目录

[Java虚拟机 3](#_Toc67295751)

[运行时数据区 3](#_Toc67295752)

[对象头的内存布局 4](#_Toc67295753)

[‌垃圾收集器 7](#_Toc67295754)

[运行时常量池 7](#_Toc67295755)

[字符串在虚拟机中的存储 7](#_Toc67295756)

[OOM如何排查？ 7](#_Toc67295757)

[类加载流程，类初始化流程？ 7](#_Toc67295758)

[计算机网络 10](#_Toc67295759)

[HTTPS协议 10](#_Toc67295760)

[http1.0，http1.1，http2.0的区别？ 10](#_Toc67295761)

[Cookie的作用是什么?和Session有什么区别？ 10](#_Toc67295762)

[如果客户端禁用了cookie，怎么使用session？ 11](#_Toc67295763)

[Get与POST的区别 11](#_Toc67295764)

[TCP协议 TCP快连接 12](#_Toc67295765)

[SYN泛洪攻击？ 12](#_Toc67295766)

[数据包从一端发到另一端的流程，从协议栈的角度说，涉及到添加首部、寻址什么的，数据包从网卡出去后经历的过程 13](#_Toc67295767)

[ARP协议的工作原理和流程，路由器是如何转发的？(路由表的工作原理) 13](#_Toc67295768)

[讲一下拥塞控制 13](#_Toc67295769)

[延迟ACK了解么？确认报文随数据一起传送过去，或者是设定超时时间到了单独发过去 13](#_Toc67295770)

[访问www.qq.com中间有多少次网络交互，分别有什么协议 13](#_Toc67295771)

[TCP如何处理丢包？TCP如何处理重传？ 13](#_Toc67295772)

[数据库 13](#_Toc67295773)

[数据库范式 13](#_Toc67295774)

[事务底层实现原理？ 15](#_Toc67295775)

[什么是索引？索引的种类？ 15](#_Toc67295776)

[聚簇索引和非聚簇索引，有什么区别？哪个快？ 15](#_Toc67295777)

[聚簇索引，二级索引(辅助索引)，联合索引？索引覆盖？ 17](#_Toc67295778)

[创建索引的原则？是什么时候适合使用索引？ 17](#_Toc67295779)

[索引的数据结构，各自优缺点？ 18](#_Toc67295780)

[最左前缀原则？最左匹配原则？ 18](#_Toc67295781)

[MySQL 用了哪种引擎？区别是什么？ 20](#_Toc67295782)

[B+树和B树的区别，B+树的优势在哪里？ 20](#_Toc67295783)

[建索引是只能对单列建索引吗？能不能对多列建索引。 21](#_Toc67295784)

[sql注入？如何处理SQL注入 21](#_Toc67295785)

[Delete \* from table where id>2执行这条语句过程中发生了什么？ 21](#_Toc67295786)

[MySQL主从复制原理 21](#_Toc67295787)

[Redis 23](#_Toc67295788)

[Redis的使用场景是什么？ 23](#_Toc67295789)

[如何基于Redis实现分布式锁？ 23](#_Toc67295790)

[‌ zset，跳跃表，为什么不用红黑树，跳跃表实现讲了一下。 26](#_Toc67295791)

[‌zset底层实现 26](#_Toc67295792)

[‌为什么zset要用跳跃表和哈希表？ 28](#_Toc67295793)

[删除策略？ 28](#_Toc67295794)

[内存淘汰策略 29](#_Toc67295795)

[‌缓存更新机制是怎么设计的？说一下三种常用的缓存读写机制。缓存数据库双写 30](#_Toc67295796)

[如何保证redis和数据库的数据一致性？ 32](#_Toc67295797)

[Redis 主从复制 32](#_Toc67295798)

[Redis的网络模型（IO模型，文件事件） 33](#_Toc67295799)

[Redis下的写时复制 34](#_Toc67295800)

[Redis与Memcached的区别 34](#_Toc67295801)

[操作系统与linux 35](#_Toc67295802)

[‌select/epoll的区别，epoll水平和边缘触发的区别，编写代码时有什么区别 35](#_Toc67295803)

[进程之间的通信与线程之间的通信？ 35](#_Toc67295804)

[僵尸进程是什么，危害和解决措施？ 39](#_Toc67295805)

[ctrl+c，发生了什么？ 40](#_Toc67295806)

[linux写时复制 40](#_Toc67295807)

[‌Linux中进程通信方式，不同主机之间传文件什么命令 (共享内存，管道，消息队列，scp http socket) 42](#_Toc67295808)

[‌Linux下载文件命令 scp/ wget 42](#_Toc67295809)

[你常用的Linux命令是什么？ 42](#_Toc67295810)

[说一下进程，线程，协程额区别？ 46](#_Toc67295811)

[协程说一下？为什么会有协程？解决了什么问题 46](#_Toc67295812)

[场景题 49](#_Toc67295813)

[如何应对高并发？ 49](#_Toc67295814)

[如何提高系统吞吐量？ 49](#_Toc67295815)

[扫码登录如何实现，微信支付怎么实现 50](#_Toc67295816)

[算法笔记 51](#_Toc67295817)

[重写比较器接口 51](#_Toc67295818)

# 数据结构

## 二叉查找树，平衡二叉树，红黑树，B-树，B树，B+树

<https://zhuanlan.zhihu.com/p/258078863> 后续再总结

**二叉查找树**

特点：

* 任意节点左子树不为空,则左子树的值均小于根节点的值.
* 任意节点右子树不为空,则右子树的值均大于于根节点的值.
* 任意节点的左右子树也分别是二叉查找树.
* 没有键值相等的节点.

存在的缺陷：

二叉树在查找数据时，时间复杂度最好情况是O(logn) ，最坏情况下时间复杂度O(n)，如a图所示，二叉树退化成一个链表了，恰好选择了最小或者最大的节点做root，节点排在了一条直线上。左倾右倾

因此,在二叉查找树的基础上,又出现了AVL树,红黑树,它们两个都是基于二叉查找树,只是在二叉查找树的基础上又对其做了限制.

**平衡二叉树**

AVL、红黑树是对二叉搜索树的改进版本。

平衡因子：节点的左右子树深度之差。在一棵平衡二叉树中，节点的平衡因子只能取 0 、1 或者 -1 ，分别对应着左右子树等高，左子树比较高，右子树比较高。

AVL树是带有平衡条件的二叉查找树，一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡，左右子树树高不超过1，和红黑树相比，它是严格的平衡二叉树，平衡条件必须满足（所有节点的左右子树高度差不超过1）。

不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而旋转是非常耗时的，由此我们可以知道AVL树适合用于插入删除次数比较少，但查找多的情况。

AVL保持平衡的四种操作

* LL右单旋转
* RR右单旋转
* LR先左后右
* RL先右后左

AVL存在的局限性

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大,故而实际的应用不多，

更多的地方是用追求局部而不是非常严格整体平衡的红黑树.当然,如果应用场景中对插入删除不频繁,只是对查找要求较高,那么AVL还是较优于红黑树.

**红黑树**

一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是red或black（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍。它是一种弱平衡二叉树(由于是弱平衡，可以推出，相同的节点情况下，AVL树的高度低于红黑树)，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索、插入、删除操作较多的情况下，我们就用红黑树。

特征

* 每个节点非红即黑；
* 根节点跟叶子节点都是黑的；
* 如果一个节点是红的，它的父节点则是黑的
* 对于任意节点而言，其到叶子的每条路径都包含相同数目的黑节点；
* 高度始终保持在h = logn

红黑树的自平衡操作

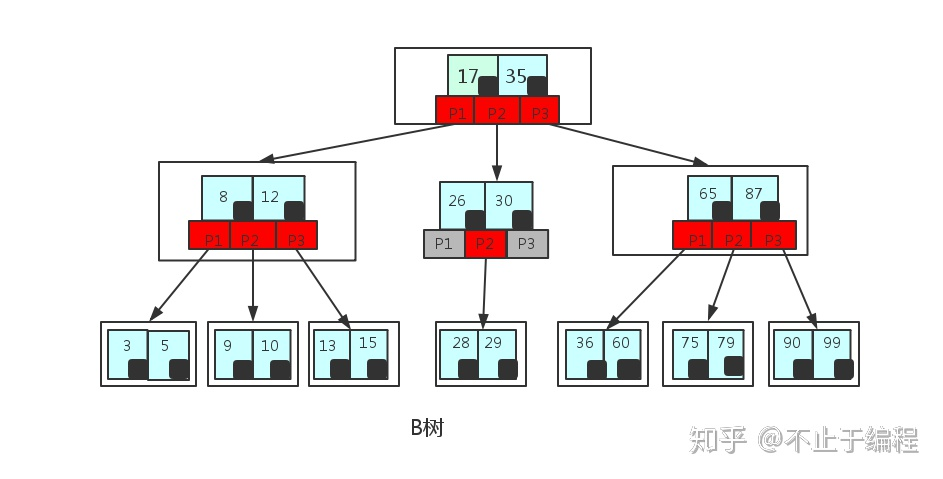
但插入、或者删除红黑树的数值时，为了重新符合红黑树的规则，需要对红黑树进行旋转、变色操作。树的旋转分为左旋和右旋。一个插入跟删除操作最多需要两次旋转操作。

红黑树解决了AVL平衡二叉树的维护起来比较麻烦的问题，红黑树，读取略逊于AVL，维护强于AVL，每次插入和删除的平均旋转次数应该是远小于平衡树。

因此：相对于要求严格的AVL树来说，红黑树的旋转次数少，所以对于插入、删除操作较多的情况下，我们就用红黑树。但是，只是对查找要求较高,那么AVL还是较优于红黑树.

**B树**

B树又名平衡多路查找树（查找路径不只两个），不同于常见的二叉树，它是一种多叉树，我们常见的使用场景一般是在数据库索引技术里，大量使用者B树和B+树的数据结构。



B树大多用在磁盘上用于查找磁盘的地址。因为磁盘会有大量的数据，有可能没有办法一次将需要的所有数据加入到内存中，所以只能逐一加载磁盘页，每个磁盘页就对应一个节点，而对于B树来说，B树很好的将树的高度降低了，这样就会减少IO查询次数，虽然一次加载到内存的数据变多了，但速度绝对快于AVL或是红黑树的。

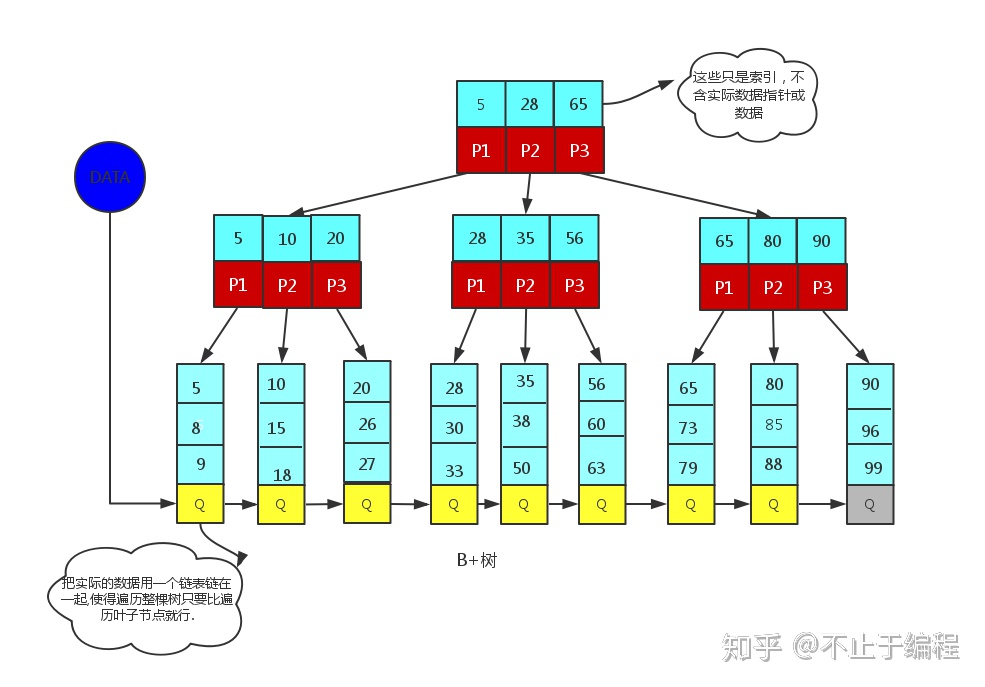
搜索B树时，很明显，访问节点（即读取磁盘）的次数与树的高度呈正比，而B树与红黑树和普通的二叉查找树相比，虽然高度都是对数数量级，但是显然B树中log函数的底可以比2更大，因此，和二叉树相比，极大地减少了磁盘读取的次数。

**B+树**

B+树时B树的一种升级版本，B+树查找的效率要比B树更高、更稳定。

B+树是应文件系统所需而产生的一种B树的变形树(文件的目录一级一级索引,只有最底层的叶子节点(文件)保存数据.),非叶子节点只保存索引,不保存实际的数据,数据都保存在叶子节点中.

这不就是文件系统文件的查找吗?我们就举个文件查找的例子:有3个文件夹,a,b,c, a包含b,b包含c,一个文件yang.c, a,b,c就是索引(存储在非叶子节点), a,b,c只是要找到的yang.c的key,而实际的数据yang.c存储在叶子节点上. 所有的非叶子节点都可以看成索引部分



特点

B+树和B树类似，但多了几条规则

* 非叶子结点的子树指针个数与关键字（节点中的元素个数）个数相同
* 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）
* 所有叶子结点有一个链指针
* 所有关键字都在叶子结点出现，只有叶子节点有Data域

B+树与B树对比

* B+树的层级更少：相较于B树，B+每个非叶子节点存储的关键字数更多，树的层级更少所以查询数据更快；
* B+树查询速度更稳定：B+所有关键字数据地址都存在叶子节点上，所以每次查找的次数都相同所以查询速度要比B树更稳定;
* B+树天然具备排序功能：B+树所有的叶子节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比B树高。
* B+树全节点遍历更快：B+树遍历整棵树只需要遍历所有的叶子节点即可，，而不需要像B树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

B树相对于B+树的优点是，如果经常访问的数据离根节点很近，而B树的非叶子节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比B+树快。

B+树相对于B树的最主要的优点：

* B+树只有叶子节点存放数据，而其他节点只存放索引，而B树每个节点都有Data域。所以相同大小的节点B+树包含的索引比B树的索引更多（因为B树每个节点还有Data域）
* B+树的叶子节点是通过链表连接的，所以找到下限后能很快进行区间查询，比B树中序遍历

**B树和红黑树的区别**

最大的区别就是树的深度较高，在磁盘I/O方面的表现不如B树。要获取磁盘上数据，必须先通过磁盘移动臂移动到数据所在的柱面，然后找到指定盘面，接着旋转盘面找到数据所在的磁道，最后对数据进行读写。磁盘IO代价主要花费在查找所需的柱面上，树的深度过大会造成磁盘IO频繁读写。根据磁盘查找存取的次数往往由树的高度所决定。

所以，在大规模数据存储的时候，红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下。在这方面，B树表现相对优异，B树可以有多个子女，从几十到上千，可以降低树的高度。

**AVL树和红黑树的区别**

红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高。

* 红黑树和AVL树都能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。 2、由于设计，红黑树的任何不平衡都会在三次旋转之内解决。AVL树增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。
* 在查找方面： 红黑树的性质(最长路径长度不超过最短路径长度的2倍)，其查找代价基本维持在O(logN)左右，但在最差情况下(最长路径是最短路径的2倍少1)，比AVL要略逊色一点。 AVL是严格平衡的二叉查找树（平衡因子不超过1）。查找过程中不会出现最差情况的单支树。因此查找效率最好，最坏情况都是O(logN)数量级的。

所以，综上： AVL比RBtree更加平衡，但是AVL的插入和删除会带来大量的旋转。 所以如果插入和删除比较多的情况，应该使用RBtree, 如果查询操作比较多，应该使用AVL。

AVL是一种高度平衡的二叉树，维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，如果场景中对插入删除不频繁，只是对查找特别有要求，AVL还是优于红黑的。

## 说一说你知道的排序算法



O(n)排序算法

计数排序，基数排序，桶排序

## 大文件排序算法

设想你有一个20GB的文件，每行一个字符串，说明如何对这个文件进行排序。

内存肯定没有20GB大，所以不可能采用传统排序法。但是可以将文件分成许多块，每块xMB,针对每个快各自进行排序，存回文件系统。

然后将这些块逐一合并，最终得到全部排好序的文件。

外排序的一个例子是外归并排序（External merge sort），它读入一些能放在内存内的数据量，在内存中排序后输出为一个顺串（即是内部数据有序的临时文件），处理完所有的数据后再进行归并。[1][2]比如，要对900MB的数据进行排序，但机器上只有100 MB的可用内存时，外归并排序按如下方法操作：

1. 读入100 MB的数据至内存中，用某种常规方式（如快速排序、堆排序、归并排序等方法）在内存中完成排序。
2. 将排序完成的数据写入磁盘。
3. 重复步骤1和2直到所有的数据都存入了不同的100 MB的块（临时文件）中。在这个例子中，有900 MB数据，单个临时文件大小为100 MB，所以会产生9个临时文件。
4. 读入每个临时文件（顺串）的前10 MB（ = 100 MB / (9块 + 1)）的数据放入内存中的输入缓冲区，最后的10 MB作为输出缓冲区。（实践中，将输入缓冲适当调小，而适当增大输出缓冲区能获得更好的效果。）
5. 执行九路归并算法，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。一旦9个输入缓冲区中的一个变空，就从这个缓冲区关联的文件，读入下一个10M数据，除非这个文件已读完。这是“外归并排序”能在主存外完成排序的关键步骤 -- 因为“归并算法”(merge algorithm)对每一个大块只是顺序地做一轮访问(进行归并)，每个大块不用完全载入主存。

