL. 若丢失, 服务端在1MSL后超时重传第三次挥手的连接释放报文段, 这个报文段到客户端的最大时间又是1MSL. 所以要等待2MSL, 没有收到超时重传的连接释放报文段, 就说明确认报文段服务端收到了, 连接正确释放了.

HTTP

在浏览器中输入url地址后显示主页的过程? (中)

1. 根据域名，进行DNS域名解析；
2. 拿到解析的IP地址，建立TCP连接；
3. 向IP地址，发送HTTP请求；
4. 服务器处理请求, 返回响应结果；
5. 关闭TCP连接；
6. 浏览器解析HTML；
7. 浏览器布局渲染；

什么是跨域, 如何解决跨域 (中)

- 跨域来源于浏览器的同源策略, 浏览器要检查协议、域名、端口号是否一致, 不一致就会跨域.
- 前端一般可以用JSONP或者代理服务器解决
- 后端一般可以用设置请求头来解决, Springboot项目中添加一个 `@CrossOrigin` 注解即可

HTTP1.0和HTTP1.1 (中)

长连接

- HTTP1.0是短连接: 浏览器每次请求都需要和服务器建立一个TCP连接, 一次请求, 一次响应后, 就会断开连接
- HTTP1.1是长连接: 在一个TCP连接上可以传送多个HTTP请求和响应, 这就减少了多次建立和关闭连接的延迟
  - 无流水的长连接: 客户端收到上一个请求的响应后才能发送下一个请求
  - 有流水的长连接: 客户端不用等待上一个请求的响应也能发送下一个请求

缓存

- HTTP1.1引入了更多可供选择的缓存头来控制缓存策略

请求头

- HTTP 1.0：请求头相对简单，功能有限。它支持的请求方法较少，常见的有 GET、POST 和 HEAD 等。
- HTTP 1.1：新增了一些请求方法，如 PUT、DELETE、OPTIONS、TRACE 等

带宽优化及传输

- HTTP 1.0：不支持请求头压缩和分块传输。当传输大文件时，需要等待整个文件内容全部传输完成后才能进行处理。
- HTTP 1.1：支持分块传输（Chunked Transfer Encoding）。服务器可以将响应数据分成多个块进行传输，每个块都有一个表示自身长度的头部，客户端可以在接收部分数据后就开始处理，而不必等待整个响应完成。同时，HTTP 1.1 还支持请求头压缩，减少了传输的数据量

HTTP的长连接是什么？（中）

HTTP协议采用的是请求-应答的模式，也就是客户端发起了请求，服务端才会返回响应，一来一回

短连接: 一次连接只能请求一次资源

- HTTP是基于TCP传输协议实现的，客户端与服务端要进行HTTP通信前，需要先建立TCP连接
- 然后客户端发送HTTP请求，服务端收到后就返回响应，至此请求-应答的模式就完成了，随后就会释放TCP连接
- 这就是短连接

长连接: 一次TCP连接可以发送多个请求, 多次应答

- HTTP长连接的特点是，只要任意一端没有明确提出断开连接，则保持TCP连接状态, 可以多次发送请求.

HTTP/2.0和HTTP1.1有什么区别 (低)

传输格式变化，采用了新的二进制格式

- HTTP1.X 的解析都是基于文本，文本的表现形式多样，不利于健壮性考虑

- HTTP2.0 采用二进制，只认0/1组合，实现更加快的方法，健壮性更加完善

多路复用 连接共享

- HTTP 1.1：虽然支持持久连接，但同一时间一个 TCP 连接只能处理一个请求，后续请求需要等待前面的请求处理完成才能发送和处理。为了提高并发性能，浏览器通常会同时打开多个 TCP 连接来处理多个请求，但这会增加服务器的负担和网络拥塞的风险
- HTTP 2.0：引入了多路复用机制，允许在一个 TCP 连接上同时并行发送多个请求和响应，并且这些请求和响应可以交错进行，互不干扰。每个请求和响应都有一个唯一的流标识符，通过流标识符可以将不同的帧正确地组装成完整的请求和响应。多路复用大大提高了连接的利用率和传输效率，减少了延迟

头部压缩

- HTTP 1.1：请求和响应的头部通常包含大量的重复信息，如`User-Agent`、`Cookie`等，而且每次请求都需要重复发送这些头部信息，导致了不必要的带宽浪费
- HTTP 2.0：使用压缩算法对头部信息进行压缩。从而显著减少了头部信息的传输量，降低了带宽消耗和延迟

HTTP为什么不安全？（高）

HTTP由于是明文传输，所以安全上存在以下三个风险：

- 窃听风险，比如通信链路上可以获取通信内容
- 篡改风险，比如强制植入垃圾广告
- 冒充风险，比如冒充京东淘宝等网站

HTTP和HTTPS区别 (高)

- HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题
- HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
- HTTP 连接建立相对简单， TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后， 还需进行 SSL/TLS 的握手过程，才可进入加密报文传输
- 两者的默认端口不一样，HTTP 默认端口号是 80，HTTPS 默认端口号是 443。
- HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的

SSL/TLS怎么确保安全通信, HTTPS怎么保证安全? (高)

SSL和TLS协议通过以下方式确保安全通信

- 加密： 使用加密算法对传输的数据进行加密，防止第三方截取和读取敏感信息
- 身份验证： 使用数字证书验证通信双方的身份，确保数据传输的可信性
- 数据完整性： 通过使用消息摘要算法，确保传输的数据在传输过程中没有被篡改或损坏

什么是对称加密和非对称加密 (中)

对称加密也称为私钥加密，使用相同的密钥来进行加密和解密

非对称加密: 使用一对不同但相关的密钥：公钥和私钥。 公钥用于加密数据，私钥用于解密数据

HTTPS是如何建立连接的 (高)

