链滴

面试锦囊

作者：sirwsl

原文链接：https://ld246.com/article/1630026311672

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20201031.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="说在最前面的话">说在最前面的话</h2>
这是一个份转载的面试锦囊，原文出处：<a href="https://ld246.com/forward?goto=https%
A%2F%2Fgithub.com%2Fyuanguangxin%2FLeetCode%2Fblob%2Fmaster%2FRocket.md" targ
t="_blank" rel="nofollow ugc">LeetCode/Rocket.md at master · yuanguangxin/LeetCode (git
ub.com)，微信公众号“JAVA 程序员社区”有原文发布，发布地址 <a href="https://ld246.c
m/forward?goto=https%3A%2F%2Fmp.weixin.qq.com%2Fs%2FAT4rIBTzd8uW3VJ8rY4GBA" t
rget="_blank" rel="nofollow ugc">https://mp.weixin.qq.com/s/AT4rIBTzd8uW3VJ8rY4GBA</
>
为了防止大佬删除自己没得地方看了，所以就转载了，大佬说过有些地方可能有些不妥，大家自
参照，哪怕不面试，当作知识积累也好啊，啊哈哈哈哈哈哈哈，毕竟提升自己最重要嘛
<h2 id="面试问题整理">面试问题整理</h2>
<h2 id="ZooKeeper">ZooKeeper</h2>
<h3 id="CAP定理-">CAP 定理：</h3>
一个分布式系统不可能在满足分区容错性（P）的情况下同时满足一致性（C）和可用
（A）。在此 ZooKeeper 保证的是 CP，ZooKeeper 不能保证每次服务请求的可用性，在极端环境
，ZooKeeper 可能会丢弃一些请求，消费者程序需要重新请求才能获得结果。另外在进行 leader 选
时集群都是不可用，所以说，ZooKeeper 不能保证服务可用性。
<h3 id="BASE理论">BASE 理论</h3>
BASE 理论是基本可用，软状态，最终一致性三个短语的缩写。BASE 理论是对 CAP
一致性和可用性（CA）权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于 CAP
理逐步演化而来的，它大大降低了我们对系统的要求。

基本可用：基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用
。但是，这绝不等价于系统不可用。比如正常情况下，一个在线搜索引擎需要在 0.5 秒之内返回给用
相应的查询结果，但由于出现故障，查询结果的响应时间增加了 1~2 秒。
软状态：软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影
系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。</strong

最终一致性：最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后
最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而
需要实时保证系统数据的强一致性。

<h3 id="ZooKeeper特点">ZooKeeper 特点</h3>

顺序一致性：同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKee
er 中去。
原子性：所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也
是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。
单一系统映像：无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据
型都是一致的。
可靠性：一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆
。

<h3 id="ZAB协议-">ZAB 协议：</h3>
ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。当整个 Zookeeper 集群刚刚
动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常
信时，所有服务器进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服
器开始与新的 Leader 服务器进行数据同步。当集群中超过半数机器与该 Leader 服务器完成数据同
之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案（
过半数同意）来进行事务请求处理。
<h3 id="选举算法和流程-FastLeaderElection-默认提供的选举算法-">选举算法和流程：
astLeaderElection(默认提供的选举算法)</h3>

目前有 5 台服务器，每台服务器均没有数据，它们的编号分别是 1,2,3,4,5,按编号依次动，它们的选择举过程如下：

服务器 1 启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不反馈信息，服务器 1 的状态一直属于 Looking。
服务器 2 启动，给自己投票，同时与之前启动的服务器 1 交换结果，由于服务器 2 的号大所以服务器 2 胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是 LOOKING。
服务器 3 启动，给自己投票，同时与之前启动的服务器 1,2 交换信息，由于服务器 3 编号最大所以服务器 3 胜出，此时投票数正好大于半数，所以服务器 3 成为 leader，服务器 1,2 成为 ollower。
服务器 4 启动，给自己投票，同时与之前启动的服务器 1,2,3 交换信息，尽管服务器 4 的编号大，但之前服务器 3 已经胜出，所以服务器 4 只能成为 follower。
服务器 5 启动，后面的逻辑同服务器 4 成为 follower。

<h3 id="zk中的监控原理">zk 中的监控原理</h3>
zk 类似于 linux 中的目录节点树方式的数据存储，即分层命名空间，zk 并不是专门存数据的，它的作用是主要是维护和监控存储数据的状态变化，通过监控这些数据状态的变化，从而可达到基于数据的集群管理，zk 中的节点的数据上限时 1M。
client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些 clint 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。
<h3 id="zk实现分布式锁">zk 实现分布式锁</h3>
zk 实现分布式锁主要利用其临时顺序节点，实现分布式锁的步骤如下：

创建一个目录 mylock
线程 A 想获取锁就在 mylock 目录下创建临时顺序节点
获取 mylock 目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则明当前线程顺序号最小，获得锁
线程 B 获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点
线程 A 处理完，删除自己的节点，线程 B 监听到变更事件，判断自己是不是最小的节，如果是则获得锁

<h2 id="Redis">Redis</h2>
<h3 id="应用场景">应用场景</h3>

缓存
共享 Session
消息队列系统
分布式锁

<h3 id="单线程的Redis为什么快">单线程的 Redis 为什么快</h3>

纯内存操作
单线程操作，避免了频繁的上下文切换
合理高效的数据结构
采用了非阻塞 I/O 多路复用机制（有一个文件描述符同时监听多个文件描述符是否有据到来）

<h3 id="Redis-的数据结构及使用场景">Redis 的数据结构及使用场景</h3>

String 字符串:字符串类型是 Redis 最基础的数据结构，首先键都是字符串类型，而且其他几种数据结构都是在字符串类型基础上构建的，我们常使用的 set key value 命令就是字符串。

用在缓存、计数、共享 Session、限速等。
Hash 哈希:在 Redis 中，哈希类型是指键值本身又是一个键值对结构，哈希可以用来放用户信息，比如实现购物车。
List 列表（双向链表）:列表（list）类型是用来存储多个有序的字符串。可以做简单的息队列的功能。
Set 集合：集合（set）类型也是用来保存多个的字符串元素，但和列表类型不一 样的，集合中不允许有重复元素，并且集合中的元素是无序的，不能通过索引下标获取元素。利用 Set 的集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。
Sorted Set 有序集合（跳表实现）：Sorted Set 多了一个权重参数 Score，集合中的素能够按 Score 进行排列。可以做排行榜应用，取 TOP N 操作。

<h3 id="Redis-的数据过期策略">Redis 的数据过期策略</h3>
Redis 中数据过期策略采用定期删除 + 惰性删除策略

定期删除策略：Redis 启用一个定时器定时监视所有的 key，判断 key 是否过期，过的话就删除。这种策略可以保证过期的 key 最终都会被删除，但是也存在严重的缺点：每次都遍历内中所有的数据，非常消耗 CPU 资源，并且当 key 已过期，但是定时器还处于未唤起状态，这段时间内 key 仍然可以用。
惰性删除策略：在获取 key 时，先判断 key 是否过期，如果过期则删除。这种方式存一个缺点：如果这个 key 一直未被使用，那么它一直在内存中，其实它已经过期了，会浪费大量的空。
这两种策略天然的互补，结合起来之后，定时删除策略就发生了一些改变，不在是每扫描全部的 key 了，而是随机抽取一部分 key 进行检查，这样就降低了对 CPU 资源的损耗，惰性删策略互补了为检查到的 key，基本上满足了所有要求。但是有时候就是那么的巧，既没有被定时器抽到，又没有被使用，这些数据又如何从内存中消失？没关系，还有内存淘汰机制，当内存不够用时，存淘汰机制就会上场。淘汰策略分为：

当内存不足以容纳新写入数据时，新写入操作会报错。（Redis 默认策略）
当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 Key。（LRU 推使用）
当内存不足以容纳新写入数据时，在键空间中，随机移除某个 Key。
当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 Key。这种情况一般是把 Redis 既当缓存，又做持久化存储的时候才用。
当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 Key
当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 K y 优先移除。

<h3 id="Redis的set和setnx">Redis 的 set 和 setnx</h3>
Redis 中 setnx 不支持设置过期时间，做分布式锁时要想避免某一客户端中断导致死，需设置 lock 过期时间，在高并发时 setnx 与 expire 不能实现原子操作，如果要用，得在程序代码显示的加锁。使用 SET 代替 SETNX ，相当于 SETNX+EXPIRE 实现了原子性，不必担心 SETNX 成，EXPIRE 失败的问题。
<h3 id="Redis的LRU具体实现-">Redis 的 LRU 具体实现：</h3>
*传统的 LRU 是使用栈的形式，每次都将最新使用的移入栈顶，但是用栈的形式会导致执行 select 的时候大量非热点数据占领头部数据，所以需要改进。Redis 每次按 key 获取一个值的候，都会更新 value 中的 lru 字段为当前秒级别的时间戳。Redis 初始的实现算法很简单，随机从 dict 中取出五个 key,淘汰一个 lru 字段值最小的。在 3.0 的时候，又改进了一版算法，首先第一次随机选的 key 都会放入一个 pool 中(pool 的大小为 16),pool 中的 key 是按 lru 大小顺序排列的。接下来次随机选取的 keylru 值必须小于 pool 中最小的 lru 才会继续放入，直到将 pool 放满。放满之后，次如果有新的 key 需要放入，需要将 pool 中 lru 最大的一个 key 取出。淘汰的时候，直接从 pool

选取一个 lru 最小的值然后将其淘汰。
<h3 id="Redis如何发现热点key">Redis 如何发现热点 key</h3>

凭借经验，进行预估：例如提前知道了某个活动的开启，那么就将此 Key 作为热点 Ke
。
服务端收集：在操作 redis 之前，加入一行代码进行数据统计。
抓包进行评估：Redis 使用 TCP 协议与客户端进行通信，通信协议采用的是 RESP，
以自己写程序监听端口也能进行拦截包进行解析。
在 proxy 层，对每一个 redis 请求进行收集上报。
Redis 自带命令查询：Redis4.0.4 版本提供了 redis-cli –hotkeys 就能找出热点 Key
（如果要用 Redis 自带命令查询时，要注意需要先把内存逐出策略设置为 allkeys-lfu 或者 volatile-lf
，否则会返回错误。进入 Redis 中使用 config set maxmemory-policy allkeys-lfu 即可。）</stro
g>

<h3 id="Redis的热点key解决方案">Redis 的热点 key 解决方案</h3>

服务端缓存：即将热点数据缓存至服务端的内存中.(利用 Redis 自带的消息通知机制来
证 Redis 和服务端热点 Key 的数据一致性，对于热点 Key 客户端建立一个监听，当热点 Key 有更新
作的时候，服务端也随之更新。)
备份热点 Key：即将热点 Key+ 随机数，随机分配至 Redis 其他节点中。这样访问热点
key 的时候就不会全部命中到一台机器上了。

<h3 id="如何解决-Redis-缓存雪崩问题">如何解决 Redis 缓存雪崩问题</h3>

使用 Redis 高可用架构：使用 Redis 集群来保证 Redis 服务不会挂掉
缓存时间不一致，给缓存的失效时间，加上一个随机值，避免集体失效

限流降级策略：有一定的备案，比如个性推荐服务不可用了，换成热点数据推荐服务</
trong>

<h3 id="如何解决-Redis-缓存穿透问题">如何解决 Redis 缓存穿透问题</h3>

在接口做校验
存 null 值（缓存击穿加锁,或设置不过期）
布隆过滤器拦截： 将所有可能的查询 key 先映射到布隆过滤器中，查询时先判断 key
否存在布隆过滤器中，存在才继续向下执行，如果不存在，则直接返回。布隆过滤器将值进行多次哈希
bit 存储，布隆过滤器说某个元素在，可能会被误判。布隆过滤器说某个元素不在，那么一定不在。</
trong>

<h3 id="Redis的持久化机制">Redis 的持久化机制</h3>
Redis 为了保证效率，数据缓存在了内存中，但是会周期性的把更新的数据写入磁盘
者把修改操作写入追加的记录文件中，以保证数据的持久化。Redis 的持久化策略有两种：</strong

RDB：快照形式是直接把内存中的数据保存到一个 dump 的文件中，定时保存，保存
略。当 Redis 需要做持久化时，Redis 会 fork 一个子进程，子进程将数据写到磁盘上一个临时 RDB
件中。当子进程完成写临时文件后，将原来的 RDB 替换掉。
AOF：把所有的对 Redis 的服务器进行修改的命令都存到一个文件里，命令的集合。<
strong>

使用 AOF 做持久化，每一个写命令都通过 write 函数追加到 appendonly.aof 中。aof
的默认策略是每秒钟 fsync 一次，在这种配置下，就算发生故障停机，也最多丢失一秒钟的数据。 缺
是对于相同的数据集来说，AOF 的文件体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略
AOF 的速度可能会慢于 RDB。 Redis 默认是快照 RDB 的持久化方式。对于主从同步来说，主从刚

连接的时候，进行全量同步（RDB）；全同步结束后，进行增量同步(AOF)。
<h3 id="Redis的事务">Redis 的事务</h3>

Redis 事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请不会插入到事务执行命令序列中。总结说：redis 事务就是一次性、顺序性、排他性的执行一个队列的一系列命令。
Redis 事务没有隔离级别的概念，批量操作在发送 EXEC 命令前被放入队列缓存，并会被实际执行，也就不存在事务内的查询要看到事务里的更新，事务外查询不能看到。</l>
Redis 中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任命令执行失败，其余的命令仍会被执行。

<h3 id="Redis事务相关命令">Redis 事务相关命令</h3>

watch key1 key2 ... : 监视一或多个 key,如果在事务执行之前，被监视的 key 被其他令改动，则事务被打断（类似乐观锁）
multi : 标记一个事务块的开始（queued）
exec : 执行所有事务块的命令（一旦执行 exec 后，之前加的监控锁都会被取消掉）</srong>
discard : 取消事务，放弃事务块中的所有命令
unwatch : 取消 watch 对所有 key 的监控

<h3 id="Redis和-memcached-的区别">Redis 和 memcached 的区别</h3>

存储方式上：memcache 会把数据全部存在内存之中，断电后会挂掉，数据不能超过存大小。redis 有部分数据存在硬盘上，这样能保证数据的持久性。
数据支持类型上：memcache 对数据类型的支持简单，只支持简单的 key-value，，而redis 支持五种数据类型。
用底层模型不同：它们之间底层实现方式以及与客户端之间通信的应用协议不一样。reis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。/strong>
value 的大小：redis 可以达到 1GB，而 memcache 只有 1MB。