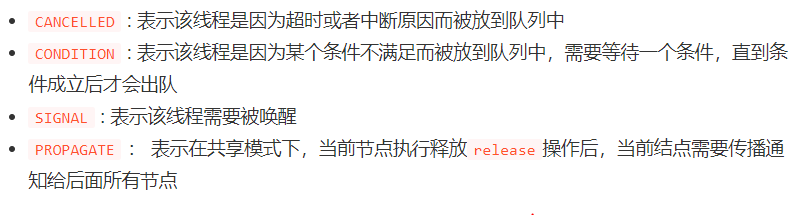
# Java基础

## AQS 抽象队列同步器

AbstractQueueSynchronizer，抽象队列同步器，定义了一套多线程访问共享资源的同步框架。

AQS基于一个FIFO双向队列实现，被设计给那些依赖一个代表状态的原子int值的同步器使用。我们都知道，既然叫同步器，那个肯定有个代表同步状态（临界资源）的东西，在AQS中即为一个叫state的int值，该值通过CAS进行原子修改，并且是volatile修饰的。

在AQS中存在一个FIFO队列，队列中的节点表示被阻塞的线程，队列节点元素有4种类型， 每种类型表示线程被阻塞的原因，这四种类型分别是：



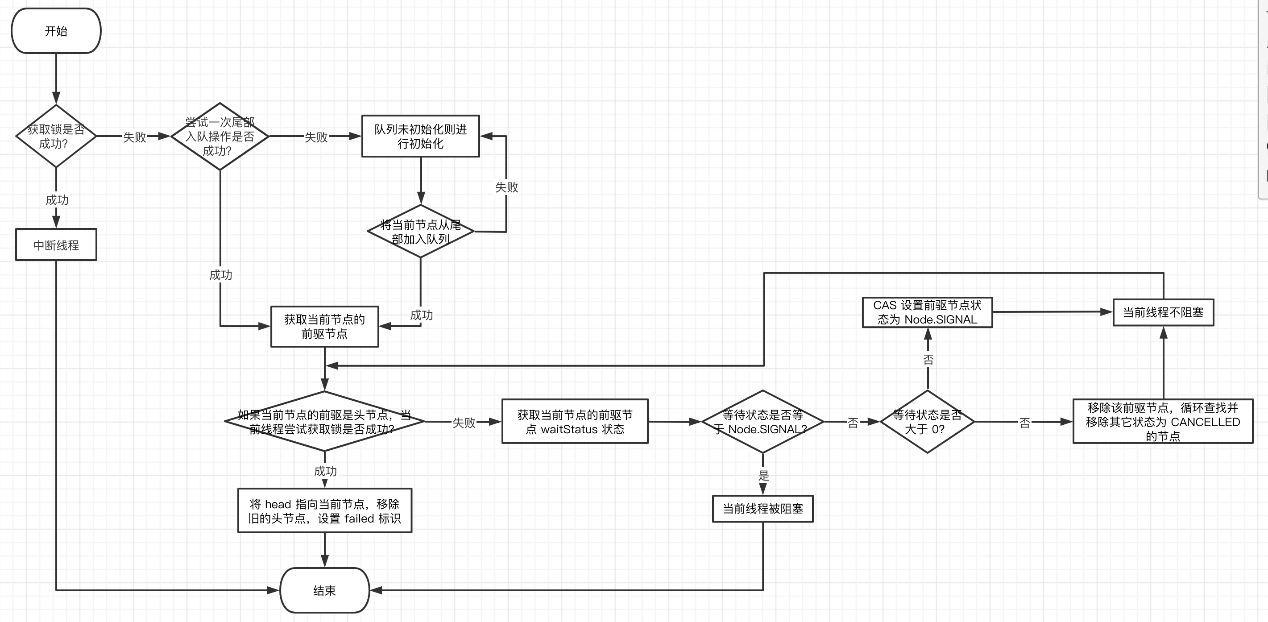
由于一个共享资源同一时间只能由一条线程持有，也可以被多个线程持有，因此AQS中存在两种模式，如下：

* 独占模式：独占模式表示共享状态值state每次能由一条线程持有，其他线程如果获取，则需要阻塞,如JUC中的ReentrantLock
* 共享模式：共享模式表示共享状态值state每次可以由多个线程持有，如JUC中的CountDownLatch

**AQS 中的核心数据结构和方法**

* AQS是基于一个FIFO队列的框架：总结下Node节点数据结构设计，队列中的元素，肯定是为了保存由于某种原因导致无法获取共享资源state而被入队的线程，因此Node中使用了waitStatus表示节点入队的原因，使用Thread对象来表示节点所关联的线程。至于prev,next，则是一般双向队列数据结构必须提供的指针，用于对队列进行相关操作。
* AQS中的共享状态值。由源码我们可以看出，AQS声明了一个int类型的state值，为了达到多线程同步的功能，必然对该值的修改必须多线程可见，因此，state采用volatile修饰，而且getState()和setState()方法采用final进行修饰，目的是限制AQS的子类只能调用这两个方法对state的值进行设置和获取，而不能对其进行重写自定义设置/获取逻辑。AQS中提供对state值修改的方法不仅仅只有setState()和getState()，还有诸如采用CAS机制进行设置的compareAndSetState()方法，同样，该方法也是采用final修饰的，不允许子类重写，只能调用。
* AQS中的tryXXX方法。一般基于AQS实现的同步器，如ReentrantLock,CountDownLatch等，对于state的获取操作，子类只需重写其tryAcquire()和tryAcquireShared()方法即可，这两个方法分别对应独占模式和共享模式下对state的获取操作；而对于释放操作，子类只需重写tryRelease()和tryReleaseShared()方法即可。至于如何维护队列的出队、入队操作，子类不用管，AQS已经帮你做好了。

<https://segmentfault.com/a/1190000022909099>



## 内部类分类以及使用场景，优缺点

* 成员内部类
* 局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。局部内部类就像方法里面的局部变量一样，是不能有public、protected、private及static修饰符的。

* 静态内部类

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。静态内部类是不需要依赖于外部类的，它不持有指向外部类对象的引用this，并且它不能使用外部类的非static成员或方法，这点很好理解，因为在没有外部类的对象的情况下，可以创建静态内部类的对象，如果允许访问外部类的非static成员就会产生矛盾，因为外部类的非static成员必须依附于具体对象。它唯一的作用就是随着类的加载（而不是随着对象的产生）而产生。

* 匿名内部类

匿名内部类应该是平时我们编写代码时用的最多的，在编写事件监听的代码时匿名内部类不但方便，而且使代码更加容易维护。匿名内部类是唯一一种没有构造器的类。正因为其没有构造器，所以匿名内部类的使用范围非常有限，大部分匿名内部类用于接口回调。

外部类与内部类访问权限

内部类与外部类之间的成员互相访问

内部类可以访问外部类的任何成员，包括private成员。

外部类访问内部类的成员需要创建内部类的对象，之后可以访问内部类的任何成员，包括private成员，需要注意的是成员内部类不可以有静态成员。

当外部类的成员和内部类的成员重名时单单用this是区分不了的。在内部类中访问外部类的成员时可以用如下语法区分

<外部类类名>.this.<外部类中需要被访问的成员名>;

优点：

* 每个内部类都能独立的集成一个接口的实现，所以无论外部类是否已经集成了某个（接口）实现，对于内部类都没有影响。内部类使得多重集成的解决方案变得完整。
* 方便将存在一定逻辑关系的类组织在一起，又可以对外界隐藏。
* 方便编写时间驱动程序。
* 方便编写线程代码。

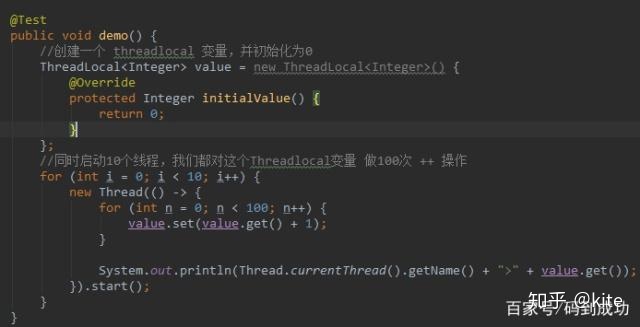
## ThreadLocal

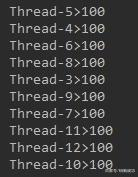
ThreadLocal提供了线程的局部变量，每个线程都可以通过set()和get()来对这个局部变量进行操作，但不会和其他线程的局部变量进行冲突，实现了线程的数据隔离～。

简要言之：往ThreadLocal中填充的变量属于当前线程，该变量对其他线程而言是隔离的。

ThreadLocal是java中一个非常常用的并发工具类，和synchronized 利用锁机制来保证线程安全不同，ThreadLocal 采用给每一个变量拷贝一个副本的方式解决线程安全问题，每一个线程都有一个自己独立的变量副本，任何线程的操作都只会影响自己的副本.

在上面的例子中，我们创建了一个ThreadLocal变量，并启动了10个线程，每个线程都对他做 ++ 操作 100次，下面我们来看一下结果



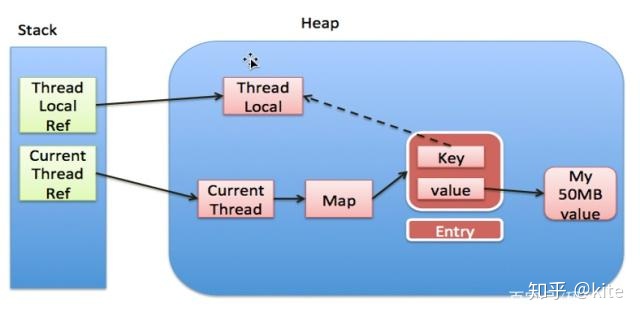


可以看到 每个线程的结果都是100，所有线程之间的值是隔离的。

底层原理解析

掌握了基本用法之后，我们就来聊聊ThreadLocal 的底层原理吧。

在java中每一个线程都会有一个 ThrealLocalMap ，里面存放了一个一个的键值对（我们一般称之为Entry），键值对的key 就是 Threadlocal本身，而value 就是他的值，正如下图所示。

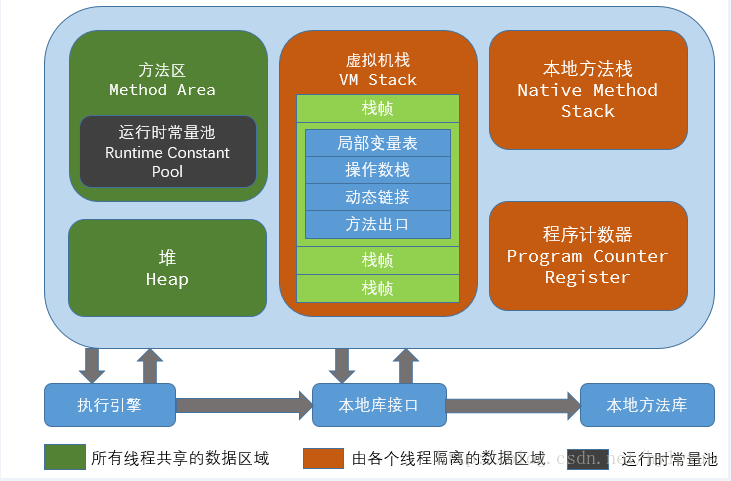


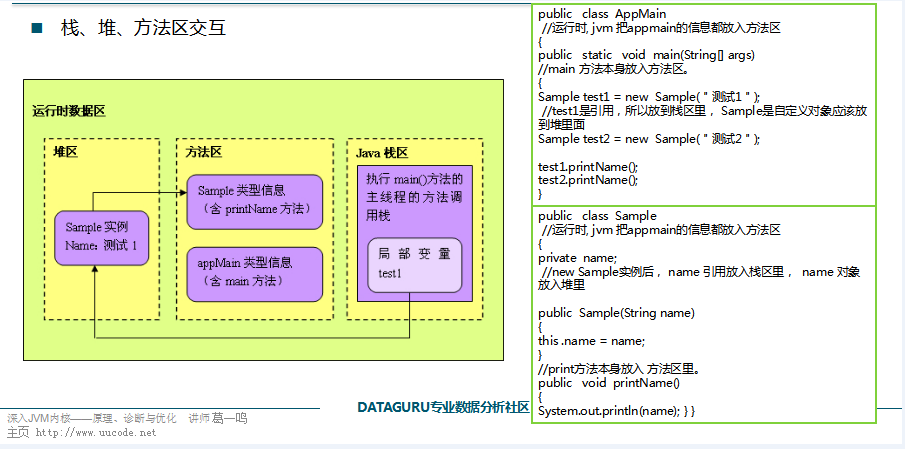
我们仔细的看一下这张图。左边是stack（栈），栈里面有 用户线程 和 threadlocal，threadlocal 用一个弱引用指向了 我们ThrealLocalMap 的key,而用户线程 则是一个强引用指向了 这个enrty的 value.现在我们就能理解为什么threadlocal 的副本是怎么回事了，线程拷贝一份值，用Threadlocal自身做key ，保存在一个map中，从而实现了每个线程之间变量的互相隔离。

## Unsafe类

# Java虚拟机

## 运行时数据区



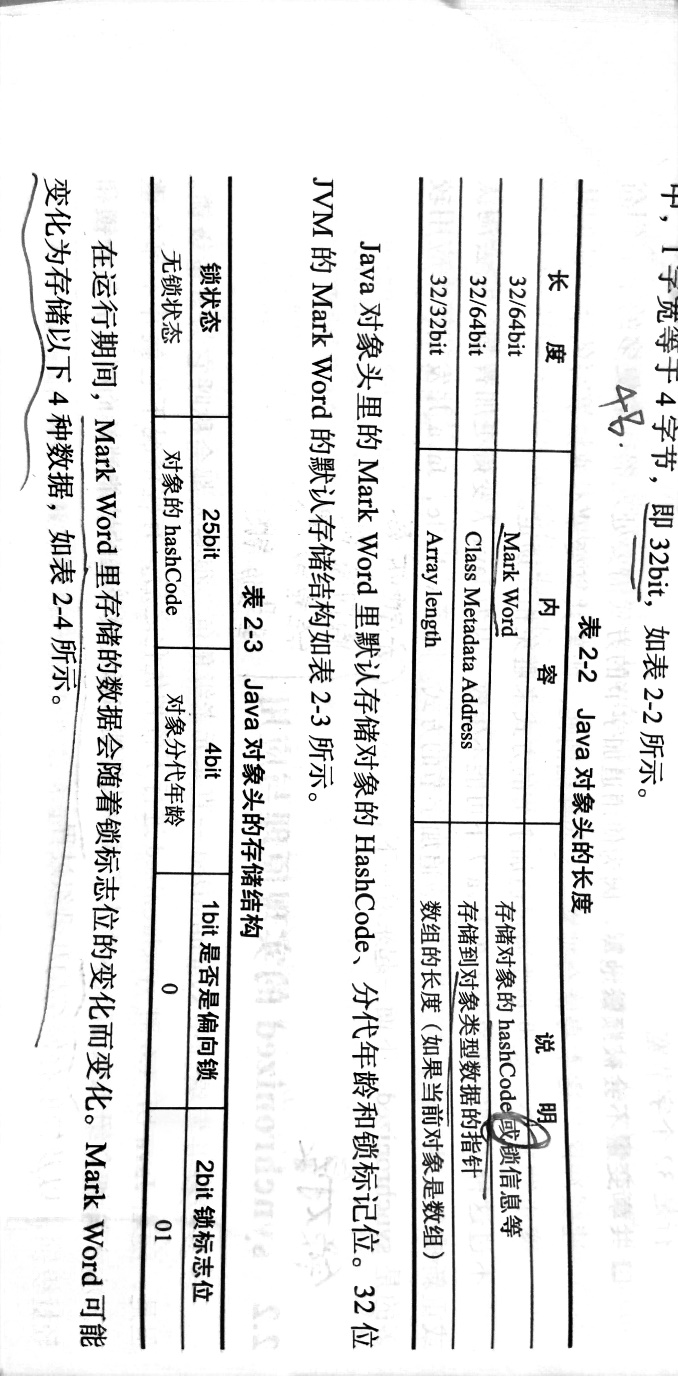


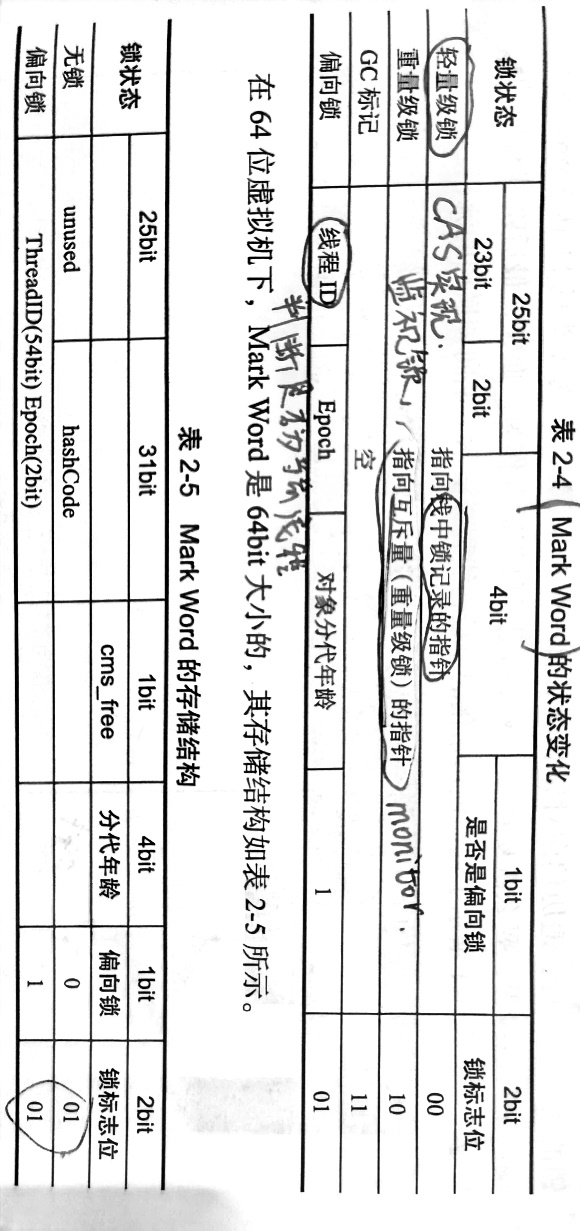
方法区（线程共享）

方法区也是所有线程共享的。主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。这个区域的内存回收目标主要针对常量池的回收和对类型的卸载。当方法区无法满足内存分配需求时，则抛出OutOfMemoryError异常。在HotSpot虚拟机中，用永久代来实现方法区，将GC分代收集扩展至方法区，但是这样容易遇到内存溢出的问题。