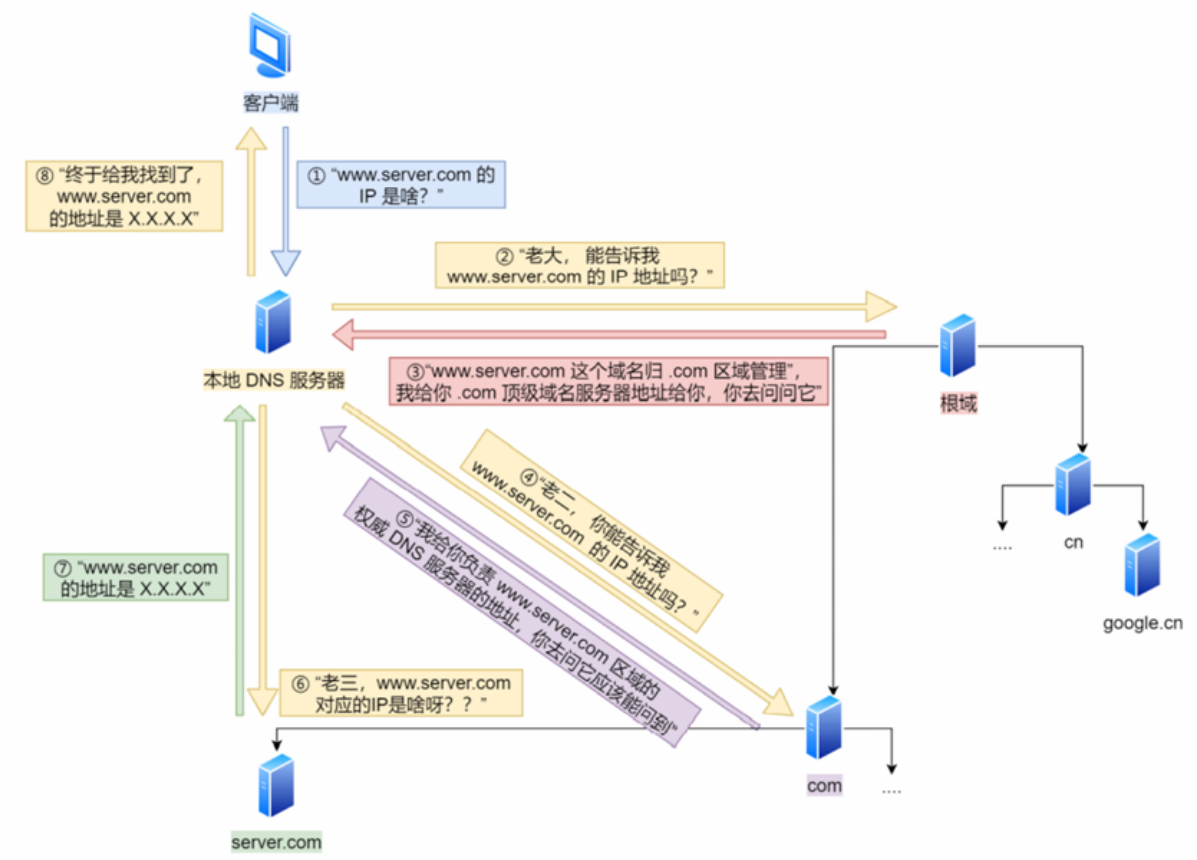
1. 客户端发送连接请求： 当客户端想要与服务器建立HTTPS连接时，它会发送一个连接请求到服务器的443端口，表明它想要使用HTTPS进行通信
2. 服务器响应： 服务器收到连接请求后，会发送一个 CA数字证书给客户端。这个证书包含了服务器的公钥、证书的颁发 者信息以及其他相关信息
3. 客户端验证证书： 客户端接收到服务器发送的数字证书后，会验证证书的合法性。这个过程包括验证证书的签名、证书是否过期、是否与预期域名匹配等
4. 生成会话密钥： 如果证书验证成功，客户端会生成一个用于该连接的随机会话密钥（对称密钥）。这个密钥将用于加密 通信数据
5. 用公钥加密会话密钥： 客户端使用服务器的公钥，将生成的会话密钥进行加密，并将加密后的会话密钥发送给服务器
6. 服务器解密会话密钥： 服务器使用自己的私钥对客户端发送的加密会话密钥进行解密，获得会话密钥
7. 建立安全通信： 从此时开始，客户端和服务器都有了相同的会话密钥，他们使用对称加密算法（如AES）来加密和解密 通信数据，保证了通信的隐私性和完整性。

DNS

DNS是什么 (中)

DNS 要解决的是**域名和 IP 地址的映射问题**, 可以将域名解析为IP地址

DNS解析流程 (中)



1. 查询浏览器缓存是否有该域名对应的IP地址
2. 如果浏览器缓存中没有，会去计算机本地的Host文件中查询是否有对应的缓存
3. 如果Host文件中也没有则会向本地的DNS服务器发送一个DNS查询请求

4. 如果本地DNS解析器有该域名的ip地址，就会直接返回，如果没有缓存该域名的解析记录，它会向根DNS服 务器发出查询请求。根DNS服务器并不负责解析域名，但它能告诉本地DNS解析器应该向哪个顶级域 （.com/.net/.org）的DNS服务器继续查询
5. 本地DNS解析器接着向指定的顶级域名DNS服务器发出查询请求。顶级域DNS服务器也不负责具体的域名解 析，但它能告诉本地DNS解析器应该前往哪个权威DNS服务器查询下一步的信息
6. 本地DNS解析器最后向权威DNS服务器发送查询请求。 权威DNS服务器是负责存储特定域名和IP地址映射的 服务器。当权威DNS服务器收到查询请求时，它会查找"example.com"域名对应的IP地址，并将结果返回给本 地DNS解析器
7. 本地DNS解析器将收到的IP地址返回给浏览器，并且还会将域名解析结果缓存在本地，以便下次访问时更快地 响应
8. 浏览器使用该IP地址与目标服 务器建立连接

DNS的底层使用TCP还是UDP? (中)

DNS基于UDP协议实现，使用UDP协议进行域名解析和数据传输。因为基于UDP实现DNS能够提供低
延迟、简单快速、轻量级的特性，更适合DNS这种需要快速响应的域名解析服务。

- 低延迟：UDP是一种无连接的协议，不需要在数据传输前建立连接，因此可以减少传输时延，适合DNS这种需要快速响应的应用场景
- 简单快速：UDP相比于TCP更简单，没有TCP的连接管理和流量控制机制，传输效率更高，适合DNS这种需要快速传输数据的场景
- 轻量级：UDP头部较小，占用较少的网络资源，对于小型请求和响应来说更加轻量级，适合DNS这种频繁且短小的数据交换

尽管UDP存在丢包和数据包损坏的风险，但在DNS的设计中，这些风险是可以被容忍的。DNS使用了一些机制来提高可靠性，例如查询超时重传、请求重试、缓存等，以确保数据传输的可靠性和正确性。

