牛客题霸-算法篇 https://www.nowcoder.com/ta/job-code-high

数据库三范式

一范式：每列或者说每个字段都不可分解，具有原子性

二范式：非主键列要完全依赖主键列，不能只是依赖主键的一部分。即表必须有主键，且没有包含在主键中的列必须完全依赖于主键

（防止出现行冗余，避免出现多行重复数据）

三范式：非主键列要直接依赖主键列，而不是通过依赖非主键列间接依赖主键列。

（防止出现列冗余，避免出现无关的列，无关的多列信息应放在其他表中）

索引数据结构：

B+-树：所有叶子节点在同一层上

B-树：

节点即存储索引也存储数据

叶子节点之间没有指针指向

B+树：

非叶子节点只存储索引，不存储数据，可有更多索引被加载到内存中，减少磁盘 IO，非叶子节点也可称作索引节点

叶子节点存储数据或者数据指针，且叶子节点之间有指针指向，方便区间查询

MYSQL：

索引 B+树（使用），Hash(对区间查询不友好)

聚簇索引：索引和数据存放在一起，同一个文件中。

非聚簇索引：索引和数据存分开存放，存放在不同的文件中。

MYSQL 中的两种存储引擎：

InnoDB 属于聚簇索引，MYISAM 属于非聚簇索引

InnoDB 存储引擎存储数据时数据文件本身就是按 B+树索引结构组织的，所以必须要有主键作为索引 key 值依据，若没有手动指定主键，数据库后台会自动选择一个唯一标识的字段建立主键；若表中没有唯一标识的列，则在表中默认添加一列作为主键字段，维护索引+数据文件。

SQL 慢查询：
开启慢查询日志 slow-log-log=1 并设置慢查询时间阈值 long qwuery time=xxx，设置慢查询日志文件 slow-query-log-file=xxx。当 SQL 语句执行时间超过指定的阈值时记录查询日志。
慢查询优化：优化在某些查询速度特别慢（慢查询日志中执行时间大于一定程度的 SQL 查询语句）

SQL 语句优化
索引优化

事务特性 ACID
1. 原子性 ： 事务里面的操作单元不可切割,要么全部成功,要么全部失败
2. 一致性 ： 事务执行前后,业务状态和其他业务状态保持一致.
3. 隔离性 ： 一个事务执行的时候最好不要受到其他事务的影响
4. 持久性 ： 一旦事务提交或者回滚.这个状态都要持久化到数据库中

脏读:在一个事务中读取到另一个事务没有提交的数据
不可重复读:在一个事务中,两次查询的结果不一致(针对的 update 操作)
虚读(幻读):在一个事务中,两次查询的结果不一致(针对的 insert 操作)
read uncommitted 读未提交 上面的三个问题都会出现
read committed 读已提交 可以避免脏读的发生
repeatable read 可重复读 可以避免脏读和不可重复读的发生
serializable 串行化 可以避免所有的问题

MVCC-多版本并发控制
MVCC-多版本并发控制，是一种并发控制方式，实现对数据库的并发访问控制，主要是为了提高数据库并发性能，实现读-写冲突不加锁，这里的读指的是快照读。维持一个数据的多个版本，使得读写操作没有冲突。

MVCC 实现的关键是 ：
1. 表的隐藏字段 事 ID 回滚指针
2. undo log 记录历史版本链
3. readView

执行更新操作时，会将旧数据行记录在 undo log 中并将数据表中当前记录更新，事务 ID 更新为当前操作的事务 ID，回滚指针指向 undo log 中的旧记录。

执行查询操作时，当前事务会生成一个 ReadVew，这个 ReadView 相当于是一个事物快照，即当前系统内活跃的事物列表，即系统内所有已开始但还未提交的事务，当前事务会根据这个 ReadView 去判断哪些数据可见，哪些数据不可见。

查询一条数据时，事务根据这个 ReadView，到 undo log 中判断，

1. 先查看 undo log 中当前记录最新数据行，若该数据行事务版本号小于 readView 中的事务最小 ID，说明该事务已提交，对于当前数据库是可见的，可直接作为结果返回
2. 若最新数据行的事务版本号大于 readView 中事务最大 ID，说明该事务是新事务，对当前事务不可见，要继续顺着 undo log 版本链继续往下寻找直到满足条件的
3. 若当前数据行的事务版本号 存在 readView 列表中，则需要遍历整个 readView，若数据行的版本号等于 readView 列表中某个事务 ID，说明当前数据行还处于活动状态，对该事务不可见，遍历 undo log 版本链，直到找到数据行的事务 ID 小于 readView 中最小事务 ID 的记录。

读已提交

事务中每次读操作都会生成一个 readView，若事务执行期间某个事务提交了，该事务 ID 将会从 readView 中移除，这样确保事务每次读操作都能读到已提交的新数据

可重复读

事务对数据行执行第一次读操作时生成 ReadView，后续的读操作重复使用这个 readView，计算在这个期间有别的事务提交更新，该更新也是不可见的。

也就是说已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的 ReadView,而可重复读隔离级别则在第一次读的时候生成一个 ReadView，之后的读都复用之前的 ReadView。

MYSQL 的 MVCC，通过版本链，实现多版本控制，可并发读-写，写-读，通过 ReadView 生成策略的不同实现不同的隔离级别。

HashSet

底层是 HashMap，add 元素时实质是 map.put（e, PRESENT），这个 value 的 PRESENT 是一个静态的常量对象

```
// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

SQL 注入

网络七层/五层模型各层的作用
应用层，应用服务的协议，文件传输、文件服务、HTTP 等
表示层，数据格式化、数据加密、转换等
会话层，解除或建立与其他结点的联系
传输层，提供端对端的传输端口或接口
网络层，为数据包的传输选择合适的路由，最短路径优先
数据链路层，传输有地址的帧并检测差错
物理层，以二进制数据形式在物理媒介上传输数据

SQL 语句 GROUP BY：

"Group By"从字面意义上理解就是根据"By"指定的规则对数据进行分组，所谓的分组就是将一个"数据集"划分成若干个"小区域"，然后针对若干个"小区域"进行数据处理

类别	数量	摘要
a	5	a2002
a	2	a2001
b	10	b2003
b	6	b2002
b	3	b2001
c	9	c2005
c	9	c2004
c	8	c2003
c	7	c2002
c	4	c2001
a	11	a2001

类别	数量之和
a	18
b	19
c	37

类别	数量之和
c	37
b	19
a	18

```
select 类别, sum(数量) as 数量之和
from A
group by 类别


select 类别, sum(数量) AS 数量之和
from A
group by 类别
order by sum(数量) desc
```

Group By 中 Select 指定的字段限制

在 select 指定的字段要么就要包含在 Group By 语句的后面，作为分组的依据；要么就要被包含在聚合函数中。

Group By 与聚合函数

group by 语句中 select 指定的字段必须是"分组依据字段"，其他字段若想出现在 select 中则必须包含在聚合函数中，常见的聚合函数如下表：

函数	作用	支持性
sum(列名)	求和	
max(列名)	最大值	
min(列名)	最小值	
avg(列名)	平均值	
first(列名)	第一条记录	仅 Access 支持
last(列名)	最后一条记录	仅 Access 支持
count(列名)	统计记录数	注意和 count(*)的区别

- where 子句的作用是在对查询结果进行分组前，将不符合 where 条件的行去掉，即在分组之前过滤数据，where 条件中不能包含聚组函数，使用 where 条件过滤出特定的行。
- having 子句的作用是筛选满足条件的组，即在分组之后过滤数据，条件中经常包含聚组函数，使用 having 条件过滤出特定的组，也可以使用多个分组标准进行分组。

微服务架构：将一个单一的应用程序开发为一组小型服务的方法，每个服务都独立运行在自己的进程中，服务间通信采用“轻量级“通信机制，通常使用 http 资源的 API 来实现。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署，这些服务共用一个最小的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术，即不受技术栈的限制。

微服务特征：

1. 每个服务独立运行在自己的进程中
2. 一系列运行的微服务共同构建起整个系统
3. 每个微服务可独立开发，开发过程中只要关注一个当前服务业务模块的特定功能，如支付服务、订单服务、用户管理服务等。
4. 可使用不同的技术栈开发不同的服务
5. 微服务之间直接通过轻量级的通信机制进行通信，如 Rest API
6. 全自动的部署机制

微服务适用场景：

1. 大型项目
2. 有快速迭代的要求
3. 访问压力过大的网站

微服务拆分：
1.依据技术栈驱动设计
2.面向对象驱动设计
按职责划分（订单服务、用户服务、商品服务等）
按通用性划分（把一些通用功能做成微服务，如用户中心、支付在中心、消息中心，一个中台由若干个微服务构成）

Zookper，本质就是一个封装了很多算法的软件，投票推选 leader，
一个分布式协调框架。
1. 发布订阅管理，作为注册中心
2. 分布式/集群管理
3. 安装 -> 启动 zkServer -> 防火墙设置 zkServer 端口通行
4.

网络 IO 模型

BIO：阻塞 IO

Server 端在 accept 和 read 或 write 时都是阻塞模式，主线程负责 accept 客户端发来的连接请求，若没有连接则一直阻塞在 accept 处，针对某个客户端发来的请求，起线程读取相应 socket 的数据作处理，若一直没有数据发来，则该线程阻塞在这里一直等待，直到 socket 中有数据传来。

1. 主线程 accpet 客户端的连接，每发来一个客户端连接，则创建一个线程来处理，这样客户端发来大量连接的情况下，会有非常多的线程，消耗系统资源；
2. 使用线程池进行优化，每当客户端发来一个连接时，将请求封装成任务发送给线程池调度，线程池管理线程，实现线程的复用，用少量的线程处理很多的客户端请求
3. 但是，用线程池能处理的客户端请求也是有限的

NIO：非阻塞 IO

Server 端的线程或进行在处理 socket 时不再阻塞，而是想方法使用事件通知的机制，当有 socket 中接收到数据可被读取时通知和处理该 socket 的程序（线程或进程）去读取。
Select 模型，当有 socket 中传来数据时，返回一个大于等于 1 的整数，但是此时只是知道有 socket 中有数据，并不知道是哪个 socket 中传来的数据，程序要去遍历所有的 socket，找出有数据的 socket 读取数据。Select 模型通过一个数组管理被监控的 socket。
Epoll 模型，维护一个双端队列-就绪 list，链接已经有数据可被处理的 socket 引

用，此时不需要再像 select 那样遍历所有的 socket；管理所有的 socket 使用一个红黑树结构。
等待队列 和 工作队列 中存放的是进程或线程的引用

额外：

网络数据传输到网卡后，网卡会发送一个中断信号给 CPU，CPU 会去优先处理将这些数据拷贝到内存和 socket 中~ 硬件有数据产生时会发送一个中断信号给 CPU，CPU 会优先处理，此种优先级比较高。

HTTP

HTTP1.0

无状态连接，每一个请求-响应都要三次握手建立连接->请求响应->断开连接

HTTP1.1

Keep-Alive 可以建立长连接，同一个连接被多个请求响应复用
HTTP1.0 和 HTTP1.1 数据都是串行传输的，文本传输

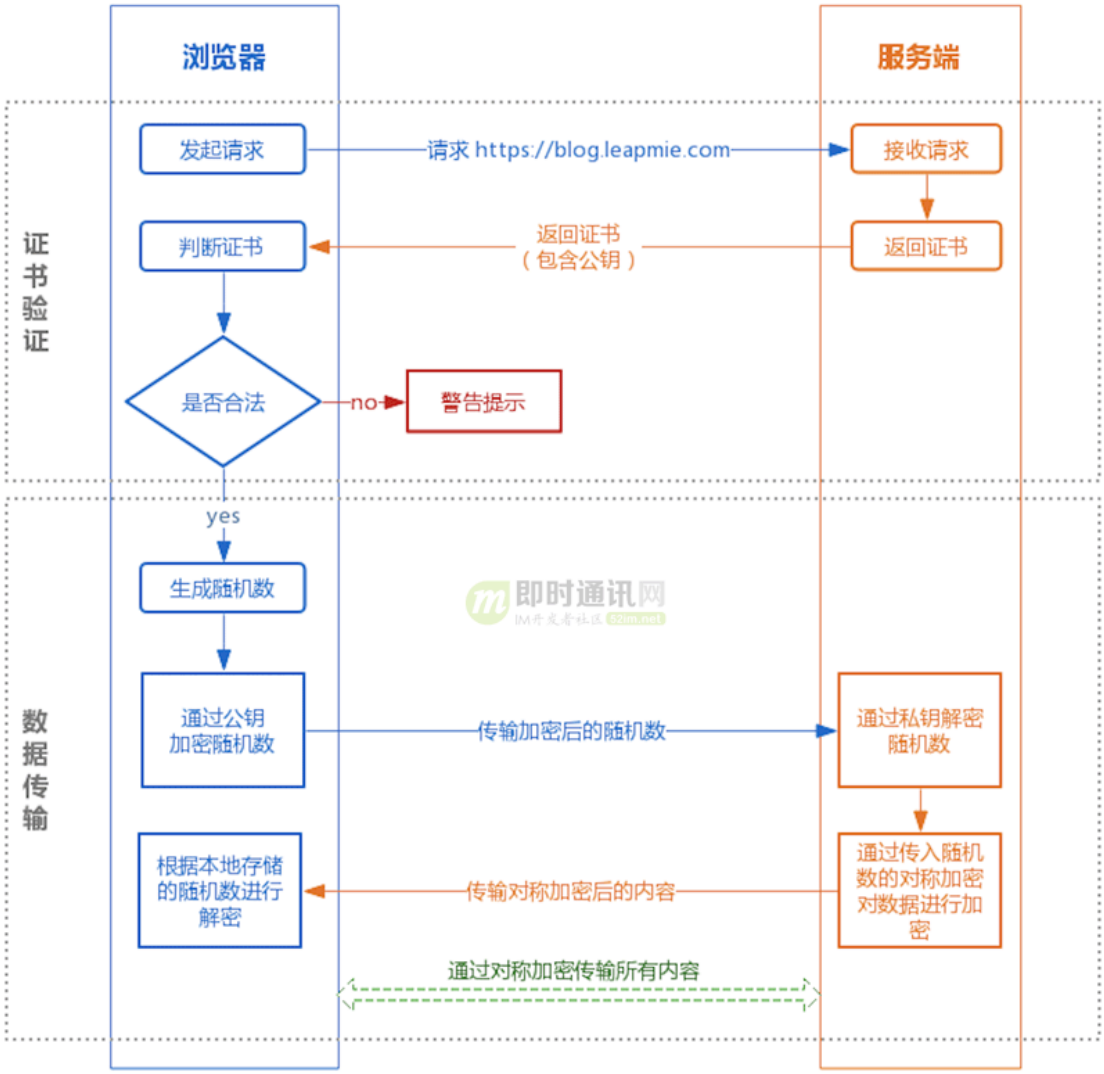
HTTP2

多路复用，数据可并行传输，使用帧为数据标记顺序，以二进制流传输，提高资源传输响应。

博客： https://www.cnblogs.com/imteck4713/p/12016313.html
HTTP: 数据明文传输，没有身份认证和加密 80
HTTPS:443
0 服务端 Server 会在第三方机构 CA 处提交公钥、组织信息、个人信息等信息并申请认证；CA 通过线上、线下等方式验证申请后会对申请者签发认证文件，即证书,证书中包含了申请者公钥、申请者各种其他信息和签发机构的 CA 信息、有效期等，同时包含签名
1 客户端向服务端发起 HTTPS 请求时，server 会先将自己的 CA 证书返回给 client，client 读取证书验证证书的合法性（客户端使用散列函数计算公开的明文信息的摘要，并利用 CA 的公钥对签名进行解密，对比证书的信息摘要若一致，则说明证书合法，即公钥合法），同时客户端还会验证证书相关的域名信息、有效期等等。客户端会内置信任的 CA 的证书信息，包含公钥，若 CA 不被信任，则找不到对应 CA 的证书，CA 证书会被认为非法。
签名产生算法：使用散列函数（哈希）计算公开的明文信息的摘要，并采用 CA 的私钥对信息摘要进行加密，密文即签名；
2 客户端使用非对称加密的方式与服务器进行通信，生成随机数，协商对称加密传输数据使用的密钥。
客户端使用服务端的公钥加密 对称密钥 的协商数据，服务端用私钥解密；服务端使用解密后随机数生成对称加密算法，对响应内容对称加密，传输给客户端，
4. 双方使用对称加密传输所有内容。

数据传输阶段使用对称加密而不使用非对称加密的原因是，非对称加密的加解密

效率非常低，http 的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的。另外：在 HTTPS 的场景中只有服务端保存了私钥，一对公私钥只能实现单向的加解密，所以 HTTPS 中内容传输加密采取的是对称加密，而不是非对称加密。



Linux 启动过程

Nginx 负载均衡策略

从 upstream 模块定义的后端服务器列表中选取一台服务器接收用户的请

求。#动态服务器组