JDK1.7中，已经把放在永久代的字符串常量池移到堆中。JDK1.8撤销永久代，引入元空间。

## 对象头的内存布局





Java对象头

synchronized是悲观锁，在操作同步资源之前需要给同步资源先加锁，这把锁就是存在Java对象头里的，而Java对象头又是什么呢？

我们以Hotspot虚拟机为例，Hotspot的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。

Mark Word：默认存储对象的HashCode，分代年龄和锁标志位信息。这些信息都是与对象自身定义无关的数据，所以Mark Word被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。它会根据对象的状态复用自己的存储空间，也就是说在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。

class Point：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

Monitor

Monitor可以理解为一个同步工具或一种同步机制，通常被描述为一个对象。每一个Java对象就有一把看不见的锁，称为内部锁或者Monitor锁。

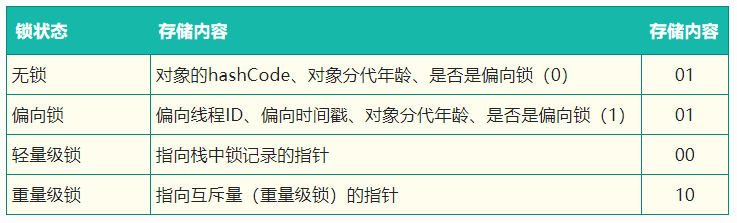
Monitor是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。

现在话题回到synchronized，synchronized通过Monitor来实现线程同步，Monitor是依赖于底层的操作系统的Mutex Lock（互斥锁）来实现的线程同步。

如同我们在自旋锁中提到的“阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长”。这种方式就是synchronized最初实现同步的方式，这就是JDK 6之前synchronized效率低的原因。这种依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”，JDK 6中为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”。

所以目前锁一共有4种状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁。锁状态只能升级不能降级。

通过上面的介绍，我们对synchronized的加锁机制以及相关知识有了一个了解，那么下面我们给出四种锁状态对应的的Mark Word内容，然后再分别讲解四种锁状态的思路以及特点：



无锁：无锁没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。

无锁的特点就是修改操作在循环内进行，线程会不断的尝试修改共享资源。如果没有冲突就修改成功并退出，否则就会继续循环尝试。如果有多个线程修改同一个值，必定会有一个线程能修改成功，而其他修改失败的线程会不断重试直到修改成功。上面我们介绍的CAS原理及应用即是无锁的实现。无锁无法全面代替有锁，但无锁在某些场合下的性能是非常高的。

偏向锁：偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁，降低获取锁的代价。在大多数情况下，锁总是由同一线程多次获得，不存在多线程竞争，所以出现了偏向锁。其目标就是在只有一个线程执行同步代码块时能够提高性能。

当一个线程访问同步代码块并获取锁时，会在Mark Word里存储锁偏向的线程ID。在线程进入和退出同步块时不再通过CAS操作来加锁和解锁，而是检测Mark Word里是否存储着指向当前线程的偏向锁。引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令即可。

偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态。撤销偏向锁后恢复到无锁（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

偏向锁在JDK 6及以后的JVM里是默认启用的。可以通过JVM参数关闭偏向锁：-XX:-UseBiasedLocking=false，关闭之后程序默认会进入轻量级锁状态。

轻量级锁：是指当锁是偏向锁的时候，被另外的线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，从而提高性能。

在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，然后拷贝对象头中的Mark Word复制到锁记录中。

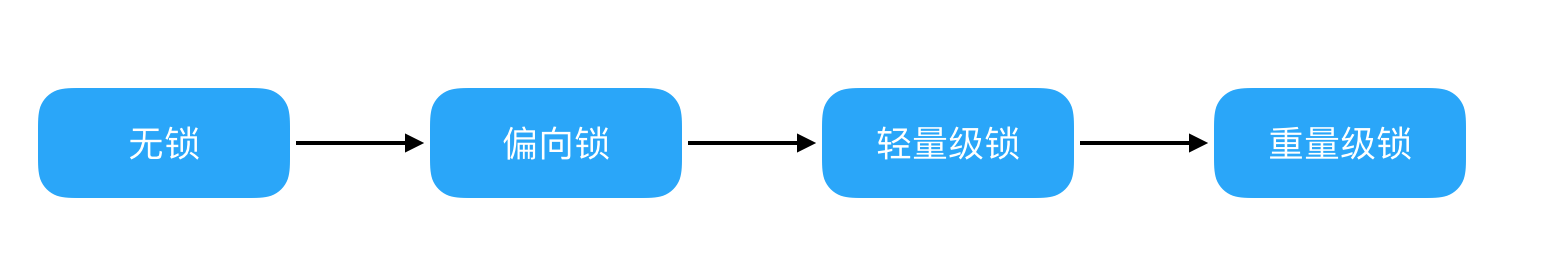
拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指针指向对象的Mark Word。

如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，表示此对象处于轻量级锁定状态。

如果轻量级锁的更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行，否则说明多个线程竞争锁。

若当前只有一个等待线程，则该线程通过自旋进行等待。但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁升级为重量级锁。

重量级锁：升级为重量级锁时，锁标志的状态值变为“10”，此时Mark Word中存储的是指向重量级锁的指针，此时等待锁的线程都会进入阻塞状态。



综上，偏向锁通过对比Mark Word解决加锁问题，避免执行CAS操作。而轻量级锁是通过用CAS操作和自旋来解决加锁问题，避免线程阻塞和唤醒而影响性能。重量级锁是将除了拥有锁的线程以外的线程都阻塞。

## JVM系统线程（后台线程/守护线程）

守护线程也称“服务线程”，他是后台线程，它有一个特性，即为用户线程 提供公共服务，在没有用户线程可服务时会自动离开。

* 虚拟机线程：这种线程的操作需要JVM达到安全点才会出现，这些操作必须在不同的线程中发生的原因是他们需要JVM达到安全点，这样堆才不会出现变化，这种线程的执行类型包括“STW”的垃圾回收，线程栈收集，线程挂起以及偏向锁撤销。
* GC线程：这种线程对在JVM里不同种类的垃圾收集行为提供了支持
* 编译线程：在运行时将字节码编译成本地代码
* 信号调度线程：接受信号并发给JVM，在它内部通过调度适当的方法进行处理。

生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周

期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依

赖于系统，与系统“同生共死”。当JVM 中所有的线程都是守护线程的时候，JVM 就可以退

出了；如果还有一个或以上的非守护线程则JVM 不会退出。

## ‌**垃圾收集器**

## **运行时常量池**

## **字符串在虚拟机中的存储**

## 内存泄漏跟内存溢出

内存溢出：简单地说内存溢出就是指程序运行过程中申请的内存大于系统能够提供的内存，导致无法申请到足够的内存，于是就发生了内存溢出。 内存泄漏：内存泄漏指程序运行过程中分配内存给临时变量，用完之后却没有被GC回收，始终占用着内存，既不能被使用也不能分配给其他程序，于是就发生了内存泄漏。

## OOM如何排查？

## Java程序编程字节码个过程

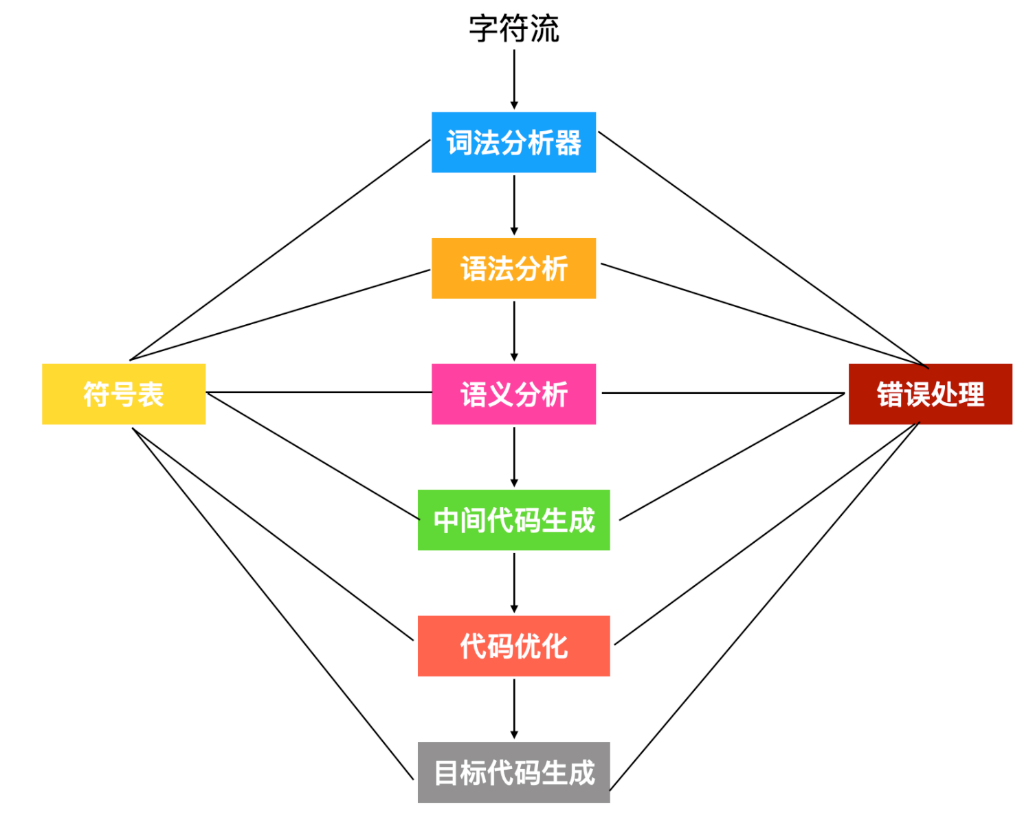
javac编译器将Java编译成为一个有效的字节码文件会经历4个步骤：

词法解析：将Java关键字排序，使得程序能有序运行。

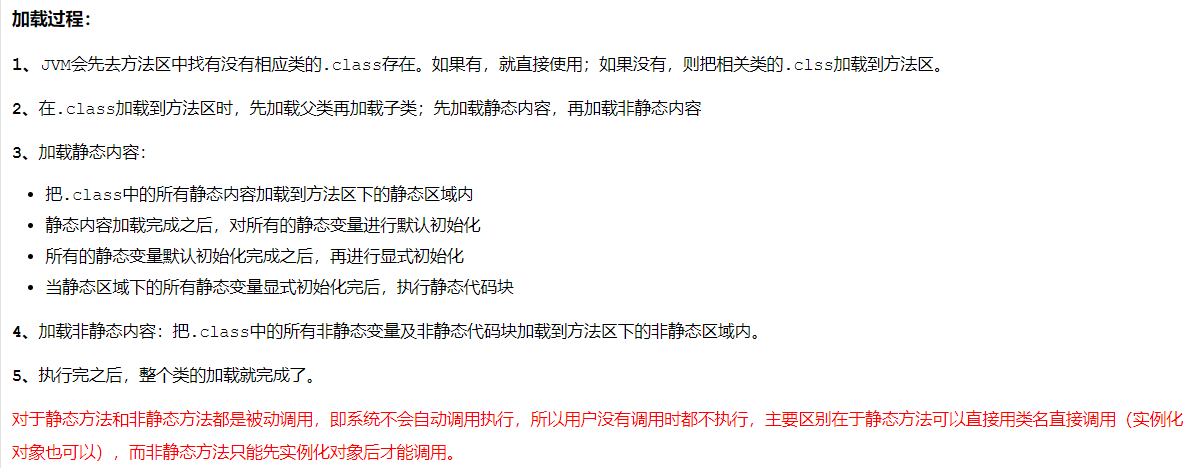
语法解析：词法解析后的Token序列整合为一颗抽象的语法树。

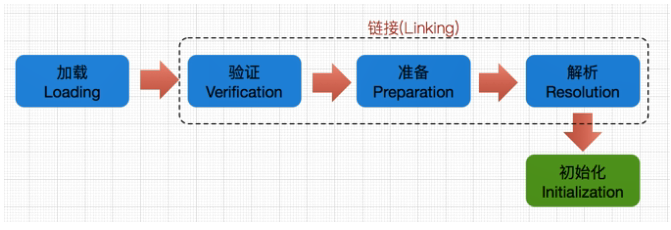
语义解析：将抽象语法树扩展地更加完善。

字节码解析：将字节码解析成完整的类。



## 类加载流程，类初始化流程？





第一步：Loading加载

通过类的全限定名（包名 + 类名），获取到该类的.class文件的二进制字节流，将二进制字节流所代表的静态存储结构，转化为方法区运行时的数据结构，在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

总结：加载二进制数据到内存—>映射成jvm能识别的结构 —> 在内存中生成class文件。

第二步：Linking链接

链接是指将上面创建好的class类合并至Java虚拟机中，使之能够执行的过程，可分为验证、准备、解析三个阶段。

① 验证（Verify）

确保class文件中的字节流包含的信息，符合当前虚拟机的要求，保证这个被加载的class类的正确性，不会危害到虚拟机的安全。

② 准备（Prepare）

为类中的**静态字段分配内存**，并设置默认的初始值，比如int类型初始值是0。被final修饰的static字段不会设置，因为final在编译的时候就分配了

③ 解析（Resolve）

解析阶段的目的，是将常量池内的符号引用转换为直接引用的过程（将常量池内的符号引用解析成为实际引用）。如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么解析将触发这个类的加载（但未必触发这个类的链接以及初始化。）

事实上，解析器操作往往会伴随着 JVM 在执行完初始化之后再执行。 符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《Java 虚拟机规范》的Class文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

解析动作主要针对类、接口、字段、类方法、接口方法、方法类型等。对应常量池中的 CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等。

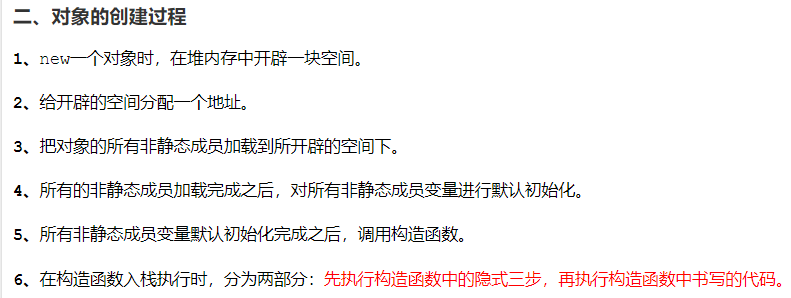
第三步：initialization初始化

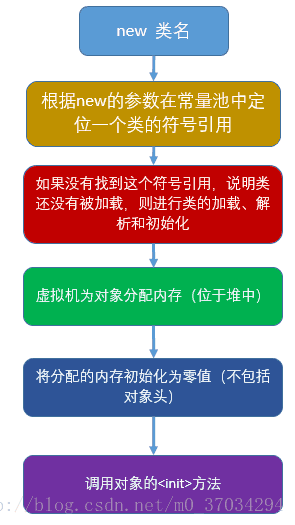
初始化就是执行类的构造器方法init()的过程。

这个方法不需要定义，是javac编译器自动收集类中所有类变量的赋值动作和静态代码块中的语句合并来的。

若该类具有父类，jvm会保证父类的init先执行，然后在执行子类的init。

实例创建？





# 计算机网络

## **HTTPS协议**

## **http1.0，http1.1，http2.0的区别？**

http1.0 1.1 与2.0的区别



## Cookie的作用是什么?和Session有什么区别？

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

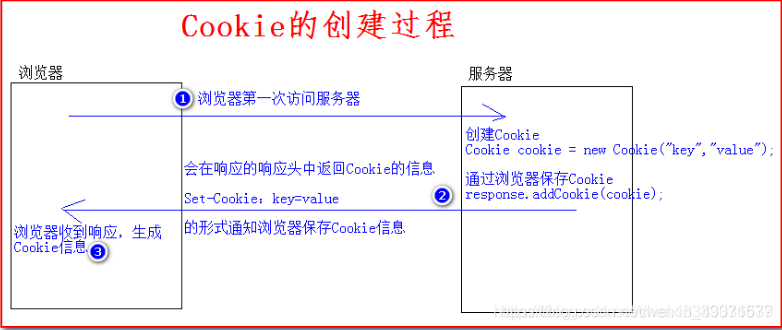
* cookie工作原理：

（1）浏览器端第一次发送请求到服务器端

（2）服务器端创建Cookie，该Cookie中包含用户的信息，然后将该Cookie发送到浏览器端

（3）浏览器端再次访问服务器端时会携带服务器端创建的Cookie

（4）服务器端通过Cookie中携带的数据区分不同的用户

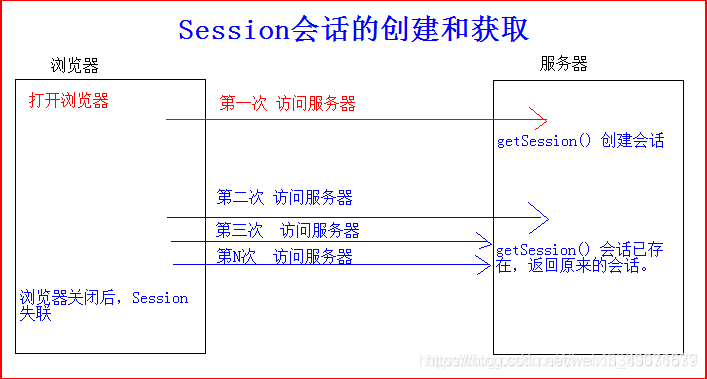


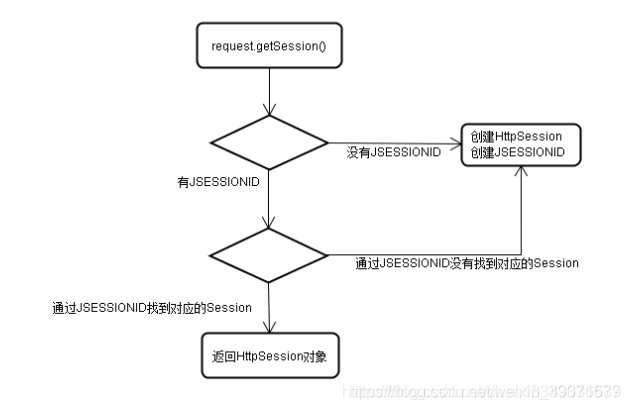
* Session的工作原理

（1）浏览器端第一次发送请求到服务器端，服务器端创建一个Session，同时会创建一个特殊的Cookie（name为JSESSIONID的固定值，value为session对象的ID），然后将该Cookie发送至浏览器端