HTTP到底是无状态的吗/解释一下HTTP的无状态 (中)

- HTTP是无状态的，每个请求都是独立的，服务器不会在多个请求之间保留关于客户端状态的信息
- 在每个HTP请求中，服务器不会记住之前的请求或会话状态，因此每个请求都是相互独立的, 这就导致了当用户登录后, 再次发起请求, 服务不知道该用户是谁, 有没有登录.
- 所以http需要通过一些机制来实现状态保持，其中最常见的方式是使用Cookie和Session来跟踪用户状态。
- 通过在客户端存储会话信息或状态信息，服务器可以识别和跟踪特定用户的状态. 就能知道用户是否登录, 是哪个用户发起了请求.

cookie和session的区别 (中)

Cookie和Session都是Web开发中用于跟踪用户状态的技术，但它们在存储位置、数据容量、安全性以及
生命周期等方面存在显著差异：

- **存储位置：**Cookie的数据存储在客户端（通常是浏览器）。当浏览器向服务器发送请求时，会自动附带Cookie中的数据。Session的数据存储在服务器端。服务器为每个用户分配一个唯一的Session ID, 这个ID通常通过Cookie或URL重写的方式发送给客户端，客户端后续的请求会带上这个Session ID, 服务器根据ID查找对应的Session数据。

- **数据容量：**单个Cookie的大小限制通常在4KB左右，而且大多数浏览器对每个域名的总Cookie数量也有限制。由于Session存储在服务器上，理论上不受数据大小的限制，主要受限于服务器的内存大小。
- **安全性：**Cookie相对不安全，因为数据存储在客户端，容易受到XSS(跨站脚本攻击）的威胁。不过，可以通过设置HttpOnly属性来防止JavaScript访问，减少XSS攻击的风险，但仍然可能受到CSRF(跨站请求伪造)的攻击。Session通常认为比Cookie更安全，因为敏感数据存储在服务器端。但仍然需要防范Session劫持(通过获取他人的SessionID)和会话固定攻击。
- **生命周期：**Cookie可以设置过期时间，过期后自动删除。也可以设置为会话Cookie,即浏览器关闭时自动删除。Session在默认情况下，当用户关闭浏览器时，Session结束。但服务器也可以设置Session的超时时间，超过这个时间未活动，Session也会失效。
- **性能：**使用Cookie时，因为数据随每个请求发送到服务器，可能会影响网络传输效率，尤其是在Cookie数据较大时。使用Session时，因为数据存储在服务器端，每次请求都需要查询服务器上的Session数据，这可能会增加服务器的负载，特别是在高并发场景下。

JWT 令牌和Session方式有什么区别？（低）

JWT

- 无状态性：JWT是无状态的令牌，不需要在服务器端存储会话信息。因为JWT令牌中包含了所有必要的信息，如用户身份、权限等。这使得JWT在分布式系统中更加适用，可以方便地进行扩展和跨域访问

- 安全性：JWT使用密钥对令牌进行签名，确保令牌的完整性和真实性。只有持有正确密钥的服务器才能对令牌进行验证和解析。这种方式比传统的基于会话和Cookie的验证更加安全，有效防止了CSRF攻击
- 跨域支持：JWT令牌可以在不同域之间传递，适用于跨域访问的场景。通过在请求的头部或参数中携带JWT令牌，可以实现无需Cookie的跨域身份验证
- 踢人下线: JWT只是一个字符串, 颁发后存在客户端, 无法立刻销毁, 所以无法做到踢人下线的功能

Session

- 有状态: session是有状态的, 需要存在服务端.
- 不支持跨域: session需要cookie, 跨域请求不携带cookie, 所以跨域时session无法使用. 一般情况会生成一个唯一token放在请求头中, 替代cookie
- 踢人下线: session在服务端保存, 需要踢人下线, 直接清除session即可.

ARP

MAC地址, IP地址, ARP协议 (中)

IP地址：每一台计算机使用的 IP 地址，是由 ISP（指提供 Internet 服务的公司，比如电信、网通、移动等）负责分配的。**ISP 会动态分配**一个 IP 地址. 对于目前广泛使用 IPv4 地址，它的资源是非常有限的，一台计算机一个 IP 地址是不现实的，往往**一个局域网才拥有一个 IP 地址**。计算机之间进行通信，必须要知道对方的 IP 地址。实际上，计算机发出的数据包中已经附带了 IP 地址，把数据包发送给路由器以后，路由器会根据 IP 地址找到位置. 通常情况下，一个局

域网才能拥有一个独立的 IP，换句话说，**IP 地址只能定位到一个局域网，无法定位到具体的某个设备**.

MAC地址：真正能**唯一标识设备**的是 MAC 地址, 也称为局域网地址，以太网地址或物理地址, 硬件地址。 计算机出厂时，**MAC 地址已经被写死到网卡里面**了, 每个网卡的 **MAC 地址在全世界都是独一无二**的。 局域网中的**路由器/交换机会记录每台计算机的 MAC 地址**。当一台计算机通过网络向另一台计算机发送数据时，**数据包中除了会附带对方的 IP 地址，还会附带对方的 MAC 地址**。当数据包达到局域网以后，路由器/交换机会根据数据包中的 MAC 地址找到对应的计算机，然后把数据包转交给它，这样就完成了数据的传递。

总而言之，**IP 地址**解决的是数据在**外网（因特网、互联网）**的传输问题，**MAC 地址**解决的是数据在**内网（局域网）**中的传输问题。对于一台计算机来说，MAC 地址是必须有的，IP 地址可有可无。如果两台通信的计算机处于同一个局域网，那么理论上只凭借 MAC 地址就可以找到对方；如果两台计算机跨网传输数据，那么 IP 地址和 MAC 地址缺一不可。

ARP 协议解决了什么问题？（低）

ARP 协议，全称 **地址解析协议（Address Resolution Protocol）**，它解决的是网络层地址和链路层地址之间的转换问题。因为一个 IP 数据报在物理上传输的过程中，总是需要知道下一跳（物理上的下一个目的地）该去往何处，但 IP 地址属于逻辑地址，而 MAC 地址才是物理地址，ARP 协议解决了 IP 地址转 MAC 地址的一些问题

ARP的过程 (低)