<h3 id="Redis的几种集群模式">Redis 的几种集群模式</h3>

主从复制
哨兵模式
cluster 模式

<h3 id="Redis的哨兵模式">Redis 的哨兵模式</h3>
哨兵是一个分布式系统,在主从复制的基础上你可以在一个架构中运行多个哨兵进程,这进程使用流言协议来接收关于 Master 是否下线的信息,并使用投票协议来决定是否执行自动故障迁移以及选择哪个 Slave 作为新的 Master。
每个哨兵会向其它哨兵、master、slave 定时发送消息,以确认对方是否活着,如果发现方在指定时间(可配置)内未回应,则暂时认为对方已挂(所谓的”主观认为宕机”)。
若“哨兵群“中的多数 sentinel,都报告某一 master 没响应,系统才认为该 master"彻死亡"(即:客观上的真正 down 机),通过一定的 vote 算法,从剩下的 slave 节点中,选一台提升为 master然后自动修改相关配置。
<h3 id="Redis的rehash">Redis 的 rehash</h3>
Redis 的 rehash 操作并不是一次性、集中式完成的，而是分多次、渐进式地完成的，rdis 会维护维持一个索引计数器变量 rehashidx 来表示 rehash 的进度。
这种渐进式的 rehash 避免了集中式 rehash 带来的庞大计算量和内存操作，但是需要意的是 redis 在进行 rehash 的时候，正常的访问请求可能需要做多要访问两次 hashtable（ht[0]， h

[1]），例如键值被 rehash 到新 ht1，则需要先访问 ht0，如果 ht0 中找不到，则去 ht1 中找。</strng>
<h3 id="Redis的hash表被扩展的条件">Redis 的 hash 表被扩展的条件</h3>

哈希表中保存的 key 数量超过了哈希表的大小.
Redis 服务器目前没有在执行 BGSAVE 命令（rdb）或 BGREWRITEAOF 命令，并且希表的负载因子大于等于 1.
Redis 服务器目前在执行 BGSAVE 命令（rdb）或 BGREWRITEAOF 命令，并且哈希的负载因子大于等于 5.(负载因子=哈希表已保存节点数量 / 哈希表大小，当哈希表的负载因子小于 0.1 时，对哈希表执行收缩操作。)

<h3 id="Redis并发竞争key的解决方案">Redis 并发竞争 key 的解决方案</h>

分布式锁 + 时间戳
利用消息队列

<h3 id="Redis与Mysql双写一致性方案">Redis 与 Mysql 双写一致性方案</h>
先更新数据库，再删缓存。数据库的读操作的速度远快于写操作的，所以脏数据很难现。可以对异步延时删除策略，保证读请求完成以后，再进行删除操作。
<h3 id="Redis的管道pipeline">Redis 的管道 pipeline</h3>
对于单线程阻塞式的 Redis，Pipeline 可以满足批量的操作，把多个命令连续的发送给 Redis Server，然后一一解析响应结果。Pipelining 可以提高批量处理性能，提升的原因主要是 TCP 连接中减少了“交互往返”的时间。pipeline 底层是通过把所有的操作封装成流，redis 有定义自己出入输出流。在 sync() 方法执行操作，每次请求放在队列里面，解析响应包。
<h2 id="Mysql">Mysql</h2>
<h3 id="事务的基本要素">事务的基本要素</h3>

原子性：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行/strong>
一致性：事务开始前和结束后，数据库的完整性约束没有被破坏。
隔离性：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干。
持久性：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。</strog>

<h3 id="Mysql的存储引擎">Mysql 的存储引擎</h3>

InnoDB 存储引擎：InnoDB 存储引擎支持事务，其设计目标主要面向在线事务处理（LTP）的应用。其特点是行锁设计，支持外键，并支持非锁定锁，即默认读取操作不会产生锁。从 Mysql5.5.8 版本开始，InnoDB 存储引擎是默认的存储引擎。
MyISAM 存储引擎：MyISAM 存储引擎不支持事务、表锁设计，支持全文索引，主要向一些 OLAP 数据库应用。InnoDB 的数据文件本身就是主索引文件，而 MyISAM 的主索引和数据分开的。
NDB 存储引擎：NDB 存储引擎是一个集群存储引擎，其结构是 share nothing 的集架构，能提供更高的可用性。NDB 的特点是数据全部放在内存中（从 MySQL 5.1 版本开始，可以将索引数据放在磁盘上），因此主键查找的速度极快，并且通过添加 NDB 数据存储节点可以线性地提数据库性能，是高可用、高性能的集群系统。NDB 存储引擎的连接操作是在 MySQL 数据库层完成，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢如果解决了这个问题，NDB 存储引擎的市场应该是非常巨大的。
Memory 存储引擎：Memory 存储引擎（之前称 HEAP 存储引擎）将表中的数据存在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时，以及数据仓库中的纬度表。Memory 存储引擎默认使用哈希索引，而不是我们熟悉的 B+ 树索引。

然 Memory 存储引擎速度非常快，但在使用上还是有一定的限制。比如，只支持表锁，并发性能较，并且不支持 TEXT 和 BLOB 列类型。最重要的是，存储变长字段时是按照定常字段的方式进行的，此会浪费内存。
Archive 存储引擎：Archive 存储引擎只支持 INSERT 和 SELECT 操作，从 MySQL 5.1 开始支持索引。Archive 存储引擎使用 zlib 算法将数据行（row）进行压缩后存储，压缩比一般可达：10。正如其名字所示，Archive 存储引擎非常适合存储归档数据，如日志信息。Archive 存储引擎用行锁来实现高并发的插入操作，但是其本身并不是事务安全的存储引擎，其设计目标主要是提供高的插入和压缩功能。
Maria 存储引擎：Maria 存储引擎是新开发的引擎，设计目标主要是用来取代原有的 yISAM 存储引擎，从而成为 MySQL 的默认存储引擎。它可以看做是 MyISAM 的后续版本。Maria 储引擎的特点是：支持缓存数据和索引文件，应用了行锁设计，提供了 MVCC 功能，支持事务和非务安全的选项，以及更好的 BLOB 字符类型的处理性能。

<h3 id="事务的并发问题">事务的并发问题</h3>

脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是数据
不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数作了更新并提交，导致事务 A 多次读取同一数据时，结果不一致。
幻读：A 事务读取了 B 事务已经提交的新增数据。注意和不可重复读的区别，这里是增，不可重复读是更改（或删除）。select 某记录是否存在，不存在，准备插入此记录，但执行 insert 时发现此记录已存在，无法插入，此时就发生了幻读。

<h3 id="MySQL事务隔离级别">MySQL 事务隔离级别</h3>
<table>
<thead>
<tr>
<th>事务隔离级别</th>
<th>脏读</th>
<th>不可重复读</th>
<th>幻读</th>
</tr>
</thead>
<tbody>
<tr>
<td>读未提交</td>
<td>是</td>
<td>是</td>
<td>是</td>
</tr>
<tr>
<td>不可重复读</td>
<td>否</td>
<td>是</td>
<td>是</td>
</tr>
<tr>
<td>可重复读</td>
<td>否</td>
<td>否</td>
<td>是</td>
</tr>
<tr>
<td>串行化</td>

<td>否</td>
<td>否</td>
<td>否</td>
</tr>
</tbody>
</table>
<h3 id="Mysql的逻辑结构">Mysql 的逻辑结构</h3>

最上层的服务类似其他 CS 结构，比如连接处理，授权处理。
第二层是 Mysql 的服务层，包括 SQL 的解析分析优化，存储过程触发器视图等也在一层实现。
最后一层是存储引擎的实现，类似于 Java 接口的实现，Mysql 的执行器在执行 SQL 时候只会关注 API 的调用，完全屏蔽了不同引擎实现间的差异。比如 Select 语句，先会判断当前用是否拥有权限，其次到缓存（内存）查询是否有相应的结果集，如果没有再执行解析 sql，检查 SQL 句语法是否正确，再优化生成执行计划，调用 API 执行。

<h3 id="SQL执行顺序">SQL 执行顺序</h3>
SQL 的执行顺序：from---where--group by---having---select---order by</stron>
<h3 id="MVCC-redolog-undolog-binlog">MVCC,redolog,undolog,binlog</h3>

undoLog 也就是我们常说的回滚日志文件 主要用于事务中执行失败，进行回滚，以及 MVCC 中对于数据历史版本的查看。由引擎层的 InnoDB 引擎实现,是逻辑日志,记录数据修改被修改的值,比如"把 id='B' 修改为 id = 'B2'，那么 undo 日志就会用来存放 id ='B'的记录”。当一条数据要更新前,会先把修改前的记录存储在 undolog 中,如果这个修改出现异常,,则会使用 undo 日志来实回滚操作,保证事务的一致性。当事务提交之后，undo log 并不能立马被删除,而是会被放到待清理链中,待判断没有事物用到该版本的信息时才可以清理相应 undolog。它保存了事务发生之前的数据的个版本，用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读。</l>
redoLog 是重做日志文件是记录数据修改之后的值，用于持久化到磁盘中。redo log 括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日文件(redo log file)，该部分日志是持久的。由引擎层的 InnoDB 引擎实现,是物理日志,记录的是物理据页修改的信息,比如“某个数据页上内容发生了哪些改动”。当一条数据需要更新时,InnoDB 会先将据更新，然后记录 redoLog 在内存中，然后找个时间将 redoLog 的操作执行到磁盘上的文件上。不是否提交成功我都记录，你要是回滚了，那我连回滚的修改也记录。它确保了事务的持久性。每个 InoDB 存储引擎至少有 1 个重做日志文件组（group），每个文件组下至少有 2 个重做日志文件，如认的 ib_logfile0 和 ib_logfile1。为了得到更高的可靠性，用户可以设置多个的镜像日志组（mirrored log groups），将不同的文件组放在不同的磁盘上，以此提高重做日志的高可用性。在日志组中每个做日志文件的大小一致，并以循环写入的方式运行。InnoDB 存储引擎先写重做日志文件 1，当达到件的最后时，会切换至重做日志文件 2，再当重做日志文件 2 也被写满时，会再切换到重做日志文件 1 中。
MVCC 多版本并发控制是 MySQL 中基于乐观锁理论实现隔离级别的方式，用于读已交和可重复读取隔离级别的实现。在 MySQL 中，会在表中每一条数据后面添加两个字段：最近修改行数据的事务 ID，指向该行（undolog 表中）回滚段的指针。Read View 判断行的可见性，创建一新事务时，copy 一份当前系统中的活跃事务列表。意思是，当前不应该被本事务看到的其他事务 id 表。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。

<h3 id="binlog和redolog的区别">binlog 和 redolog 的区别</h3>

redolog 是在 InnoDB 存储引擎层产生，而 binlog 是 MySQL 数据库的上层服务层生的。
两种日志记录的内容形式不同。MySQL 的 binlog 是逻辑日志，其记录是对应的 SQL

语句，对应的事务。而 innodb 存储引擎层面的重做日志是物理日志，是关于每个页（Page）的更的物理情况。
两种日志与记录写入磁盘的时间点不同，binlog 日志只在事务提交完成后进行一次写。而 innodb 存储引擎的重做日志在事务进行中不断地被写入，并日志不是随事务提交的顺序进行写的。
binlog 不是循环使用，在写满或者重启之后，会生成新的 binlog 文件，redolog 是环使用。
binlog 可以作为恢复数据使用，主从复制搭建，redolog 作为异常宕机或者介质故障的数据恢复使用。

<h3 id="Mysql读写分离以及主从同步">Mysql 读写分离以及主从同步</h3>

原理：主库将变更写 binlog 日志，然后从库连接到主库后，从库有一个 IO 线程，将库的 binlog 日志拷贝到自己本地，写入一个中继日志中，接着从库中有一个 sql 线程会从中继日志取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再执行一遍 sql，这样就可以保证自跟主库的数据一致。
问题：这里有很重要一点，就是从库同步主库数据的过程是串行化的，也就是说主库并行操作，在从库上会串行化执行，由于从库从主库拷贝日志以及串行化执行 sql 特点，在高并发情下，从库数据一定比主库慢一点，是有延时的，所以经常出现，刚写入主库的数据可能读不到了，要几十毫秒，甚至几百毫秒才能读取到。还有一个问题，如果突然主库宕机了，然后恰巧数据还没有同到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。所以 mysql 实际上有两个制，一个是半同步复制，用来解决主库数据丢失问题，一个是并行复制，用来解决主从同步延时问题
半同步复制：semi-sync 复制，指的就是主库写入 binlog 日志后，就会将强制此时立将数据同步到从库，从库将日志写入自己本地的 relay log 之后，接着会返回一个 ack 给主库，主库收到至少一个从库 ack 之后才会认为写完成。
并发复制：指的是从库开启多个线程，并行读取 relay log 中不同库的日志，然后并行放不同库的日志，这样库级别的并行。（将主库分库也可缓解延迟问题）

<h3 id="Next-Key-Lock">Next-Key Lock</h3>
InnoDB 采用 Next-Key Lock 解决幻读问题。在 <code>insert into test(xd) values (1), (3), (5), (8), (11);</code> 后，由于 xid 上是有索引的，该算法总是会去锁索引记录。现在，该索引可能被锁住的范围如下：(-∞, 1], (1, 3], (3, 5], (5, 8], (8, 11], (11, +∞)。Sesson A（<code>select * from test where id = 8 for update</code>）执行会锁住的范围：(5, 8], (8, 11]。除了锁住 8 所在的范围，还会锁住下一个范围，所谓 Next-Key。
<h3 id="InnoDB的关键特性">InnoDB 的关键特性</h3>

插入缓冲：对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个 Insert Buer 对象中。然后再以一定的频率和情况进行 Insert Buffer 和辅助索引页子节点的 merge（合并）操，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索插入的性能。
两次写：两次写带给 InnoDB 存储引擎的是数据页的可靠性，有经验的 DBA 也许会，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚地认识到，如果这个本身已经发生了损坏（物理到 page 页的物理日志成功页内逻辑日志失败），再对其进行重做是没有义的。这就是说，在应用（apply）重做日志前，用户需要一个页的副本，当写入失效发生时，先通页的副本来还原该页，再进行重做。在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过 mmcpy 函数将脏页先复制到内存中的 doublewrite buffer，之后通过 doublewrite buffer 再分两次每次 1MB 顺序地写入共享表空间的物理磁盘上，这就是 doublewrite。
自适应哈希索引：InnoDB 存储引擎会监控对表上各索引页的查询。如果观察到建立希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引。
异步 IO：为了提高磁盘操作性能，当前的数据库系统都采用异步 IO（AIO）的方式来理磁盘操作。AIO 的另一个优势是可以进行 IO Merge 操作，也就是将多个 IO 合并为 1 个 IO，这