```
upstream dynamic_zuoyu {
    server localhost:8080;  #tomcat 7.0
    server localhost:8081;  #tomcat 8.0
    server localhost:8082;  #tomcat 8.5
    server localhost:8083;  #tomcat 9.0
}
```

有 6 中方式的分配策略：

1. 轮询，将请求均匀分配给不同的服务器
2. 权重，按服务器性能不同赋予不同的权重，权重越高分配到处理的请求越多
3. Ip_hash，根据请求 ip 即客户端 IP 的 Hash 值分配处理的服务器，即相同的客户端的请求会被发送到同一个服务器处理
4. Least_conn，最少连接方式，即处理请求最少的服务器
5. url_hash，请求的 url 的 Hash 值分配处理服务器
6. fair， 响应时间方式

心跳机制

UCP 与 TCP 区别

1. 基于连接与无连接；
2. 对系统资源的要求（TCP 较多，UDP 少）；
3. UDP 程序结构较简单；
4. 流模式与数据报模式 ；
5. TCP 保证数据正确性，UDP 可能丢包，TCP 保证数据顺序，UDP 不保证。

TCP 拥塞控制

发送方维持一个拥塞窗口变量

如何辨别网络出现拥塞：超时或出现 3 个冗余 ACK

拥塞控制算法：加性增乘性减、慢启动

慢启动：TCP 连接建立初始阶段，拥塞窗口 congWin 大小为 1，TCP 发送方在初始阶段不是线性增加发送速率的，而是以指数的速度增加，没经过一个 RTT 将 congWin 的值翻倍，直到发生一个丢包事件为止，此时 congWin 将被降为一半，

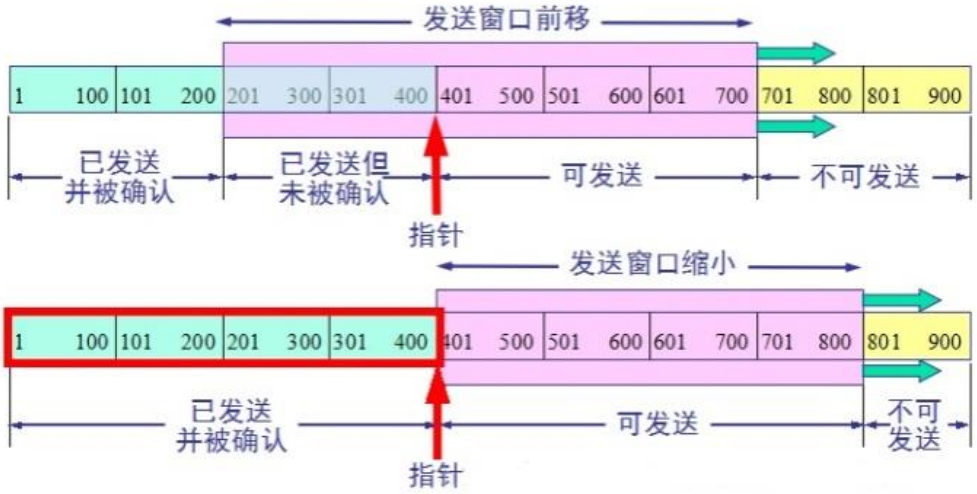
然后开始线性增长。

乘性减：每发生一次丢包事件将当前 CongWin 值减半
加性增：每收到一个 ACK 确认包后将 CongWin 增大一点

对因超时和收到 3 个冗余 ACK 而检测到的丢包事件作出的反应是不同的，
收到 3 个冗余 ACK 后，拥塞窗口减半，开始线性增长
超时事件发生，TCP 发送方进入慢启动阶段，congWin 被设为 1，然后窗口指数增长，直到 congWin 达到超时事件前窗口值得一半为止，
阈值：每当发生一个丢包事件，Threshold 值会被设置为当前 CongWin 的一半。
因此，TCP 发送方在发生超时事件后进入慢启动阶段，慢启动阶段 congWin 以指数快速增长，直至 congWin 达到阈值 Threshold 为止，当 congWin 达到阈值时，TCP 就进入拥塞避免阶段，在该阶段中，congWin 线性增长。

TCP 的窗口如何设置

https://blog.csdn.net/weixin_39966130/article/details/111108350?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-3.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-3.control



发送窗口中相关的有四个概念：已发送并收到确认的数据（不再发送窗口和发送缓冲区之内）、已发送但未收到确认的数据（位于发送窗口之中）、允许发送但尚未发送的数据以及发送窗口外发送缓冲区内暂时不允许发送的数据；

流量控制是通过接收方传递窗口大小给发送方，根据这个接收窗口大小控制数据发送速率的。

滑动窗口需掌握的知识点：

A、数据接收端将自己可以接受的缓冲区大小放入 TCP 首部中“窗口大小”字段，通过 ACK 来通知数据传输端。B、窗口大小字段越大，说明网络的吞吐率越高
C、窗口大小指的是**无需等待确认应答而可以继续发送数据的最大值**，即就是说不需要数据接收端的应答，可以一次连续的发送数据。
D、操作系统内核为了维护滑动窗口，需要开辟发送缓冲区，来记录当前还有哪些数据没有应答，只有确认应答过的数据，才能从缓冲区删除。PS：发送缓冲区如果太大，会有空间开销。
E、数据接收端一旦发现自己的缓冲区快满了，就会将窗口大小设置成一个更小的值通知给数据传输端，数据传输端收到这个值后，就会减慢自己的发送速度。
F、如果数据接收端发现自己的缓冲区满了，就会将窗口大小设置为 0，此时数据传输端不再传输数据，但是需要在定期发送一个窗口探测数据段，使数据接收端把窗口大小告诉数据传输端。
PS：在 TCP 的首部，有一个 16 为窗口字段，此字段就是用来存放窗口大小信息的。
TCP 报文段发送时机的选择
1、TCP 维持一个变量，它等于最大报文段长度 MSS，只要缓存中存放的数据达到 MSS 字节就组装成一个 TCP 报文段发送出去。
2、由发送方的应用程序指明要求发送报文段，即 TCP 支持的推送操作。
3、是发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段发送出去。

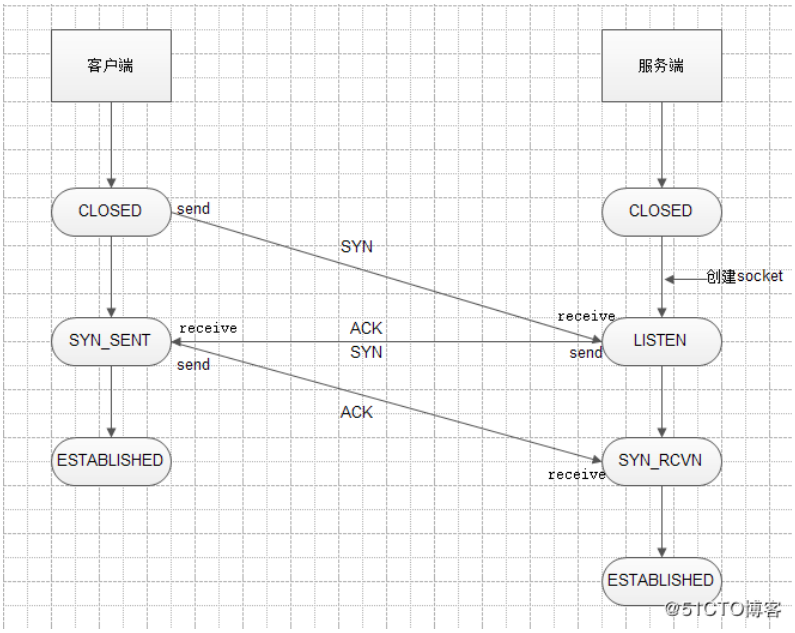
SYN 泛洪攻击

攻击者发送大量的 TCP SYN 报文段，而不完成三次握手的第三步，通过从多个源发送 SYN 能够加大供给力度，从而形成 DDos 分布式拒绝服务 SYN 泛洪攻击。随着这种 SYN 报文的到来，服务器要不断为这些半开连接分配资源，结果导致服务器连接资源迅速消耗殆尽。半连接队列爆满，导致正常连接无法进行。
半连接队列，未完成第三次握手的半连接会进入半连接队列

全连接队列，第三步 server 端收到第三次握手时建立起连接，将这条连接从半连接队列中取出 push 到全连接队列中。

三次握手和四次挥手状态

1、一开始，建立连接之前服务器和客户端的状态都为 CLOSED；
2、服务器创建 socket 后开始监听，变为 LISTEN 状态；
3、客户端请求建立连接，向服务器发送 SYN 报文，客户端的状态变味 SYN_SENT；
4、服务器收到客户端的报文后向客户端发送 ACK 和 SYN 报文，此时服务器的状态变为 SYN_RCVD；
5、然后，客户端收到 ACK、SYN，就向服务器发送 ACK，客户端状态变为 ESTABLISHED；
6、服务器端收到客户端的 ACK 后变为 ESTABLISHED。此时 3 次握手完成，连接建立！



由于 TCP 连接是全双工的，断开连接会比建立连接麻烦一点点。

1、客户端先向服务器发送 FIN 报文，请求断开连接，其状态变为 FIN_WAIT1；

2、服务器收到 FIN 后向客户端发送 ACK，服务器的状态围边 CLOSE_WAIT；

3、客户端收到 ACK 后就进入 FIN_WAIT2 状态，此时连接已经断开了一半了。如果服务器还有数据要发送给客户端，就会继续发送；

4、直到发完数据，就会发送 FIN 报文，此时服务器进入 LAST_ACK 状态；

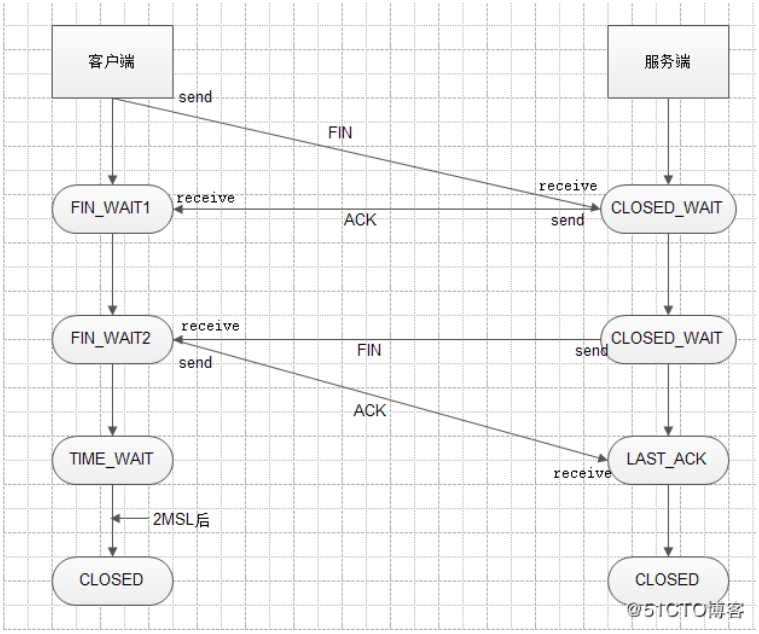
5、客户端收到服务器的 FIN 后，马上发送 ACK 给服务器，此时客户端进入 TIME_WAIT 状态；

6、再过了 2MSL 长的时间后进入 CLOSED 状态。服务器收到客户端的 ACK 就进入 CLOSED 状态。

至此，还有一个状态没有出来：CLOSING 状态。

CLOSING 状态表示：

客户端发送了 FIN，但是没有收到服务器的 ACK，却收到了服务器的 FIN，这种情况发生在服务器发送的 ACK 丢包的时候，因为网络传输有时会有意外。



LISTEN：等待从任何远端 TCP 和端口的连接请求。
SYN_SENT：发送完一个连接请求后等待一个匹配的连接请求。
SYN_RECEIVED：发送连接请求并且接收到匹配的连接请求以后等待连接请求确认。
ESTABLISHED：表示一个打开的连接，接收到的数据可以被投递给用户。连接的数据传输阶段的正常状态。
FIN_WAIT_1：等待远端 TCP 的连接终止请求，或者等待之前发送的连接终止请求的确认。
FIN_WAIT_2：等待远端 TCP 的连接终止请求。
CLOSE_WAIT：等待本地用户的连接终止请求。
CLOSING：等待远端 TCP 的连接终止请求确认。
LAST_ACK：等待先前发送给远端 TCP 的连接终止请求的确认（包括它字节的连接终止请求的确认）
TIME_WAIT：等待足够的时间过去以确保远端 TCP 接收到它的连接终止请求的确认。
TIME_WAIT 两个存在的理由：
1.可靠的实现 tcp 全双工连接的终止；
2.允许老的重复分节在网络中消逝。
CLOSED：不在连接状态（这是为方便描述假想的状态，实际不存在）

对象存储结构：对象头、实例变量

对象头， 含 4 个字节的 markWord 标记字段 和 4 字节的 class 类型对象指针，指向该对象对应的 java.lang.class 对象的地址

标记位：存储**如哈希码、 对象的 GC 分代年龄、锁状态标志、**轻量级锁标记位、偏向锁标记位等、**线程持有的锁、偏向的线程 ID、**偏向时间戳等、（标志位是不固定的 bitmap）