ARP(Address Resolution Protocol)协议

- **地址解析协议**．**根据IP地址获取物理地址的一个协议。**
- 主机发送信息时将包含目标IP地址的**ARP请求广播到局域网络上的所有主机**，并**接收返回消息**，以此**确定目标的物理地址**；
- 收到返回消息后将该**IP地址和物理地址存入本机ARP缓存**中并保留一定时间，下次请求时**直接查询ARP缓存**以节约资源。
- 地址解析协议是建立在网络中各个主机互相信任的基础上的，局域网络上的主机可以**自主发送ARP应答消息**，其他主机收到应答报文时**不会检测该报文的真实性就会将其记入本机ARP缓存**

操作系统

基础

什么是用户态和内核态/什么是系统调用 (中)

根据进程访问资源的特点，我们可以把进程在系统上的运行分为两个级别：

1. 用户态 : 用户态运行的进程可以直接读取用户程序的数据。
2. 内核态: 可以简单的理解内核态运行的进程或程序几乎可以访问计算机的任何资源，不受限制

我们运行的程序基本都是运行在用户态，如果我们调用操作系统提供的系统态级别的子功能咋办呢？那就需要系统调用了！

也就是说在我们运行的用户程序中，凡是与系统态级别的资源有关的操作（如文件管理、进程控制、内存管理等)，都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成

如java的io操作, 需要操作系统从用户态陷入到内核态, 然后读取文件, 放入系统内存, 然后放入java的缓冲区, 然后操作系统再从内核态切换到用户态, 此时java就可以操作文件了. 为了避免复制, 所以出现了nio和直接内存.

进程管理

进程的几种状态 (中)

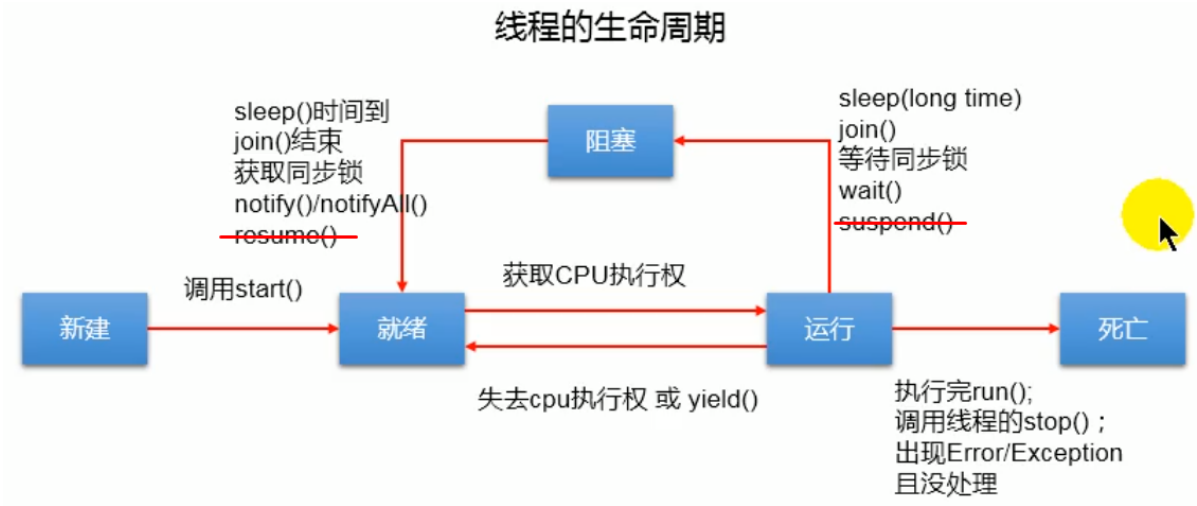
创建状态(new)：进程正在被创建，尚未到就绪状态。

就绪状态(ready)：进程已处于准备运行状态，即进程获得了除了处理器之外的一切所需资源，一旦得到处理器资源(处理器分配的时间片)即可运行。

运行状态(running)：进程正在处理器上上运行(单核 CPU 下任意时刻只有一个进程处于运行状态)。

阻塞状态(waiting)：又称为等待状态，进程正在等待某一事件而暂停运行如等待某资源为可用或等待 IO 操作完成。即使处理器空闲，该进程也不能运行。

结束状态(terminated)：进程正在从系统中消失。可能是进程正常结束或其他原因中断退出运行。



进程间的通信方式 (中)

进程是分配系统资源的单位（包括内存地址空间）,因此**各进程拥有的内存地址空间相互独立。**一个进程不能访问另一个进程的内存地址空间, 所以就需要进程通信.

1. **共享内存(Shared memory)** : 在内存中划分出一块共享存储区, 使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。**各个进程要互斥地访问共享内存**, 可以用互斥锁和信号量等实现互斥操作。这个的典型使用就是剪贴板.
2. **管道通信 (字符流)**: 操作系统**建立管道连接两个软件, 再传输字符流. 这个管道是一种特殊的pipe文件**, 管道大小固定, 采用半双工方式通信, **各进程互斥访问管道**. 数据以字符流的形式写入管道，当管道写满时，写进程的write()系统调用将被阻塞，等待另一个进程将数据取走。当读进程将数据全部取走后，管道变空，此时读进程的read()系统调用将被阻塞, 第一个进程就可以开始写了。(**写满了才能读, 读完了才能写**)
3. **消息传递**:
    1. 直接通信方式: 每个进程会有一个消息缓冲队列, 其他进程想要通信, 就**把要发送的数据封装为一个消息, 然后放入目标进程的消息缓冲队列中**, 目标进程就能读取消息缓冲队列, 获得数据
    2. 间接通信方式: **进程发送消息时, 会发送到一个中间实体中, 称为信箱**. 消息的消息头中存放了发送进程ID和接收进程ID, 所以目标进程想知道消息, 直接从信箱取即可.
4. **socket套接字**: 此方法主要用于在客户端和服务器之间通过网络进行通信

进程的调度算法有哪些 (中)

1. 先来先服务 first-come first-serverd（FCFS)

把需要调度的进程放到就绪队列中, 先进先出, 从就绪队列中选择一个最先进入该队列的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。