可以提高 IOPS 的性能。
刷新邻接页：当刷新一个脏页时，InnoDB 存储引擎会检测该页所在区（extent）的有页，如果是脏页，那么一起进行刷新。这样做的好处显而易见，通过 AIO 可以将多个 IO 写入操作并为一个 IO 操作，故该工作机制在传统机械磁盘下有着显著的优势。

<h3 id="Mysql如何保证一致性和持久性">Mysql 如何保证一致性和持久性</3>
MySQL 为了保证 ACID 中的一致性和持久性，使用了 WAL(Write-Ahead Logging,写日志再写磁盘)。Redo log 就是一种 WAL 的应用。当数据库忽然掉电，再重新启动时，MySQL 以通过 Redo log 还原数据。也就是说，每次事务提交时，不用同步刷新磁盘数据文件，只需要同步新 Redo log 就足够了。
<h3 id="InnoDB的行锁模式">InnoDB 的行锁模式</h3>

共享锁(S)：用法 lock in share mode，又称读锁，允许一个事务去读一行，阻止其他务获得相同数据集的排他锁。若事务 T 对数据对象 A 加上 S 锁，则事务 T 可以读 A 但不能修改 A，他事务只能再对 A 加 S 锁，而不能加 X 锁，直到 T 释放 A 上的 S 锁。这保证了其他事务可以读 A，在 T 释放 A 上的 S 锁之前不能对 A 做任何修改。
排他锁(X)：用法 for update，又称写锁，允许获取排他锁的事务更新数据，阻止其他务取得相同的数据集共享读锁和排他写锁。若事务 T 对数据对象 A 加上 X 锁，事务 T 可以读 A 也可修改 A，其他事务不能再对 A 加任何锁，直到 T 释放 A 上的锁。在没有索引的情况下，InnoDB 只使用表锁。

<h3 id="为什么选择B-树作为索引结构">为什么选择 B+ 树作为索引结构</h3>

Hash 索引：Hash 索引底层是哈希表，哈希表是一种以 key-value 存储数据的结构，以多个数据在存储关系上是完全没有任何顺序关系的，所以，对于区间查询是无法直接通过索引查询，就需要全表扫描。所以，哈希索引只适用于等值查询的场景。而 B+ 树是一种多路平衡查询树，所他的节点是天然有序的（左子节点小于父节点、父节点小于右子节点），所以对于范围查询的时候不要做全表扫描
二叉查找树：解决了排序的基本问题，但是由于无法保证平衡，可能退化为链表。
平衡二叉树：通过旋转解决了平衡的问题，但是旋转操作效率太低。
红黑树：通过舍弃严格的平衡和引入红黑节点，解决了 AVL 旋转效率过低的问题，但在磁盘等场景下，树仍然太高，IO 次数太多。
B+ 树：在 B 树的基础上，将非叶节点改造为不存储数据纯索引节点，进一步降低了的高度；此外将叶节点使用指针连接成链表，范围查询更加高效。

<h3 id="B-树的叶子节点都可以存哪些东西">B+ 树的叶子节点都可以存哪些东西</h3>
可能存储的是整行数据，也有可能是主键的值。B+ 树的叶子节点存储了整行数据的主键索引，也被称之为聚簇索引。而索引 B+ Tree 的叶子节点存储了主键的值的是非主键索引，也被之为非聚簇索引
<h3 id="覆盖索引">覆盖索引</h3>
指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之实现了索引覆盖。
<h3 id="查询在什么时候不走-预期中的-索引">查询在什么时候不走（预期中的）索引</h3>

模糊查询 %like
索引列参与计算,使用了函数
非最左前缀顺序
where 单列索引对 null 判断
where 不等于

or 操作有至少一个字段没有索引
需要回表的查询结果集过大（超过配置的范围）

<h3 id="为什么Mysql数据库存储不建议使用NULL">为什么 Mysql 数据库存储不建议使用NULL</h3>

NOT IN 子查询在有 NULL 值的情况下返回永远为空结果，查询容易出错。
索引问题，单列索引无法存储 NULL 值，where 对 null 判断会不走索引。
如果在两个字段进行拼接（CONCAT 函数），首先要各字段进行非 null 判断，否则要任意一个字段为空都会造成拼接的结果为 null
如果有 Null column 存在的情况下，count(Null column)需要格外注意，null 值不会与统计。
Null 列需要更多的存储空间：需要一个额外的字节作为判断是否为 NULL 的标志位

<h3 id="explain命令概要">explain 命令概要</h3>

id:select 选择标识符
select_type:表示查询的类型。
table:输出结果集的表
partitions:匹配的分区
type:表示表的连接类型
possible_keys:表示查询时，可能使用的索引
key:表示实际使用的索引
key_len:索引字段的长度
ref:列与索引的比较
rows:扫描出的行数(估算的行数)
filtered:按表条件过滤的行百分比
Extra:执行情况的描述和说明

<h3 id="explain-中的-select-type-查询的类型-">explain 中的 select_type（查询的类）</h3>

SIMPLE(简单 SELECT，不使用 UNION 或子查询等)
PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的 select 标记为 PRIMARY)
UNION(UNION 中的第二个或后面的 SELECT 语句)
DEPENDENT UNION(UNION 中的第二个或后面的 SELECT 语句，取决于外面的查询
UNION RESULT(UNION 的结果，union 语句中第二个 select 开始后面所有 select)
SUBQUERY(子查询中的第一个 SELECT，结果不依赖于外部查询)
DEPENDENT SUBQUERY(子查询中的第一个 SELECT，依赖于外部查询)
DERIVED(派生表的 SELECT, FROM 子句的子查询)
UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的一行)

<h3 id="explain-中的-type-表的连接类型-">explain 中的 type（表的连接类型）</h3>

system：最快，主键或唯一索引查找常量值，只有一条记录，很少能出现<

li>
const：PK 或者 unique 上的等值查询
eq_ref：PK 或者 unique 上的 join 查询，等值匹配，对于前表的每一行(row)，后表有一行命中
ref：非唯一索引，等值匹配，可能有多行命中
range：索引上的范围扫描，例如：between/in
index：索引上的全集扫描，例如：InnoDB 的 count
ALL：最慢，全表扫描(full table scan)

<h3 id="explain-中的-Extra-执行情况的描述和说明-">explain 中的 Extra（执行情况的述和说明）</h3>

Using where:不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对的全部的请求列都是同一个索引的部分的时候，表示 mysql 服务器将在存储引擎检索行后再进行过滤/strong>
Using temporary：表示 MySQL 需要使用临时表来存储结果集，常见于排序和分组询，常见 group by ; order by
Using filesort：当 Query 中包含 order by 操作，而且无法利用索引完成的排序操作为“文件排序”
Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能
Impossible where：这个值强调了 where 语句会导致没有符合条件的行（通过收集计信息不可能存在结果）。
Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚函数结果中返回一行
No tables used：Query 语句中使用 from dual 或不含任何 from 子句</l
>

<h3 id="数据库优化指南">数据库优化指南</h3>

创建并使用正确的索引
只返回需要的字段
减少交互次数（批量提交）
设置合理的 Fetch Size（数据每次返回给客户端的条数）

<h2 id="JVM">JVM</h2>
<h3 id="运行时数据区域">运行时数据区域</h3>

程序计数器：程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计器来完成。是线程私有”的内存。
Java 虚拟机栈：与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方在执行的同时都会创建一个栈帧 ，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。<
li>
本地方法栈：本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法则为虚拟机使用到的 Native 方法服务。
Java 堆：对于大多数应用来说，Java 堆是 Java 虚拟机所管理的内存中最大的一块。Ja a 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象例，几乎所有的对象实例都在这里分配内存。

方法区：方法区用于存储已被虚拟机加载的类信息、常量、静态变量，如 static 修饰变量加载类的时候就被加载到方法区中。运行时常量池是方法区的一部分，class 文件除了有类的字、接口、方法等描述信息之外，还有常量池用于存放编译期间生成的各种字面量和符号引用。在老版 jk，方法区也被称为永久代。在 1.8 之后，由于永久代内存经常不够用或发生内存泄露，爆出异常 java.ang.OutOfMemoryError，所以在 1.8 之后废弃永久代，引入元空间的概念。元空间是方法区的在 HtSpot jvm 中的实现，元空间的本质和永久代类似，都是对 JVM 规范中方法区的实现。不过元空间永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。理论上取决于 32 位/64 系统可虚拟的内存大小。可见也不是无限制的，需要配置参数。

<h3 id="分代回收">分代回收</h3>
HotSpot JVM 把年轻代分为了三部分：1 个 Eden 区和 2 个 Survivor 区（分别叫 frm 和 to）。一般情况下，新创建的对象都会被分配到 Eden 区(一些大对象特殊处理),这些对象经过一次 Minor GC 后，如果仍然存活，将会被移到 Survivor 区。对象在 Survivor 区中每熬过一次 Mino GC，年龄就会增加 1 岁，当它的年龄增加到一定程度时，就会被移动到年老代中。
因为年轻代中的对象基本都是朝生夕死的，所以在年轻代的垃圾回收算法使用的是复算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活的对象复制到另外一块上面。复制算法不会产生内存碎片。
在 GC 开始的时候，对象只会存在于 Eden 区和名为“From”的 Survivor 区，Survivr 区“To”是空的。紧接着进行 GC，Eden 区中所有存活的对象都会被复制到“To”，而在“From区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxenuringThreshold 来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域经过这次 GC 后，Eden 区和 From 区已经被清空。这个时候，“From”和“To”会交换他们的角，也就是新的“To”就是上次 GC 前的“From”，新的“From”就是上次 GC 前的“To”。不管样，都会保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程，直到“To”区填满，“To”区被填满之后，会将所有对象移动到年老代中。
<h3 id="动态年龄计算">动态年龄计算</h3>
Hotspot 在遍历所有对象时，按照年龄从小到大对其所占用的大小进行累积，当累积某个年龄大小超过了 survivor 区的一半时，取这个年龄和 MaxTenuringThreshold 中更小的一个值作为新的晋升年龄阈值。
JVM 引入动态年龄计算，主要基于如下两点考虑：

如果固定按照 MaxTenuringThreshold 设定的阈值作为晋升条件： a）MaxTenuringhreshold 设置的过大，原本应该晋升的对象一直停留在 Survivor 区，直到 Survivor 区溢出，一旦出发生，Eden+Svuvivor 中对象将不再依据年龄全部提升到老年代，这样对象老化的机制就失效了。b）MaxTenuringThreshold 设置的过小，“过早晋升”即对象不能在新生代充分被回收，大量短期象被晋升到老年代，老年代空间迅速增长，引起频繁的 Major GC。分代回收失去了意义，严重影响C 性能。
相同应用在不同时间的表现不同：特殊任务的执行或者流量成分的变化，都会导致对的生命周期分布发生波动，那么固定的阈值设定，因为无法动态适应变化，会造成和上面相同的问题

<h3 id="常见的垃圾回收机制">常见的垃圾回收机制</h3>

引用计数法：引用计数法是一种简单但速度很慢的垃圾回收技术。每个对象都含有一引用计数器,当有引用连接至对象时,引用计数加 1。当引用离开作用域或被置为 null 时,引用计数减 1虽然管理引用计数的开销不大,但这项开销在整个程序生命周期中将持续发生。垃圾回收器会在含有全对象的列表上遍历,当发现某个对象引用计数为 0 时,就释放其占用的空间。
可达性分析算法：这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 没有任引用链相连（用图论的话来说，就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的

<h3 id="CMS的执行过程">CMS 的执行过程</h3>

初始标记(STW initial mark)：这个过程从垃圾回收的"根对象"开始，只扫描到能够和"对象"直接关联的对象，并作标记。所以这个过程虽然暂停了整个 JVM，但是很快就完成了。
并发标记(Concurrent marking)：这个阶段紧随初始标记阶段，在初始标记的基础上续向下追溯标记。并发标记阶段，应用程序的线程和并发标记的线程并发执行，所以用户不会感受到顿。
并发预清理(Concurrent precleaning)：并发预清理阶段仍然是并发的。在这个阶段虚拟机查找在执行并发标记阶段新进入老年代的对象(可能会有一些对象从新生代晋升到老年代， 或有一些对象被分配到老年代)。通过重新扫描，减少下一个阶段"重新标记"的工作，因为下一个阶段会top The World。
重新标记(STW remark)：这个阶段会暂停虚拟机，收集器线程扫描在 CMS 堆中剩余对象。扫描从"跟对象"开始向下追溯，并处理对象关联。
并发清理(Concurrent sweeping)：清理垃圾对象，这个阶段收集器线程和应用程序程并发执行。
并发重置(Concurrent reset)：这个阶段，重置 CMS 收集器的数据结构状态，等待下次垃圾回收。

<h3 id="G1的执行过程">G1 的执行过程</h3>

标记阶段：首先是初始标记(Initial-Mark),这个阶段也是停顿的(stop-the-word)，并会稍带触发一次 yong GC。
并发标记：这个过程在整个堆中进行，并且和应用程序并发运行。并发标记过程可能被 yong GC 中断。在并发标记阶段，如果发现区域对象中的所有对象都是垃圾，那个这个区域会被立回收(图中打 X)。同时，并发标记过程中，每个区域的对象活性(区域中存活对象的比例)被计算。</strng>
再标记：这个阶段是用来补充收集并发标记阶段产新的新垃圾。与之不同的是，G1 中用了更快的算法:SATB。
清理阶段：选择活性低的区域(同时考虑停顿时间)，等待下次 yong GC 一起收集，这过程也会有停顿(STW)。
回收/完成：新的 yong GC 清理被计算好的区域。但是有一些区域还是可能存在垃圾象，可能是这些区域中对象活性较高，回收不划算，也肯能是为了迎合用户设置的时间，不得不舍弃些区域的收集。

<h3 id="G1和CMS的比较">G1 和 CMS 的比较</h3>

CMS 收集器是获取最短回收停顿时间为目标的收集器，因为 CMS 工作时，GC 工作程与用户线程可以并发执行，以此来达到降低停顿时间的目的（只有初始标记和重新标记会 STW）但是 CMS 收集器对 CPU 资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用 CPU 资源而导致引用程序变慢，总吞吐量下降。
CMS 仅作用于老年代，是基于标记清除算法，所以清理的过程中会有大量的空间碎片
CMS 收集器无法处理浮动垃圾，由于 CMS 并发清理阶段用户线程还在运行，伴随程的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在本次收集中处它们，只好留待下一次 GC 时将其清理掉。
G1 是一款面向服务端应用的垃圾收集器，适用于多核处理器、大内存容量的服务端系。G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 STW 的停顿时间，它满足短时间停顿的同时达到一个高的吞吐量。
从 JDK 9 开始，G1 成为默认的垃圾回收器。当应用有以下任何一种特性时非常适合用 G1：Full GC 持续时间太长或者太频繁；对象的创建速率和存活率变动很大；应用不希望停顿时间长(于 0.5s 甚至 1s)。
G1 将空间划分成很多块（Region），然后他们各自进行回收。堆比较大的时候可以用，采用复制算法，碎片化问题不严重。整体上看属于标记整理算法,局部(region 之间)属于复制算法
G1 需要记忆集来记录新生代和老年代之间的引用关系，这种数据结构在 G1 中需要占