若对象是数组类型，还需要三个机器码记录数组长度，

类型指针：指向该对象对应的 java.lang.class 对象的地址

对象头信息是与对象自身定义的数据无关的额外存储成本，设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据。

实例变量， 含实例所有的成员变量，大小由各个成员变量大小决定，如 byte 和 boolean 是 1 个字节，short 和 char 是 2 个字节，int 和 float 是 4 个字节，long 和 double 是 8 个字节，reference 是 4 个字节

对齐填充位， 按 8 个字节的整数倍填充

这里要特别注意一下标志位，是不固定的数据结构，随着对象的状态不同而变化。例如在 32 位的 HotSpot 虚拟机 中对象未被锁定的状态下，Mark Word 的 32 个 Bits 空间中的 25Bits 用于存储对象哈希码（HashCode），4Bits 用于存储对象分代年龄，2Bits 用于存储锁标志 位，1Bit 固定为 0，在其他状态（轻量级锁定、重量级锁定、GC 标记、可偏向）

32位虚拟机			
25Bit	4Bit	1Bit	2Bit
对象的hashCode	对象的分代年龄	是否是偏向锁	锁标志位

无锁状态：markWord – 对象的 HashCode + 对象的分代年龄 + 状态位 001

Synchronized：

每个对象都自带了一把锁或者说内部锁 Monitor ，每个上锁的对象都关联了一个 Monitor 对象

Monitorenter – 同步尝试获得锁 Monitorexit – 释放锁

https://blog.csdn.net/wj1314250/article/details/111694820

锁的状态：无锁、偏向锁、轻量级锁、重量级锁，这些锁状态会随着竞争的激烈而逐渐升级。锁可以升级但不可以降级，锁升级优化是为了减少锁竞争带来的上下文切换。

Synchronized 同步锁的升级优化路径：偏向锁->轻量级锁->重量级锁

偏向锁：用来优化同一线程多次申请同一个锁的竞争，在某些情况下，大部分时间都是同一个线程竞争锁资源

当一个线程再次访问同一个同步代码块时，该线程只需要判断该对象头的 Mark word 中是否有偏向锁指向它，即线程 ID，而无需在进入 Monitor 竞争对象（这可以避免用户态和内核态的切换），

当对象被作为同步锁时且有一个线程抢到锁时，锁的标志位仍为 01，但是否偏向锁标志位置为 1，且记录抢到锁的线程 ID，进入偏向锁状态

当线程 1 再次获取锁时，会比较当前线程 ID 与锁对象头中标志位的线程 ID 是否一致，若一致，则无需 CAS 来抢占锁，否则需要查看对象头中 Markword 记录的线程是否存活，若没有存活，锁对象被重置为无锁对象（也是一种撤销），然后对象锁重新偏向线程 2，如果存活，则查找线程 1 的栈帧信息，若线程 1 仍需要继续持有该锁对象，那么要暂停线程 1，撤销偏向锁，升级为轻量级锁，若线程 1 不再使用该对象锁，则将该锁对象置为无锁状态，然后重新偏向线程 2；一旦出现其他线程竞争锁资源时，偏向锁就会被撤销。偏向锁的撤销可能需要等待全局安全点，暂停持有该锁的线程，同时检查该线程是否还在执行该方法。如果还没有执行完，说明此刻有多个线程竞争，升级为轻量级锁；如果已经执行完毕，唤醒其他线程继续 CAS 抢占；在高并发场景下，当大量线程同时竞争同一个锁资源时，偏向锁会被撤销，发生 STW，加大了性能开销

当有另外一个线程竞争锁时，由于当前的对象锁处于偏向锁状态，且发现对象头 Mark Word 中的线程 ID 不是自己的线程 ID，该线程就会执行 CAS 操作 获取锁，若获取成功，直接替换 MarkWord 中的线程 ID 为自己的线程 ID，该锁保持偏向锁状态，若获取失败，说明当前锁有一定竞争，偏向锁升级为轻量级锁。线程获取轻量级锁时分两步：

先复制锁对象头中 MarkWord 一份到线程的栈帧中（DisplacedMarkWord），主要为了保留现场

使用 CAS，把对象头中的内容替换为线程栈中 DisplacedMarkWord 的地址

在线程 1 复制对象头 Mark Word 的同时（CAS 之前），线程 2 也准备获取锁，也复制了对象头 Mark Word；在线程 2 进行 CAS 时，发现线程 1 已经把对象头换了，线程 2 的 CAS 失败，线程 2 会尝试使用自旋锁来等待线程 1 释放锁；轻量级锁的适用场景：线程交替执行同步块，绝大部分的锁在整个同步周期内都不存在长时间的竞争

轻量级锁 CAS 抢占失败，线程将会被挂起进入阻塞状态，如果正在持有锁的线程在很短的时间内释放锁资源，那么进入阻塞状态的线程被唤醒后又要重新抢占锁资源，JVM 提供了自旋锁，通过自旋方式不断尝试获取锁，避免线程被挂起阻塞，自旋锁重试达到一定的次数之后若依然抢锁失败，则锁会升级至重量级锁，锁标志位置为 10. 在这个状态下，未抢到锁的线程都会进入 Monitor，之后会被阻塞在 WaitSet 中，

在锁竞争不激烈且锁占用时间非常短的场景下，使用自旋锁可以提高系统性能，一旦锁竞争激烈或者锁占用的时间过长，自旋锁将会导致大量线程一直处于 CAS 重试状态，占用 CPU 资源。在高并发场景下，可通过关闭自旋锁来优化系统性能。

小结：

1.JVM 在 JDK1.6 中引入了分级锁机制来优化 synchronized

2.当线程获取锁时，首先对象锁成为一个偏向锁，

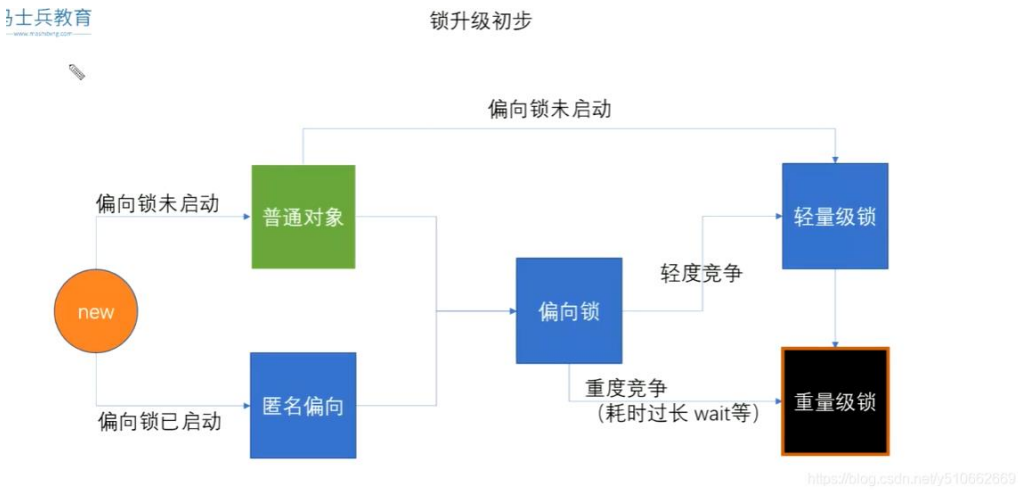
这是为了避免同一个线程重复获取同一把锁时，用户态和内核态的频繁切换

4. 若有多个线程竞争锁资源，锁将会升级为轻量级锁

这适用于在短时间内持有锁，且分锁交替切换的场景，轻量级锁还结合了自旋锁来避免线程用户态和内核态的频繁切换

5 如果锁竞争太激烈（自旋锁失败），同步锁会升级为重量级锁、、

6 优化 synchronized 同步锁的关键：减少锁竞争



1) 当 new 一个对象，偏向锁未启动时，产生的是轻量级锁，即 cas 自旋锁，当竞争愈发激烈，假如说有一万个线程在做 cas 操作，那么会消耗大量的 cpu 资源，这个时候就会升级成重量级锁，由操作系统来统一管控，得不到锁的统一放到一个队列里面去阻塞等待。

2) 当 new 一个对象，偏向锁已经启动，默认偏向锁会在系统启动后延迟启动，这个时候产生的是匿名偏向对象，再升级到偏向锁，通俗的说偏向锁，就是第一个来的线程，直接在 markword 内即一个标记而已，后面来的线程越来越多，竞争愈发激烈，升级成轻量级锁，就是多个线程在进行 cas 操作（cas 操作过多会影响 cpu 资源），看谁能争抢到记录这个标记，如果争抢的线程越来越多，会进一步升级，升级为重量级锁，这个时候需要操作系统出面调控，让其他竞争的线程进行阻塞排队等待，停止自旋操作。

Synchronized 锁优化或者说锁升级过程

锁状态	31bit(25bit unused)		4bit	1bit	2bit
	54bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向轻量级锁的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

CAS： compare and swap 比较并交换

CAS 直接将当前期望值与内存中的当前值比较，若相等认为没有别的线程修改过，将内存中的值更新为新的值；若不相同认为有别的线程更改过，则重新取出内存中的当前值作为新的期望值，并再次进行 CAS 比较。

CAS 是一条原子操作的指令，不可拆分。

CAS 中有 ABA 的问题，通过时间戳，加一个时间戳字段或者版本号

java.util.concurrent.atomic.AtomicStampedReference

JVM 内存区域划分：

线程共享：方法区、堆

方法区：存储类的信息、常量池、类方法、类变量等

JDK1.6、1.7 时方法区就是永久代，JDK1.8 方法区升级成为元空间

元空间和永久代最大的区别在于：元空间并不再虚拟机中，而是使用本地内存。

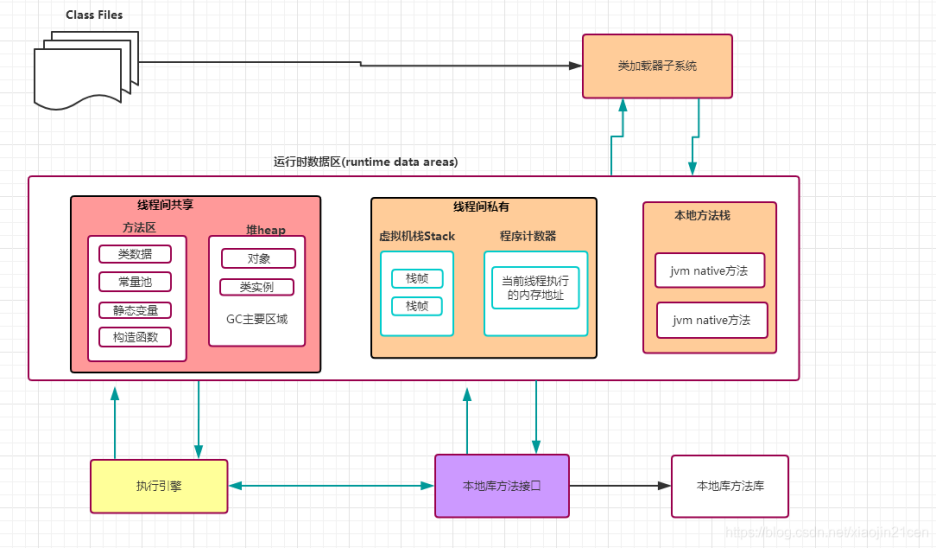
堆：存储对象实例信息、GC 回收、OOM 溢出

线程私有：程序计数器、虚拟机栈、本地方法栈

程序计数器：记录当前线程执行的字节码的行号指示器。

虚拟机栈：线程执行函数的栈帧，记录了局部变量表、操作数、方法动态链接及方法返回等信息

本地方法栈：线程执行 native 本地方法时的栈帧



JVM 可达性分析：

可达性分析算法：通过一系列的名为“GC Root”的对象作为起点，从这些节点向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到 GC Root 没有任何引用链相连时（或者该对象不是 GCRoot 对象时），则认为该对象不可达，后面该对象将会被垃圾收集器回收其所占的内存。