2. 短作业优先 shortest job first（SJF）

按估计运行时间最短的顺序进行调度

会导致饥饿, 长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

3. 优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

满足紧急作业的要求, 特别适合用在实时系统中

4. 时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。

当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。
- 而如果时间片过长，那么实时性就不能得到保证。

5. 多级反馈队列调度算法

1. 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大
2. 新进程到达时先进入第1级队列，按FCFS原则排队等待被分配时间片，若用完时间片进程还未结束，则进程进入下一级队列队尾。
3. 当第一级队列的进程被服务完, 开始运行第二级队列的进程, 类推

多级反馈队列调度算法综合了FCFS, 短作业优先, 时间片轮转, 优先级调度的优点, 是当前认为比较好的一种调度算法

线程是不是越多越好 (中)

多线程不一定越多越好，过多的线程可能会导致一些问题。

- 切换开销：线程的创建和切换会消耗系统资源，包括内存和CPU。如果创建太多线程，会占用大量的系统资源，导致系统负载过高，某个线程崩溃后，可能会导致进程崩溃。

- 死锁的问题：过多的线程可能会导致竞争条件和死锁。竞争条件指的是多个线程同时访问和修改共享资
  源，如果没有合适的同步机制，可能会导致数据不一致或错误的结果

进程切换和线程切换的区别?

- 进程切换：进程切换涉及到更多的内容，包括整个进程的地址空间、全局变量、文件描述符等。因此，进程切换的开销通常比线程切换大。
- 线程切换：线程切换只涉及到线程的堆栈、寄存器和程序计数器等，不涉及进程级别的资源，因此线程切换的开销较小。

线程切换为什么比进程切换快，节省了什么资源? (中)

- 线程切换比进程切换快是因为线程共享同一进程的地址空间和资源，线程切换时只需切换堆栈和程序计数器等少量信息，而不需要切换地址空间
- 避免了进程切换时需要切换内存映射表等大量资源的开销，从而节省了时间和系统资源

死锁的四个必要条件 (中)

1. **互斥条件**: 只有对**临界资源**(需要互斥访问的资源)的争夺才会产生死锁
2. **不可剥夺条件**: 进程在所获得的资源未释放前，**不能被其他进程强行夺走，只能自己释放。**

3. **请求保持条件**: 两个进程各占有一部分资源, 保持**占有一部分资源的同时都请求对方让出另一部分资源**
4. **循环等待条件**: 双方都**等待对面让出资源, 产生僵持**

死锁检测, 预防, 避免 (中)

检测死锁

- `jps` 可以查看定位进程号, `jstack 进程号` 可以查看栈信息, 来排查死锁
- `jconsole` 可以用来检测死锁
- arthas这种工具也可以用来检测排查死锁

预防死锁

- **破坏互斥条件**: Java的ThreadLocal, 每个线程都拥有自己数据副本, 自己访问自己的, 不需要互斥访问.
- **破坏请求与保持条件**： 一次性申请所有的资源
- **破坏不可剥夺条件**： 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源
  - 超时放弃 (**破坏不可剥夺条件**)
    - Lock接口提供了boolean tryLock(long time, TimeUnit unit) 方法, 如果一定时间没获取到锁就放弃.
- **破坏循环等待条件**： 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件
  - 指定获取锁的顺序 (**破坏循环等待条件**)
    - 比如某个线程只有获得A锁和B锁才能对某资源进行操作.
    - 规定获取锁的顺序，比如只有获得A锁的线程才有资格获取B锁，按顺序获取锁就可以避免死锁

内存管理

内存管理机制/内存分配方式 (低)

简单分为**连续分配管理方式**和**非连续分配管理方式**这两种。连续分配管理方式是指为一个用户程序分配一个连续的内存空间，常见的如 **块式管理** 。同样地，非连续分配管理方式允许一个程序使用的内存分布在离散或者说不相邻的内存中，常见的如**页式管理** 和 **段式管理**

- **块式管理**： 远古时代的计算机操作系统的内存管理方式。将内存分为几个固定大小的块，每个块中只包含一个进程。如果程序运行需要内存的话，操作系统就分配给它一块，如果程序运行只需要很小的空间的话，分配的这块内存很大一部分几乎被浪费了。这些在每个块中未被利用的空间，我们称之为碎片
- **页式管理**： 把主存分为大小相等且固定的一页一页的形式，页非常小，相比于块式管理的划分粒度更小. 页式管理提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址
- **段式管理**： 页式管理虽然提高了内存利用率，但是页式管理其中的页并无任何实际意义。 段式管理把主存分为一段段的，段是有实际意义的，每个段定义了一组逻辑信息，例如,有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。 段式管理通过段表对应逻辑地址和物理地址

简单来说：页是物理单位，段是逻辑单位。分页可以有效提高内存利用率，分段可以更好满足程序员需求。

- **段页式管理机制** 。段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说 **段页式管理机制** 中段与段之间以及段的内部的都是离散的。

分页机制和分段机制的共同点和区别 (低)

1. 共同点
    - 分页机制和分段机制都是为了提高内存利用率，减少内存碎片。
    - 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是，每个页和段中的内存是连续的。
2. 区别
    - 页的大小是固定的，由操作系统决定；而段的大小不固定，取决于我们当前运行的程序。
    - 分页仅仅是为了满足操作系统内存管理的需求，而段是逻辑信息的单位，在程序中可以体现为代码段，数据段，能够更好满足用户的需要

快表和多级页表 (低)

快表

为了提高虚拟地址到物理地址的转换速度，操作系统在 页表方案 基础之上引入了快表来加速虚拟地址到物理地址的转换。我们可以把快表理解为一种特殊的高速缓冲存储器（Cache），其中的内容是页表的一部分或者全部内容.

每次查询时, 先查快表, 如果快表命中, 就能直接拿到物理地址. 如果快表未命中, 则去查页表, 同时将页表中的该映射表项添加到快表中.

快表本质上就是一个缓存, 和redis或高速缓冲存储器（Cache）的作用类似.

快表实际上用了程序的局部性原理，一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问. 某条指令一旦执行，不久以后该指令可能再次执行；如果某数据被访问过，不久以后该数据可能再次被访问.

多级页表

引入多级页表的主要目的是为了避免把全部页表一直放在内存中占用过多空间，特别是那些根本就不需要的页表就不需要保留在内存中.