大量的内存，可能达到整个堆内存容量的 20% 甚至更多。而且 G1 中维护记忆集的成本较高，带来更高的执行负载，影响效率。所以 CMS 在小内存应用上的表现要优于 G1，而大内存应用上 G1 更有势，大小内存的界限是 6GB 到 8GB。（Card Table（CMS 中）的结构是一个连续的 byte[]数组，描 Card Table 的时间比扫描整个老年代的代价要小很多！G1 也参照了这个思路，不过采用了一种新数据结构 Remembered Set 简称 Rset。RSet 记录了其他 Region 中的对象引用本 Region 中对象关系，属于 points-into 结构（谁引用了我的对象）。而 Card Table 则是一种 points-out（我引用谁的对象）的结构，每个 Card 覆盖一定范围的 Heap（一般为 512Bytes）。G1 的 RSet 是在 Card able 的基础上实现的：每个 Region 会记录下别的 Region 有指向自己的指针，并标记这些指针分别哪些 Card 的范围内。 这个 RSet 其实是一个 Hash Table，Key 是别的 Region 的起始地址，Value 是一个集合，里面的元素是 Card Table 的 Index。每个 Region 都有一个对应的 Rset。）</strong

<h3 id="哪些对象可以作为GC-Roots">哪些对象可以作为 GC Roots</h3>

虚拟机栈（栈帧中的本地变量表）中引用的对象。
方法区中类静态属性引用的对象。
方法区中常量引用的对象。
本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。

<h3 id="GC中Stop-the-world-STW-">GC 中 Stop the world（STW）</h3

在执行垃圾收集算法时，Java 应用程序的其他所有除了垃圾收集收集器线程之外的线都被挂起。此时，系统只能允许 GC 线程进行运行，其他线程则会全部暂停，等待 GC 线程执行完毕才能再次运行。这些工作都是由虚拟机在后台自动发起和自动完成的，是在用户不可见的情况下把用正常工作的线程全部停下来，这对于很多的应用程序，尤其是那些对于实时性要求很高的程序来说是以接受的。
但不是说 GC 必须 STW,你也可以选择降低运行速度但是可以并发执行的收集算法，取决于你的业务。
<h3 id="垃圾回收算法">垃圾回收算法</h3>

停止-复制：先暂停程序的运行,然后将所有存活的对象从当前堆复制到另一个堆,没有复制的对象全部都是垃圾。当对象被复制到新堆时,它们是一个挨着一个的,所以新堆保持紧凑排列,然就可以按前述方法简单,直接的分配了。缺点是一浪费空间,两个堆之间要来回倒腾,二是当程序进入稳态时,可能只会产生极少的垃圾,甚至不产生垃圾,尽管如此,复制式回收器仍会将所有内存自一处复制到一处。
标记-清除：同样是从堆栈和静态存储区出发,遍历所有的引用,进而找出所有存活的对。每当它找到一个存活的对象,就会给对象一个标记,这个过程中不会回收任何对象。只有全部标记工完成的时候,清理动作才会开始。在清理过程中,没有标记的对象会被释放,不会发生任何复制动作。所剩下的堆空间是不连续的,垃圾回收器如果要希望得到连续空间的话,就得重新整理剩下的对象。</strog>
标记-整理：它的第一个阶段与标记/清除算法是一模一样的，均是遍历 GC Roots，然将存活的对象标记。移动所有存活的对象，且按照内存地址次序依次排列，然后将末端内存地址以后内存全部回收。因此，第二阶段才称为整理阶段。
分代收集算法：把 Java 堆分为新生代和老年代，然后根据各个年代的特点采用最合适收集算法。新生代中，对象的存活率比较低，所以选用复制算法，老年代中对象存活率高且没有额外间对它进行分配担保，所以使用“标记-清除”或“标记-整理”算法进行回收。

<h3 id="Minor-GC和Full-GC触发条件">Minor GC 和 Full GC 触发条件</h3

Minor GC 触发条件：当 Eden 区满时，触发 Minor GC。
Full GC 触发条件：

调用 System.gc 时，系统建议执行 Full GC，但是不必然执行

老年代空间不足
方法区空间不足
通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存
由 Eden 区、From Space 区向 To Space 区复制时，对象大小大于 To Space 可用内，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

<h3 id="对象什么时候进入老年代">对象什么时候进入老年代</h3>

大对象直接进入老年代。 虚拟机提供了一个阈值参数，令大于这个设置值的对象直接老年代中分配。如果大对象进入新生代，新生代采用的复制算法收集内存，会导致在 Eden 区和两个urvivor 区之间发生大量的内存复制，应该避免这种情况。
长期存活的对象进入老年代。 虚拟机给每个对象定义了一个年龄计数器，对象在 Eden 区出生，经过一次 Minor GC 后仍然存活，并且能被 Survivor 区容纳的话，将被移动到 Survivor 区，此时对象年龄设为 1。然后对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1，当年龄超设定的阈值时，就会被移动到老年代中。
动态对象年龄判定： 如果在 Survivor 空间中所有相同年龄的对象，大小总和大于 Survvor 空间的一半，那么年龄大于或等于该年龄的对象就直接进入老年代，无须等到阈值中要求的年龄
空间分配担保： 如果老年代中最大可用的连续空间大于新生代所有对象的总空间，那么Minor GC 是安全的。如果老年代中最大可用的连续空间大于历代晋升到老年代的对象的平均大小，进行一次有风险的 Minor GC，如果小于平均值，就进行 Full GC 来让老年代腾出更多的空间。因为生代使用的是复制算法，为了内存利用率，只使用其中一个 Survivor 空间来做轮换备份，因此如果量对象在 Minor GC 后仍然存活，导致 Survivor 空间不够用，就会通过分配担保机制，将多出来的象提前转到老年代，但老年代要进行担保的前提是自己本身还有容纳这些对象的剩余空间，由于无法前知道会有多少对象存活下来，所以取之前每次晋升到老年代的对象的平均大小作为经验值，与老年的剩余空间做比较。

<h3 id="TLAB">TLAB</h3>
在 Java 中，典型的对象不再堆上分配的情况有两种：TLAB 和栈上分配（通过逃逸分）。JVM 在内存新生代 Eden Space 中开辟了一小块线程私有的区域，称作 TLAB（Thread-local allcation buffer）。默认设定为占用 Eden Space 的 1%。在 Java 程序中很多对象都是小对象且用过丢，它们不存在线程共享也适合被快速 GC，所以对于小对象通常 JVM 会优先分配在 TLAB 上，并且LAB 上的分配由于是线程私有所以没有锁开销。因此在实践中分配多个小对象的效率通常比分配一个对象的效率要高。也就是说，Java 中每个线程都会有自己的缓冲区称作 TLAB（Thread-local allocatin buffer），每个 TLAB 都只有一个线程可以操作，TLAB 结合 bump-the-pointer 技术可以实现快的对象分配，而不需要任何的锁进行同步，也就是说，在对象分配的时候不用锁住整个堆，而只需要自己的缓冲区分配即可。
<h3 id="Java对象分配的过程">Java 对象分配的过程</h3>

编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配。如果是在堆上分配，进入 2.
如果 tlab_top + size <= tlab_end，则在在 TLAB 上直接分配对象并增加 tlab_top 的值，如果现有的 TLAB 不足以存放当前对象则 3.
重新申请一个 TLAB，并再次尝试存放当前对象。如果放不下，则 4。
在 Eden 区加锁（这个区是多线程共享的），如果 eden_top + size <= eden_end 则将对象存放在 Eden 区，增加 eden_top 的值，如果 Eden 区不足以存放，则 5。
执行一次 Young GC（minor collection）
经过 Young GC 之后，如果 Eden 区任然不足以存放当前对象，则直接分配到老年代

<h3 id="对象内存分配的两种方法">对象内存分配的两种方法</h3>

指针碰撞(Serial、ParNew 等带 Compact 过程的收集器)：假设 Java 堆中内存是绝规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式为“指针碰撞”（Bump the Pointer）。
空闲列表(CMS 这种基于 Mark-Sweep 算法的收集器)：如果 Java 堆中的内存并不是整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实，并更新列表上的记录，这种分配方式称为“空闲列表”（Free List）。

<h3 id="JVM类加载过程">JVM 类加载过程</h3>
类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载验证、准备、解析、初始化、使用和卸载 7 个阶段。

加载：通过一个类的全限定名来获取定义此类的二进制字节流，将这个字节流所代表静态存储结构转化为方法区的运行时数据结构，在内存中生成一个代表这个类的 Class 对象，作为方去这个类的各种数据的访问入口
验证：验证是连接阶段的第一步，这一阶段的目的是确保 Class 文件的字节流中包含信息符合当前虚拟机的要求，并且不会危害虚拟自身的安全。
准备：准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使的内存都将在方法去中进行分配。这时候进行内存分配的仅包括类变量（static），而不包括实例变，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。
解析：解析阶段是虚拟机将常量池内的符号（Class 文件内的符号）引用替换为直接引（指针）的过程。
初始化：初始化阶段是类加载过程的最后一步，开始执行类中定义的 Java 程序代码（节码）。

<h3 id="双亲委派模型">双亲委派模型</h3>
双亲委派的意思是如果一个类加载器需要加载类，那么首先它会把这个类请求委派给类加载器去完成，每一层都是如此。一直递归到顶层，当父加载器无法完成这个请求时，子类才会尝去加载。
<h3 id="双亲委派模型的-破坏-">双亲委派模型的"破坏"</h3>
一个典型的例子便是 JNDI 服务，JNDI 现在已经是 Java 的标准服务，它的代码由启类加载器去加载(在 JDK 1.3 时放进去的 rt.jar)，但 JNDI 的目的就是对资源进行集中管理和查找，它要调用由独立厂商实现并部署在应用程序的 ClassPath 下的 JNDI 接口提供者(SPI,Service Provider Interface)的代码，但启动类加载器不可能“认识”这些代码那该怎么办?
为了解决这个问题，Java 设计团队只好引入了一个不太优雅的设计:线程上下文类加载(Thread Context ClassLoader)。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoser()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承 一个，如果在应用程序的全范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。
有了线程上下文类加载器，就可以做一些“舞弊”的事情了，JNDI 服务使用这个线程下 文类加载器去加载所需要的 SPI 代码，也就是父类加载器请求子类加载器去完成类加载的动 作，种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，实际上已经 违背了双亲委派型的一般性原则，但这也是无可奈何的事情。Java 中所有涉及 SPI 的加载动 作基本上都采用这种方，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。
<h3 id="JVM锁优化和膨胀过程">JVM 锁优化和膨胀过程</h3>

自旋锁：自旋锁其实就是在拿锁时发现已经有线程拿了锁，自己如果去拿会阻塞自己这个时候会选择进行一次忙循环尝试。也就是不停循环看是否能等到上个线程自己释放锁。自适应自锁指的是例如第一次设置最多自旋 10 次，结果在自旋的过程中成功获得了锁，那么下一次就可以设成最多自旋 20 次。
锁粗化：虚拟机通过适当扩大加锁的范围以避免频繁的拿锁释放锁的过程。
锁消除：通过逃逸分析发现其实根本就没有别的线程产生竞争的可能（别的线程没有界量的引用），或者同步块内进行的是原子操作，而“自作多情”地给自己加上了锁。有可能虚拟机

直接去掉这个锁。
偏向锁：在大多数的情况下，锁不仅不存在多线程的竞争，而且总是由同一个线程获。因此为了让线程获得锁的代价更低引入了偏向锁的概念。偏向锁的意思是如果一个线程获得了一个向锁，如果在接下来的一段时间中没有其他线程来竞争锁，那么持有偏向锁的线程再次进入或者退出一个同步代码块，不需要再次进行抢占锁和释放锁的操作。
轻量级锁：当存在超过一个线程在竞争同一个同步代码块时，会发生偏向锁的撤销。前线程会尝试使用 CAS 来获取锁，当自旋超过指定次数(可以自定义)时仍然无法获得锁，此时锁会膨升级为重量级锁。
重量级锁：重量级锁依赖对象内部的 monitor 锁来实现，而 monitor 又依赖操作系的 MutexLock（互斥锁）。当系统检查到是重量级锁之后，会把等待想要获取锁的线程阻塞，被阻的线程不会消耗 CPU，但是阻塞或者唤醒一个线程，都需要通过操作系统来实现。

<h3 id="什么情况下需要开始类加载过程的第一个阶段加载">什么情况下需要开始类加载程的第一个阶段加载</h3>

遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 new 关键实例化对象的时候、读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的态字段除外）的时候，以及调用一个类的静态方法的时候。
使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化则需要先触发其初始化。
当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类初始化。
当虚拟机启动时，用户需要指定一个要执行的主类（包含 main（）方法的那个类），拟机会先初始化这个主类。

<h3 id="i--操作的字节码指令">i++ 操作的字节码指令</h3>

将 int 类型常量加载到操作数栈顶
将 int 类型数值从操作数栈顶取出，并存储到到局部变量表的第 1 个 Slot 中
将 int 类型变量从局部变量表的第 1 个 Slot 中取出，并放到操作数栈顶
将局部变量表的第 1 个 Slot 中的 int 类型变量加 1
表示将 int 类型数值从操作数栈顶取出，并存储到到局部变量表的第 1 个 Slot 中，即 i中

<h3 id="JVM性能监控">JVM 性能监控</h3>

JDK 的命令行工具

jps(虚拟机进程状况工具)：jps 可以列出正在运行的虚拟机进程，并显示虚拟机执行主(Main Class,main()函数所在的类)名称 以及这些进程的本地虚拟机唯一 ID(Local Virtual Machine Identifier,LVMID)。
jstat(虚拟机统计信息监视工具)：jstat 是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据。
jinfo(Java 配置信息工具)：jinfo 的作用是实时地查看和调整虚拟机各项参数。
jmap(Java 内存映像工具)：命令用于生成堆转储快照(一般称为 heapdump 或 dump 文 件)。如果不使用 jmap 命令，要想获取 Java 堆转储快照，还有一些比较“暴力”的手段:譬如 在第 2 章中用过的-XX:+HeapDumpOnOutOfMemoryError 参数，可以让虚拟机在 OOM 异常出 现之自动生成 dump 文件。jmap 的作用并不仅仅是为了获取 dump 文件，它还可以查询 finalize 执行列、Java 堆和永 久代的详细信息，如空间使用率、当前用的是哪种收集器等。

jhat(虚拟机堆转储快照分析工具)：jhat 命令与 jmap 搭配使用，来分析 jmap 生成的 转储快照。jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 文件的分析结果后，可以在览器中查看。
jstack(Java 堆栈跟踪工具)：jstack 命令用于生成虚拟机当前时刻的线程快照。线程快就是当前虚拟机内每一条线程正在执行的方法堆栈 的集合，生成线程快照的主要目的是定位线程出现时间停顿的原因，如线程间死锁、死循 环、请求外部资源导致的长时间等待等都是导致线程长时间停的常见原因。线程出现停顿 的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的程到底在后台做些 什么事情，或者等待着什么资源。