什么对象可作为 GCRoot 的对象？

a. java 虚拟机栈中的引用的对象。

b.方法区中的类静态属性引用的对象。（一般指被 static 修饰的对象，加载类的时候就加载到内存中。）

c.方法区中的常量引用的对象

d.本地方法栈中的 JNI（native 方法）引用的对象

堆内存：

1. 为啥要有 survivor 区，新生代使用复制的垃圾回收算法，若不设 survivor 区，存活的对象将被复制到老年代，这样导致老年代频繁的 full GC，所以在新生代设了 survivor 区。通过 survior 区也保证经历过默认是 15 次 minorGC 的对象才能进入老年代，对象的分代年龄存放在对象头的 markword 中。每经历过一次 minorGC 的对象分代年龄+1，“老不死的”

对象，再呆在幸存区没有必要（而且老是在两个幸存区之间反复地复制也需要消耗资源），才会把它转移到老年代。

2. 新生代使用复制算法的原因是因为大多数对象都是朝生夕灭的，生命周期比较短，少数对象存活，使用复制算法复制少数存活对象到新内存中效率比较高。
3. 为什么使用两个 survivor 区，而不是一个？建立两块 Survivor 区，刚刚新建的对象在 Eden 中，经历一次 Minor GC，Eden 中的存活对象就会被移动到第一块 survivor space S0，Eden 被清空；等 Eden 区再满了，就再触发一次 Minor GC，Eden 和 S0 中的存活对象又会被复制送入第二块 survivor space S1（这个过程非常重要，因为这种复制算法保证了 S1 中来自 S0 和 Eden 两部分的存活对象占用连续的内存空间，避免了碎片化的发生）。S0 和 Eden 被清空，然后下一轮 S0 与 S1 交换角色，如此循环往复。如果对象的复制次数达到 16 次，该对象就会被送到老年代中。
4. Eden:survior 默认使用 8：1 是因为新生对象都是在 Eden 区中创建，而大部分对象都是朝生夕灭，GC 的时候只有少部分对象存活。

Java 对象从创建到消亡的过程？

1.Student stu = new Student(“zhangsan”); 运行时会先在方法区中寻找 Student 类的类型信息，若没有会使用类加载器将 Student 类的字节码文件 Student.class 加载至内存中的方法区，并将 Student 类的类型信息存放至方法区。

2.JVM 在堆中为新的 Student 实例分配内存空间，对象投+实例数据+填充

3.stu 是个引用，是在当前执行线程虚拟机栈上的，指向堆中的 student 对象

4.程序执行使用对象

5.使用结束后 stu=null，置空

6.JVM GC，这里 GC 时会查看当前对象类是否重写了 finalize 方法，若没有重写，直接 GC 回收，若重写了，查看 finalize 方法是否被执行，若被执行则直接回收，没有执行则进入 F-QUEUE 队列，等待执行该对象类的 finalize 方法后再次被 GC 判断是否可达，不可达则被回收，否则对象复活。finalize 使得对象有复活的机会。

操作系统内存管理：

https://www.cnblogs.com/myseries/p/12487211.html

虚拟内存：虚拟内存是计算机内存管理的一种技术，这种技术使得应用程序认为它使用了连续的内存空间（即一个连续完整的地址空间），实际上，它通常是被由不连续的物理内存碎片形成的，还有部分暂时存储在外部磁盘上。

虚拟内存：将地址空间定义为连续的虚拟内存地址，使其认为自己正在使用一大块连续地址。但实际上，这一大块连续的虚拟内存是被映射在多个不连续的物理

内存碎片上的，还有部分映射到了外部磁盘存储器上。

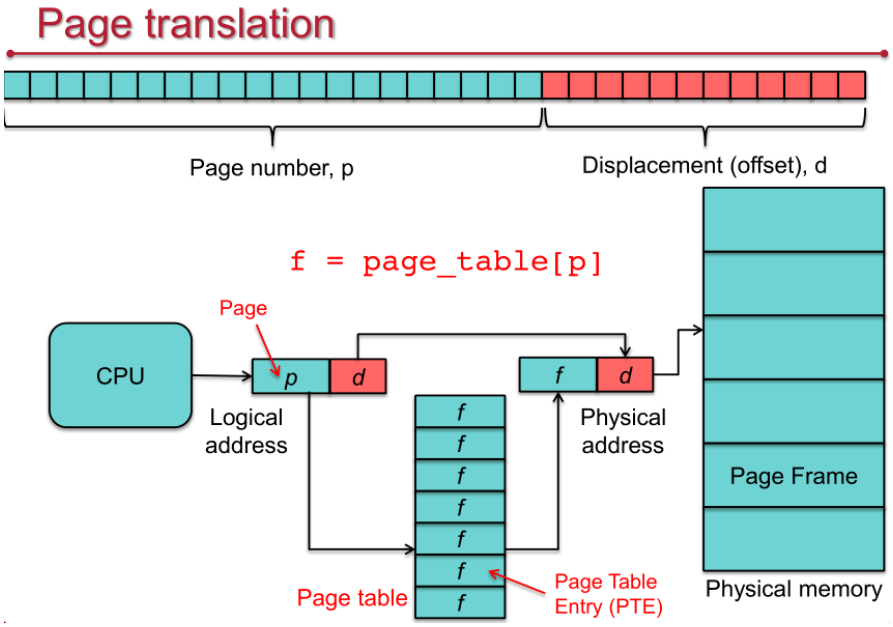
思想：每个程序都有自己的地址空间，这个空间被分割成多个块，每一个块被称作一页或页面。每一页有连续的地址范围。这些页被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行必要的映射。当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令。

1. 虚拟内存地址空间是连续的、没有碎片
2. 虚拟内存的最大空间是 CPU 的最大寻址空间，不受内存大小限制，能提供比内存更大的地址空间

SWAP：

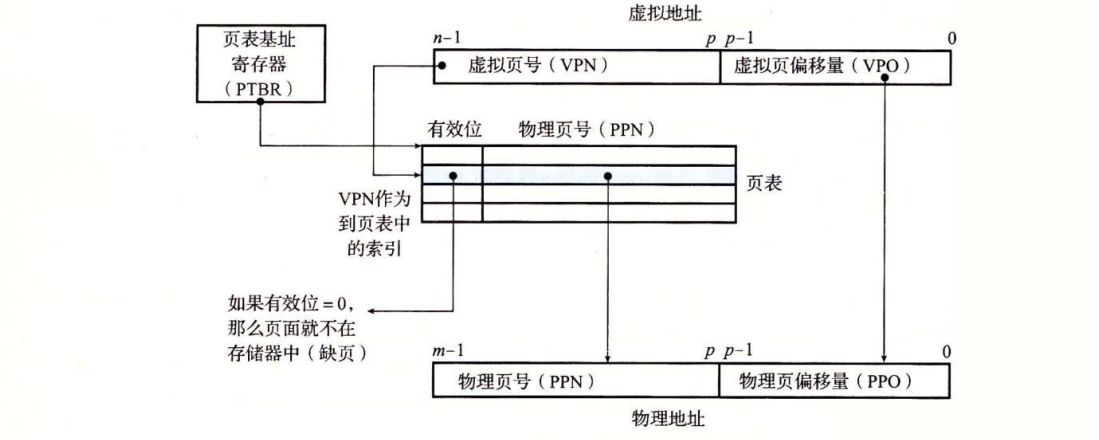
原理：虚拟内存被分成大小固定的称作虚拟页的块，物理内存也被分成物理页，虚拟内存
虚拟地址（虚拟页、偏移地址） –> 页表 –> 物理地址（物理页+虚拟偏移地址）
页表存储的是虚拟页 到 物理块的映射，物理块作为物理地址高位，虚拟地址中的偏移地址作为物理地址低位。

页表的有效标识位标志虚拟页是否被缓存在主存中，若为 1 说明该虚拟页已被缓存在主存中，若位 0 分两种情况，可能是虚拟内存还未创建虚拟页，也有可能是已创建虚拟页但还没被缓存到主存，所以虚拟页集合由 3 个子集组成：1.虚拟内存系统还未分配或未创建 2.已缓存在物理内存种的已分配的虚拟页 3.未缓存在物理内存中但已被创建或分配的虚拟页。

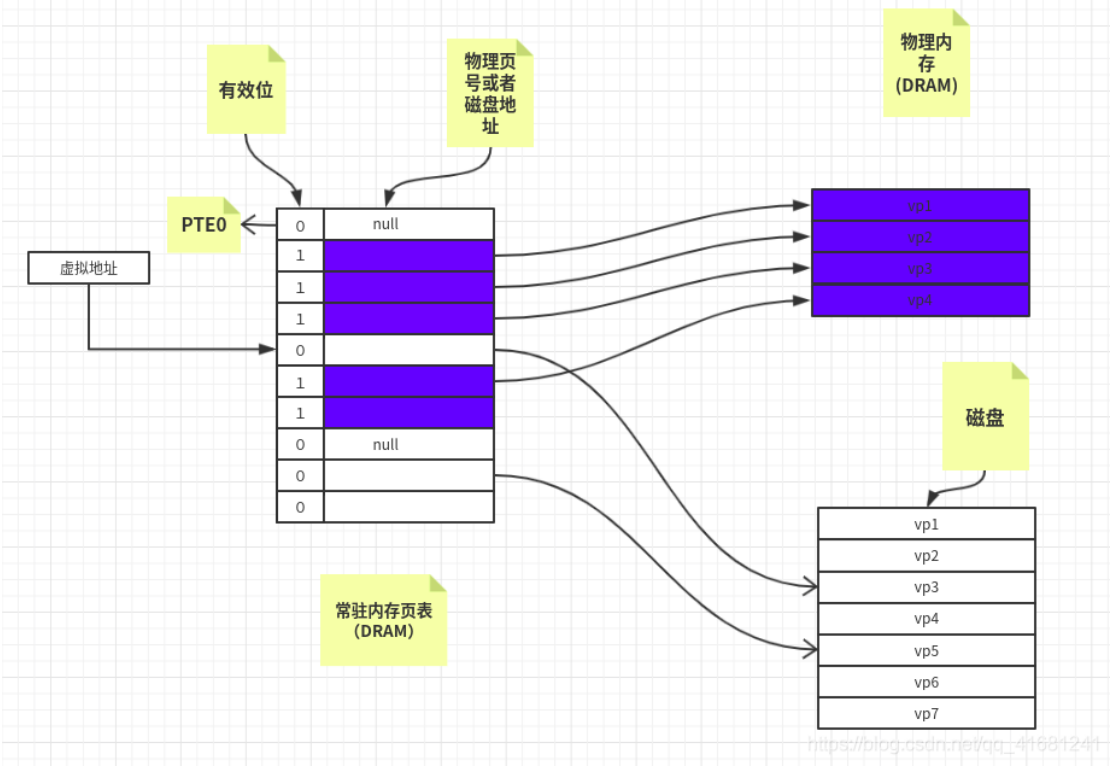


MMU 将虚拟地址映射为物理地址的详细过程：CPU 产生虚拟地址，然后从虚拟地址中得到虚拟页号，接下来通过将虚拟页号作为索引去查找页表，通过得到对

应 PTE 上的有效位来判断当前虚拟页是否在主存中，若命中则将对应 PTE 上的物理页号和虚拟地址中的虚拟页偏移进行串联从而构造出主存中的物理地址，否则未命中（专业名词称为“缺页”），此时 MMU 将引发缺页异常，从 CPU 传递到操作系统内核处理缺页异常处理程序，此时将选择一个牺牲页并将对应所缺虚拟页调入并更新页表上的 PTE，缺页处理程序再次返回到原来的进程，再次执行缺页指令，CPU 重新将虚拟地址发给 MMU，此时虚拟页已存在物理内存中，所以命中，最终将请求的字返回给处理器。详细过程如下：



虚拟内存 可以保证进程由各自独立的地址空间，保证了数据访问的安全性，



进程间的通信方式：

https://www.cnblogs.com/sunsky303/p/13589696.html

进程通信：每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据通过内核，在内核中开辟一块缓存区，进程进程 A 把数据从用户空间拷到内核缓冲区,进程 B 再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信。

1. 匿名管道通信：
2. 高级管道通信
3. 有名管道通信
4. 消息队列通信
5. 信号量通信
6. 信号通信
7. 共享内存通信
8. 套接字通信

项目：

1. 秒杀

描述：千万级用户秒杀商品，将参与秒杀的商品提前放入 redis 队列中，秒杀的时候每有一个用户发起秒杀请求，将对应商品的 list 左 pop 一个元素，pop 成功则表示抢到，并将此用户 id 存入 set 集合中，防止超买，否则 pop 出为空表示没有秒杀到，秒杀失败了。

String 类型，**Key-value，可进行自增自减**

redis 设计了一种简单动态字符串(SDS[Simple Dynamic String])作为底实现：

定义 **SDS** 对象，此对象中包含三个属性：**String**

- len buf 中已经占有的长度(表示此字符串的实际长度)
- free buf 中未使用的缓冲区长度
- char buf[] 实际保存字符串数据的地方

Key - value (List) – > value1-> value2 -> value3

在 redis 中，list 类型是按照插入顺序排序的字符串链表（ziplist，linkedlist,quicklist）

Ziplist 在 redis 是将表中的每一项存放在前后连续的地址空间中，一个 ziplist 整体占用一大块内存，它是一个表，不是一个链表。由表头和 N 个 entry 节点和压缩列表尾部标识符 zlend 组成的一个连续的内存块，用于存储整数和比较短的字符。

链表 linkedlist 普通的双端链表

Quicklist -> 每个节点为 ziplist 的 linkedlist，整体上是 linkedlist，即双向链表结构，但每个 quicklist 节点是一个 ziplist，具备压缩列表的特性。

Key - value (hash) – >(key1,value1)->(key2, value2) -> value3

Ziplist

hashTable

Hash , (ziplist， hashtable)

Set, hashtable

Key ->value(set) -> (value1,value2,value3……)