总结

为了提高内存的空间性能，提出了多级页表的概念；但是提到空间性能是以浪费时间性能为基础的，因此为了补充损失的时间性能，提出了快表（即 TLB）的概念。 不论是快表还是多级页表实际上都利用到了程序的局部性原理

逻辑(虚拟)地址和物理地址 (低)

编程一般只有可能和逻辑地址打交道，比如在 C 语言中，指针里面存储的数值就可以理解成为内存里的一个地址，这个地址也就是我们说的逻辑地址，逻辑地址由操作系统决定。

物理地址指的是真实物理内存中地址，更具体一点来说就是内存地址寄存器中的地址。物理地址是内存单元真正的地址.

CPU 寻址了解吗?为什么需要虚拟(逻辑地址)地址空间? (低)

现代处理器使用的是一种称为 **虚拟寻址(Virtual Addressing)** 的寻址方式。**使用虚拟寻址，CPU 需要将虚拟地址翻译成物理地址，这样才能访问到真实的物理内存**

没有虚拟地址空间的时候，**程序直接访问和操作的都是物理内存**

- 直接使用物理地址让用户程序可以访问任意内存，寻址内存的每个字节，这样就很容易（有意或者无意）破坏操作系统，造成操作系统崩溃, 或者破坏其他程序.
- 直接使用物理地址会导致程序在装入的时候会出问题, 想要同时运行多个程序特别困难，比如你想同时运行一个微信和一个 QQ 音乐都不行。为什么呢？举个简单的例子：微信在运行的时候给内存地址 1xxx 赋值后，QQ 音乐也同样给内存地址 1xxx 赋值，那么 QQ 音乐对内存的赋值就会覆盖微信之前所赋的值，这就造成了微信这个程序就会崩溃

所以每个程序都使用虚拟地址, 然后由操作系统将虚拟地址转化为物理地址来装入内存. 避免用户程序破坏系统, 方便多个程序装入内存.

虚拟内存

局部性原理 (低)

局部性原理是虚拟内存技术的基础，正是因为程序运行具有局部性原理，才可以只装入部分程序到内存就开始运行

- **时间局部性**：如果程序中的某条指令一旦执行，不久以后该指令可能再次执行；如果某数据被访问过，不久以后该数据可能再次被访问。产生时间局部性的典型原因，是由于在程序中存在着大量的循环操作。
- **空间局部性**：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问. 这是因为指令通常是顺序存放、顺序执行的，很多数据也一般是以数组的形式存储的.

传统存储管理的缺点 (低)

1. **一次性：程序必须一次性全部装入内存后才能开始运行。**这会造成两个问题：
    - 程序很大时，不能全部装入内存，导致大程序无法运行；
    - 当大量程序要求运行时，由于内存无法容纳所有程序，因此有少量程序能运行，导致并发度下降
2. **驻留性：一旦作业被装入内存，就会一直驻留在内存中，直至作业运行结束。**
    - 事实上，在一个时间段内，只需要访问程序的一小部分代码即可正常运行
    - 驻留性就导致了内存中会驻留大量的、暂时用不到的数据和代码，浪费了宝贵的内存资源

什么是虚拟内存/虚拟存储技术/虚拟存储器 (低)

我们一个游戏, 像3A大作, 下载需要50G, 甚至100G. 但是我们电脑的内存一般是8G, 16G, 32G, 远远小于游戏所需空间, 但是游戏却可以顺利运行在我们的电脑上. 这就用到了虚拟存储技术.

因为程序有局部性原理, 所以就可以用虚拟存储技术:

1. 基于局部性原理，在程序装入时，可以将**程序中很快会用到的部分装入内存，暂时用不到的部分留在外存**，就可以让程序开始执行。
2. 在程序执行过程中，当所**访问的信息不在内存时，就会产生缺页中断, 由操作系统负责将所需信息从外存调入内存**，然后继续执行程序。
3. 若**内存空间不够**，由操作系统负责**将内存中暂时用不到的信息换出到外存。**

这样，计算机好像为用户提供了一个比实际内存大得多的存储器——**虚拟存储器**

虚拟存储技术的实现 (低)

虚拟内存的实现需要建立在离散分配的内存管理方式的基础上。 虚拟内存的实现有以下三种方式：

- **请求分页存储管理**：
  - 建立在分页管理之上，为了支持虚拟存储器功能而增加了请求调页功能和页面置换功能。请求分页是目前最常用的一种实现虚拟存储器的方法。
  - 请求分页存储管理系统中，在程序开始运行之前，仅装入当前要执行的部分页面即可运行。
  - 假如在程序运行的过程中发现要访问的页面不在内存，则由处理器通知操作系统按照对应的页面置换算法将相应的页面调入到主存，同时操作系统也可以将暂时不用的页面置换到外存中。
- **请求分段存储管理**：
  - 建立在分段存储管理之上，增加了请求调段功能、分段置换功能。
  - 请求分段储存管理方式就如同请求分页储存管理方式一样，在作业开始运行之前，仅装入当前要执行的部分段即可运行；
  - 在执行过程中，可使用请求调入中断动态装入要访问但又不在内存的程序段；当内存空间已满，而又需要装入新的段时，根据置换功能适当调出某个段，以便腾出空间而装入新的段
- **请求段页式存储管理**

- 结合请求分页和请求分段

请求分页与分页存储管理有何不同呢（低）

请求分页存储管理建立在基本分页管理之上。他们的根本区别是**是否将程序全部所需的全部地址空间都装入主存**，这也是请求分页存储管理可以实现虚拟内存的原因.

页面置换算法 (低)

虚拟内存需要用到请求段页式内存管理, 缺页时会发生缺页中断, 就需要页面置换算法将页面换入或者换出内存.