JDK 的可视化工具

JConsole
VisualVM

<h3 id="JVM常见参数">JVM 常见参数</h3>

-Xms20M：表示设置 JVM 启动内存的最小值为 20M，必须以 M 为单位</i>
-Xmx20M：表示设置 JVM 启动内存的最大值为 20M，必须以 M 为单位。将-Xmx 和 Xms 设置为一样可以避免 JVM 内存自动扩展。大的项目-Xmx 和-Xms 一般都要设置到 10G、20G 至还要高
-verbose:gc：表示输出虚拟机中 GC 的详细情况
-Xss128k：表示可以设置虚拟机栈的大小为 128k
-Xoss128k：表示设置本地方法栈的大小为 128k。不过 HotSpot 并不区分虚拟机栈本地方法栈，因此对于 HotSpot 来说这个参数是无效的
-XX:PermSize=10M：表示 JVM 初始分配的永久代（方法区）的容量，必须以 M 为位
-XX:MaxPermSize=10M：表示 JVM 允许分配的永久代（方法区）的最大容量，必以 M 为单位，大部分情况下这个参数默认为 64M
-Xnoclassgc：表示关闭 JVM 对类的垃圾回收
-XX:+TraceClassLoading 表示查看类的加载信息
-XX:+TraceClassUnLoading：表示查看类的卸载信息
-XX:NewRatio=4：表示设置年轻代（包括 Eden 和两个 Survivor 区）/老年代 的大比值为 1：4，这意味着年轻代占整个堆的 1/5
-XX:SurvivorRatio=8：表示设置 2 个 Survivor 区：1 个 Eden 区的大小比值为 2:8 这意味着 Survivor 区占整个年轻代的 1/5，这个参数默认为 8
-Xmn20M：表示设置年轻代的大小为 20M
-XX:+HeapDumpOnOutOfMemoryError：表示可以让虚拟机在出现内存溢出异常时 Dump 出当前的堆内存转储快照
-XX:+UseG1GC：表示让 JVM 使用 G1 垃圾收集器
-XX:+PrintGCDetails：表示在控制台上打印出 GC 具体细节
-XX:+PrintGC：表示在控制台上打印出 GC 信息
-XX:PretenureSizeThreshold=3145728：表示对象大于 3145728（3M）时直接进老年代分配，这里只能以字节作为单位
-XX:MaxTenuringThreshold=1：表示对象年龄大于 1，自动进入老年代,如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象年轻代的存活时间，增加在年轻代被回收的概率。
-XX:CompileThreshold=1000：表示一个方法被调用 1000 次之后，会被认为是热点码，并触发即时编译
-XX:+PrintHeapAtGC：表示可以看到每次 GC 前后堆内存布局

-XX:+PrintTLAB：表示可以看到 TLAB 的使用情况
-XX:+UseSpining：开启自旋锁
-XX:PreBlockSpin：更改自旋锁的自旋次数，使用这个参数必须先开启自旋锁</stron
>
-XX:+UseSerialGC：表示使用 jvm 的串行垃圾回收机制，该机制适用于单核 cpu 的
境下
-XX:+UseParallelGC：表示使用 jvm 的并行垃圾回收机制，该机制适合用于多 cpu
制，同时对响应时间无强硬要求的环境下，使用-XX:ParallelGCThreads=设置并行垃圾回收的线程
，此值可以设置与机器处理器数量相等。
-XX:+UseParallelOldGC：表示年老代使用并行的垃圾回收机制
-XX:+UseConcMarkSweepGC：表示使用并发模式的垃圾回收机制，该模式适用于
响应时间要求高，具有多 cpu 的环境下
-XX:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间，如果无法满足
时间，JVM 会自动调整年轻代大小，以满足此值。
-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小
相应的 Survivor 区比例，以达到目标系统规定的最低响应时间或者收集频率等，此值建议使用并行
集器时，一直打开

<h3 id="JVM调优目标-何时需要做jvm调优">JVM 调优目标-何时需要做 jvm 调优</stro
g></h3>

heap 内存（老年代）持续上涨达到设置的最大内存值；
Full GC 次数频繁；
GC 停顿时间过长（超过 1 秒）；
应用出现 OutOfMemory 等内存异常；
应用中有使用本地缓存且占用大量内存空间；
系统吞吐量与响应性能不高或下降。

<h3 id="JVM调优实战">JVM 调优实战</h3>

Major GC 和 Minor GC 频繁**首先优化 Minor GC 频繁问题。通常情况下，由于新生代空
较小，Eden 区很快被填满，就会导致频繁 Minor GC，因此可以通过增大新生代空间来降低 Minor
C 的频率。例如在相同的内存分配率的前提下，新生代中的 Eden 区增加一倍，Minor GC 的次数就
减少一半。**扩容 Eden 区虽然可以减少 Minor GC 的次数，但会增加单次 Minor GC 时
么？扩容后，Minor GC 时增加了 T1（扫描时间），但省去 T2（复制对象）的时间，更重要的是对
虚拟机来说，复制对象的成本要远高于扫描成本，所以，单次 Minor GC 时间更多取决于 GC 后存活
象的数量，而非 Eden 区的大小。因此如果堆中短期对象很多，那么扩容新生代，单次 Minor GC 时
不会显著增加。
请求高峰期发生 GC，导致服务可用性下降****由于跨代引用的存在，CMS 在 Remark
阶段必须扫描整个堆，同时为了避免扫描时新生代有很多对象，增加了可中断的预清理阶段用来等待
inor GC 的发生。只是该阶段有时间限制，如果超时等不到 Minor GC，Remark 时新生代仍然有很
对象，我们的调优策略是，通过参数强制 Remark 前进行一次 Minor GC，从而降低 Remark 阶段的
间。 另外，类似的 JVM 是如何避免 Minor GC 时扫描全堆的？ 经过统计信息显示，老年代持有新
代对象引用的情况不足 1%，根据这一特性 JVM 引入了卡表（card table）来实现这一目的。卡表的
体策略是将老年代的空间分成大小为 512B 的若干张卡（card）。卡表本身是单字节数组，数组中的
个元素对应着一张卡，当发生老年代引用新生代时，虚拟机将该卡对应的卡表元素设置为适当的值。
上图所示，卡表 3 被标记为脏（卡表还有另外的作用，标识并发标记阶段哪些块被修改过），之后 Mi
or GC 时通过扫描卡表就可以很快的识别哪些卡中存在老年代指向新生代的引用。这样虚拟机通过空
换时间的方式，避免了全堆扫描。
STW 过长的 GC****对于性能要求很高的服务，建议将 MaxPermSize 和 MinPermSiz
设置成一致（JDK8 开始，Perm 区完全消失，转而使用元空间。而元空间是直接存在内存中，不在 J
M 中），Xms 和 Xmx 也设置为相同，这样可以减少内存自动扩容和收缩带来的性能损失。虚拟机启
的时候就会把参数中所设定的内存全部化为私有，即使扩容前有一部分内存不会被用户代码用到，这
分内存在虚拟机中被标识为虚拟内存，也不会交给其他进程使用。

外部命令导致系统缓慢****一个数字校园应用系统，发现请求响应时间比较慢，通过作系统的 mpstat 工具发现 CPU 使用率很高，并且系统占用绝大多数的 CPU 资 源的程序并不是应系统本身。每个用户请求的处理都需要执行一个外部 shell 脚本来获得系统的一些信息，执行这个 shel 脚本是通过 Java 的 Runtime.getRuntime().exec()方法来调用的。这种调用方式可以达到目的，但它在 Java 虚拟机中是非常消耗资源的操作，即使外部命令本身能很快执行完毕，频繁调用时创建进程的开销也非常可观。Java 虚拟机执行这个命令的过程是:首先克隆一个和当前虚拟机拥有一样环境变的进程，再用这个新的进程去执行外部命令，最后再退出这个进程。如果频繁执行这个操作，系统的耗会很大，不仅是 CPU，内存负担也很重。用户根据建议去掉这个 Shell 脚本执行的语句，改为使用 ava 的 API 去获取这些信息后，系统很快恢复了正常。
由 Windows 虚拟内存导致的长时间停顿**一个带心跳检测功能的 GUI 桌面程序，每 15 秒发送一次心跳检测信号，如果对方 30 秒以内都没有信号返回，那就认为和对方程序的连接已经断开程序上线后发现心跳 检测有误报的概率，查询日志发现误报的原因是程序会偶尔出现间隔约一分钟左的时间完 全无日志输出，处于停顿状态。****因为是桌面程序，所需的内存并不大(-Xmx256m)，所开始并没有想到是 GC 导致的 程序停顿，但是加入参数-XX:+PrintGCApplicationStoppedTime-XX:PrintGCDateStamps- Xloggc:gclog.log 后，从 GC 日志文件中确认了停顿确实是由 GC 导致的，部分 GC 时间都控 制在 100 毫秒以内，但偶尔就会出现一次接近 1 分钟的 GC。****从 GC 日志中找长时间停顿的具体日志信息(添加了-XX:+PrintReferenceGC 参数)， 找到的日志片段如下所示。从志中可以看出，真正执行 GC 动作的时间不是很长，但从准 备开始 GC，到真正开始 GC 之间所消耗时间却占了绝大部分。**除 GC 日志之外，还观察到这个 GUI 程序内存变化的一个特点，它最小化的时候，资源 管理中显示的占用内存大幅度减小，但是虚拟内存则没有变化，因此怀疑程序最小化时它的工作内存被自动交换到磁盘的页面文件之中了，这样发生 GC 时就有可能因为恢复页面件的操作而导致不正常的 GC 停顿。在 Java 的 GUI 程序中要避免这种现象，可以 加入参数“-Dsun.wt.keepWorkingSetOnMinimize=true”来解决。

<h2 id="Java基础">Java 基础</h2>
<h3 id="HashMap和ConcurrentHashMap">HashMap 和 ConcurrentHashMap</h3>
由于 HashMap 是线程不同步的，虽然处理数据的效率高，但是在多线程的情况下存着安全问题，因此设计了 CurrentHashMap 来解决多线程安全问题。
HashMap 在 put 的时候，插入的元素超过了容量（由负载因子决定）的范围就会触扩容操作，就是 rehash，这个会重新将原数组的内容重新 hash 到新的扩容数组中，在多线程的环境，存在同时其他的元素也在进行 put 操作，如果 hash 值相同，可能出现同时在同一数组下用链表表，造成闭环，导致在 get 时会出现死循环，所以 HashMap 是线程不安全的。
HashMap 的环：若当前线程此时获得 ertry 节点，但是被线程中断无法继续执行，时线程二进入 transfer 函数，并把函数顺利执行，此时新表中的某个位置有了节点，之后线程一获得行权继续执行，因为并发 transfer，所以两者都是扩容的同一个链表，当线程一执行到 e.next = newable[i] 的时候，由于线程二之前数据迁移的原因导致此时 new table[i] 上就有 ertry 存在，所以线程执行的时候，会将 next 节点，设置为自己，导致自己互相使用 next 引用对方，因此产生链表，导致循环。
在 JDK1.7 版本中，ConcurrentHashMap 维护了一个 Segment 数组，Segment 这类继承了重入锁 ReentrantLock，并且该类里面维护了一个 HashEntry<K,V>[] table 数组，写操作 put，remove，扩容的时候，会对 Segment 加锁，所以仅仅影响这个 Segment，不同的 Sement 还是可以并发的，所以解决了线程的安全问题，同时又采用了分段锁也提升了并发的效率。在 JK1.8 版本中，ConcurrentHashMap 摒弃了 Segment 的概念，而是直接用 Node 数组 + 链表 + 红树的数据结构来实现，并发控制使用 Synchronized 和 CAS 来操作，整个看起来就像是优化过且线安全的 HashMap。
<h3 id="HashMap如果我想要让自己的Object作为K应该怎么办">HashMap 如果我想要自己的 Object 作为 K 应该怎么办</h3>

重写 hashCode()是因为需要计算存储数据的存储位置，需要注意不要试图从散列码计中排除掉一个对象的关键部分来提高性能，这样虽然能更快但可能会导致更多的 Hash 碰撞；
重写 equals()方法，需要遵守自反性、对称性、传递性、一致性以及对于任何非 null 引用值 x，x.equals(null)必须返回 false 的这几个特性，目的是为了保证 key 在哈希表中的唯一性（J

va 建议重写 equal 方法的时候需重写 hashcode 的方法）

<h3 id="volatile">volatile</h3>
volatile 在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线修改一个共享变量时，另外一个线程能读到这个修改的值。(共享内存，私有内存)
<h3 id="Atomic类的CAS操作">Atomic 类的 CAS 操作</h3>
CAS 是英文单词 CompareAndSwap 的缩写，中文意思是：比较并替换。CAS 需要有 3 个操作数：内存地址 V，旧的预期值 A，即将要更新的目标值 B。CAS 指令执行时，当且仅当内存址 V 的值与预期值 A 相等时，将内存地址 V 的值修改为 B，否则就什么都不做。整个比较并替换的作是一个原子操作。如 Intel 处理器，比较并交换通过指令的 cmpxchg 系列实现。
<h3 id="CAS操作ABA问题-">CAS 操作 ABA 问题：</h3>
如果在这段期间它的值曾经被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为从来没有被改变过。Java 并发包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStapedReference”，它可以通过控制变量值的版本来保证 CAS 的正确性。
<h3 id="Synchronized的四种使用方式">Synchronized 的四种使用方式</h
>

synchronized(this)：当 a 线程执行到该语句时，锁住该语句所在对象 object，其它程无法访问 object 中的所有 synchronized 代码块。
synchronized(obj)：锁住对象 obj，其它线程对 obj 中的所有 synchronized 代码块访问被阻塞。
synchronized method()：与（1）类似，区别是（1）中线程执行到某方法中的该语才会获得锁，而对方法加锁则是当方法调用时立刻获得锁。
synchronized static method()：当线程执行到该语句时，获得锁，所有调用该方法其它线程阻塞，但是这个类中的其它非 static 声明的方法可以访问，即使这些方法是用 synchronized 声明的，但是 static 声明的方法会被阻塞；注意，这个锁与对象无关。