Key - value (zset) – >(member,score)->(member, score)

ziplist

压缩列表结构示意图：

zlbytes	zltail	zllen	Entry1	Entry2	…	EntryN	zlend

压缩列表元素 entry 结构示意图

Previous_entry_length	encoding	content

每个 entry 的占用空间大小不是固定相同的，每个元素可以是一个字节数组或一个整数，encoding 字段标识的当前元素存储的数据类型和数据长度，编码时首先尝试将内容解析为整数，若解析成功，则按压缩列表整数类型编码存储；若解析失败，则按照压缩列表字节数组类型编码。

ziplist 插入元素时，先对新插入的元素进行编码，重新分配空间，复制数据，插入新元素，并更改 next 元素相应的 previous_entry_len 字段，更新 zltail 尾元素的偏移量。

压缩列表本质上是一个字节数组，是为了节约内存而设计的一种线性存储结构，redis 的有序集合、散列表和列表都都会直接或间接地使用了压缩列表，当有序集合或散列表中的元素个数比较少，且元素都是短字符串时，redis 使用压缩列表作为其底层数据存储结构。

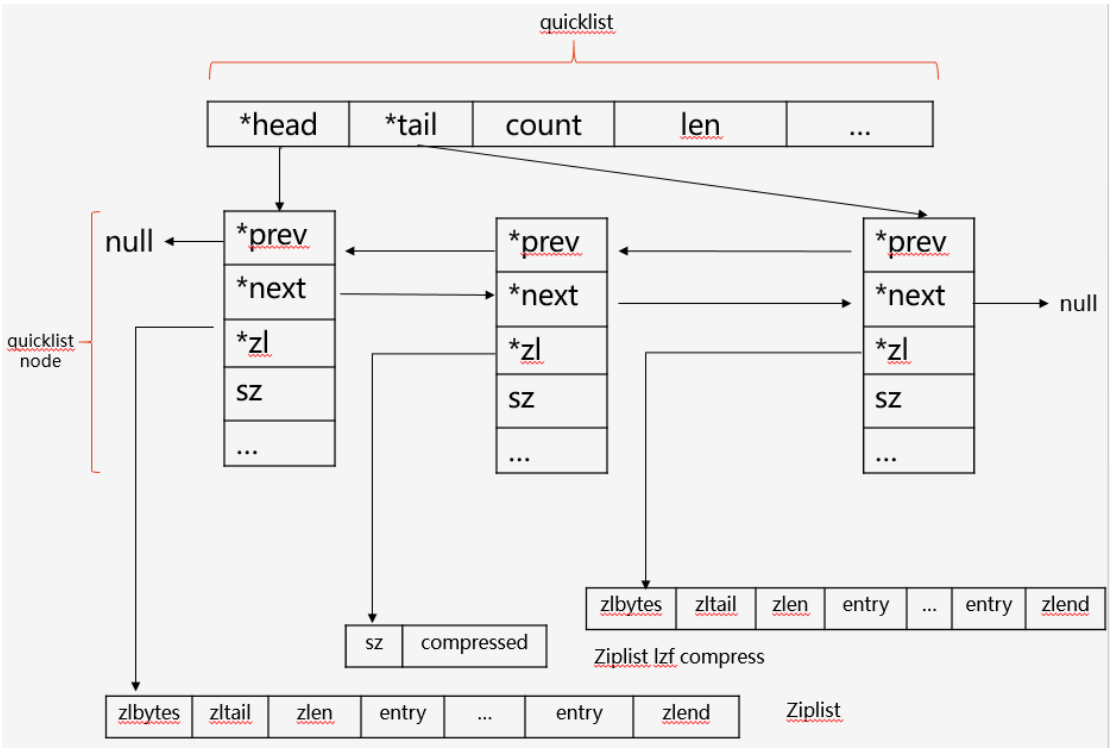
Redis3.2 版本引入 quicklist, 3.2 及之后列表使用 quicklist 数据结构存储，而 quicklist 是 ziplist（压缩列表）和 list（双向链表）的组合。

Redis3.2 版本之前的列表，当元素个数比较少且元素长度比较小时，采用 ziplist 作为底层存储；当任意一个条件不满足时，redis 使用 adlist 作为底层存储。这么做的原因是：当元素长度小时，使用 ziplist 可有效节省空间，但 ziplist 的存储空间是连续的，若元素个数比较多时，修改元素时，需要重新分配存储空间，会影响 redis 的执行效率。

Quicklist, a doubly linked list of ziplist

quicklist 是一个双向链表，链表中的每个结点是一个 ziplist 结构，即用双向链表将若干小型的 ziplist 连接到一起组成的一种数据结构。

考虑到 quicklistNode 结点个数较多时，由于经常访问的是两端的元素（redis 中 list 列表经常使用的操作是 lpush lpop rpush rpop），为进一步节省空间，redis 允许对中间的 quicklistNode 节点进行压缩，通过修改参数 list-compress-depth 进行配置。



Hash 散列表的底层存储：

1. ziplist （所有键值对的字符串长度均小于指定值或散列对象保存的键值对个数小于指定值时）
2. hashtable

ZSet 有序集合

1. ziplist
2. dict+skiplist

商品减库存

静态数据优化

对于稳定数据，在一段时间内保持稳定不变的数据，通常使用两种方式处理：利用缓存或者静态化技术。如商品信息等，可使用 Hash 数据类型存储

RabbitMQ 异步消息队列流量削峰

生成订单，使用了 rabbitMQ 异步消息队列，用户秒杀成功以后使用 UUID 生成一个订单号并返回，将用户订单信息发送到消息队列中，Order 消费者即订单操作程序会将订单信息异步的持久化到数据库中，这里可以对消费者一次性处理的请求进行限流，即设置 prefetch 可以限制。

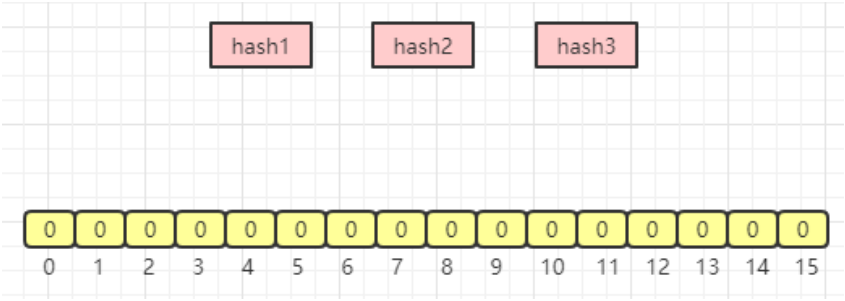
秒杀页面要定时（3 秒）checkoutOrder 是否处理好，若没有生成 Order 则依然在 wait…页面，若 Order 处理成功则进入订单详情页。

订单若在一定时间内没有被支付，那该订单关闭成为历史订单。

主动说 ，redis 缓存穿透、缓存击穿、雪崩及热键

缓存穿透：假设查询一条 id=3 在 redis 中不存在且在后端数据库中也不存在的数据，高并发下可使用布隆过滤器，判断，系统启动时将后端数据库中要被访问的信息 ID 根据多个 HASH 函数映射到布隆过滤器的相应位 bitmap，当有请求过来时先经过布隆过滤器的 hash 判断，若所有 hash 位均位 1，请求往后传，否则返回，说明数据库中不存在请求的记录。 或者在缓存中设置一个 null 空数据，并设置有效期，需要设置有效期有效期额的原因是如果过一会儿新增了一条记录为 3 的数据，如果缓存不设置过期时间，那么这条数据就永远获取不到。

布隆过滤器：布隆过滤器是一个 bit 数组，一个很长的 bit 数组和一系列的 hash 函数构成。**它能判断某个元素一定不存在或者是可能存在**。



缓存击穿：查询一条在 redis 中不存在但在后端数据库中存在的数据，这样使用 redis 互斥锁控制只能有一个线程处理段程序，若如果不再 redis 里边，上锁访问后端数据库，访问到返回并存储到 redis 中，这样下一个请求就无需再去访问数据库，没有获得锁就执行从 redis 中直接取数据的操作即可。也可以为热键设置不过期时间

缓存雪崩：设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。有效期=固定时间+随机时间

缓存预热：将一些可能经常使用的数据在系统启动的时候预先设置到缓存中，这样可以避免在使用到的时候先去数据库查询，如可以将一些可能成为热点的 key 添加到缓存中

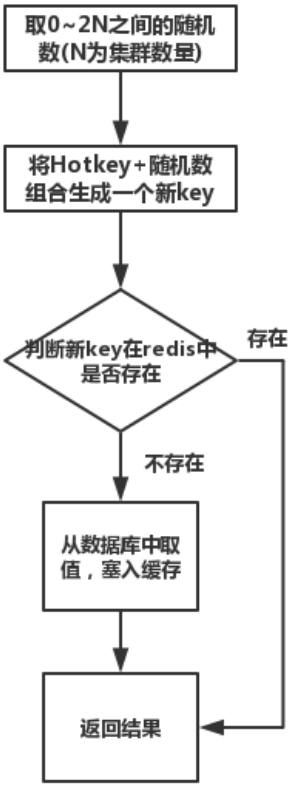
热 key（热数据）问题：瞬间有几十万的请求去访问 redis 上某个固定的 key，这样会造成流量过于集中，达到物理网卡上限，从而压垮缓存服务的情况，接下来这个 key 的请求，就会直接怼到你的数据库上，导致你的服务不可用。

发现热 key：（1）根据业务经验，进行预估那些可能成为热 key；如商品秒杀（2）客户端统计收集，本地统计或者上报；（3）若服务端有代理层，可在代理层进行收集上报

解决热 key：（1）利用二级缓存，可在本地缓存热 key，使用本地缓存，发现

热 key 后，将热 key 对应数据加载到应用服务器本地缓存中，访问热 key 数据时，直接从本地缓存中获取，而不会请求到 redis 服务器。（2）备份热数据，将热 key 在多个 redis 都存一份，有热 key 请求进来的时候，在有备份的 redis 上随机选取一台，进行访问取值并返回数据。将热 key 的 key 设计为 key+一个随机数，这样对请求的 key 进行 hash 以后落入不同的 redis server。热 key 备份，比如 key，备份为 key1,key2……keyN，同样的数据 N 个备份，N 个备份分布到不同分片，访问时可随机访问 N 个备份中的一个,进一步分担读流量。

假设 redis 的集群数量为 n，



Redis 的过期策略：

1. 定时删除，定时扫描设置过期时间的 key，但过期扫描不会遍历过期字典中所有的 key，而是采用一种简单的策略，1.从过期字典中随机扫描 20 个 key，2.删除这 20 个 key 中过期的 key，3.若 20 个 key 中过期的 key 占比超过 1/4，则重复步骤 1；redis 默认每秒扫描 10 次，即 100ms 一次，每个设置了过期时间 key 被存放在一个独立的字典中，定期扫描这个字典删除到期的 key
2. 惰性删除，客户端访问到这个 key 时，redis 对该 key 的过期时间进行检查，若过期了就立即删除，不会返回任何值。

定期删除可能会导致很多过期的 key 并没有被删除掉，所以有了惰性删除，若某个过期的 key，靠定期策略没有被扫描到，还在内存中，除非被系统访问下这个 key，才能被 redis 发现过期和删除，这就是惰性删除。

定期删除相当于集中处理，而惰性删除相当于零散处理。

Redis 淘汰策略：

1. noeviction: 返回错误，不驱逐任何键
2. allkeys-lru: 从所有 key 中删除最近最久未使用的部分 key
3. volatile-lru: 从设置了过期时间的 key 中删除最近最久未使用的部分 key
4. allkeys-random：从所有 key 中随机删除部分 key
5. volatile-random: 从设置了过期时间的 key 中随机删除部分 key
6. volatile-ttl: 从设置了过期时间的 key 中驱逐马上就要过期的 key
7. allkeys-lfu：从所有 key 中删除使用频率最少的 key
8. volatile-lfu: 从设置了过期时间的 key 中删除使用频率最少的 key

Nginx 环境下存在 session 不同步的问题