（2）浏览器端发送第N（N>1）次请求到服务器端,浏览器端访问服务器端时就会携带该name为JSESSIONID的Cookie对象

（3）服务器端根据name为JSESSIONID的Cookie的value(sessionId),去查询Session对象，从而区分不同用户。





Cookie一般用来保存用户信息 比如①我们在 Cookie 中保存已经登录过得用户信息，下次访问网站的时候页面可以自动帮你登录的一些基本信息给填了；②一般的网站都会有保持登录也就是说下次你再访问网站的时候就不需要重新登录了，这是因为用户登录的时候我们可以存放了一个 Token 在 Cookie 中，下次登录的时候只需要根据 Token 值来查找用户即可(为了安全考虑，重新登录一般要将 Token 重写)；③登录一次网站后访问网站其他页面不需要重新登录。Session 的主要作用就是通过服务端记录用户的状态。 典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了。

Cookie 存储在客户端中，而Session存储在服务器上，相对来说 Session 安全性更高。如果要在 Cookie 中存储一些敏感信息，不要直接写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

* 存储位置不同： cookie数据存放在客户的浏览器上，session数据放在服务器上。
* 存储容量不同： 单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。对于Session并没有上限，但出于对服务器端的性能考虑，Session内不要存放过多的东西，并设置session删除机制。
* 存取方式不同： cookie中只能保管ASCII字符串，并需要通过编码方式存储为Unicode字符或者二进制数据。session中能够存储任何类型的数据，包括且不限于string，integer，list，map等。
* 隐私策略不同： Cookie对客户端是可见的，别有用心的人可以分析放在本地的Cookie上面并进行Cookie欺骗，所以它是不安全的。
* 有效期不同： 开发可以通过设置Cookie的属性，达到Cookie长期有效的效果，只要关闭窗口该Session就会失效，因为假如设置Session的超过时间过长，服务器累计的Session就会越多，越容易导致内存溢出。
* 跨域支持上的不同： Cookie 支持跨域名访问，例如，将 domain 属性设置为“.biaodianfu.com”，则以“.biaodianfu.com”为后缀的一切域名均能够访问该Cookie。Session则不会支持跨域名访问。Session仅在它所在的域名内有效。

## 如果客户端禁用了cookie，怎么使用session？

（1）使用URL重写，将sessionid附在URL的后面，传给服务器。

（2）表单隐藏字段，服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把sessionid传回服务器。

## Get与POST的区别

GET与POST是我们常用的两种HTTP Method，二者之间的区别主要包括如下五个方面：

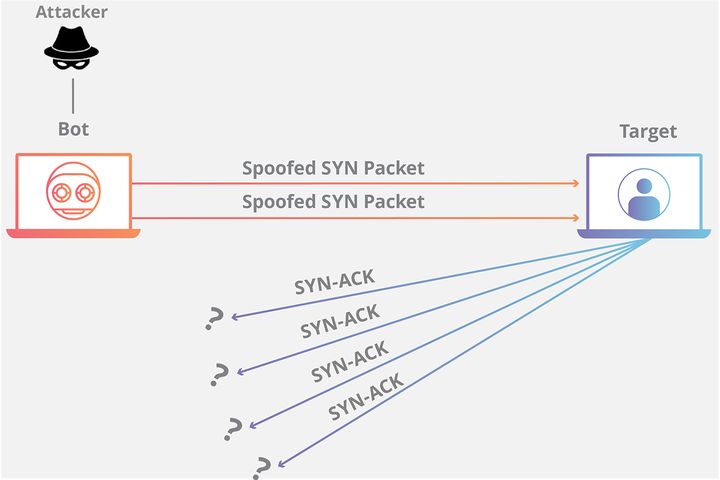
* 从功能上讲，GET一般用来从服务器上获取资源，POST一般用来更新服务器上的资源
* 从REST服务角度上说，GET是幂等的，即读取同一个资源，总是得到相同的数据，而POST不是幂等的，因为每次请求对资源的改变并不是相同的；进一步地，GET不会改变服务器上的资源，而POST会对服务器资源进行改变；
* 从请求参数形式上看，GET请求的数据会附在URL之后，即将请求数据放置在HTTP报文的 请求头 中，以?分割URL和传输数据，参数之间以&相连。特别地，如果数据是英文字母/数字，原样发送；否则，会将其编码为 application/x-www-form-urlencoded MIME 字符串(如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如：%E4%BD%A0%E5%A5%BD，其中％XX中的XX为该符号以16进制表示的ASCII)；而POST请求会把提交的数据则放置在是HTTP请求报文的 请求体中。
* 就安全性而言，POST的安全性要比GET的安全性高，因为GET请求提交的数据将明文出现在URL上，而且POST请求参数则被包装到请求体中，相对更安全。
* 从请求的大小看，GET请求的长度受限于浏览器或服务器对URL长度的限制，允许发送的数据量比较小，而POST请求则是没有大小限制的。



## TCP协议 TCP快连接

## SYN泛洪攻击？

SYN攻击利用的是TCP的三次握手机制，攻击端利用伪造的IP地址向被攻击端发出请求，而被攻击端发出的响应报文将永远发送不到目的地，那么被攻击端在等待关闭这个连接的过程中消耗了资源，如果有成千上万的这种连接，主机资源将被耗尽，从而达到攻击的目的。



从防御角度来讲，存在几种的解决方法：

* 第一种是缩短SYN Timeout时间。由于SYN Flood攻击的效果取决于服务器上保持的SYN半连接数，这个值=SYN攻击的频度 x SYN Timeout，所以通过缩短从接收到SYN报文到确定这个报文无效并丢弃改连接的时间
* 第二种方法是设置SYN Cookie，就是给每一个请求连接的IP地址分配一个Cookie，如果短时间内连续受到某个IP的重复SYN报文，就认定是受到了攻击，以后从这个IP地址来的包会被一概丢弃。

## 数据包从一端发到另一端的流程，从协议栈的角度说，涉及到添加首部、寻址什么的，数据包从网卡出去后经历的过程

## ARP协议的工作原理和流程，路由器是如何转发的？(路由表的工作原理)

## 讲一下拥塞控制

## 延迟ACK了解么？确认报文随数据一起传送过去，或者是设定超时时间到了单独发过去

## 访问www.qq.com中间有多少次网络交互，分别有什么协议

## TCP如何处理丢包？TCP如何处理重传？

# 数据库

## 数据库范式



## 事务底层实现原理？

回顾数据库事务总结word版

## 什么是索引？索引的种类？

索引是一种文件，它包含着对数据库表里所有记录的引用指针。

索引（在 MySQL 中也叫“键key”）是存储引擎快速找到记录的一种数据结构，通俗来说类似书本的目录。为查找书中的内容，通过内容建立索引形成目录。

种类：

* 主键索引
* 唯一索引
* 普通索引
* 全文索引

## 聚簇索引和非聚簇索引，有什么区别？哪个快？

聚集索引与非聚集索引的区别是：索引文件与数据文件是否放在一起。

InnoDB 主键使用的是聚簇索引，MyISAM 不管是主键索引，还是二级索引使用的都是非聚簇索引。

下图形象说明了聚簇索引表(InnoDB)和非聚簇索引(MyISAM)的区别：



MyLSAM中的记录并不是通过主键从小到大进行排序的，因此不能进行聚簇索引查找。InnoDB的主键都是通过主键从小到大排序的。

使用MyLSAM存储引擎的表会将索引信息单独存储到另外一个文件中，称为索引文件索引。MyLSAM会为表的主键单独创建一个索引文件，只不过索引的叶子节点中存储的不是完整的用户记录，而是主键与行号的组合。先通过主键获取行号，然后在通过行号获取相应的记录。

在InnoDB存储引擎中，我们只需要根据主键值对聚簇索引进行一次查找就可以找到相应的记录，而在MyLSAM中却需要进行一个回表操作，这也意味着MyLsam中建立的索引全部都是二级索引。

* 对于非聚簇索引表来说（右图），表数据和索引是分成两部分存储的，主键索引和二级索引存储上没有任何区别。使用的是B+树作为索引的存储结构，所有的节点都是索引，叶子节点存储的是索引+索引对应的记录的数据。
* 对于聚簇索引表来说（左图），表数据是和主键一起存储的，主键索引的叶结点存储行数据(包含了主键值)，二级索引的叶结点存储行的主键值。使用的是B+树作为索引的存储结构，非叶子节点都是索引关键字，但非叶子节点中的关键字中不存储对应记录的具体内容或内容地址。叶子节点上的数据是主键与具体记录(数据内容)。

**聚簇索引的优点**：

* 当你需要取出一定范围内的数据时，用聚簇索引也比用非聚簇索引好。
* 当通过聚簇索引查找目标数据时理论上比非聚簇索引要快，因为非聚簇索引定位到对应主键时还要多一次目标记录寻址,即多一次I/O。
* 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

**聚簇索引的缺点**：

* 插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个自增的ID列为主键
* 更新主键的代价很高，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。
* **二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。**二级索引的叶节点存储的是主键值，而不是行指针（非聚簇索引存储的是指针或者说是地址），这是为了减少当出现行移动或数据页分裂时二级索引的维护工作，但会让二级索引占用更多的空间。
* 采用聚簇索引插入新值比采用非聚簇索引插入新值的速度要慢很多，因为插入要保证主键不能重复，判断主键不能重复，采用的方式在不同的索引下面会有很大的性能差距，聚簇索引遍历所有的叶子节点，非聚簇索引也判断所有的叶子节点，但是聚簇索引的叶子节点除了带有主键还有记录值，记录的大小往往比主键要大的多。这样就会导致聚簇索引在判定新记录携带的主键是否重复时进行昂贵的I/O代价。

## 聚簇索引，二级索引(辅助索引)，联合索引？索引覆盖？

**聚簇索引**的叶子包含所有的行记录信息。聚簇索引只能在搜索条件是主键时才能发挥作用。原因是，B+树中的数据都是按照主键进行排序的。

二级索引的叶子只包含主键的信息。在耳机索引中进行查找时，查找的结果是得到主键，然后再通过主键进行回表查询。根据主键再从聚簇索引进行查找。

通过携带主键信息到聚簇索引中重新定位完整的用户记录也称为回表。

**联合索引**是指同时以多个列的大小作为排序规则，也就是为多个列建立索引。从左往右建立索引。

覆盖索引（covering index ，或称为索引覆盖）即从非主键索引中就能查到的记录，而不需要查询主键索引中的记录，避免了回表的产生减少了树的搜索次数，显著提升性能。

## 创建索引的原则？是什么时候适合使用索引？

1. 选择唯一性索引

唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。例如，学生表中学号是具有唯一性的字段。为该字段建立唯一性索引可以很快的确定某个学生的信息。如果使用姓名的话，可能存在同名现象，从而降低查询速度。

1. 为经常需要排序、分组和联合操作的字段建立索引

经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。

1. 为常作为查询条件的字段建立索引

如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。

1. 限制索引的数目

索引的数目不是越多越好。每个索引都需要占用磁盘空间，索引越多，需要的磁盘空间就越大。修改表时，对索引的重构和更新很麻烦。越多的索引，会使更新表变得很浪费时间。

1. 最左前缀匹配原则，非常重要的原则。

mysql会一直向右匹配直到遇到**范围查询**(>、<、between、like)就停止匹配，比如a 1=”” and=”” b=”2” c=”“> 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

1. 尽量选择区分度高的列作为索引。

区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就 是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条 记录

1. 尽量使用数据量少的索引

如果索引的值很长，那么查询的速度会受到影响。例如，对一个CHAR(100)类型的字段进行全文检索需要的时间肯定要比对CHAR(10)类型的字段需要的时间要多。

## 索引的数据结构，各自优缺点？

B+树与Hash表。

B+索引的优势：  
支持范围查找。

Hash索引优点：

* 等值查询。哈希索引具有绝对优势（前提是：没有大量重复键值，如果大量重复键值时，哈希索引的效率很低，因为存在所谓的哈希碰撞问题。）

缺点：

* 不支持范围查询
* 不支持索引完成排序
* 不支持联合索引的最左前缀匹配规则

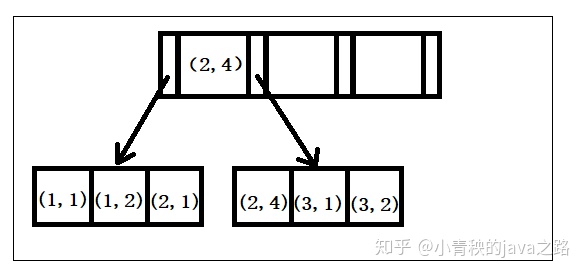
两者使用场景：

* 如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据。
* 如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索。
* 同理，哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）。
* 哈希索引也不支持多列联合索引的最左匹配规则。
* B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

## 最左前缀原则？最左匹配原则

**最左前缀匹配原则**：在MySQL建立联合索引时会遵守最左前缀匹配原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。

要想理解联合索引的最左匹配原则，先来理解下索引的底层原理。索引的底层是一颗B+树，那么联合索引的底层也就是一颗B+树，只不过联合索引的B+树节点中存储的是键值。由于构建一棵B+树只能根据一个值来确定索引关系，所以数据库依赖联合索引最左的字段来构建。举例：创建一个（a,b）的联合索引，那么它的索引树就是下图的样子。



可以看到a的值是有顺序的，1，1，2，2，3，3，而b的值是没有顺序的1，2，1，4，1，2。但是我们又可发现a在等值的情况下，b值又是按顺序排列的，但是这种顺序是相对的。这是因为MySQL创建联合索引的规则是首先会对联合索引的最左边第一个字段排序，在第一个字段的排序基础上，然后在对第二个字段进行排序。所以b=2这种查询条件没有办法利用索引。

多个单列索引和联合索引的区别详解：配合使用，写的很好！

<https://blog.csdn.net/Abysscarry/article/details/80792876>

通俗理解：

利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引 不同于使用两个单独的索引。复合索引的结构与电话簿类似，人名由姓和名构成，电话簿首先按姓氏对进行排序，然后按名字对有相同姓氏的人进行排序。如果您知道姓，电话簿将非常有用；如果您知道姓和名，电话簿则更为有用，但如果您只知道名不姓，电话簿将没有用处。

所以说创建复合索引时，应该仔细考虑列的顺序。对索引中的所有列执行搜索或仅对前几列执行搜索时，复合索引非常有用；仅对后面的任意列执行搜索时，复合索引则没有用处。

重点：多个单列索引在多条件查询时优化器会选择最优索引策略，可能只用一个索引，也可能将多个索引全用上！ 但多个单列索引底层会建立多个B+索引树，比较占用空间，也会浪费一定搜索效率，故如果只有多条件联合查询时最好建联合索引！

最左前缀原则：

顾名思义是最左优先，以最左边的为起点任何连续的索引都能匹配上，

注：如果第一个字段是范围查询需要单独建一个索引

注：在创建联合索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。这样的话扩展性较好，比如 userid 经常需要作为查询条件，而 mobile 不常常用，则需要把 userid 放在联合索引的第一位置，即最左边

同时存在联合索引和单列索引（字段有重复的），这个时候查询mysql会怎么用索引呢？

这个涉及到mysql本身的查询优化器策略了，当一个表有多条索引可走时, Mysql 根据查询语句的成本来选择走哪条索引；

联合索引本质：

当创建\*\*(a,b,c)联合索引时，相当于创建了(a)单列索引\*\*，(a,b)联合索引以及\*\*(a,b,c)联合索引\*\*想要索引生效的话,只能使用 a和a,b和a,b,c三种组合；当然，我们上面测试过，a,c组合也可以，但实际上只用到了a的索引，c并没有用到！

## MySQL 用了哪种引擎？区别是什么？

InnoDB与MyLSAM。

区别：

1. InnoDB 支持事务，MyISAM 不支持事务。这是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一
2. InnoDB 支持外键，而 MyISAM 不支持。对一个包含外键的 InnoDB 表转为 MYISAM 会失败；
3. InnoDB 是聚集索引，MyISAM 是非聚集索引。聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而 MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
4. InnoDB 不保存表的具体行数，执行 select count(\*) from table 时需要全表扫描。而MyISAM 用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快
5. InnoDB并发量更好。InnoDB 最小的锁粒度是行锁，MyISAM 最小的锁粒度是表锁。一个更新语句会锁住整张表，导致其他查询和更新都会被阻塞，因此并发访问受限。这也是 MySQL 将默认存储引擎从 MyISAM 变成 InnoDB 的重要原因之一；

## B+树和B树的区别，B+树的优势在哪里？

<https://leetcode-cn.com/circle/discuss/F7bKlM/>

**B+ 树和 B 树的区别：**

* B 树非叶子结点和叶子结点都存储数据,因此查询数据时，时间复杂度最好为 O(1),最坏为 O(log n)。B+ 树只在叶子结点存储数据，非叶子结点存储关键字，且不同非叶子结点的关键字可能重复，因此查询数据时，时间复杂度固定为 O(log n)。
* B+ 树叶子结点之间用链表相互连接，因而只需扫描叶子结点的链表就可以完成一次遍历操作，B树只能通过中序遍历。