最佳置换算法: (OPT页面置换算法)

算法思想: 淘汰以后永不访问或将来最长时间内不再访问的页面

特点: 不能预测未来, 所以该算法不能实现

先进先出置换算法: (FIFO页面置换算法)

算法思想: 将最早进入内存的页面淘汰

特点: 有可能调出主程序, 所以性能最差

最近最久未使用置换算法: (LRU)

算法思想: 每次淘汰的页面是最近最久未使用的页面

特点: 性能优异, 接近最佳置换算法, 但是需要硬件栈支持, 开销大

最近最少使用算法(LFU)

最近最少使用算法。它是基于“如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小”的思路。

LRU是淘汰最长时间没有被使用的页面。 LFU是淘汰一段时间内，使用次数最少的页面。

磁盘和IO

(操作系统中, 磁盘和IO并不是常考点, 所以这里不多做赘述. IO常考的BIO, NIO, IO多路复用等在JavaSE中已有讲过)

数据结构

(数据结构一般不会单独考察, 往往配合Java的集合, 或者配合算法/场景题考察, 这里没办法列出具体数据结构的问题, 只能给出常考数据结构的定义. 数据结构建议在刷算法题时好好理解和应用)

常见数据结构 (高)

数组

- 使用一块连续的内存来存储
- 提供随机存取, 我们直接可以利用元素的索引（index）可以计算出该元素对应的存储地址
- 适合查找, 不适合增加删除

链表

- 使用的非连续的内存空间来存储数据。
- 链表的插入和删除操作的复杂度为 O(1)，只需要知道目标位置元素的上一个元素即可
- 适合插入删除, 不适合查找, 不具有随机存取特点

栈

后进先出

- 最经典的使用就是维护函数调用(虚拟机栈), 最后一个被调用的函数必须先完成执行

队列

先进先出

- 最经典的应用: MQ, MQ本质就是一个阻塞队列, 先进先出.

树

二叉排序树/二叉查找树/二叉搜索树

特点

- 左孩子 < 根 < 右孩子
- 对二叉排序树进行中序遍历，可以得到一个升序排列的节点值序列
- 在二叉排序树中查找一个特定值的节点时，平均时间复杂度为 O(logn)

- 插入操作时，根据节点值的大小关系，从根节点开始比较，找到合适的位置插入新节点，保持二叉排序树的性质不变。插入操作的平均时间复杂度也是O(logn)
- 删除操作相对复杂一些，但也可以在保持树的性质的前提下完成，平均时间复杂度同样为 O(logn)

平衡二叉树

平衡二叉树 一般指平衡二叉排序树，且具有以下性质

1. 如果不是空树，它的左右两个子树的高度差的绝对值不超过 1
2. 并且左右两个子树都是一棵平衡二叉树

二叉排序树可能退化为链表，而平衡二叉树不会：普通的二叉排序树在某些极端情况下（如按顺序插入节点）会导致树的高度过大，从而使查找效率降低；而平衡二叉树通过自动调整树的结构，始终保持树的平衡，树高度不会过高, 保证了操作的高效性

堆

什么是堆

堆是一种特殊的树, 任意一个节点的值都大于等于（或小于等于）所有子节点的值

堆的作用

当我们只关心所有数据中的最大值或者最小值，存在多次获取最大值或者最小值，多次插入或删除数据时，就可以使用堆

堆的主要优势在于插入和删除数据效率较高。 因为堆是基于完全二叉树实现的，所以在插入和删除数据时，只需要在二叉树中上下移动节点，时间复杂度为 `O(log(n))`，相比有序数组的 `O(n)`，效率更高。

哈希表

详情见javaSE中的HashMap原理

常见题

数组和链表区别是什么？（中）

- 访问效率：数组可以通过索引直接访问任何位置的元素，访问效率高，时间复杂度为O(1),而链表需要从头节点开始遍历到目标位置，访问效率较低，时间复杂度为O(n)。
- 插入和删除操作效率：数组插入和删除操作可能需要移动其他元素，时间复杂度为O(n),而链表只需要修改指针指向，时间复杂度为O(1)。
- 缓存命中率：由于数组元素在内存中连续存储，可以提高CPU缓存的命中率，而链表节点不连续存储，可能导致CPU缓存的命中率较低，频繁的缓存失效会影响性能。
- 应用场景：数组适合静态大小、频繁访问元素的场景，而链表适合动态大小、频繁插入、删除操作的场景

说一下队列和栈的区别和应用场景 (中)

- 队列: 先进先出.
  - 用于
- 栈: 先进后出
  - 用于函数调用, 比如函数栈.

有大量的数据在磁盘中, 但是内存很小, 如何排序 (中)

根据内存大小，将待排序的文件拆成多个部分，让每个部分都是足以存入内存中的。然后使用快速排序对内存中的数据进行排序，时间复杂度是O(nlogn). 排序后的数据成为顺段.

此时对前面的多个“顺段”进行合并，使用k路归并排序

- 每次将k个连续的顺段合并成一个更大的顺段
- 因为内存限制，每次可能只能读入k个顺段的部分内容，所以我们需要一部分一部分读入，在内存里排序，并输出到外存里的文件中
- 不断重复这个过程，直至k个顺段被完整遍历
- 这样经过多层的归并之后，最终会得到一个完整的顺序文件

内存很小, 但是有大量的数据, 如何快速找出最大的前K个值? (中)

内存小，无法一次性将所有数据加载到内存中排序. 可以使用堆结构求top k.

- 使用一个最小堆，维护K个最大的元素

- 遍历数据，当堆的大小小于K时直接加入，否则比较当前元素和堆顶，如果更大就替换堆顶，并调整堆
- 遍历所有数据, 堆中的数据就是top k

时间复杂度是`O(N * log K)`, 因为每次堆操作是`O(log K)`, 需要进行N次

设计模式

(设计模式常考的就是单例和工厂, 其他设计模式只要你简历不写, 一般不会问. 设计模式也不是面试重点, 不要花太多时间)

单例模式 (高)

确保一个类只有一个实例，并提供一个全局访问点来访问该实例

关键概念

- 一个私有构造函数（确保只能单例类自己创建实例）单例类通常会将其构造函数设为私有，以防止外部代码直接实例化对象
- 一个私有静态变量（确保只有一个实例） 单例类通常包含一个私有的静态变量，用于保存该类的唯一实例
- 一个公有静态函数（给使用者提供调用方法）

单例模式的应用场景

资源共享：当多个模块或系统需要共享某一资源时，可以使用单例模式确保该资源只被创建一次，避免重复创 建和浪费资源

配置管理器：当整个应用程序需要共享一些配置信息时，可以使用单例模式将配置信息存储在单例类中，方便 全局访问和管理

日志记录器：单例模式可以用于创建一个全局的日志记录器，用于记录系统中的日志信息

饿汉式单例

```
public class Singleton {
    // 类加载时就创建实例
    private static final Singleton instance = new Singleton();

    // 私有构造函数
    private Singleton() {}

    // 公共静态方法，返回实例
    public static Singleton getInstance() {
        return instance;
    }
}
```