每个 session 默认存放在不同的后端服务器上，Nginx 将同一个客户端的请求负载均衡分发到不同的后端服务器，导致 session 不同步，为了 session 信息共享，将 session 存放在一个独立的 redis 服务器中，将 session 转存到 redis 中，即全局只存储一份，所有后端服务器共享这一份，实现 session 的共享。直接使用的 spring-session，导入依赖包，并在程序入口处注释 @EnableRedisHttpSession，这样会框架会自动监听将 session 存储到 redis 中，这样不同后端服务器共享了同一个 session，实现了 session 同步问题。

防重和幂等性：

幂等性：用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。

如何保证幂等性：

insert 前先 select （不适用于并发场景）、状态机，

更新时的 where 条件中包含了该条记录的状态码

1. 生成订单时的幂等性，根据用户 ID 和商品 ID 查看是否有这条记录，有则不操作，没有生成新订单。

   客户端可使用一个全局唯一的流水号 ID，用来表示是不是同一个请求或交易，让客户端生成这个 ID，每个抢购或生成订单的请求生成一个唯一 ID,并将 ID 保存到一个流水表中，设为唯一约束 UNIQUE KEY，若插入出现冲突，说明这个创建订单的请求已经被处理过了，直接返回之前的操作结果即可。

   当调用方携带流水号 ID 调用创建订单的接口，如果出现超时了，调用方不知道订单到底创建成功还是失败，这个时候，用同一个流水号进行重试，订单系统虽然收到了两个请求，但是由于流水号 ID 是同一个，可以根据流水表来做幂等操作。并告知对方订单创建成功与否。

2. 支付订单时的幂等性，支付订单时先查询当前订单的支付状态，若状态为待支付那直接支付，若已支付那就不要再支付了，这样保证幂等性，不会多支付。

   支付的时候产生一个唯一的支付订单号，支付接口保证幂等性，

电商

1. 动态页面静态化

   （1）影片信息是存储在数据库中的，准备首页和详情页模板，提前读取模板

和数据库影片信息，使用 io 流生成 html 静态文件存储，用户访问的时候直接返回 html 页面，而无需动态加载，
（2）当然，这里的影片信息也可以在系统启动时存放在 redis 中，请求来时直接从 redis 中取渲染页面，这样也可，无需到后端数据库读取
图片存储在一个独立的虚拟图片服务器上
（3）这些静态资源也可以选择在 CDN 内容分发，CDN 会根据用户线路及位置，为用户选择靠近的位置和相同的运营商，提升用户体验。即将用户的请求重新导向到离用户最近的服务结点上，使用户就近取得所访问的内容，提高用户访问网站的响应速度。

在 redis 中对象存储的三种方式：1.序列化 2.JSON 串 3.HASH 数据类型
https://blog.csdn.net/weixin_44242474/article/details/85244976?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-2.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-2.control

2. 多线程抢购同一张票或秒杀同一件商品
使用事务，select ticket … where state = .. for update，这会在数据库中对这行数据加上行锁，此时只能有一条线程访问，-付款-口库存-生成订单使用事务控制。
这里可以根据场次售票时间提前将该场次对应的票加入 redis 缓存，这样用户买票的时候直接从 redis 中取，由于每张票只有一张，我觉得可以将场次的票放入一个 set 中，场次 ID 为 key，ticket 的 id 为 set 中 value，当用户购票请求发来时，会携带场次和 ticket 的 id 信息，查询对应的场次 set 中是否存在该 id，存在则秒啥成功，从 set 中移除，不存在则表示该票已经售卖，这个售卖也要减库存-扣款等，

3. 跨域问题，安全 （浏览器同源策略的限制，不同源的资源不能共享，）
由于浏览器的安全限制，JS 只能访问与所在页面同一域（相同协议、域名、端口）的内容，通过 ajax 请求后端接口并返回数据，有时会受到浏览器的安全限制，产生跨域问题
JSONP，<script>中的 src 不受同源策略的限制，利用 srcipt 标签的 src 属性来实现，src 请求服务端，服务端返回的是 callback，执行函数与返回的数据。
服务端设置 add-allow-origin- 为*号

悲观锁

当要对数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制，在修改数据之前先锁定，再修改的方式被称之为悲观并发控制。悲观锁具有强烈的独占和排他特性

乐观锁

乐观锁是相对悲观锁而言的，乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回给用户错误的信息，让用户决定如何去做。乐观锁适用于读操作多的场景，这样可以提高程序的吞吐量。

乐观锁实现：

- CAS 实现：Java 中 java.util.concurrent.atomic 包下面的原子变量使用了乐观锁的一种 CAS 实现方式。
- 版本号控制：一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数。当数据被修改时，version 值会+1。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值与当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

博客：https://www.jianshu.com/p/d2ac26ca6525

i++不是原子操作，是三个原子操作的组合

1. 读取内存中的 i 值赋值给局部变量 temp，
2. 执行 temp+1 操作
3. 将 temp 复制给 i

所以如果两个线程同时执行 count++ 的话，不能保证线程一按顺序执行完上述三步后线程二才开始执行。

2. 自动化测试平台

任务

伴鱼：

0 之前面试体验怎么样

面试体验很好，伴鱼是我春招面试的第一家公司，匆匆准备了不到一周的时间，面试过程中很多基础忘记了，但还是感谢能给我面试的机会。一面过程主要是算法代码，在面试官的引导下终于斯出来了，面试过程中一些基础我知道自己回答的有点乱且不太好，一面面试官都会及时为我纠正一下正确的，体验真的很好，而且一面和二面的面试官都针对我的问题给出了很实用的建议，耐心的告诉我要坚持刷 leetcode，每日一题，二面面试官针对我实战少，提议我要做技术去一线城市实习，是这样的。而且还愿意给我一次机会，十分感谢。

1. **对互联网方面的了解，**

因为我还没有在真正的互联网实习过，所以对互联网的了解可能不是很充分。不过我认为现在互联网已经成为整个社会的基础设施，任何方面都离不开互联网，生活中处处都需要互联网的支撑。无论餐饮、娱乐、购物还是出行等各种场景都有互联网产品的支撑，互联网大大遍历了人们的生活，促进了社会进步。在互联网行业也有很好的就业前景，但是对人员的技术要求比较高，也需要较强的技术支撑，要有技术领头人，其实做互联网产品产生的职业成就感会让自己产生比较强的价值认同，而且做技术确实真的要趁年轻去到一线城市做技术也会让自己提升很快，但是需要自身的付出，无论精力还是其他方面。

2. 项目是自己做的还是实习
3. 平常技术怎么学，学习途径

4. 校园经历
   大学的时候，一个是清泽心雨，网站开发部门做前端，三农支教，
   现在班长，组织活动，研招办助管，研究生考试现场报名和考场安排等工作
5. 自己的职业规划
   刚毕业还是要提高自己的技术，不管如何考虑了半年还是决定先去一线城市发展，多学习多接触优秀的人，因为想做后端开发，我觉得后端虽然开发语言和开发框架很多，重要的还是思想，希望自己能够在以后的工作中多思考多总结，积累经验的同时学习别人的设计思想，而不仅仅只是完成工作而已。两年到三年的成长时间吧，希望一两年左右也能当小组长带新人，
6. 目前其他的进展怎么样
   嗯，伴鱼是进展最快的，因为 4 月初我才预答辩完，前段时间一直忙着准备毕业论文的事情，所以工作这块没有上心。伴鱼是我春招面试的第一家公司，前天晚上二面完感觉自己表现不太好以为要凉了，所以又在牛客上投了一些别家的简历，然后竟然意外的收到了伴鱼 hr 的通知。
7. 秋招没有参加么
   参加了，有互联网、银行和研究所的 offer，就是我个人想去北京，所以我找的是互联网，但是男朋友让去研究所，两个人一直谈不拢，我后来签了老家的研究所，但是我并不想去。
8. 期望薪资 不低于 20k 吧，虽然面试表现不太好，但是我觉得我值这个价
9. 确定毕业之后能来北京工作么~ 嗯是的，我正在努力找
10. 反问，嗯，我想问一下，如果幸运通过面试，公司会安排提前实习么~ 我是如果盲审和答辩顺利的话，7 月份 1 日左右拿到毕业证，5 月初预答辩完可能会稍微清闲一点，早点实习早点适应新环境。
11. 对于实习怎么看待的