**B+树的优势：**

* B+ 树更加适应磁盘的特性，相比 B 树减少了 I/O 读写的次数。由于索引文件很大因此索引文件存储在磁盘上，B+ 树的非叶子结点只存关键字不存数据，因而单个页可以存储更多的关键字，即一次性读入内存的需要查找的关键字也就越多，磁盘的随机 I/O 读取次数相对就减少了。在一般情况下，B+树的高度不超多4层。因此在通过主键值去查找某条记录时，最多仅需要进行四个页面内的查找，查找三个存储目录项记录的页与一个存储用户记录的页。
* B+ 树的查询效率相比B树更加稳定，由于数据只存在在叶子结点上，所以查找效率固定为 O(log n)。
* B+ 树叶子结点之间用链表有序连接，所以扫描全部数据只需扫描一遍叶子结点，利于扫库和范围查询；B 树由于非叶子结点也存数据，所以只能通过中序遍历按序来扫。也就是说，对于范围查询和有序遍历而言，B+ 树的效率更高。

## 建索引是只能对单列建索引吗？能不能对多列建索引。

联合索引。要遵守最左前缀匹配原则。

## sql注入？如何处理SQL注入

Sql 注入攻击是通过将恶意的 Sql 查询或添加语句插入到应用的输入参数中，再在后台 Sql 服务器上解析执行进行恶意的增删改查，它目前黑客对数据库进行攻击的最常用手段之一。

Sql 注入产生原因及威胁：

当我们访问动态网页时, Web 服务器会向数据访问层发起 Sql 查询请求，如果权限验证通过就会执行 Sql 语句。这种网站内部直接发送的Sql请求一般不会有危险。但实际情况是很多时候需要结合用户的输入数据动态构造 Sql 语句，如果用户输入的数据被构造成恶意 Sql 代码，Web 应用又未对动态构造的 Sql 语句使用的参数进行审查，则会带来意想不到的危险。

Sql 注入带来的威胁主要有如下几点

* 猜解后台数据库，这是利用最多的方式，盗取网站的敏感信息。
* 绕过认证，列如绕过验证登录网站后台。
* 注入可以借助数据库的存储过程进行提权等操作

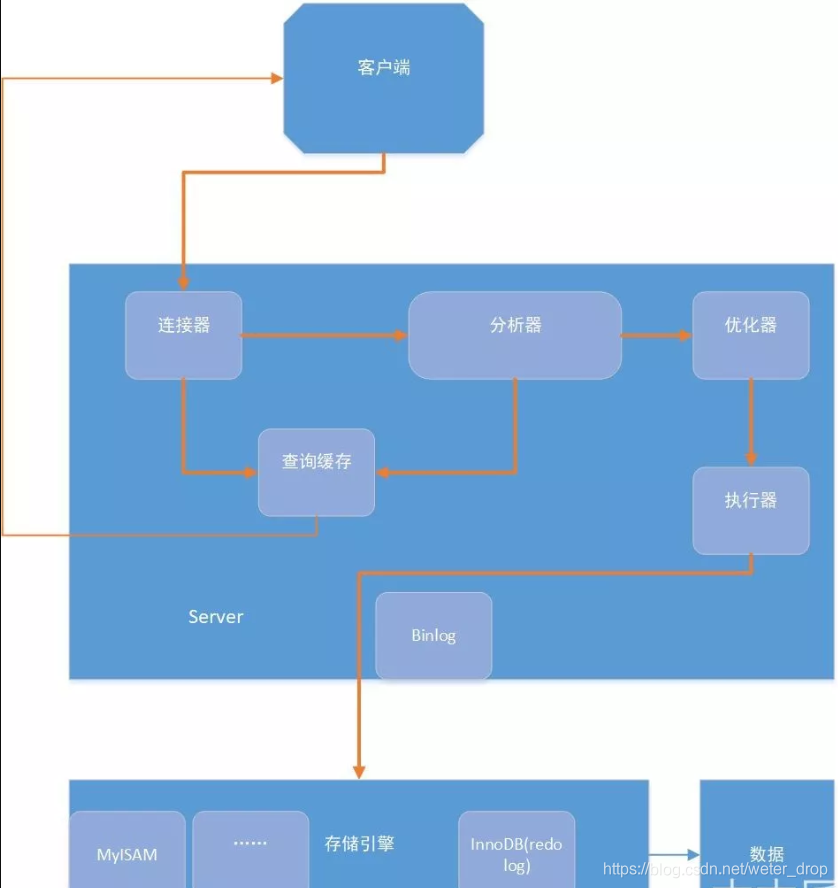
防御：

* 预编译 SQL（Java 中使用 PreparedStatement），参数化查询方式，避免 SQL 拼接
* 严格限制 Web 应用的数据库的操作权限，给连接数据库的用户提供满足需要的最低权限，最大限度的减少注入攻击对数据库的危害
* 校验参数的数据格式是否合法（可以使用正则或特殊字符的判断），使用过滤器
* 报错信息不要包含 SQL 信息输出到 Web 页面

## Delete \* from table where id>2执行这条语句过程中发生了什么？

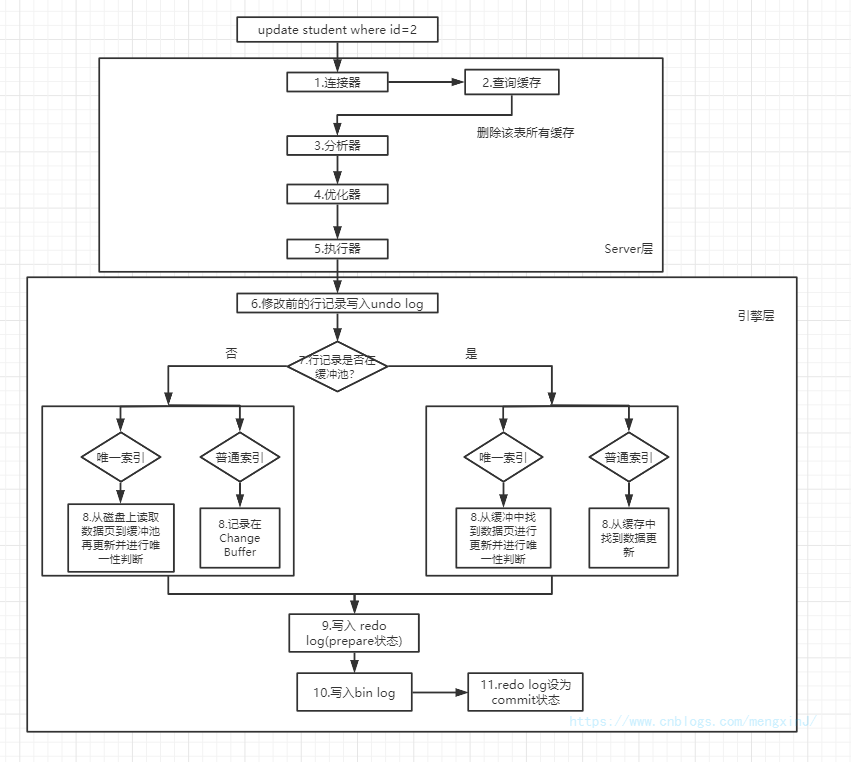
MySQL 基本架构概览：

* 连接器： 身份认证和权限相关(登录 MySQL 的时候)。
* 查询缓存: 执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。
* 分析器: 没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。
* 优化器： 按照 MySQL 认为最优的方案去执行。
* 执行器: 执行语句，然后从存储引擎返回数据



简单来说 MySQL 主要分为 Server 层和存储引擎层：

* Server 层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。
* 存储引擎： 主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory 等多个存储引擎，其中 InnoDB 引擎有自有的日志模块 redolog 模块。现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了。



## 数据库语句执行流程

去哪里取数据？→select

取哪些数据？→where

数据分组吗？→group by

取哪些分组？→having

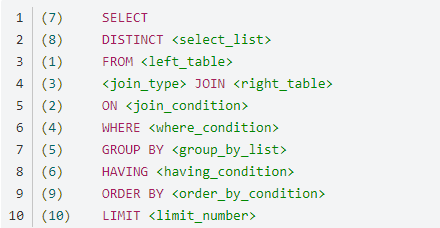
得到数据→select

得到数据后如何表示？

去除重复→distinct

输出顺序→order by

取多少？→limit



1. 执行FROM语句

在这些SQL语句的执行过程中，都会产生一个虚拟表，用来保存SQL语句的执行结果（这是重点），现在就来跟踪这个虚拟表的变化，得到最终的查询结果的过程，来分析整个SQL逻辑查询的执行顺序和过程。

第一步，执行FROM语句。我们首先需要知道最开始从哪个表开始的，这就是FROM告诉我们的。现在有了<left\_table>和<right\_table>两个表，我们到底从哪个表开始，还是从两个表进行某种联系以后再开始呢？它们之间如何产生联系呢？——笛卡尔积

关于什么是笛卡尔积，请自行Google补脑。经过FROM语句对两个表执行笛卡尔积，会得到一个虚拟表

1. 执行ON过滤

执行完笛卡尔积以后，接着就进行ON a.customer\_id = b.customer\_id条件过滤，根据ON中指定的条件，去掉那些不符合条件的数据

1. 执行WHERE过滤

对添加外部行得到的VT3进行WHERE过滤，只有符合<where\_condition>的记录才会输出到虚拟表

1. 执行GROUP BY分组

GROU BY子句主要是对使用WHERE子句得到的虚拟表进行分组操作。

1. 执行HAVING过滤

HAVING子句主要和GROUP BY子句配合使用，对分组得到的VT5虚拟表进行条件过滤。

1. SELECT列表

现在才会执行到SELECT子句，不要以为SELECT子句被写在第一行，就是第一个被执行的。

1. 执行DISTINCT子句

如果在查询中指定了DISTINCT子句，则会创建一张内存临时表（如果内存放不下，就需要存放在硬盘了）。这张临时表的表结构和上一步产生的虚拟表VT7是一样的，不同的是对进行DISTINCT操作的列增加了一个唯一索引，以此来除重复数据。

1. 执行ORDER BY子句

对虚拟表中的内容按照指定的列进行排序，然后返回一个新的虚拟表

1. 执行LIMIT子句

LIMIT子句从上一步得到的VT8虚拟表中选出从指定位置开始的指定行数据。对于没有应用ORDER BY的LIMIT子句，得到的结果同样是无序的，所以，很多时候，我们都会看到LIMIT子句会和ORDER BY子句一起使用。

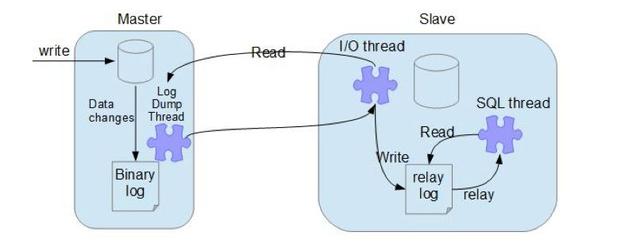
## MySQL主从复制原理

MySQL 主从复制是指数据可以从一个MySQL数据库服务器主节点复制到一个或多个从节点。MySQL 默认采用异步复制方式，这样从节点不用一直访问主服务器来更新自己的数据，数据的更新可以在远程连接上进行，从节点可以复制主数据库中的所有数据库或者特定的数据库，或者特定的表。

MySQL 主从复制主要用途

* 读写分离：在开发工作中，有时候会遇见某个sql 语句需要锁表，导致暂时不能使用读的服务，这样就会影响现有业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运作。
* 数据实时备份，当系统中某个节点发生故障时，可以方便的故障切换
* 高可用HA
* 架构扩展：随着系统中业务访问量的增大，如果是单机部署数据库，就会导致I/O访问频率过高。有了主从复制，增加多个数据存储节点，将负载分布在多个从节点上，降低单机磁盘I/O访问的频率，提高单个机器的I/O性能。

MySQL主从复制涉及到三个线程，一个运行在主节点（log dump thread），其余两个(I/O thread, SQL thread)运行在从节点，如下图所示:



* 主节点 binary log dump 线程

当从节点连接主节点时，主节点会创建一个log dump 线程，用于发送bin-log的内容。在读取bin-log中的操作时，此线程会对主节点上的bin-log加锁，当读取完成，甚至在发动给从节点之前，锁会被释放。

* 从节点I/O线程

当从节点上执行`start slave`命令之后，从节点会创建一个I/O线程用来连接主节点，请求主库中更新的bin-log。I/O线程接收到主节点binlog dump 进程发来的更新之后，保存在本地relay-log中。

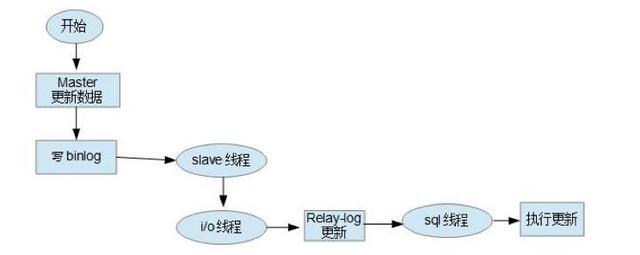
* 从节点SQL线程

SQL线程负责读取relay log中的内容，解析成具体的操作并执行，最终保证主从数据的一致性。

对于每一个主从连接，都需要三个进程来完成。当主节点有多个从节点时，主节点会为每一个当前连接的从节点建一个binary log dump 进程，而每个从节点都有自己的I/O进程，SQL进程。从节点用两个线程将从主库拉取更新和执行分成独立的任务，这样在执行同步数据任务的时候，不会降低读操作的性能。比如，如果从节点没有运行，此时I/O进程可以很快从主节点获取更新，尽管SQL进程还没有执行。如果在SQL进程执行之前从节点服务停止，至少I/O进程已经从主节点拉取到了最新的变更并且保存在本地relay日志中，当服务再次起来之后，就可以完成数据的同步。

要实施复制，首先必须打开Master 端的binary log（bin-log）功能，否则无法实现。

因为整个复制过程实际上就是Slave 从Master 端获取该日志然后再在自己身上完全顺序的执行日志中所记录的各种操作。如下图所示：



复制的基本过程如下：

1. 从节点上的I/O 进程连接主节点，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；
2. 主节点接收到来自从节点的I/O请求后，通过负责复制的I/O进程根据请求信息读取指定日志指定位置之后的日志信息，返回给从节点。返回信息中除了日志所包含的信息之外，还包括本次返回的信息的bin-log file 的以及bin-log position；从节点的I/O进程接收到内容后，将接收到的日志内容更新到本机的relay log中，并将读取到的binary log文件名和位置保存到master-info 文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log 的哪个位置开始往后的日志内容，请发给我”；
3. Slave 的 SQL线程检测到relay-log 中新增加了内容后，会将relay-log的内容解析成在祝节点上实际执行过的操作，并在本数据库中执行。

# Redis

## Redis的使用场景是什么？

* 缓存

缓存现在几乎是所有中大型网站都在用的必杀技，合理的利用缓存不仅能够提升网站访问速度，还能大大降低数据库的压力。Redis提供了键过期功能，也提供了灵活的键淘汰策略，所以，现在Redis用在缓存的场合非常多。

* 排行榜

很多网站都有排行榜应用的，如京东的月度销量榜单、商品按时间的上新排行榜等。Redis提供的有序集合数据类构能实现各种复杂的排行榜应用。

* 计数器

什么是计数器，如电商网站商品的浏览量、视频网站视频的播放数等。为了保证数据实时效，每次浏览都得给+1，并发量高时如果每次都请求数据库操作无疑是种挑战和压力。Redis提供的incr命令来实现计数器功能，内存操作，性能非常好，非常适用于这些计数场景。

* 分布式会话

集群模式下，在应用不多的情况下一般使用容器自带的session复制功能就能满足，当应用增多相对复杂的系统中，一般都会搭建以Redis等内存数据库为中心的session服务，session不再由容器管理，而是由session服务及内存数据库管理。

* 分布式锁

在很多互联网公司中都使用了分布式技术，分布式技术带来的技术挑战是对同一个资源的并发访问，如全局ID、减库存、秒杀等场景，并发量不大的场景可以使用数据库的悲观锁、乐观锁来实现，但在并发量高的场合中，利用数据库锁来控制资源的并发访问是不太理想的，大大影响了数据库的性能。可以利用Redis的setnx功能来编写分布式的锁，如果设置返回1说明获取锁成功，否则获取锁失败，实际应用中要考虑的细节要更多。

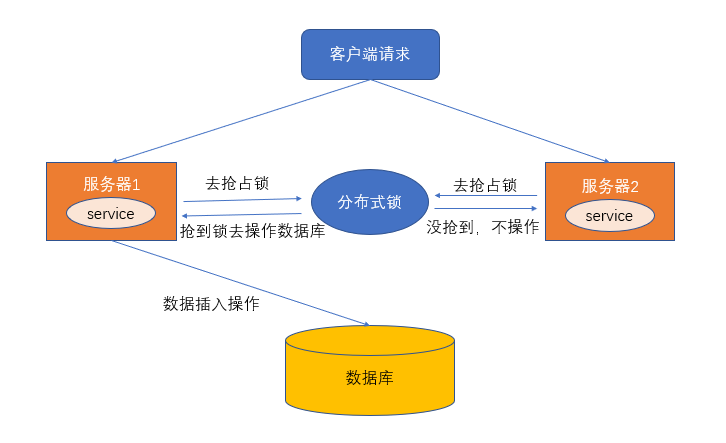
## 如何基于Redis实现分布式锁？

分布式锁概览

在多线程的环境下，为了保证一个代码块在同一时间只能由一个线程访问，Java中我们一般可以使用synchronized语法和ReetrantLock去保证，这实际上是本地锁的方式。但是现在公司都是流行分布式架构，在分布式环境下，如何保证不同节点的线程同步执行呢？

实际上，对于分布式场景，我们可以使用分布式锁，它是控制分布式系统进程之间互斥访问共享资源的一种方式。

比如说在一个分布式系统中，多台机器上部署了多个服务，当客户端一个用户发起一个数据插入请求时，如果没有分布式锁机制保证，那么那多台机器上的多个服务可能进行并发插入操作，导致数据重复插入，对于某些不允许有多余数据的业务来说，这就会造成问题。而分布式锁机制就是为了解决类似这类问题，保证多个服务之间互斥的访问共享资源，如果一个服务抢占了分布式锁，其他服务没获取到锁，就不进行后续操作。大致意思如下图所示