前三种方式加的是对象锁，但是如果（2）中 obj 是一个 class 类型的对象，那么加的类锁，并且锁的范围比（4）还要大；如果该 class 类型的某个实例对象获得了类锁，那么该 class 类的所有实例对象的 synchronized 代码块均无法访问。
<h3 id="Synchronized和Lock的区别">Synchronized 和 Lock 的区别</h3>

首先 synchronized 是 java 内置关键字在 jvm 层面，Lock 是个 java 类。
/li>
synchronized 无法判断是否获取锁的状态，Lock 可以判断是否获取到锁，并且可以动尝试去获取锁。
synchronized 会自动释放锁(a 线程执行完同步代码会释放锁； b 线程执行过程中发异常会释放锁)，Lock 需在 finally 中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁。<
strong>
用 synchronized 关键字的两个线程 1 和线程 2，如果当前线程 1 获得锁，线程 2 线等待。如果线程 1 阻塞，线程 2 则会一直等待下去，而 Lock 锁就不一定会等待下去，如果尝试获取到锁，线程可以不用一直等待就结束了。
synchronized 的锁可重入、不可中断、非公平，而 Lock 锁可重入、可判断、可公平两者皆可）
Lock 锁适合大量同步的代码的同步问题，synchronized 锁适合代码少量的同步问题

<h3 id="AQS理论的数据结构">AQS 理论的数据结构</h3>
AQS 内部有 3 个对象，一个是 state（用于计数器，类似 gc 的回收计数器），一个线程标记（当前线程是谁加锁的），一个是阻塞队列。
AQS 是自旋锁，在等待唤醒的时候，经常会使用自旋的方式，不停地尝试获取锁，直被其他线程获取成功。
AQS 有两个队列，同步对列和条件队列。同步队列依赖一个双向链表来完成同步状态管理，当前线程获取同步状态失败后，同步器会将线程构建成一个节点，并将其加入同步队列中。通过

signal 或 signalAll 将条件队列中的节点转移到同步队列。
<h3 id="如何指定多个线程的执行顺序">如何指定多个线程的执行顺序</h3>

设定一个 orderNum，每个线程执行结束之后，更新 orderNum，指明下一个要执行线程。并且唤醒所有的等待线程。
在每一个线程的开始，要 while 判断 orderNum 是否等于自己的要求值，不是，则 wait，是则执行本线程。

<h3 id="为什么要使用线程池">为什么要使用线程池</h3>

减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
可以根据系统的承受能力，调整线程池中工作线程的数目，放置因为消耗过多的内存而把服务器累趴下。

<h3 id="核心线程池ThreadPoolExecutor内部参数">核心线程池 ThreadPoolExecutor 部参数</h3>

corePoolSize：指定了线程池中的线程数量
maximumPoolSize：指定了线程池中的最大线程数量
keepAliveTime：线程池维护线程所允许的空闲时间
unit: keepAliveTime 的单位。
workQueue：任务队列，被提交但尚未被执行的任务。
threadFactory：线程工厂，用于创建线程，一般用默认的即可。
handler：拒绝策略。当任务太多来不及处理，如何拒绝任务。

<h3 id="线程池的执行流程">线程池的执行流程</h3>

如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务
如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列
如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是创建非核心线程立刻运行这个任务
如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程会抛出异常 RejectExecutionException

<h3 id="线程池都有哪几种工作队列">线程池都有哪几种工作队列</h3>

ArrayBlockingQueue：底层是数组，有界队列，如果我们要使用生产者-消费者模式这是非常好的选择。
LinkedBlockingQueue：底层是链表，可以当做无界和有界队列来使用，所以大家不以为它就是无界队列。
SynchronousQueue：本身不带有空间来存储任何元素，使用上可以选择公平模式和公平模式。
PriorityBlockingQueue：无界队列，基于数组，数据结构为二叉堆，数组第一个也是的根节点总是最小值。

举例 ArrayBlockingQueue 实现并发同步的原理：原理就是读操作和写操作都需要获到 AQS 独占锁才能进行操作。如果队列为空，这个时候读操作的线程进入到读线程队列排队，等待线程写入新的元素，然后唤醒读线程队列的第一个等待线程。如果队列已满，这个时候写操作的线程入到写线程队列排队，等待读线程将队列元素移除腾出空间，然后唤醒写线程队列的第一个等待线程
<h3 id="线程池的拒绝策略">线程池的拒绝策略</h3>

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出 RejectedExecutionException 异
。
ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新提交
拒绝的任务
ThreadPoolExecutor.CallerRunsPolicy：由调用线程（提交任务的线程）处理该任务
/strong>

<h3 id="线程池的正确创建方式">线程池的正确创建方式</h3>
<p>不能用 Executors，newFixed 和 newSingle，因为队列无限大，容易造成耗尽资源和
OOM，newCached 和 newScheduled 最大线程数是 Integer.MAX_VALUE，线程创建过多和 OO
。应该通过 ThreadPoolExecutor 手动创建。</p>
<h3 id="线程提交submit--和execute--有什么区别">线程提交 submit()和 execute()有
么区别</h3>

submit()相比于 excute()，支持 callable 接口，也可以获取到任务抛出来的异常</str
ng>
可以获取到任务返回结果
用 submit()方法执行任务，用 Future.get()获取异常

<h3 id="线程池的线程数量怎么确定">线程池的线程数量怎么确定</h3>

一般来说，如果是 CPU 密集型应用，则线程池大小设置为 N+1。
一般来说，如果是 IO 密集型应用，则线程池大小设置为 2N+1。
在 IO 优化中，线程等待时间所占比例越高，需要越多线程，线程 CPU 时间所占比
越高，需要越少线程。这样的估算公式可能更适合：最佳线程数目 = （（线程等待时间 + 线程 CPU
间）/线程 CPU 时间 ） CPU 数目*

<h3 id="如何实现一个带优先级的线程池">如何实现一个带优先级的线程池</
3>
<p>利用 priority 参数，继承 ThreadPoolExecutor 使用 PriorityBlockingQueue 优先级
列。</p>
<h3 id="ThreadLocal的原理和实现">ThreadLocal 的原理和实现</h3>
<p>ThreadLoal 变量，线程局部变量，同一个 ThreadLocal 所包含的对象，在不同的 Thr
ad 中有不同的副本。ThreadLocal 变量通常被 private static 修饰。当一个线程结束时，它所使用的
有 ThreadLocal 相对的实例副本都可被回收。</p>
<p>一个线程内可以存在多个 ThreadLocal 对象，所以其实是 ThreadLocal 内部维护了
个 Map ， 这个 Map 不是直接使用的 HashMap ， 而是 ThreadLocal 实现的一个叫做 ThreadLocal
ap 的静态内部类。而我们使用的 get()、set() 方法其实都是调用了这个 ThreadLocalMap 类对应的
et()、set() 方法。这个储值的 Map 并非 ThreadLocal 的成员变量，而是 java.lang.Thread 类的成
变量。ThreadLocalMap 实例是作为 java.lang.Thread 的成员变量存储的，每个线程有唯一的一个 th
eadLocalMap。这个 map 以 ThreadLocal 对象为 key，”线程局部变量”为值，所以一个线程下
以保存多个”线程局部变量”。对 ThreadLocal 的操作，实际委托给当前 Thread，每个 Thread 都
有自己独立的 ThreadLocalMap 实例，存储的仓库是 Entry[] table；Entry 的 key 为 ThreadLocal，
alue 为存储内容；因此在并发环境下，对 ThreadLocal 的 set 或 get，不会有任何问题。由于 Tomc
t 线程池的原因，我最初使用的”线程局部变量”保存的值，在下一次请求依然存在（同一个线程处
），这样每次请求都是在本线程中取值。所以在线程池的情况下，处理完成后主动调用该业务 treadL
cal 的 remove()方法，将”线程局部变量”清空，避免本线程下次处理的时候依然存在旧数据。</str
ng></p>
<h3 id="ThreadLocal为什么要使用弱引用和内存泄露问题">ThreadLocal 为什么要使用
引用和内存泄露问题</h3>
<p>在 ThreadLocal 中内存泄漏是指 ThreadLocalMap 中的 Entry 中的 key 为 null，而 v
lue 不为 null。因为 key 为 null 导致 value 一直访问不到，而根据可达性分析导致在垃圾回收的时

进行可达性分析的时候,value 可达从而不会被回收掉，但是该 value 永远不能被访问到，这样就存在内存泄漏。如果 key 是强引用，那么发生 GC 时 ThreadLocalMap 还持有 ThreadLocal 的强引用，导致 ThreadLocal 不会被回收，从而导致内存泄漏。弱引用 ThreadLocal 不会内存泄漏，对应的 val e 在下一次 ThreadLocalMap 调用 set、get、remove 方法时被清除，这算是最优的解决方案。</st ong></p>
<p>Map 中的 key 为一个 threadlocal 实例.如果使用强引用，当 ThreadLocal 对象（假为 ThreadLocal@123456）的引用被回收了，ThreadLocalMap 本身依然还有 ThreadLocal@123456 的强引用，如果没有手动删除这个 key，则 Thread ocal@123456 不会被回收，所以只要当前线程不消亡，ThreadLocalMap 用的那些对象就不会被回收，可以认为这导致 Entry 内存泄漏。</p>
<p>如果使用弱引用，那指向 ThreadLocal@123456 对象的引用两个：ThreadLocal 强引用和 ThreadLocalMap 中 Entry 的弱引用。一旦 ThreadLocal 强引用被回，则指向 ThreadLocal@123456 的就只有弱引用了，在下次 gc 的时候，这 ThreadLocal@123456 就会被回收。</p>
<p>虽然上述的弱引用解决了 key，也就是线程的 ThreadLocal 能及时被回收，但是 value 却依然存在内存泄漏的问题。当把 threadlocal 实例置为 null 以后,没有任何强引用指向 threadlocal 实例,所以 threadlocal 将会被 gc 回收.map 里面的 value 却没有被回收.而这块 value 永远不会被访到了. 所以存在着内存泄露,因为存在一条从 current thread 连接过来的强引用.只有当前 thread 结束后, current thread 就不会存在栈中,强引用断开, Current Thread, Map, value 将全部被 GC 回收.所当线程的某个 localThread 使用完了，马上调用 threadlocal 的 remove 方法,就不会发生这种情况。</p>
<p>另外其实只要这个线程对象及时被 gc 回收，这个内存泄露问题影响不大，但在 threa Local 设为 null 到线程结束中间这段时间不会被回收的，就发生了我们认为的内存泄露。最要命的是程对象不被回收的情况，这就发生了真正意义上的内存泄露。比如使用线程池的时候，线程结束是不销毁的，会再次使用，就可能出现内存泄露。</p>
<h3 id="HashSet和HashMap">HashSet 和 HashMap</h3>
<p>HashSet 的 value 存的是一个 static finial PRESENT = newObject()。而 HashSet 的 remove 是使用 HashMap 实现,则是 map.remove 而 map 的移除会返回 value,如果底层 value 都存 null,显然将无法分辨是否移除成功。</p>
<h3 id="Boolean占几个字节">Boolean 占几个字节</h3>
<p>未精确定义字节。Java 语言表达式所操作的 boolean 值，在编译之后都使用 Java 虚机中的 int 数据类型来代替，而 boolean 数组将会被编码成 Java 虚拟机的 byte 数组，每个元素 boo ean 元素占 8 位。</p>
<h3 id="阻塞非阻塞与同步异步的区别">阻塞非阻塞与同步异步的区别</h3>

同步和异步关注的是消息通信机制，所谓同步，就是在发出一个调用时，在没有得到果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。而异步则是相反，调用在发出之后这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调。你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等”我查一下"，然后开始查啊查，等查好了（可能是 5 秒，也可能是一天）告诉你结果（返回结果）而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。</l >
阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。阻塞调用是调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不立刻得到结果之前，该调用不会阻塞当前线程。你打电话问书店老板有没有《分布式系统》这本书，如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用

你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟 check 一下老板有没有
回结果。在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。</strong

<h3 id="Java-SPI">Java SPI</h3>
<p>由于双亲委派模型损失了一丢丢灵活性。就比如 java.sql.Driver 这个东西。JDK 只能
供一个规范接口，而不能提供实现。提供实现的是实际的数据库提供商。提供商的库总不能放 JDK
录里吧。Java 从 1.6 搞出了 SPI 就是为了优雅的解决这类问题——JDK 提供接口，供应商提供服务
编程人员编码时面向接口编程，然后 JDK 能够自动找到合适的实现。</p>
<h2 id="Spring">Spring</h2>
<h3 id="什么是三级缓存">什么是三级缓存</h3>

第一级缓存：单例缓存池 singletonObjects。
第二级缓存：早期提前暴露的对象缓存 earlySingletonObjects。（属性还没有值对
也没有被初始化）
第三级缓存：singletonFactories 单例对象工厂缓存。

<p>三级缓存详解：<a href="https://ld246.com/forward?goto=https%3A
2F%2Fzhuanlan.zhihu.com%2Fp%2F144627581" target="_blank" rel="nofollow ugc"><stron
>根据 Spring 源码写一个带有三级缓存的 IOC</p>
<h3 id="Spring如何解决循环依赖问题">Spring 如何解决循环依赖问题</h3>
<p>Spring 使用了三级缓存解决了循环依赖的问题。在 populateBean()给属性赋值阶段
面 Spring 会解析你的属性，并且赋值，当发现，A 对象里面依赖了 B，此时又会走 getBean 方法，
这个时候，你去缓存中是可以拿的到的。因为我们在对 createBeanInstance 对象创建完成以后已经
入了缓存当中，所以创建 B 的时候发现依赖 A，直接就从缓存中去拿，此时 B 创建完，A 也创建完
一共执行了 4 次。至此 Bean 的创建完成，最后将创建好的 Bean 放入单例缓存池中。（非单例的实
作用域是不允许出现循环依赖）</p>
<h3 id="BeanFactory和ApplicationContext的区别">BeanFactory 和 ApplicationCont
xt 的区别</h3>

BeanFactory 是 Spring 里面最低层的接口，提供了最简单的容器的功能，只提供了
例化对象和拿对象的功能。
ApplicationContext 应用上下文，继承 BeanFactory 接口，它是 Spring 的一各更高
的容器，提供了更多的有用的功能。如国际化，访问资源，载入多个（有继承关系）上下文 ，使得每
个上下文都专注于一个特定的层次，消息发送、响应机制，AOP 等。
BeanFactory 在启动的时候不会去实例化 Bean，中有从容器中拿 Bean 的时候才会
实例化。ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了。它还可以为 Bean 配置
azy-init=true 来让 Bean 延迟实例化