    实习还是很有必要的，不管实习时间长短，或多或少都会让我们有额外的收获。但是，还是要找优质的公司去实习，做技术要去互联网，

12 缺点：我最大的缺点可能就是不够自信，有时候过于不够果断，导致有时候很多机会被错过。专注好当下吧，不要想别的，把手下的工作做好，多相信自己一点，还有就是要逐渐学会接受失败，从失败中找方法，复盘总结经验，为下一次的成功做准备，而不是把时间浪费在无意义的事情上。

13 优点：可能就是做一件事情不是那么容易轻易放弃，有时候可能做的不太好，但是心里放不下就会一直。

1. 长度为 n 的正整数数组中求和大于等于 s 的最短连续子序列长度
   思路：使用滑动窗口
2. 虚拟内存
3. TCP 三次握手、为何不用两次握手、SYN 泛洪攻击、状态变化、半、全连接队列
4. DNS 使用的是 TCP 还是 UDP
5. SWAP 这个是置换算法效率低下

6 cookie 和 session 的区别 https://www.cnblogs.com/l199616j/p/11195667.html

会话（Session）跟踪是 Web 程序中常用的技术，用来**跟踪用户的整个会话**。常用的会话跟踪技术是 Cookie 与 Session。1.**Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份**。2.由于 cookie 存放在客户端，相对 session 来说不是很安全，别人可以分析存放在本地的

COOKIE 并进行 COOKIE 欺骗，所以考虑到安全应当使用 session。3. session 会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能，考虑到减轻服务器性能方面，应当使用 cookie。

Cookie 是保存在客户端的，session 是保存在服务端的，它们都是表示会话的一种方式。

由于 HTTP 是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是 Cookie 的工作原理。

Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改 Cookie 的内容。

Session 是服务器端使用的一种记录客户端状态的机制，使用上比 Cookie 简单一些，相应的也增加了服务器的存储压力。

6. GET 和 POST 的区别

（1）GET 相当于从服务端读取或拉取资源，不改变服务端上的任何资源、

POST 请求通常是向服务端提交数据，可能会修改服务器上的资源

（2）GET 请求的参数直接跟在 url 后边，用?分割 url 与携带参数，多个参数之间用&连接

POST 请求携带的参数封装在请求体中，所以说 GET 请求参数是暴露在地址栏中的，而 POST 的参数则不会暴露在地址栏。

（3）GET 请求在实际开发中，特定的浏览器和服务器对 URL 长度有限制，所以 GET 请求的传输数据会受到 URL 长度的限制

POST 请求携带数据理论上不会受限制，但实际上各个服务器也会规定对 POST 提交数据的大小进行限制的配置的

（4）POST 请求的安全性相比 GET 更高，因为 GET 的参数暴漏在地址栏

HTTP 响应常见的状态码以及含义如下：

200 OK 服务器成功处理请求

301/302 Moved Permanently（重定向）请求的 URL 已移走。响应报文中应该包含一个 Location URL，说明资源现在所处的位置

304 Not Modified（未修改） 客户的缓存资源是最新的，要客户端使用缓存内容

404 Not Found 未找到资源

501 Internal Server Error 服务器遇到错误，使其无法对请求提供服务

7 如何防止 POAT 请求重复提交：

本质：如何识别当前的请求是重复提交的内容

1. 比对内容相同的提交是否重复出现。
2. 为每次提交标号，比对相同编号的提交是否重复出现。即依赖提交内容作了标号

客户端请求页面时，服务器为每次产生的 form 表单分配唯一的随机标识号，并且在 form 的一个隐藏字段中设置这个标识号，同时在当前用户的 session 中保存这个标识号，当提交表单时，服务器比较 hidden 的 ID 和 session 中的标识号是否相同，session 中的标识表单的 ID 可以保存一定时间间隔后清空。

8 JAVA 三大特性 ，

继承 子类继承父类的特征与行为，实现代码复用

封装 将一类事物的共性（属性和方法）封装到同一类中，实现细节的封装，保证安全性。

多态 同一个行为具有不同的表现形式或形态的能力

子类继承父类

方法重写

父类引用指向子类对象

9 访问权限问题

10 重载 Java 中由参数类型和参数个数识别方法的重载，重载是同一个类中实现名字相同但参数不同的方法，比如不同的构造方法

11 synchronized 和 lock 的区别

12 线程池的执行流程

13 设计模式有哪些

14 说下某个集合的底层结构（ArrayList LinkedList HashMap ConcurrentHashMap）

15 七层网络协议

16 DNS 域名解析过程

1、在浏览器中输入 www . qq .com 域名，操作系统会先检查自己本地的 hosts 文件是否有这个网址映射关系，如果有，就先调用这个 IP 地址映射，完成域名解析。

2、如果 hosts 里没有这个域名的映射，则查找本地 DNS 解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

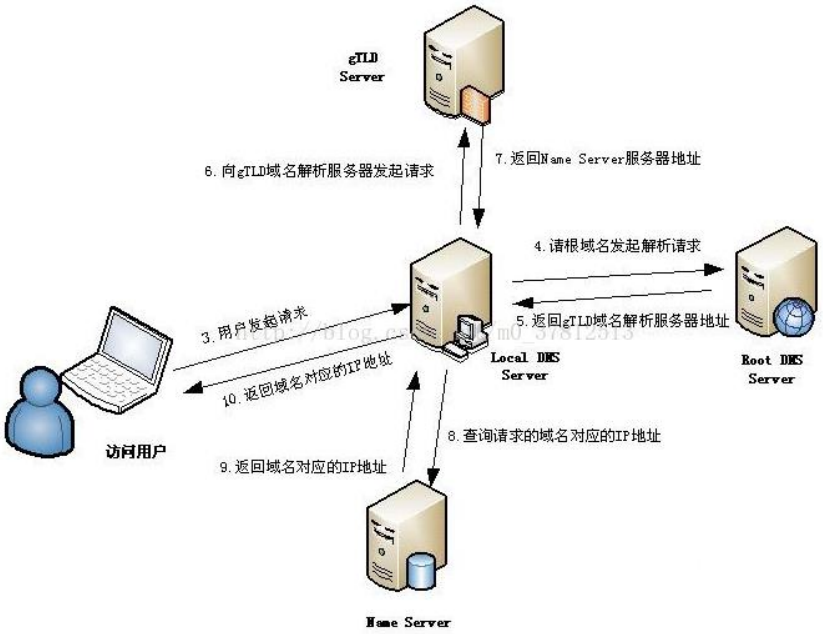
3、如果 hosts 与本地 DNS 解析器缓存都没有相应的网址映射关系，首先会找 TCP/ip 参数中设置的首选 DNS 服务器，在此我们叫它本地 DNS 服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

4、如果要查询的域名，不由本地 DNS 服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个 IP 地址映射，完成域名解析，此解析不具有权威性。

5、如果本地 DNS 服务器本地区域文件与缓存解析都失效，则根据本地 DNS 服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地 DNS 就把请求发至 13 台根 DNS，根 DNS 服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个 IP。本地 DNS 服务器收到 IP 信息后，将会联系负责.com 域的这台服务器。这台负责.com 域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com 域的下一级 DNS 服务器地址(http://qq.com)给本地 DNS 服务器。当本地 DNS 服务器收到这个地址后，就会找 http://qq.com 域服务器，重复上面的动作，进行查询，直至找到 www . qq .com 主机。

6、如果用的是转发模式，此 DNS 服务器就会把请求转发至上一级 DNS 服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根 DNS 或把转请求转至上上级，以此循环。不管是本地 DNS 服务器用是是转发，还是根提示，最后都是把结果返回给本地 DNS 服务器，由此 DNS 服务器再返回给客户机。

从客户端到本地 DNS 服务器是属于递归查询，而 DNS 服务器之间就是的交互查询就是迭代查询。



16 反射

17 类加载机制

18 StringBuilder 和 StringBuffer 哪个是线程安全的

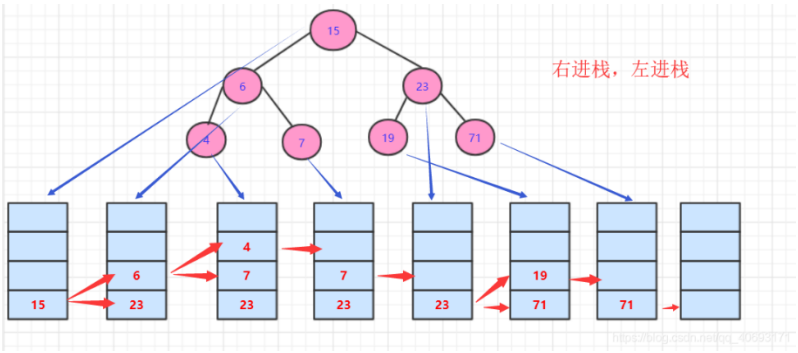
18 二叉树四种遍历的非递归方式 ，---- 前序、中序、后序（栈）、层序（队列）

（1） 前序非递归 根 左 右

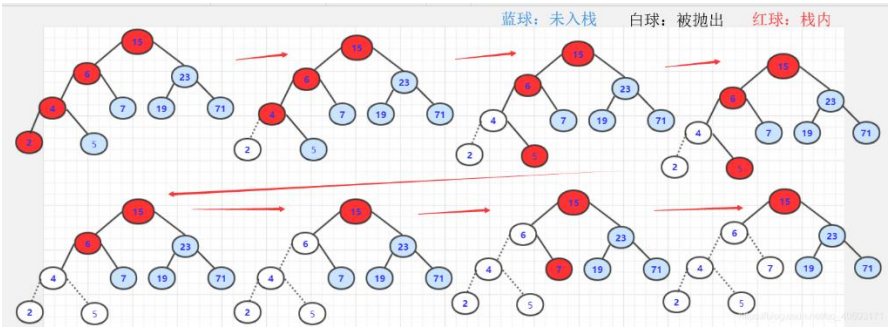
递归是左递归，右递归。但是利用栈要相反，右进栈、左进栈

利用递归的思路，需要先放右节点进栈，再放左节点进栈，这个下次·再取节点取到左节点·，这个节点再右节点进栈，左节点进栈。然后循环一直到最后会一直优先取到左节点。达到和递归顺序相仿效果。（方法一）

方法二和中序遍历的非递归方式类似，入栈出栈一样，只是输出地点有点差别



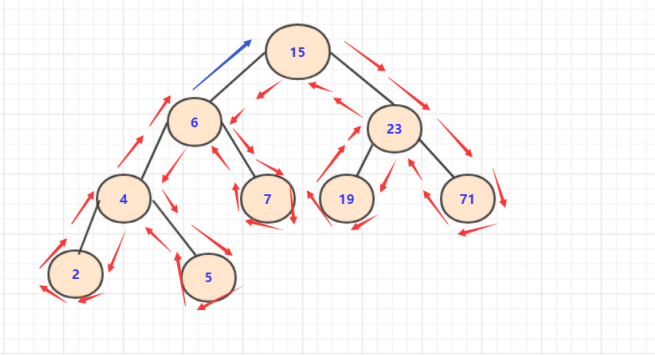
（2） 非递归中序 左 根 右



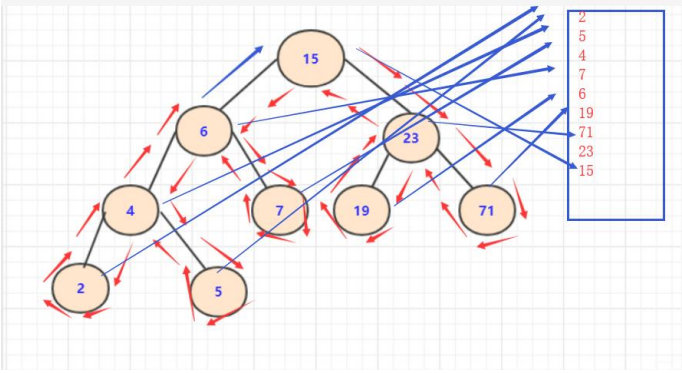
（3） 非递归后序

左 右 根 --- 逆序 是 根 右 左，

可以先 中 右 左 然后逆序输出即为后序遍历



左 右 中遍历，中 右 左 遍历 （类前序）



自我介绍

ArrayList 扩容