分布式锁的特点

分布式锁一般有如下的特点：

* 互斥性： 同一时刻只能有一个线程持有锁
* 可重入性： 同一节点上的同一个线程如果获取了锁之后能够再次获取锁
* 锁超时：和J.U.C中的锁一样支持锁超时，防止死锁
* 高性能和高可用： 加锁和解锁需要高效，同时也需要保证高可用，防止分布式锁失效
* 具备阻塞和非阻塞性：能够及时从阻塞状态中被唤醒

分布式锁的实现方式

我们一般实现分布式锁有以下几种方式：

* 基于 DB 的唯一索引。
* 基于 ZK 的临时有序节点。
* 基于 Redis 的 NX EX 参数。





Redisson分布式锁的实现

Redisson 分布式重入锁用法。Redisson 支持单点模式、主从模式、哨兵模式、集群模式，这里以单点模式为例：

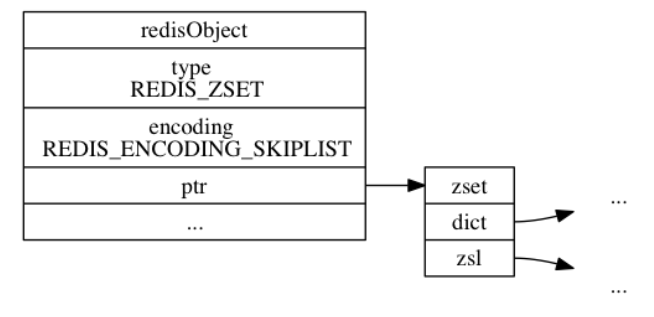


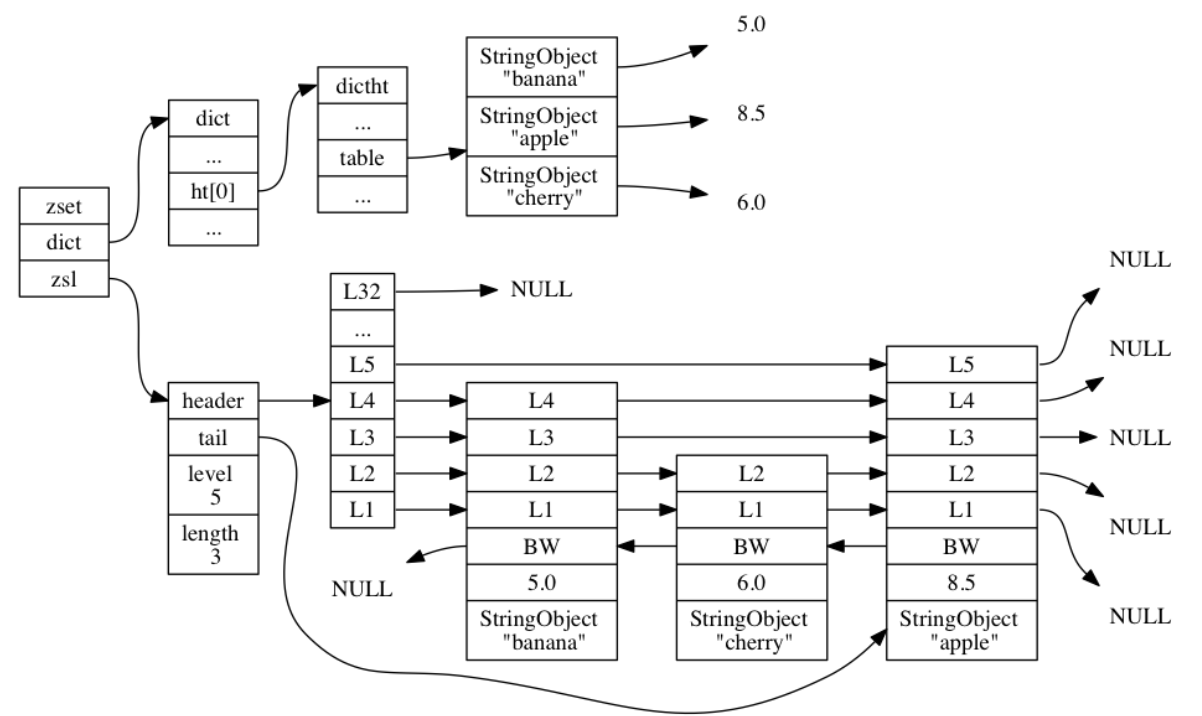
## ‌ zset，跳跃表，为什么不用红黑树，跳跃表实现讲了一下。

因为跳跃表的实现比平衡树更简单，并且效率可以跟平衡树相媲美。

## ‌zset底层实现

Zset是一个复合的数据结构，一方面需要一个hash结构来存储value与score的对应关系，另一方面需要提供按照score排序的功能，还需能通过指定score的范围来获取value列表的功能。Zset的内部实现是通过一个hash字典加一个跳跃列表。

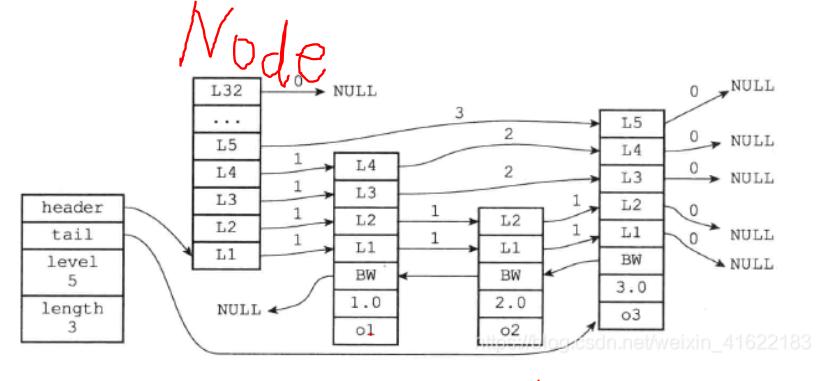




Hash表与跳跃表是会共享元素的成员值域分值，因此不会造成任何的数据重复，也不会造成空间浪费。

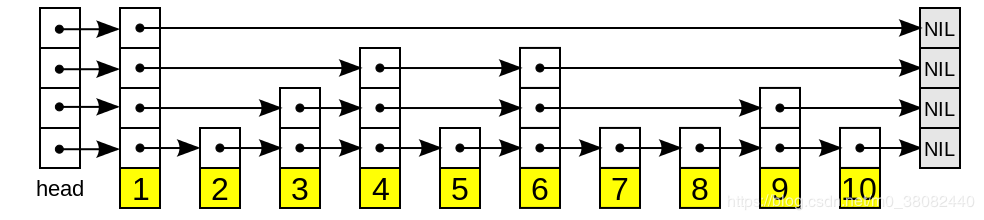
Zset的内部的排序功能是通过“跳跃列表”数据结构来实现的。zset需要进行随机插入与删除，因此它不适合使用数据来表示。zset需要进行随机查找，它也不适合采用传统链表来存储。数组无法进行随机插入删除，传统链表无法进行随机查找。

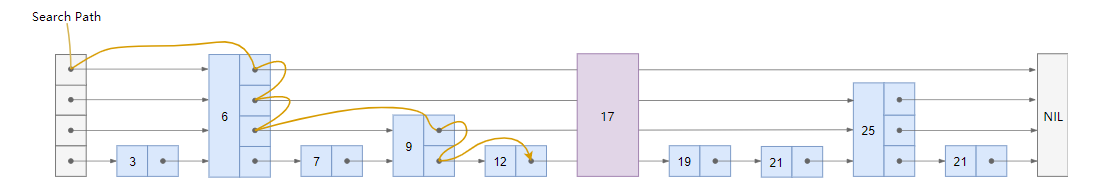
**跳跃表时一种有序的数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快点访问节点的目的。跳跃表的效率可以和平衡树想媲美了，最关键是它的实现相对于平衡树来说，代码的实现上简单很多。**平均O(longN),最坏O(N)时间复杂度的节点查找，还可以通过顺序性操作来批处理节点。



Redis的跳跃表实现由zsliplist与zskiplistNode两个结构组成，前者用来保存跳跃表信息(表头节点，表尾节点，长度，层高)，后者用来表示跳跃表的节点。

原理就是对有序的链表增加上附加的前进链接，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分列表，因此得名。所有操作都以对数随机化的时间进行。





随机层数：对于每一个新插入的节点，都需要调用一个随机算法来给它分配一个合理的层数。

时间复杂度：跳跃表可以实现二分的查找。O(logN)，最坏的是所有节点的值相等，时间负责度为0(n).

空间负责度：假如原始链表包含 n 个元素，则一级索引元素个数为 n/2、二级索引元素个数为 n/4、三级索引元素个数为 n/8 以此类推。所以，索引节点的总和是：n/2 + n/4 + n/8 + … + 8 + 4 + 2 = n-2，空间复杂度是 O(n)。

插入数据：插入数据的复杂度主要是在查找插入位置，查找的过程是O(logN).

删除操作：也是查找O(logN).

Redis中跳跃表的应用：实现有序集合键，在集群节点中用作内部数据结构。

## ‌为什么zset要用跳跃表和哈希表？

使用跳跃表可以在Log(N)的时间负责度内进行范围查找。

使用哈希表可以在O(1)的时间负责度内进行给定值查找。

单独使用跳跃表时，查找给定值的时间负责度是O(logN).单独使用哈希表进行范围查找时，会使用O(n)的空间进行存放对象，再使用O（Nlog(N)）的复杂度进行排序。

同时使用跳跃表跟哈希表一种空间换时间的策略。既可以快速进行区间查找，又可以O(1)进行定值查找。

## 删除策略？

过期删除策略

通常删除某个key，我们有如下三种方式进行处理。

①、定时删除

　　在设置某个key 的过期时间同时，我们创建一个定时器，让定时器在该过期时间到来时，立即执行对其进行删除的操作。

　　优点：定时删除对内存是最友好的，能够保存内存的key一旦过期就能立即从内存中删除。

　　缺点：对CPU最不友好，在过期键比较多的时候，删除过期键会占用一部分 CPU 时间，对服务器的响应时间和吞吐量造成影响。

②、惰性删除

　　设置该key 过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key。

　　优点：对 CPU友好，我们只会在使用该键时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查。

　　缺点：对内存不友好，如果一个键已经过期，但是一直没有使用，那么该键就会一直存在内存中，如果数据库中有很多这种使用不到的过期键，这些键便永远不会被删除，内存永远不会释放。从而造成内存泄漏。

③、定期删除

　　每隔一段时间，我们就对一些key进行检查，删除里面过期的key。

　　优点：可以通过限制删除操作执行的时长和频率来减少删除操作对 CPU 的影响。另外定期删除，也能有效释放过期键占用的内存。

　　缺点：难以确定删除操作执行的时长和频率。

　　　　　如果执行的太频繁，定期删除策略变得和定时删除策略一样，对CPU不友好。

　　　　　如果执行的太少，那又和惰性删除一样了，过期键占用的内存不会及时得到释放。

　　　　　另外最重要的是，在获取某个键时，如果某个键的过期时间已经到了，但是还没执行定期删除，那么就会返回这个键的值，这是业务不能忍受的错误。

## 内存淘汰策略

过期键淘汰：

* volatile-lru：从设置过期时间的数据集(server.db[i].expires)中挑选出最近最少使用的数据淘汰。没有设置过期时间的key不会被淘汰，这样就可以在增加内存空间的同时保证需要持久化的数据不会丢失。
* volatile-ttl：除了淘汰机制采用LRU，策略基本上与volatile-lru相似，从设置过期时间的数据集(server.db[i].expires)中挑选将要过期的数据淘汰，ttl值越大越优先被淘汰。
* volatile-random：从已设置过期时间的数据集(server.db[i].expires)中任意选择数据淘汰。当内存达到限制无法写入非过期时间的数据集时，可以通过该淘汰策略在主键空间中随机移除某个key。

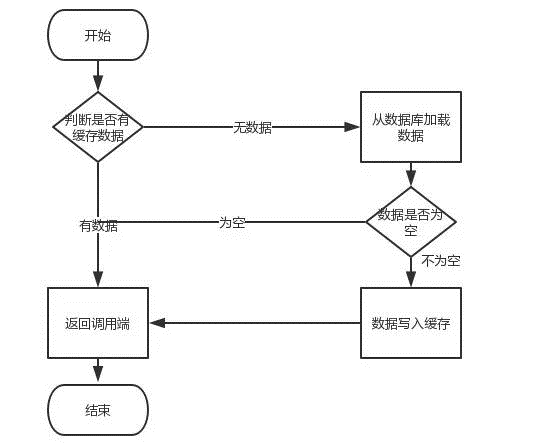
全局键淘汰

* allkeys-lru：从数据集(server.db[i].dict)中挑选最近最少使用的数据淘汰，该策略要淘汰的key面向的是全体key集合，而非过期的key集合。
* allkeys-random：从数据集(server.db[i].dict)中选择任意数据淘汰。

报错

* no-enviction：禁止驱逐数据，也就是当内存不足以容纳新入数据时，新写入操作就会报错，请求可以继续进行，线上任务也不能持续进行，采用no-enviction策略可以保证数据不被丢失，这也是系统默认的一种淘汰策略。

## ‌缓存更新机制是怎么设计的？说一下三种常用的缓存读写机制。缓存数据库双写



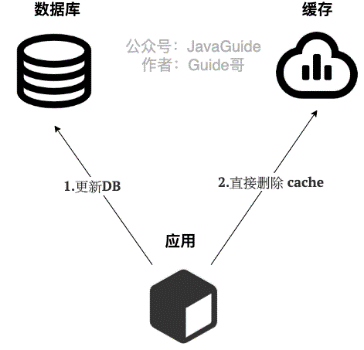
* Cache Aside Pattern（旁路缓存模式）先写数据库，然后删除缓存

Cache Aside Pattern 是我们平时使用比较多的一个缓存读写模式，比较适合读请求比较多的场景。Cache Aside Pattern 中服务端需要同时维系 DB 和 cache，并且是以 DB 的结果为准。

写 ：

先更新 DB

然后直接删除 cache 。

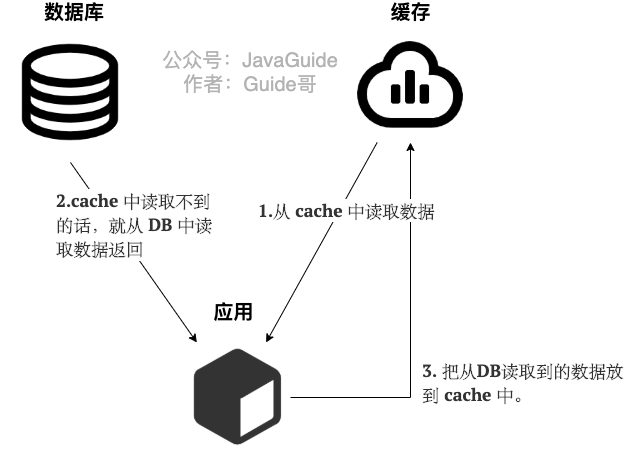


读 :

从 cache 中读取数据，读取到就直接返回

cache中读取不到的话，就从 DB 中读取数据返回

再把数据放到 cache 中。



“在写数据的过程中，可以先删除 cache ，后更新 DB 么？”

答：那肯定是不行的！因为这样可能会造成数据库（DB）和缓存（Cache）数据不一致的问题。为什么呢？比如说请求1 先写数据A，请求2随后读数据A的话就很有可能产生数据不一致性的问题。这个过程可以简单描述为：请求1先把cache中的A数据删除 -> 请求2从DB中读取数据->请求1再把DB中的A数据更新。

**Cache Aside Pattern 的缺陷。**

缺陷1：首次请求数据一定不在 cache 的问题

解决办法：可以将热点数据可以提前放入cache 中。

缺陷2：写操作比较频繁的话导致cache中的数据会被频繁被删除，这样会影响缓存命中率 。

解决办法：数据库和缓存数据强一致场景 ：更新DB的时候同样更新cache，不过我们需要加一个锁/分布式锁来保证更新cache的时候不存在线程安全问题。

可以短暂地允许数据库和缓存数据不一致的场景 ：更新DB的时候同样更新cache，但是给缓存加一个比较短的过期时间，这样的话就可以保证即使数据不一致的话影响也比较小。

* Read/Write Through Pattern（读写穿透） 先写缓存再写数据库

Read/Write Through Pattern 中服务端把 cache 视为主要数据存储，从中读取数据并将数据写入其中。cache 服务负责将此数据读取和写入 DB，从而减轻了应用程序的职责。

这种缓存读写策略小伙伴们应该也发现了在平时在开发过程中非常少见。抛去性能方面的影响，大概率是因为我们经常使用的分布式缓存 Redis 并没有提供 cache 将数据写入DB的功能。

写（Write Through）：

先查 cache，cache 中不存在，直接更新 DB。

cache 中存在，则先更新 cache，然后 cache 服务自己更新 DB（同步更新 cache 和 DB）。

读(Read Through)：

从 cache 中读取数据，读取到就直接返回 。

读取不到的话，先从 DB 加载，写入到 cache 后返回响应。

* Write Behind Pattern（异步缓存写入） 先写缓存，固定时间再写数据库。

Write Behind Pattern 和 Read/Write Through Pattern 很相似，两者都是由 cache 服务来负责 cache 和 DB 的读写。

但是，两个又有很大的不同：Read/Write Through 是同步更新 cache 和 DB，而 Write Behind Caching 则是只更新缓存，不直接更新 DB，而是改为异步批量的方式来更新 DB。

很明显，这种方式对数据一致性带来了更大的挑战，比如cache数据可能还没异步更新DB的话，cache服务可能就就挂掉了。

这种策略在我们平时开发过程中也非常非常少见，但是不代表它的应用场景少，比如消息队列中消息的异步写入磁盘、MySQL 的 InnoDB Buffer Pool 机制都用到了这种策略。