<h3 id="动态代理的实现方式-AOP的实现方式">动态代理的实现方式，AOP 的实现方式<
strong></h3>

JDK 动态代理：利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用
nvokeHandler 来处理。
CGlib 动态代理：利用 ASM（开源的 Java 字节码编辑库，操作字节码）开源包，将
理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。
<strong>区别：JDK 代理只能对实现接口的类生成代理；CGlib 是针对类实现代理，对指定的
生成一个子类，并覆盖其中的方法，这种通过继承类的实现方式，不能代理 final 修饰的类。</stron
>

<h3 id="-Transactional错误使用失效场景"><strong>@Transactional 错误使用失效场景</stron
></h3>

<strong>@Transactional 在 private 上：当标记在 protected、private、package-visible 方

上时，不会产生错误，但也不会表现出为它指定的事务配置。可以认为它作为一个普通的方法参与到个 public 方法的事务中。
@Transactional 的事务传播方式配置错误。
@Transactional 注解属性 rollbackFor 设置错误：Spring 默认抛出了未检查 unchecked 异常（继承自 RuntimeException 的异常）或者 Error 才回滚事务；其他异常不会触发回滚事务
同一个类中方法调用，导致 @Transactional 失效：由于使用 Spring AOP 代理造成，因为只有当事务方法被当前类以外的代码调用时，才会由 Spring 生成的代理对象来管理。
异常被 catch 捕获导致 @Transactional 失效。
数据库引擎不支持事务。

<h3 id="Spring中的事务传播机制">Spring 中的事务传播机制</h3>

REQUIRED（默认，常用）：支持使用当前事务，如果当前事务不存在，创建一个新事务。eg:方法 B 用 REQUIRED 修饰，方法 A 调用方法 B，如果方法 A 当前没有事务，方法 B 就新建个事务（若还有 C 则 B 和 C 在各自的事务中独立执行），如果方法 A 有事务，方法 B 就加入到这个事务中，当成一个事务。
SUPPORTS：支持使用当前事务，如果当前事务不存在，则不使用事务。
MANDATORY：强制，支持使用当前事务，如果当前事务不存在，则抛出 Exception
REQUIRES_NEW（常用）：创建一个新事务，如果当前事务存在，把当前事务挂起。g:方法 B 用 REQUIRES_NEW 修饰，方法 A 调用方法 B，不管方法 A 上有没有事务方法 B 都新建一事务，在该事务执行。
NOT_SUPPORTED：无事务执行，如果当前事务存在，把当前事务挂起。
NEVER：无事务执行，如果当前有事务则抛出 Exception。
NESTED：嵌套事务，如果当前事务存在，那么在嵌套的事务中执行。如果当前事务存在，则表现跟 REQUIRED 一样。

<h3 id="Spring中Bean的生命周期">Spring 中 Bean 的生命周期</h3>

实例化 Instantiation
属性赋值 Populate
初始化 Initialization
销毁 Destruction

<h3 id="Spring的后置处理器">Spring 的后置处理器</h3>

BeanPostProcessor：Bean 的后置处理器，主要在 bean 初始化前后工作。（before 和 after 两个回调中间只处理了 init-method）
InstantiationAwareBeanPostProcessor：继承于 BeanPostProcessor，主要在实例化 bean 前后工作（TargetSource 的 AOP 创建代理对象就是通过该接口实现）
BeanFactoryPostProcessor：Bean 工厂的后置处理器，在 bean 定义(bean definitions)加载完成后，bean 尚未初始化前执行。
BeanDefinitionRegistryPostProcessor：继承于 BeanFactoryPostProcessor。其自义的方法 postProcessBeanDefinitionRegistry 会在 bean 定义(bean definitions)将要加载，bean 未初始化前真执行，即在 BeanFactoryPostProcessor 的 postProcessBeanFactory 方法前被调用。

<h3 id="Spring-MVC的工作流程-源码层面-">Spring MVC 的工作流程（源码层面）</h3>
<p>参考文章：<a href="https://ld246.com/forward?goto=https%3A%2F

2Fzhuanlan.zhihu.com%2Fp%2F139751932" target="_blank" rel="nofollow ugc">己写个 Spring MVC
<h2 id="消息队列">消息队列</h2>
<h3 id="为什么需要消息队列">为什么需要消息队列</h3>
解耦，异步处理，削峰/限流
<h3 id="Kafka的文件存储机制">Kafka 的文件存储机制</h3>
Kafka 中消息是以 topic 进行分类的，生产者通过 topic 向 Kafka broker 发送消息，费者通过 topic 读取数据。然而 topic 在物理层面又能以 partition 为分组，一个 topic 可以分成若个 partition。partition 还可以细分为 segment，一个 partition 物理上由多个 segment 组成，seg ent 文件由两部分组成，分别为“.index”文件和“.log”文件，分别表示为 segment 索引文件和数文件。这两个文件的命令规则为：partition 全局的第一个 segment 从 0 开始，后续每个 segment 件名为上一个 segment 文件最后一条消息的 offset 值。
<h3 id="Kafka-如何保证可靠性">Kafka 如何保证可靠性</h3>
如果我们要往 Kafka 对应的主题发送消息，我们需要通过 Producer 完成。前面我们过 Kafka 主题对应了多个分区，每个分区下面又对应了多个副本；为了让用户设置数据可靠性， Kafk 在 Producer 里面提供了消息确认机制。也就是说我们可以通过配置来决定消息发送到对应分区的几副本才算消息发送成功。可以在定义 Producer 时通过 acks 参数指定。这个参数支持以下三种值：</ trong>

acks = 0：意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功入 Kafka 。在这种情况下还是有可能发生错误，比如发送的对象无能被序列化或者网卡发生故障，但果是分区离线或整个集群长时间不可用，那就不会收到任何错误。在 acks=0 模式下的运行速度是非快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不如果选择了这种模式， 一定会丢失一些消息。
acks = 1：意味若 Leader 在收到消息并把它写入到分区数据文件（不一定同步到磁上）时会返回确认或错误响应。在这个模式下，如果发生正常的 Leader 选举，生产者会在选举时收一个 LeaderNotAvailableException 异常，如果生产者能恰当地处理这个错误，它会重试发送悄息最终消息会安全到达新的 Leader 那里。不过在这个模式下仍然有可能丢失数据，比如消息已经成功入 Leader，但在消息被复制到 follower 副本之前 Leader 发生崩溃。
acks = all（这个和 request.required.acks = -1 含义一样）：意味着 Leader 在返回认或错误响应之前，会等待所有同步副本都收到悄息。如果和 min.insync.replicas 参数结合起来，可以决定在返回确认前至少有多少个副本能够收到悄息，生产者会一直重试直到消息被成功提交。不这也是最慢的做法，因为生产者在继续发送其他消息之前需要等待所有副本都收到当前的消息。</strng>

<h3 id="Kafka消息是采用Pull模式-还是Push模式">Kafka 消息是采用 Pull 模式，还是 Psh 模式</h3>
Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的计：producer 将消息推送到 broker，consumer 从 broker 拉取消息。push 模式下，当 broker 推的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达。
<h3 id="Kafka是如何实现高吞吐率的">Kafka 是如何实现高吞吐率的</h3>

顺序读写：kafka 的消息是不断追加到文件中的，这个特性使 kafka 可以充分利用磁的顺序读写性能
零拷贝：跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存的直接映射，数据不复制到“用户态缓冲区”
文件分段：kafka 的队列 topic 被分为了多个区 partition，每个 partition 又分为多段 segment，所以一个队列中的消息实际上是保存在 N 多个片段文件中
批量发送：Kafka 允许进行批量发送消息，先将消息缓存在内存中，然后一次请求批发送出去

数据压缩：Kafka 还支持对消息集合进行压缩，Producer 可以通过 GZIP 或 Snappy 式对消息集合进行压缩

<h3 id="Kafka判断一个节点还活着的两个条件">Kafka 判断一个节点还活着的两个条件</trong></h3>

节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的接
如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久</stron>

<h2 id="Dubbo">Dubbo</h2>
<h3 id="Dubbo的容错机制">Dubbo 的容错机制</h3>

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长迟。可通过 retries="2" 来设置重试次数
快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增录。
失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。
失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。</l>
并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。
广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更缓存或日志等本地资源信息

<h3 id="Dubbo注册中心挂了还可以继续通信么">Dubbo 注册中心挂了还可以继续通信</h3>
可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，以注册中心挂了可以继续通信。
<h3 id="Dubbo提供的线程池">Dubbo 提供的线程池</h3>

**fixed：固定大小线程池，启动时建立线程，不关闭，一直持有。 **
**cached：缓存线程池，空闲一分钟自动删除，需要时重建。 **
limited：可伸缩线程池，但池中的线程数只会增长不会收缩。(为避免收缩时突然来了流量引起的性能问题)。

<h3 id="Dubbo框架设计结构">Dubbo 框架设计结构</h3>

服务接口层：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设对应的接口和实现。
配置层：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心，可以直接 nw 配置类，也可以通过 spring 解析配置生成配置类。
服务代理层：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton，以 SrviceProxy 为中心，扩展接口为 ProxyFactory。
服务注册层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFctory、Registry 和 RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。</strng>
集群层：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，展接口为 Cluster、Directory、Router 和 LoadBalance。将多个服务提供方组合为一个服务提供方实现对服务消费方来透明，只需要与一个服务提供方进行交互。
监控层：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFctory、Monitor 和 MonitorService。
远程调用层：封将 RPC 调用，以 Invocation 和 Result 为中心，扩展接口为 Protoco

、Invoker 和 Exporter。Protocol 是服务域，它是 Invoker 暴露和引用的主功能入口，它负责 Invok
r 的生命周期管理。Invoker 是实体域，它是 Dubbo 的核心模型，其它模型都向它靠扰，或转换成
，它代表一个可执行体，可向它发起 invoke 调用，它有可能是一个本地的实现，也可能是一个远程
实现，也可能一个集群实现。
信息交换层：封装请求响应模式，同步转异步，以 Request 和 Response 为中心，扩
接口为 Exchanger、ExchangeChannel、ExchangeClient 和 ExchangeServer。
网络传输层：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Cha
nel、Transporter、Client、Server 和 Codec。
数据序列化层：可复用的一些工具，扩展接口为 Serialization、 ObjectInput、Objec
Output 和 ThreadPool。

<h2 id="操作系统">操作系统</h2>
<h3 id="进程和线程">进程和线程</h3>

进程是操作系统资源分配的最小单位，线程是 CPU 任务调度的最小单位。一个进程可
包含多个线程，所以进程和线程都是一个时间段的描述，是 CPU 工作时间段的描述，不过是颗粒大
不同。
不同进程间数据很难共享，同一进程下不同线程间数据很易共享。
每个进程都有独立的代码和数据空间，进程要比线程消耗更多的计算机资源。线程可
看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器
线程之间切换的开销小。
进程间不会相互影响，一个线程挂掉将导致整个进程挂掉。
系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了 CPU 外，
统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。</s
rong>

<h3 id="多线程和单线程">多线程和单线程</h3>
*线程不是越多越好，假如你的业务逻辑全部是计算型的（CPU 密集型）,不涉及到 IO，
且只有一个核心。那肯定一个线程最好，多一个线程就多一点线程切换的计算，CPU 不能完完全全的
计算能力放在业务计算上面，线程越多就会造成 CPU 利用率（用在业务计算的时间/总的时间）下降
但是在 WEB 场景下，业务并不是 CPU 密集型任务，而是 IO 密集型的任务，一个线程是不合适，如
一个线程在等待数据时，把 CPU 的计算能力交给其他线程，这样也能充分的利用 CPU 资源。但是线
数量也要有个限度，一般线程数有一个公式：最佳启动线程数=[任务执行时间/(任务执行时间-IO 等
时间)]CPU 内核数超过这个数量，CPU 要进行多余的线程切换从而浪费计算能力，低于这个数
，CPU 要进行 IO 等待从而造成计算能力不饱和。总之就是要尽可能的榨取 CPU 的计算能力。如果
的 CPU 处于饱和状态，并且没有多余的线程切换浪费，那么此时就是你服务的完美状态，如果再加
并发量，势必会造成性能上的下降。
<h3 id="进程的组成部分">进程的组成部分</h3>
进程由进程控制块（PCB）、程序段、数据段三部分组成。
<h3 id="进程的通信方式">进程的通信方式</h3>

无名管道：半双工的，即数据只能在一个方向上流动，只能用于具有亲缘关系的进程
间的通信，可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。但是
不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。
FIFO 命名管道：FIFO 是一种文件类型，可以在无关的进程之间交换数据，与无名管
不同，FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。
消息队列：消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符
即队列 ID）来标识。
信号量：信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存
进程间通信数据。
共享内存：共享内存指两个或多个进程共享一个给定的存储区，一般配合信号量使用。
/strong>

<h3 id="进程间五种通信方式的比较">进程间五种通信方式的比较</h3>

管道：速度慢，容量有限，只有父子进程能通讯。
FIFO：任何进程间都能通讯，但速度慢。
消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数的问题。
信号量：不能传递复杂消息，只能用来同步。
共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存。

<h3 id="内存管理有哪几种方式">内存管理有哪几种方式</h3>

块式管理：把主存分为一大块、一大块的，当所需的程序片断不在主存时就分配一块存空间，把程序片断 load 入主存，就算所需的程序片度只有几个字节也只能把这一块分配给它。这会造成很大的浪费，平均浪费了 50%的内存空间，但是易于管理。
页式管理：把主存分为一页一页的，每一页的空间要比一块一块的空间小很多，显然种方法的空间利用率要比块式管理高很多。
段式管理：把主存分为一段一段的，每一段的空间又要比一页一页的空间小很多，这方法在空间利用率上又比页式管理高很多，但是也有另外一个缺点。一个程序片断可能会被分为几十，这样很多时间就会被浪费在计算每一段的物理地址上。
段页式管理：结合了段式管理和页式管理的优点。将程序分成若干段，每个段分成若页。段页式管理每取一数据，要访问 3 次内存。

<h3 id="页面置换算法">页面置换算法</h3>

最佳置换算法 OPT：只具有理论意义的算法，用来评价其他页面置换算法。置换策略将当前页面中在未来最长时间内不会被访问的页置换出去。
先进先出置换算法 FIFO：简单粗暴的一种置换算法，没有考虑页面访问频率信息。每淘汰最早调入的页面。
最近最久未使用算法 LRU：算法赋予每个页面一个访问字段，用来记录上次页面被访到现在所经历的时间 t，每次置换的时候把 t 值最大的页面置换出去(实现方面可以采用寄存器或者栈方式实现)。
时钟算法 clock(也被称为是最近未使用算法 NRU)：页面设置一个访问位，并将页面接为一个环形队列，页面被访问的时候访问位设置为 1。页面置换的时候，如果当前指针所指页面访为为 0，那么置换，否则将其置为 0，循环直到遇到一个访问为位 0 的页面。
改进型 Clock 算法：在 Clock 算法的基础上添加一个修改位，替换时根究访问位和修位综合判断。优先替换访问位和修改位都是 0 的页面，其次是访问位为 0 修改位为 1 的页面。
LFU 最少使用算法 LFU：设置寄存器记录页面被访问次数，每次置换的时候置换当前问次数最少的。