先不管需不需要使用这个实例，直接先实例化好实例（饿死鬼一样，所以称为饿汉式），然后当需要使用的时候， 直接调方法就可以使用了

优点： 提起实例化好了一个实例，避免了线程不安全问题的出现，

缺点： 直接实例化了实例，若系统没有使用这个实例，或者系统运行很久之后才需要使用这个实例，都会使操作系统的资源浪费。

懒汉式单例

单检测锁版本

```
public class Singleton {
    private static Singleton uniqueInstance;

    private static singleton() {}

    private static synchronized Sing leton
getUinqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

多个线程访问，每次只有拿到锁的的线程能够进入该方法，避免了多线程不安全问题的出现。

优点： 延迟实例化，节约了资源，并且是线程安全的

缺点： 虽然解决了线程安全问题，但是性能降低了。因为，即使实例已经实例化了，既后续不会再出现线程安全问题了， 但是锁还在，每次还是只能拿到锁的线程进入该方法使线程阻塞，等待时间过长。

双重检测锁版本

```
public class Singleton {
    private volatile static Singleton
uniqueInstance;

    private Singleton() {}
```

```
5
6       // 使用双重检查锁定保证线程安全
7       public static Singleton getUniqueInstance() {
8           if (uniqueInstance == null) {
9               synchronized (Singleton.class) {
10                  if (uniqueInstance == null) {
11                      uniqueInstance = new
    Singleton();
12                  }
13              }
14          }
15
16          return uniqueInstance;
17      }
18  }
```

双重检查锁相当于是线程安全的懒汉式

先判断实例是否已经存在，若已经存在了，则直接返回对象, 不会去争抢锁.

如果还没有实例化的时候，多个线程进去了，也没有事，因为里面的方法有锁，只会让一个线程进入最内层方法并实例化实例。如此一来，最多最多，也就是第一次实例化的时候，会有线程 阻塞的情况，后续便不会再有线程阻塞的问题

为什么使用 volatile 关键字修饰了 uniqueInstance 实例变量?

uniqueInstance = new Singleton(); 这段代码执行时分为三步

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

正常的执行顺序当然是 1>2>3 ，但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1>3>2

单线程环境时，指令重排并没有什么问题；多线程环境时，会导致有些线程可能会获取到还没初始化的实例. 所以必须使用volatile

优点： volatile 会禁止 JVM 的指令重排，就可以保证延迟实例化，节约了资源；线程安全；并且相对于线程安全的懒汉式，性能提高了

缺点： volatile 关键字，对性能也有一些影响

静态内部类版本

```
public class Singleton {
    private Singleton() {}

    // 静态内部类持有实例
    private static class SingletonHolder {
        private static final Singleton instance = new Singleton();
    }

    // 公共静态方法，返回实例
    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

当外部类 Singleton 被加载时，静态内部类 SingletonHolder 并没有被加载进内存。

当调用 getUniqueInstance() 方法时，会运行 return SingletonHolder.INSTANCE; 触发了 SingletonHolder.INSTANCE，此时静态内部类 SingletonHolder 才会被加载进内存，并且初始化 INSTANCE 实例

而且 JVM 会确保 INSTANCE 只被实例化一次

优点： 延迟实例化，节约了资源，且线程安全，性能也提高了

工厂模式 (中)

工厂模式就是一个工厂类, 用来生产对象的.

可以分为简单工厂(静态简单工厂) / 工厂方法 / 抽象工厂

简单工厂

在简单工厂模式中，一个工厂类负责创建多个产品类的实例，通过传入不同的参数来决定创建哪种产品

```
public interface Product {
}


public class ProductA implements Product {
}


public class ProductB implements Product {
}


public class Factory {
    public static Product getProduct(String type)
{
        if (type.equals("ProductA")) {
            return new ProductA();
        }
```

```
        if (type.equals("ProductB")) {
            return new ProductB();
        }

        throw new RuntimeException("Unknown product type: " + type);
    }
}
```

缺点:

- 虽然实现了对象的创建和使用的分离，但是不够灵活，工厂类集合了所有产品的创建逻辑，职责过重
- 同时新增一个产品就需要在原工厂类内部添加一个分支，违反了开闭原则
- 并且若是有多个判断条件共同决定创建对象，则后期修改会越来越复杂.

spring采用了简单工厂+配置文件的方式进行解耦, 使用反射创建对象, 大大增加了灵活的, 避免了耦合. 这里实现一下简单工厂+配置文件解耦的方式.

商品接口和商品实现类

```java
public interface Product {
}


public class ProductA implements Product {
}


public class ProductB implements Product {
}
```

商品类型, 用于存放配置数据

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ProductType {
    private String type;
    private String className;
}
```

```json
// spring使用yml配置, 这里使用json配置, 配置类型和全类名
[
  {
    "type": "ProductA",
    "className": "com.pan.mvntest.test.ProductA"
  },
  {
    "type": "ProductB",
    "className": "com.pan.mvntest.test.ProductB"
  }
]
```

核心工厂类实现

```
public class Factory {
    private static final ConcurrentHashMap<String, Product> PRODUCTS = new ConcurrentHashMap<>();

    static {
        // 读配置
        String json = ResourceUtil.readStr("config.json", StandardCharsets.UTF_8);

        // 将json转为List<ProductType>
        List<ProductType> productTypes = JSONUtil.toBean(json, new TypeReference<List<ProductType>>() {
        }, true);

        // 创建对象放入map中
        for (ProductType productType : productTypes) {
            String type = productType.getType();
            Product product;

            try {
                Class<Product> clazz = (Class<Product>) Class.forName(productType.getClassName());
                product = clazz.getConstructor().newInstance();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
```

```
23
24              PRODUCTS.put(productType.getType(), product);
25          }
26      }
27
28      public static Product getProduct(String type) {
29          return PRODUCTS.get(type);
30      }
31  }
32
```

从核心工厂类中可以看到, 将类型和对应的全类名抽取到json配置中, 然后读json, 通过反射创建对象然后放入map中, 后续从map中读.

这就是spring的实现方式. 当然了, spring实现了更复杂的东西. 但是仅从工厂模式的使用来看, 这和spring的思想是一致的.

后续新增ProductC, 只需要写出对应Class, 然后修改配置文件即可, 工厂代码不需要更改. 这就符合了开闭原则.

(工厂方法和抽象工厂并不常考, 这里不就多做解释)