Write Behind Pattern 下 DB 的写性能非常高，非常适合一些数据经常变化又对数据一致性要求没那么高的场景，比如浏览量、点赞量。

## 如何保证redis和数据库的数据一致性？

回答缓存更新操作即可

客户端对数据库中的数据主要有两类操作，读(select)与写(DML)。针对放入redis中缓存的热点数据，当客户端想读取的数据在缓存中就直接返回数据，即命中缓存(cache hit)，当读取的数据不在缓存内，就需要从数据库中将数据读入缓存，即未命中缓存(cache miss)。所以读操作并不会导致缓存与数据库中的数据不一致。

  对于写操作(DML)，缓存与数据库中的内容都需要被修改，但两者的执行必定存在一个先后顺序，这可能会导致缓冲与数据库中的数据不再一致，此时主要需要考虑两个问题：

  1、执行顺序的问题：先更新缓存还是先更新数据库？

  2、更新缓存的策略问题：当缓存中的内容变化时，是选择修改缓存(update)，还是直接淘汰缓存(delete)？

针对这两点问题，一共可以分为四种方案：

  1、先更新缓存，再更新数据库；

  2、先更新数据库，再更新缓存；

  3、先淘汰缓存，再更新数据库；

  4、先更新数据库，再淘汰缓存。

<https://developer.aliyun.com/article/712285>

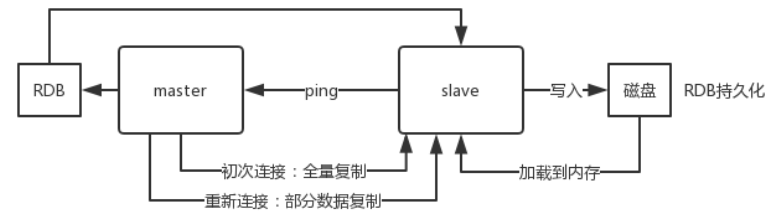
## Redis 主从复制

主从复制是指新启动一个Slave节点，master将缓存数据复制到slave的过程

具体过程：

1. 当启动一个slave node时，slave node 会发送一个PSYNC 命令给master 节点
2. 如果这是slave节点第一次连接到master节点，那么会触发一下full resynchronization 全员复制，此时master节点会fork()启动一个子线程生成一份RDB快照文件，不会阻塞master节点。For()使用了写时复制原理。同时还将客户端收到的写命令写到内存中，RDB文件生成后，master节点将RDB文件发送给slave。
3. Slave节点接收时首先写入磁盘，然后再从磁盘加载RDB文件
4. 接着master节点会将内存中缓存的写命令发给slave节点，然后slave同步这些写命令。

如果slave节点与master节点有网络断链，连接之后master仅会复制给slave缺少的文件。



过程原理：

1. 当从库和主库建立MS关系后，会向主数据库发送SYNC命令
2. 主库接收到SYNC命令后会开始在后台保存快照(RDB持久化过程)，并将期间接收到的写命令缓存起来
3. 当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从Redis
4. 从Redis接收到后，会载入快照文件并且执行收到的缓存的命令
5. 之后，主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致

## Redis的网络模型（IO模型，文件事件）

文件事件：Redis服务器通过套接字与客服端进行连接，而文件事件就是服务器对套接字操作的抽象，服务器与客户端的通信会产生相应的文件事件，而服务器则通过监听并处理这些事件来完成网络通信操作。

Redis基于Reactor模式开发了自己的网络事件处理器，这个处理器被称为文件事件处理器。Redis线程模型包含四个内容：多个Socket，IO多路复用模型，文件事件分派器，事件处理器。



## Redis为什么使用单线程都这么快？

## Redis下的写时复制

<https://juejin.cn/post/6844903702373859335>

Redis在持久化时，如果是采用BGSAVE命令或者BGREWRITEAOF的方式，那Redis会fork出一个子进程来读取数据，从而写到磁盘中。

总体来看，Redis还是读操作比较多。如果子进程存在期间，发生了大量的写操作，那可能就会出现很多的分页错误(页异常中断page-fault，fork()之后，kernel把父进程中所有的内存页的权限都设为read-only)，这样就得耗费不少性能在复制上。而在rehash阶段上，写操作是无法避免的。所以Redis在fork出子进程之后，将负载因子阈值提高，尽量减少写操作，避免不必要的内存写入操作，最大限度地节约内存。

AOF重写时也会发生写时复制。AOF重写时重新生成RDB文件，会创建一个子进程来负责，同时将RDB过程的写命令写入缓存中。

## Redis与Memcached的区别

那肯定是内存中。memcached和redis就是将数据存储在内存中，按照key-value的方式查询，可以大幅度提高效率。所以一般它们都用做缓存服务器，缓存常用的数据，需要查询的时候，直接从它们那儿获取，减少查询数据库的次数，提高查询效率。

但Redis与Memcached存在诸多不同。

* 数据存储类型

Redis支持多种数据类型，比如String，List，Hash，Set，Zset，而Memcached仅支持文本型数据与二进制数据。

* 查询操作

Redis支持批量操作，支持事务，每个类型不同的CRUD，Memcached仅支持少量的CUED。

* 网络IO模型

Redis使用单线程的IO多路复用模型，自己封装了一个简单的AeEvent事件处理框架，主要实现了epoll, kqueue和select，对于单存只有IO操作来说，单线程可以将速度优势发挥到最大

Memcached使用多线程，非阻塞IO模型。分为监听主线程和worker子线程，监听线程监听网络连接，接受请求后，将连接描述字pipe传递给worker线程，进行读写IO，网络层使用libevent封装的事件库，多线程模型可以发挥多核作用

* 数据一致性问题

Memcached提供了cas命令，可以保证多个并发访问操作同一份数据的一致性问题。 Redis没有提供cas 命令，并不能保证这点，不过Redis提供了事务的功能，可以保证一串 命令的原子性，中间不会被任何操作打断。

* 持久化支持

Redis支持RDB与AOF两种持久化方式，可以将内存中的数据备份到磁盘中，实现数据的永久保存。

* 集群模式

Redis支持多种集群，可实现主从模式，哨兵模式，Cluster模式，Memcached不支持集群。

# 操作系统与linux

## 说一说操作系统中的堆栈

堆和栈

同：两者在操作系统中都是为了存储数据而存在的。

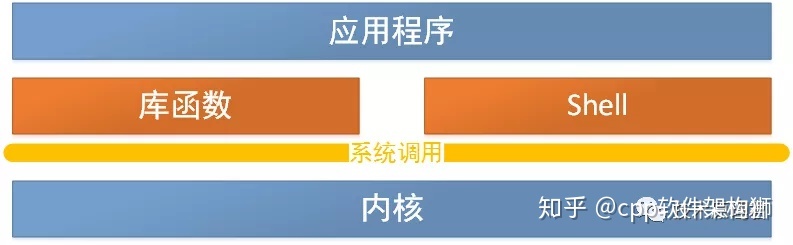
异：1.出现错误的情况：栈由编译器自动分配释放，最大的程序问题会出现栈溢出。堆由程序员手动管理稍有不当就会产生内存泄漏还有各种segment fault。

2.效率：栈申请速度快，堆要经过算法筛选比较慢。操作系统有一个记录当前空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，并且寻找第一个大于当前申请的堆空间（这取决于采取何种算法-首次适应算法，循环首次适应算法，最佳适应算法，最坏适应算法，快速适应算法），然后重新建立新的空闲链表，将这个节点分配。另一个堆慢的原因就是比如int \*p = &a;他是要先在p里面找到这个指针的地址然后再到这个地址指向的空间寻找所需的数据，多了一个步骤固然变慢了。

3.内容：栈一般放当前所需要的数据，堆中随时存储。

我们常说堆栈堆栈，其实堆栈是两个不同的概念，最直观的理解，堆是由用户来控制的，我们可以使用 malloc 这种命令来在堆中申请内存，而栈是由操作系统控制的，在栈中存储的是这个进程的局部变量等，比如我们用 malloc 来申请一块内存，内存本身是在堆中开辟的，而指向这块内存的指针存储在栈中。

## 内核态与用户态



从本质上说就是我们所说的内核，它是一种特殊的软件程序，特殊在哪儿呢？控制计算机的硬件资源，例如协调CPU资源，分配内存资源，并且提供稳定的环境供应用程序运行。

用户态就是提供应用程序运行的空间，为了使应用程序访问到内核管理的资源例如CPU，内存，I/O。内核必须提供一组通用的访问接口，这些接口就叫系统调用。

**为什么要区分内核态与用户态？**

因为在CPU的指令中，有一些是非常危险的，比如清理内存、设置时钟等，如果所有的程序都能使用，就可能造成系统的崩溃，所以，CPU 将指令分为特权指令和非特权指令，对于那些危险的指令，只允许操作系统使用。CPU 的特权级别有四级，从 Ring0 到 Ring3，正常使用时一般只有两级，即用户态的 Ring3 和内核态的 Ring0。Ring3 状态不能访问 Ring0 的地址空间，包括代码和数据。

**用户态切换到内核态的三种方式**

* 系统调用，这个上面已经讲解过了，其实系统调用本身就是中断，但是软件中断，跟硬中断不同。
* 异常：如果当前进程运行在用户态，如果这个时候发生了异常事件，就会触发切换。例如：缺页异常。
* 外设中断：当外设完成用户的请求时，会向CPU发送中断信号。

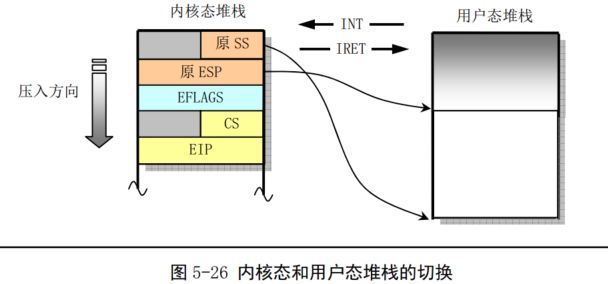
**进程的堆栈**

 内核在创建进程的时候，会为进程创建相应的堆栈。每个进程会有两个栈，一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。当进程在用户空间运行时，CPU堆栈指针寄存器里面的内容是用户堆栈地址，使用用户栈；当进程在内核空间时，CPU堆栈指针寄存器里面的内容是内核栈空间地址，使用内核栈。

**用户态切换到内核态的过程？**

1. 从当前进程的描述符中提取其内核栈的ss0及esp0信息。
2. 使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。
3. 将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。
4. 内核态程序执行完毕时如果要从内核态返回用户态，可以通过执行指令iret来完成，指令iret会将先前压栈的进入内核态前的cs,eip,eflags,ss,esp信息从栈里弹出，加载到各个对应的寄存器中，重新开始执行用户态的程序.

在Linux0.11系统中，所有中断服务程序都属于内核代码。若中断时，进程在执行用户态的代码，该中断会引起CPU特权级从3级到0级的切换，此时CPU会进行堆栈的切换，CPU会从当前任务的TSS中取到新堆栈的段选择符和偏移值；CPU首先会把原用户态的堆栈指针ss和esp压入内核态堆栈，随后把标志寄存器eflags的内容和此次中断的返回位置cs，eip压入内核态堆栈。当中断处理函数结束后，将恢复内核栈中的数据，并继续处理被中断的进程。若中断时，进程正在执行内核态的代码，则不需要堆栈的切换，CPU仅把eflags的内容和此次中断的返回位置cs，eip压入内核态堆栈，然后执行中断服务程序。



## 五种IO模式

BIO

NIO

IO多路复用

信号驱动IO

AIO

## BIO与NIO的区别

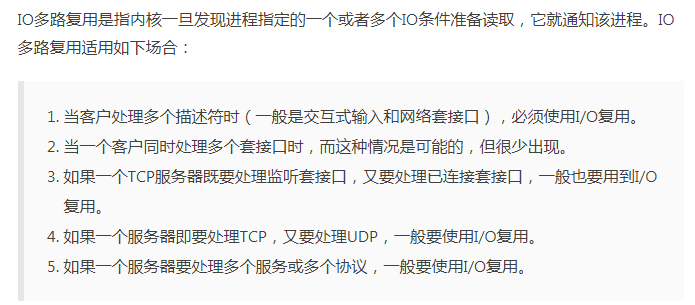
* BIO是以流的方式处理数据，NIO是以块的方式处理数据，面向缓冲区的，块的速度比流的速度快很多
* BIO是阻塞的，NIO是非阻塞的
* BIO基于字节流和字符流进行操作，而NO基于 Channel（通道）和 Buffer（缓冲区）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。 Selector（选择器）庯于监听多个通道的事件（比如：连接请求，数据到达等），因此使用单个线程就可以监听多个客户端通道

面向缓冲区有什么好处？

Java NIO和BIO之间第一个最大的区别是，BIO是面向流的，NIO是面向缓冲区的。 Java BIO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

## ‌select/poll/epoll的区别？

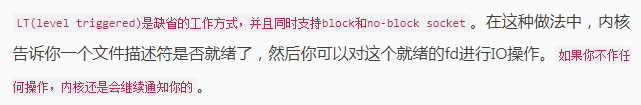
select，poll，epoll都是IO多路复用的机制。I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

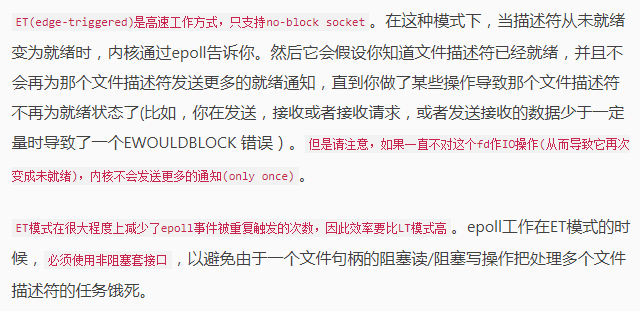


## epoll水平和边缘触发的区别，编写代码时有什么区别？边缘触发为什么高效？

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：





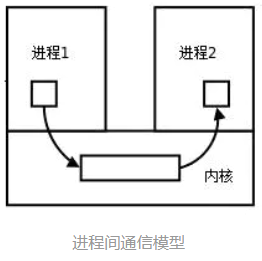


LT可以理解为水平触发，只要有数据可以读，不管怎样都会通知。而ET为边缘触发，只有状态发生变化时才会通知，可以理解为电平变化。



## 进程之间的通信与线程之间的通信？

**进程之间通信：**

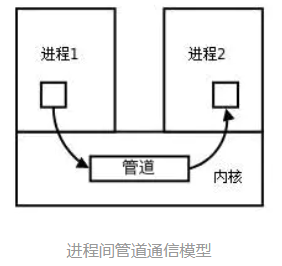


1. 无名管道：半双工的、只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）、只存在于内存中的文件。

管道的实质是一个内核缓冲区，进程以先进先出的方式从缓冲区存取数据，管道一端的进程顺序的将数据写入缓冲区，另一端的进程则顺序的读出数据。

该缓冲区可以看做是一个循环队列，读和写的位置都是自动增长的，不能随意改变，一个数据只能被读一次，读出来以后在缓冲区就不复存在了。

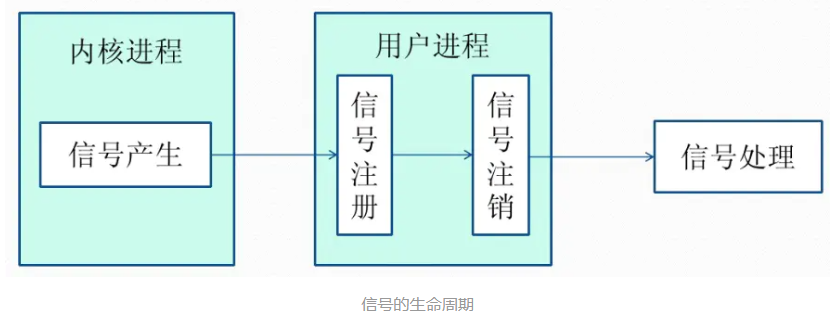
当缓冲区读空或者写满时，有一定的规则控制相应的读进程或者写进程进入等待队列，当空的缓冲区有新数据写入或者满的缓冲区有数据读出来时，就唤醒等待队列中的进程继续读写。



1. 有名管道：遵循先进先出(first in first out)。命名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。有名管道不同于匿名管道之处在于它提供了一个路径名与之关联，以有名管道的文件形式存在于文件系统中，这样，即使与有名管道的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过有名管道相互通信，因此，通过有名管道不相关的进程也能交换数据。值的注意的是，有名管道严格遵循先进先出(first in first out),对匿名管道及有名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如lseek()等文件定位操作。有名管道的名字存在于文件系统中，内容存放在内存中。
2. 消息队列 ：消息的链接表，存放在内核中，消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。
   * 消息队列是存放在内核中的消息链表，每个消息队列由消息队列标识符表示。
   * 与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列存放在内核中，只有在内核重启(即，操作系统重启)或者显示地删除一个消息队列时，该消息队列才会被真正的删除。
   * 另外与管道不同的是，消息队列在某个进程往一个队列写入消息之前，并不需要另外某个进程在该队列上等待消息的到达。
3. 信号：信号是Linux系统中用于进程间互相通信或者操作的一种机制，信号可以在任何时候发给某一进程，而无需知道该进程的状态

信号生命周期和处理流程：

1. 信号被某个进程产生，并设置此信号传递的对象（一般为对应进程的pid），然后传递给操作系统；
2. 操作系统根据接收进程的设置（是否阻塞）而选择性的发送给接收者，如果接收者阻塞该信号（且该信号是可以阻塞的），操作系统将暂时保留该信号，而不传递，直到该进程解除了对此信号的阻塞（如果对应进程已经退出，则丢弃此信号），如果对应进程没有阻塞，操作系统将传递此信号。
3. 目的进程接收到此信号后，将根据当前进程对此信号设置的预处理方式，暂时终止当前代码的执行，保护上下文（主要包括临时寄存器数据，当前程序位置以及当前CPU的状态）、转而执行中断服务程序，执行完成后在回复到中断的位置。当然，对于抢占式内核，在中断返回时还将引发新的调度。