```
private static final int DEFAULT_CAPACITY = 10;
```

```
/**
 * Appends the specified element to the end of this list.
 * 先确保容量够不够，能不能够容纳的下将要添加的元素，即capacity是否能够达到size+1 (ensureCapacityInternal判断容量是否足够)
 * 不够，则进行扩容，若够容纳，不扩容，add到list之后即可。
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        // 若默认容量小于size+1，即添加元素时需要占用一个位置，size+1就是当前共需要多少个，查看是否足够
    }

    ensureExplicitCapacity(minCapacity);
}
```

HashMap 扩容
事务 ACID
隔离级别
HTTP 和 HTTPS
索引如何创建
为何使用 B+树不使用红黑树、平衡树
使用索引的查找过程，回表
自动化测试平台负责的东西
学习方式
深入研究过什么吗
MYSQL 中的 innodb 是聚簇索引，非叶子节点是主键，叶子节点存储的数据页，即表记录（一整条记录，所有的字段）
ID 自增主键　Userid 建立的索引　name　age
Select name from user where userid = 100
去根据 userid 这个辅助索引找到 userid 的叶子节点，这个叶子节点存储的是什么（id 和 userid 吗，还是只有 id），然后根据 id 去到主键索引查询到 id 为 userid 为 100 的这个记录 id 值得叶子节点，取出 age 字段。

聚簇索引和辅助索引得不同：叶子节点存放的是否是一整行的信息

辅助索引的叶子节点不包含行记录的全部字段的数据，叶子节点存放的是那条数据的主键字段的值，除此之外，每个叶子节点中的索引行中还包含一个书签，这个书签你可以理解为是一个{'name 字段'，name 的值，主键 id 值}的这么一个数据。该书签用来告诉 InnoDB 存储引擎去哪里可以找到与索引相对应的行数据。如果我们 select 后面要的是 name，我们直接就可以在辅助索引的叶子节点找到对应的 name 值，比如：select name from tb1 where name='xx'；这个 xx 值你直接就在辅助索引的叶子节点就能找到，这种我们也可以称为覆盖索引。如果你 select 后面的字段不是 name，例如：select age from tb1 where name='xx'；也就是说，我通过辅助索引的叶子节点不能直接拿到 age 的值，需要通过辅助索引的叶子节点中保存的主键 id 的值再去通过聚集索引来找到完整的一条记录，然后从这个记录里面拿出 age 的值，这种操作有时候也成为回表操作，就是从头再回去查一遍，这种的查询效率也很高，但是比覆盖索引低一些，再说一下昂，再辅助索引的叶子节点就能找到你想找的数据

可称为覆盖索引。

Select name from user where age=3 and userid=100; 建立联合索引时顺序 userid 在前，age 在后，因为 userid 的区分度更高

IO 流实现文件复制

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
/**
 * 文件拷贝：文件字节输入流，输出流
 * @author Administrator
 *
 */
public class Copy {

	public static void main(String[] args) {
		copy("src/com/jianshun/Copy.java","copy.txt");
	}

	public static void copy(String srcPath,String destPath){
		//,2，创建源
		File src = new File(srcPath);
		File dest = new File(destPath);
		//,2，选择流
		InputStream in = null;
		OutputStream out = null;
		try {
			in = new FileInputStream(src);
			out = new FileOutputStream(destPath);
			//3，操作，分段读取
			byte[] flush = new byte[1024];
			int len = -1;//接收长度
			while((len = in.read(flush)) != -1){
				out.write(flush, 0, len);//分段写出
			}
			out.flush();
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}finally{
```

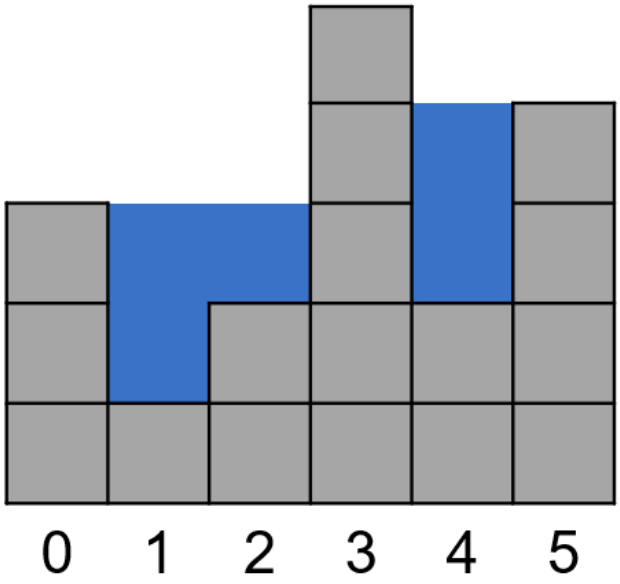
```
            //4，释放资源
            if(in !=null){
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if(out != null){
                try {
                    out.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

主观

1. 值得称道的工作细节
2. 曾经克服过的最大挑战
3. 找工作时最看重的方面
4. 引以为傲的个人项目或事迹
5. 你有什么优势或展示你优势的事情

MYSQL 数据库事务实现机制

NC128 容器盛水问题



快手面经：

1. 数据结构

ArrayList

ensureExplicitCapacity - >

if minCapacity - elementData.length > 0 grow(minCapacity)

```
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {  //minCapicity+1
    // overflow-conscious code
    int oldCapacity = elementData.length;       //旧数组容量大小
    int newCapacity = oldCapacity + (oldCapacity >> 1);     //新数组容量大小1.5倍的oldCapacity
    if (newCapacity - minCapacity < 0)          //若扩容的1.5倍oldCapacity仍小于minCapacity 则扩容为minCapacity,否则是1.5倍的oldCapacity
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)       //再判断newCapacity的
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);          //Arrays.copyOf，使用数组复制，将旧数组中的元素复制到扩容后的新的数组中去
}
```

```
/**
 * Inserts the specified element at the specified position in this
 * list. Shifts the element currently at that position (if any) and
 * any subsequent elements to the right (adds one to their indices).
 * 在指定位置插入指定元素时候需要移动右侧的元素
 * @param index index at which the specified element is to be inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
    rangeCheckForAdd(index);  //判断index是否越界，若越界则插入失败，否则继续往下执行，判断是否需要扩容，然后在index右侧移动元素，将指定元素插入Index处

    ensureCapacityInternal( minCapacity: size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData,  destPos: index + 1,
                     length: size - index);
    elementData[index] = element;
    size++;
}
```

ArrayList 中的 add(index)和 remove(index)都是使用了 System.arraycopy(elementData, index+1, elementData, index, numMoved) 即 copy 从 index+1 到最后的元素覆盖 elementData 数组 index 位置到最后，并将 elementData[size--]=null，垃圾回收。

Array.copy

System.arraycopy

LinkedList、

LinkedList,双向链表，每个元素是个 node，通过 pre 和 next 指针相连，LinkedList 有 first 和 last 标识第一个 node 和最后一个 node，add 和 remove 只需改变指针指向即可，无需扩容操作。

HashMap

jdk1.7 数组+链表 jdk1.8 数组+链表+红黑树

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; //16
static final int MAXIMUM_CAPACITY = 1 << 30;
static final float DEFAULT_LOAD_FACTOR = 0.75f;
static final int TREEIFY_THRESHOLD = 8;
static final int UNTREEIFY_THRESHOLD = 6;
static final int MIN_TREEIFY_CAPACITY = 64;
transient Node<K,V>[] table;
transient Set<Map.Entry<K,V>> entrySet;
```

```
    int threshold;
    // The next size value at which to resize (capacity *
load factor).
```

Get

Put

```
    public V put(K key, V value)
       return putVal(hash(key), key, value, false, true);
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
             boolean evict)
```

Put 过程简述：

1.判断 tab 数组是否为 null 或 length 为 0，若是则 resize 扩容；否则

2.hash&tab.length-1 得到当前插入 key 所处下标 i，若 tab[i]==null 则直接插入 tab[i]=newNode(hash,key,value,null);否则说明有 hash 冲突，要去遍历冲突元素

3.从 tab[i]这条链上（或这条树上的结点）遍历，判断 e.hash==hash(key)&&e.key==key 若是，则 break,用新值覆盖旧值；若遍历完仍没有相同的 key，则进入尾插，尾插后判断当前 bin 的 count 是否大于等于 treefy_threshold；若是则执行树化操作；否则尾插法完成后退出循环，modCount++;插入完成后若当前 hashMap 中包含的元素数量 size>threshold 扩容阈值时，执行扩容操作 resize;

resize 过程简述：

1.若 oldTab 为 null，为 HashMap 的初始化，生成数组返回 null 即可

```
  newCap = DEFAULT_INITIAL_CAPACITY;
  newThr = (int)(DEFAULT_LOAD_FACTOR *DEFAULT_INITIAL_CAPACITY);
  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
```

2.若 oldTab 不为 null，

```
if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
        oldCap >= DEFAULT_INITIAL_CAPACITY)
    newThr = oldThr << 1; // double threshold
threshold = newThr;
```

```
Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
```

HashSet
TreeSet
HashTable
CorrentHashMap
TreeMap
Object 类
Semphor
CountDownLatch
CycleBarrier

克隆指的均是对象的克隆，对象实例包含了实例基本数据类型和引用数据类型的成员变量，深浅克隆主要针对的是对象中的引用所指向的对象是否进行了内存的重新分配及复制。
浅克隆：

实现 cloneable 方法，重写 Object 的 clone 方法，super.clone()

浅克隆，8 种基本类型的数据变量被复制，而引用类型的变量也被复制，只是拷贝了一份引用，新旧引用指向同一块存储对象的内存地址，通过其中一个引用修改指向内存的对象后，使用另一个引用访问时发生变化。
深克隆：

重写 clone 方法，为对象中存在的引用指向重新分配新的内存，将指向的成员对象拷贝一份到新的内存中，这样新旧引用指向不同的内存中存储的对象。这样通过其中一个引用修改指向对象时，修改的自己指向的那一份，不会对别的空间产生影响。

实现方式（1）super.clone();this.xxObj = new XXObj(); (2)使用序列化

2. 设计模式

3 项目复习

4