<h3 id="操作系统中进程调度策略有哪几种">操作系统中进程调度策略有哪几种</h3>

先来先服务调度算法 FCFS：队列实现，非抢占，先请求 CPU 的进程先分配到 CPU，以作为作业调度算法也可以作为进程调度算法；按作业或者进程到达的先后顺序依次调度，对于长作比较有利.
最短作业优先调度算法 SJF：作业调度算法，算法从就绪队列中选择估计时间最短的业进行处理，直到得出结果或者无法继续执行，平均等待时间最短，但难以知道下一个 CPU 区间长；缺点：不利于长作业；未考虑作业的重要性；运行时间是预估的，并不靠谱.
优先级调度算法(可以是抢占的，也可以是非抢占的)：优先级越高越先分配到 CPU，同优先级先到先服务，存在的主要问题是：低优先级进程无穷等待 CPU，会导致无穷阻塞或饥饿.
时间片轮转调度算法(可抢占的)：按到达的先后对进程放入队列中，然后给队首进程分

CPU 时间片，时间片用完之后计时器发出中断，暂停当前进程并将其放到队列尾部，循环 ;队列中没进程被分配超过一个时间片的 CPU 时间，除非它是唯一可运行的进程。如果进程的 CPU 区间超过了个时间片，那么该进程就被抢占并放回就绪队列。

<h3 id="死锁的4个必要条件">死锁的 4 个必要条件</h3>

互斥条件：一个资源每次只能被一个线程使用；
请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放；</stron>
不剥夺条件：进程已经获得的资源，在未使用完之前，不能强行剥夺；

循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

<h3 id="如何避免-预防-死锁">如何避免（预防）死锁</h3>

破坏“请求和保持”条件：让进程在申请资源时，一次性申请所有需要用到的资源，要一次一次来申请，当申请的资源有一些没空，那就让线程等待。不过这个方法比较浪费资源，进程能经常处于饥饿状态。还有一种方法是，要求进程在申请资源前，要释放自己拥有的资源。
破坏“不可抢占”条件：允许进程进行抢占，方法一：如果去抢资源，被拒绝，就释自己的资源。方法二：操作系统允许抢，只要你优先级大，可以抢到。
破坏“循环等待”条件：将系统中的所有资源统一编号，进程可在任何时刻提出资源请，但所有申请必须按照资源的编号顺序提出（指定获取锁的顺序，顺序加锁）。

<h2 id="计算机网路">计算机网路</h2>
<h3 id="Get和Post区别">Get 和 Post 区别</h3>

Get 是不安全的，因为在传输过程，数据被放在请求的 URL 中；Post 的所有操作对户来说都是不可见的。
Get 传送的数据量较小，这主要是因为受 URL 长度限制；Post 传送的数据量较大，般被默认为不受限制。
Get 限制 Form 表单的数据集的值必须为 ASCII 字符；而 Post 支持整个 ISO10646 符集。
Get 执行效率却比 Post 方法好。Get 是 form 提交的默认方法。
GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。（非必然，客户端可灵活定）

<h3 id="Http请求的完全过程">Http 请求的完全过程</h3>

浏览器根据域名解析 IP 地址（DNS）,并查 DNS 缓存
浏览器与 WEB 服务器建立一个 TCP 连接
浏览器给 WEB 服务器发送一个 HTTP 请求（GET/POST）：一个 HTTP 请求报文由求行（request line）、请求头部（headers）、空行（blank line）和请求数据（request body）4 部分组成。
服务端响应 HTTP 响应报文，报文由状态行（status line）、相应头部（headers）、行（blank line）和响应数据（response body）4 个部分组成。
浏览器解析渲染

<h3 id="计算机网络的五层模型">计算机网络的五层模型</h3>

应用层：为操作系统或网络应用程序提供访问网络服务的接口 ，通过应用进程间的交完成特定网络应用。应用层定义的是应用进程间通信和交互的规则。（HTTP，FTP，SMTP，RPC）

传输层：负责向两个主机中进程之间的通信提供通用数据服务。（TCP,UDP）</stron
>
网络层：负责对数据包进行路由选择和存储转发。（IP，ICMP(ping 命令)）</strong

数据链路层：两个相邻节点之间传送数据时，数据链路层将网络层交下来的 IP 数据报
装成帧，在两个相邻的链路上传送帧（frame)。每一帧包括数据和必要的控制信息。
物理层：物理层所传数据单位是比特（bit)。物理层要考虑用多大的电压代表 1 或 0
以及接受方如何识别发送方所发送的比特。

<h3 id="tcp和udp区别">tcp 和 udp 区别</h3>

TCP 面向连接，UDP 是无连接的，即发送数据之前不需要建立连接。
TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重
，且按序到达;UDP 尽最大努力交付，即不保证可靠交付。
TCP 面向字节流，实际上是 TCP 把数据看成一连串无结构的字节流，UDP 是面向报
的，UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP
电话，实时视频会议等）
每一条 TCP 连接只能是点到点的，UDP 支持一对一，一对多，多对一和多对多的交
通信。
TCP 首部开销 20 字节，UDP 的首部开销小，只有 8 个字节。
TCP 的逻辑通信信道是全双工的可靠信道，UDP 则是不可靠信道。

<h3 id="tcp和udp的优点">tcp 和 udp 的优点</h3>

TCP 的优点： 可靠，稳定 TCP 的可靠体现在 TCP 在传递数据之前，会有三次握手来
立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用
节约系统资源。 TCP 的缺点： 慢，效率低，占用系统资源高，易被攻击 TCP 在传递数据之前，要先
连接，这会消耗时间，而且在数据传递时，确认机制、重传机制、拥塞控制机制等都会消耗大量的时
，而且要在每台设备上维护所有的传输连接，事实上，每个连接都会占用系统的 CPU、内存等硬件资
。 而且，因为 TCP 有确认机制、三次握手机制，这些也导致 TCP 容易被人利用，实现 DOS、DDO
、CC 等攻击。
UDP 的优点： 快，比 TCP 稍安全 UDP 没有 TCP 的握手、确认、窗口、重传、拥塞
制等机制，UDP 是一个无状态的传输协议，所以它在传递数据时非常快。没有 TCP 的这些机制，UDP
较 TCP 被攻击者利用的漏洞就要少一些。但 UDP 也是无法避免攻击的，比如：UDP Flood 攻击……
DP 的缺点： 不可靠，不稳定 因为 UDP 没有 TCP 那些可靠的机制，在数据传递时，如果网络质量
好，就会很容易丢包。 基于上面的优缺点，那么： 什么时候应该使用 TCP： 当对网络通讯质量有要
的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如 HTTP、H
TPS、FTP 等传输文件的协议，POP、SMTP 等邮件传输的协议。 在日常生活中，常见使用 TCP 协
的应用如下： 浏览器，用的 HTTP FlashFXP，用的 FTP Outlook，用的 POP、SMTP Putty，用的 T
lnet、SSH QQ 文件传输。什么时候应该使用 UDP： 当对网络通讯质量要求不高的时候，要求网络
讯速度能尽量的快，这时就可以使用 UDP。 比如，日常生活中，常见使用 UDP 协议的应用如下： Q
语音 QQ 视频 TFTP。

<h3 id="三次握手">三次握手</h3>

第一次握手：建立连接时，客户端发送 syn 包（syn=x）到服务器，并进入 SYN_SENT
状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。</l
>
**第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=x+1），同时自己也发送一个 S
N 包（syn=y），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；**
第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK(ack=y+
），此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。</st
ong>

<h3 id="为什么不能两次握手">为什么不能两次握手</h3>
TCP 是一个双向通信协议，通信双方都有能力发送信息，并接收响应。如果只是两次手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认</p

<h3 id="四次挥手">四次挥手</h3>

客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。
服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时。
客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。</l>
服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LST-ACK（最后确认）状态，等待客户端的确认。
客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放必须经过 2□□MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状。
服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些</li

<h3 id="为什么连接的时候是三次握手-关闭的时候却是四次握手">为什么连接的时候是次握手，关闭的时候却是四次握手</h3>
因为当 Server 端收到 Client 端的 SYN 连接请求报文后，可以直接发送 SYN+ACK文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 Server 端收到 FIN报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，"你发的IN 报文我收到了"。只有等到我 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能起发送。故需要四步握手。
<h2 id="数据结构与算法">数据结构与算法</h2>
<h3 id="排序算法">排序算法</h3>

冒泡排序
选择排序：选择排序与冒泡排序有点像，只不过选择排序每次都是在确定了最小数的标之后再进行交换，大大减少了交换的次数
插入排序：将一个记录插入到已排序的有序表中，从而得到一个新的，记录数增 1 的序表
快速排序：通过一趟排序将序列分成左右两部分，其中左半部分的的值均比右半部分值小，然后再分别对左右部分的记录进行排序，直到整个序列有序。

<pre><code class="highlight-chroma">int partition(int a[], int low, int high){
    int key = a[low];
    while(lo < high){
     nbsp; while(low < high && a[high] >= key) high--;

```
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  a[low] = a[high];
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  while(low &lt; high &amp;&amp; a[low] &lt;= key) low++;
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  a[high] = a[low];
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    a[low] =
ey;
</span></span><span class="highlight-line"><span class="highlight-cl">    return lo
;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">void quick_sort(int
a[], int low, int high){
</span></span><span class="highlight-line"><span class="highlight-cl">    if(low &g
;= high) return;
</span></span><span class="highlight-line"><span class="highlight-cl">    int keypo
 = partition(a, low, high);
</span></span><span class="highlight-line"><span class="highlight-cl">    quick_sor
(a, low, keypos-1);
</span></span><span class="highlight-line"><span class="highlight-cl">    quick_sor
(a, keypos+1, high);
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<ol>
<li><strong>堆排序：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节
。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余 n-1 个元素重新构造成一个堆，这样
得到 n 个元素的次小值。如此反复执行，便能得到一个有序序列了。</strong></li>
</ol>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Test {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public vo
d sort(int[] arr) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  for (int i = arr.length / 2 - 1; i &gt;= 0; i--) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      adjustHeap(arr, i, arr.length);
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  }
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  for (int j = arr.length - 1; j &gt; 0; j--) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      swap(arr, 0, j);
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      adjustHeap(arr, 0, j);
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public vo
d adjustHeap(int[] arr, int i, int length) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
```

```
nbsp;  int temp = arr[i];
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  for (int k = i * 2 + 1; k &lt; length; k = k * 2 + 1) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      if (k + 1 &lt; length &amp;&amp; arr[k] &lt; arr[k + 1]) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;          k++;
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      }
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      if (arr[k] &gt; temp) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;          arr[i] = arr[k];
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;          i = k;
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      } else {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;          break;
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;      }
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  }
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  arr[i] = temp;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public vo
d swap(int[] arr, int a, int b) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  int temp = arr[a];
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  arr[a] = arr[b];
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  arr[b] = temp;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public st
tic void main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  int[] arr = {9, 8, 7, 6, 5, 4, 3, 2, 1};
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  new Test().sort(arr);
</span></span><span class="highlight-line"><span class="highlight-cl">    
nbsp;  System.out.println(Arrays.toString(arr));
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<ol>
<li><strong>希尔排序：先将整个待排记录序列分割成为若干子序列分别进行直接插入排序，待整
序列中的记录基本有序时再对全体记录进行一次直接插入排序。</strong></li>
<li><strong>归并排序：把有序表划分成元素个数尽量相等的两半，把两半元素分别排序，两个有
表合并成一个</strong></li>
</ol>
```

<h2 id="其他">其他</h2>
<h3 id="高并发系统的设计与实现">高并发系统的设计与实现</h3>
在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。

缓存：缓存比较好理解，在大型高并发系统中，如果没有缓存数据库将分分钟被爆，统也会瞬间瘫痪。使用缓存不单单能够提升系统访问速度、提高并发访问量，也是保护数据库、保护统的有效方式。大型网站一般主要是“读”，缓存的使用很容易被想到。在大型“写”系统中，缓存常常扮演者非常重要的角色。比如累积一些数据批量写入，内存里面的缓存队列（生产消费），以及Base 写数据的机制等等也都是通过缓存提升系统的吞吐量或者实现系统的保护措施。甚至消息中间，你也可以认为是一种分布式的数据缓存。
降级：服务降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。降级往往会指定不同的级别，临不同的异常等级执行不同的处理。根据服务方式：可以拒接服务，可以延迟服务，也有时候可以随服务。根据服务范围：可以砍掉某个功能，也可以砍掉某些模块。总之服务降级需要根据不同的业务求采用不同的降级策略。主要的目的就是服务虽然有损但是总比没有好。
限流：限流可以认为服务降级的一种，限流就是限制系统的输入和输出流量已达到保系统的目的。一般来说系统的吞吐量是可以被测算的，为了保证系统的稳定运行，一旦达到的需要限的阈值，就需要限制流量并采取一些措施以完成限制流量的目的。比如：延迟处理，拒绝处理，或者分拒绝处理等等。

<h3 id="负载均衡算法-">负载均衡算法：</h3>

轮询
加权轮询
随机算法
一致性 Hash

<h3 id="常见的限流算法-">常见的限流算法：</h3>
常见的限流算法有计数器、漏桶和令牌桶算法。漏桶算法在分布式环境中消息中间件者 Redis 都是可选的方案。发放令牌的频率增加可以提升整体数据处理的速度，而通过每次获取令牌个数增加或者放慢令牌的发放速度和降低整体数据处理速度。而漏桶不行，因为它的流出速率是固定，程序处理速度也是固定的。
<h3 id="秒杀并发情况下库存为负数问题">秒杀并发情况下库存为负数问题</3>

for update 显示加锁
把 update 语句写在前边，先把数量-1，之后 select 出库存如果 >-1 就 commit，则 rollback。

<pre><code class="highlight-chroma">update products set quantity = quantity-1 WHERE id=3;
select quantity fr m products WHERE id=3 for update;
</code></pre>
3.update 语句在更新的同时加上一个条件
<pre><code class="highlight-chroma">quantity = select quantity from products WHERE id=3;
update products s t quantity = ($quantity-1) WHERE id=3 and queantity = $quantity;
</code></pre>