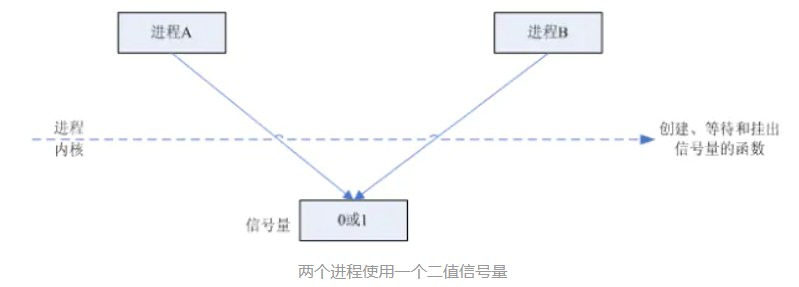


1. 信号量 ：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。

为了获得共享资源，进程需要执行下列操作：

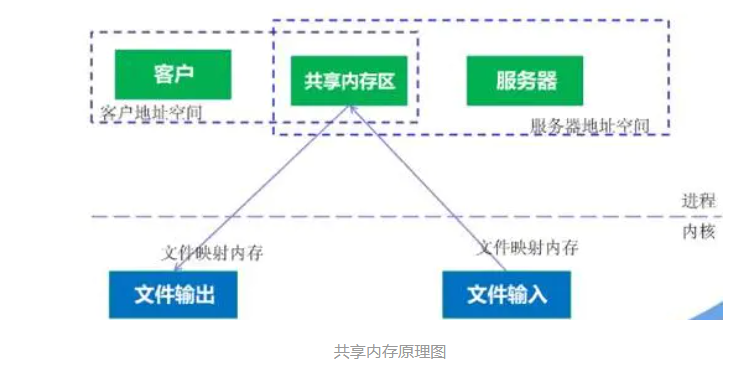
1. 创建一个信号量：这要求调用者指定初始值，对于二值信号量来说，它通常是1，也可是0。
2. 等待一个信号量：该操作会测试这个信号量的值，如果小于0，就阻塞。也称为P操作。
3. 挂出一个信号量：该操作将信号量的值加1，也称为V操作。

为了正确地实现信号量，信号量值的测试及减1操作应当是原子操作。

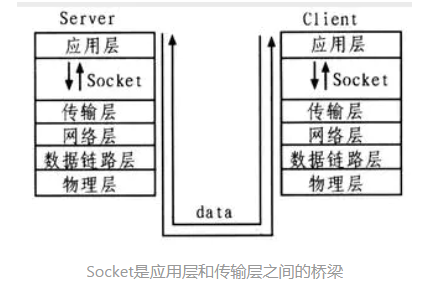


1. 共享内存： 使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。

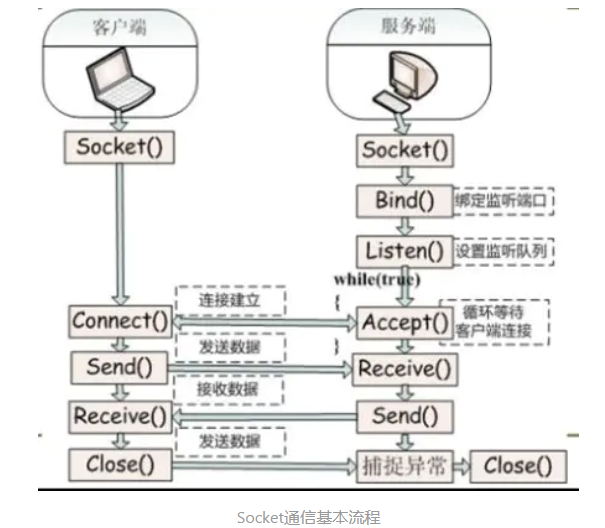
* 使得多个进程可以可以直接读写同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。
* 为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间。进程就可以直接读写这一块内存而不需要进行数据的拷贝，从而大大提高效率。
* 由于多个进程共享一段内存，因此需要依靠某种同步机制（如信号量）来达到进程间的同步及互斥。



1. 套接字：套接字是一种通信机制，凭借这种机制，客户/服务器（即要进行通信的进程）系统的开发工作既可以在本地单机上进行，也可以跨网络进行。也就是说它可以让不在同一台计算机但通过网络连接计算机上的进程进行通信。



套接字是支持TCP/IP的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。



**线程之间的通信：**

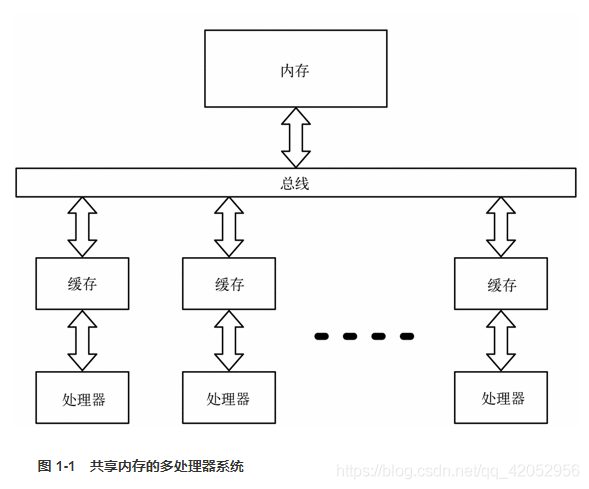
1. Volatile和sonchronized关键字
2. 等待/通知机制

* Notify()通知一个在对象上等待的线程
* Notifyall()通知所有线程
* Wait()释放锁，等另外的线程通知或者被中断才会返回

1. 管道输入/输出流

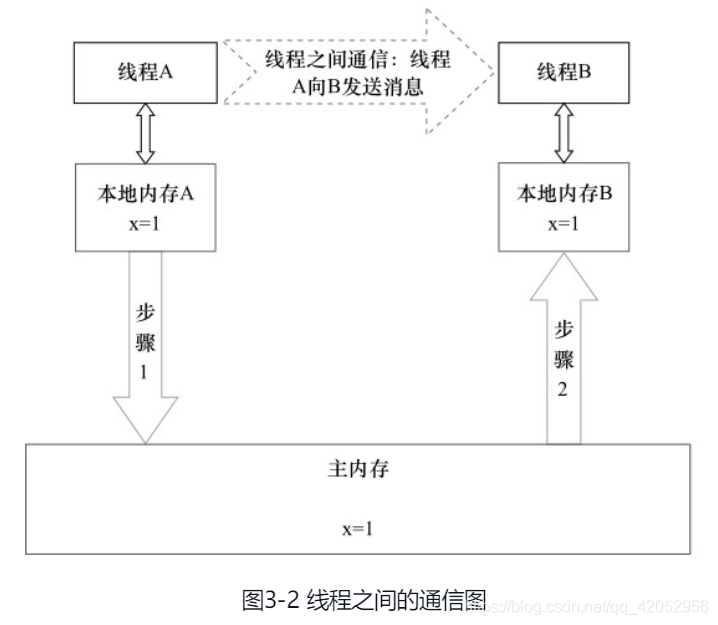
访问临界资源的顺序，从根本上来说，线程间的通信有两种方式：

* 一种是在共享内存的并发模型中，线程之间通过读-写共享内存来实现通信（公共状态）
* 一种是在消息传递的并发模型里，线程之间通过发送消息来进行通信（没有公共状态）



而Java的并发采用的是共享内存模型，所以Java线程之间的通信是基于共享内存实现的。具体来讲Java线程之间的通信由Java内存模型控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。如果线程A与B之间要通信的话，必须经历下面两个步骤：

1. 线程A把本地内存中更新过的共享变量刷新到主内存中去。
2. 线程B到住内存中去读取线程A之前更新过的共享变量。（如图）



现在问题来了，假设共享内存中的共享变量a，如果多个线程同时读a，不会出现任何问题，但是如果这些线程中有至少一个线程对a执行写操作，就可能出现数据不一致问题，也就是线程不安全了，那这是绝对不允许的。怎么办？答案就是依靠线程同步，来保证即使多个线程并发访问共享变量时，依然能够保证数据一致！

* Volatile
* Synchronized
* RentrantLock

线程之间的通信（锁 信号机制 信号量机制 线程之间通信主要是为了线程同步，所以线程没有像进程通信中的用于数据交换的通信机制）（锁 信号机制 信号量机制 线程之间通信主要是为了线程同步，所以线程没有像进程通信中的用于数据交换的通信机制）

## 僵尸进程是什么，危害和解决措施？

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了init进程身上，init进程就好像是一个民政局，专门负责处理孤儿进程的善后工作。每当出现一个孤儿进程的时候，内核就把孤 儿进程的父进程设置为init，而init进程会循环地wait()它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。

僵尸进程危害场景：

例如有个进程，它定期的产 生一个子进程，这个子进程需要做的事情很少，做完它该做的事情之后就退出了，因此这个子进程的生命周期很短，但是，父进程只管生成新的子进程，至于子进程 退出之后的事情，则一概不闻不问，这样，系统运行上一段时间之后，系统中就会存在很多的僵死进程，倘若用ps命令查看的话，就会看到很多状态为Z的进程。 严格地来说，僵死进程并不是问题的根源，罪魁祸首是产生出大量僵死进程的那个父进程。因此，当我们寻求如何消灭系统中大量的僵死进程时，答案就是把产生大 量僵死进程的那个元凶枪毙掉（也就是通过kill发送SIGTERM或者SIGKILL信号啦）。枪毙了元凶进程之后，它产生的僵死进程就变成了孤儿进 程，这些孤儿进程会被init进程接管，init进程会wait()这些孤儿进程，释放它们占用的系统进程表中的资源，这样，这些已经僵死的孤儿进程 就能瞑目而去了。

僵尸进程解决办法：

* 通过信号机制：子进程退出时向父进程发送SIGCHILD信号，父进程处理SIGCHILD信号。在信号处理函数中调用wait进行处理僵尸进程。
* fork两次：《Unix 环境高级编程》8.6节说的非常详细。原理是将子进程成为孤儿进程，从而其的父进程变为init进程，通过init进程可以处理僵尸进程。

## ctrl+c，发生了什么？

1. 用户输入命令，在Shell下启动一个前台进程。
2. 用户按下Ctrl-C，这个键盘输入产生一个硬件中断。
3. 如果CPU当前正在执行这个进程的代码，则该进程的用户空间代码暂停执行，CPU从用户态切换到内核态处理硬件中断。
4. 终端驱动程序将Ctrl-C解释成一个SIGINT信号，记在该进程的PCB中（也可以说发送了一个SIGINT信号给该进程）。
5. 当某个时刻要从内核返回到该进程的用户空间代码继续执行之前，首先处理PCB中记录的信号，发现有一个SIGINT信号待处理，而这个信号的默认处理动作是终止进程，所以直接终止进程而不再返回它的用户空间代码执行。

## linux写时复制

<https://juejin.cn/post/6844903702373859335>

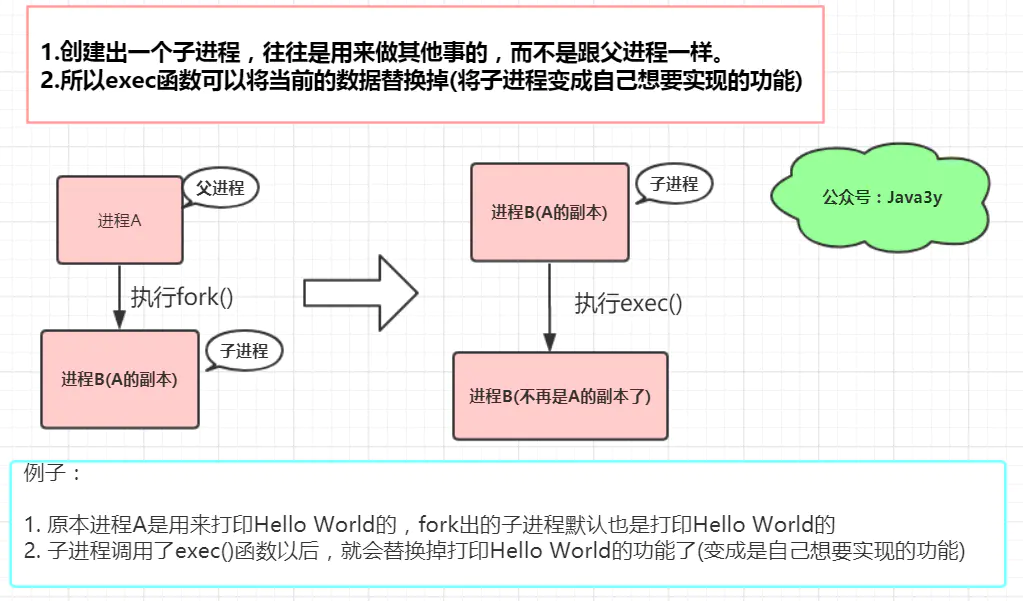
fork()函数：fork是类Unix操作系统上创建进程的主要方法。fork用于创建子进程(等同于当前进程的副本)。新的进程要通过老的进程复制自身得到，这就是fork！

fork作为一个函数被调用。这个函数会有两次返回，将子进程的PID返回给父进程，0返回给子进程。(如果小于0，则说明创建子进程失败)。

再次说明：当前进程调用fork()，会创建一个跟当前进程完全相同的子进程(除了pid)，所以子进程同样是会执行fork()之后的代码。

exec()函数：exec函数的作用就是：装载一个新的程序（可执行映像）覆盖当前进程内存空间中的映像，从而执行不同的任务。exec系列函数在执行时会直接替换掉当前进程的地址空间。

Fork()是创建一个父进程的副本，子进程功能跟父进程一模一样。



如果按传统的做法，会直接将父进程的数据拷贝到子进程中，拷贝完之后，父进程和子进程之间的数据段和堆栈是相互独立的。但是，以我们的使用经验来说：往往子进程都会执行exec()来做自己想要实现的功能。所以，如果按照上面的做法的话，创建子进程时复制过去的数据是没用的(因为子进程执行exec()，原有的数据会被清空).既然很多时候复制给子进程的数据是无效的，于是就有了Copy On Write这项技术了，原理也很简单：



Copy On Write技术实现原理：

fork()之后，kernel把父进程中所有的内存页的权限都设为read-only，然后子进程的地址空间指向父进程。当父子进程都只读内存时，相安无事。当其中某个进程写内存时，CPU硬件检测到内存页是read-only的，于是触发页异常中断（page-fault），陷入kernel的一个中断例程。中断例程中，kernel就会把触发的异常的页复制一份，于是父子进程各自持有独立的一份。

Copy On Write技术好处是什么？

* COW技术可减少分配和复制大量资源时带来的瞬间延时。
* COW技术可减少不必要的资源分配。比如fork进程时，并不是所有的页面都需要复制，父进程的代码段和只读数据段都不被允许修改，所以无需复制。

Copy On Write技术缺点是什么？

* 如果在fork()之后，父子进程都还需要继续进行写操作，那么会产生大量的分页错误(页异常中断page-fault)，这样就得不偿失。

几句话总结Linux的Copy On Write技术：

* fork出的子进程共享父进程的物理空间，当父子进程有内存写入操作时，read-only内存页发生中断，将触发的异常的内存页复制一份(其余的页还是共享父进程的)。
* fork出的子进程功能实现和父进程是一样的。如果有需要，我们会用exec()把当前进程映像替换成新的进程文件，完成自己想要实现的功能。

## ‌Linux中进程通信方式，不同主机之间传文件什么命令 (共享内存，管道，消息队列，scp http socket)

## ‌Linux下载文件命令 scp/ wget

## 你常用的Linux命令是什么？

* --help 指令的基本用法与选项介绍。
* Wget yum下载命令
* 命令行运行java程序



* 进程管理

查看进程

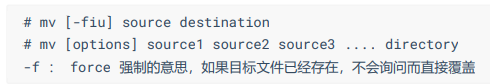


ps aux | grep threadx



* 文件与目录的基本操作ls列出文件或者目录的信息，目录的信息就是其中包含的文件。

mv 移动文件



Rm 删除文件 -r 递归删除

Cp 复制文件。如果源文件有两个以上，则目的文件一定要是目录才行

Rmdir 删除目录，目录必须为空。

 mkdir 创建目录。

Cd 更换当前目录。 修改文件权限：chmod

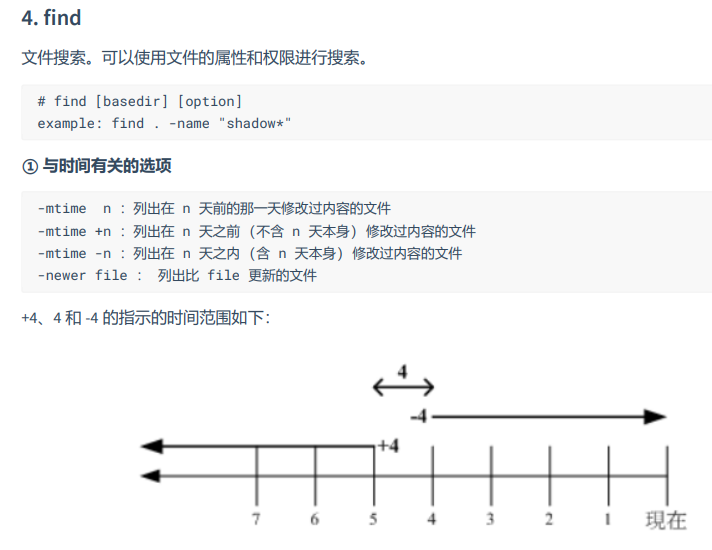
文件默认权限：文件默认没有可执行权限，因此为 666，也就是 -rw-rw-rw- 。

目录默认权限：目录必须要能够进入，也就是必须拥有可执行权限，因此为 777 ，也就是 drwxrwxrwx。 cat 获取文件内容

More 和 cat 不同的是它可以一页一页查看文件内容，比较适合大文件的查看。 

* 指令与文件搜索







## 说一下进程，线程，协程的区别？

进程与线程的区别：

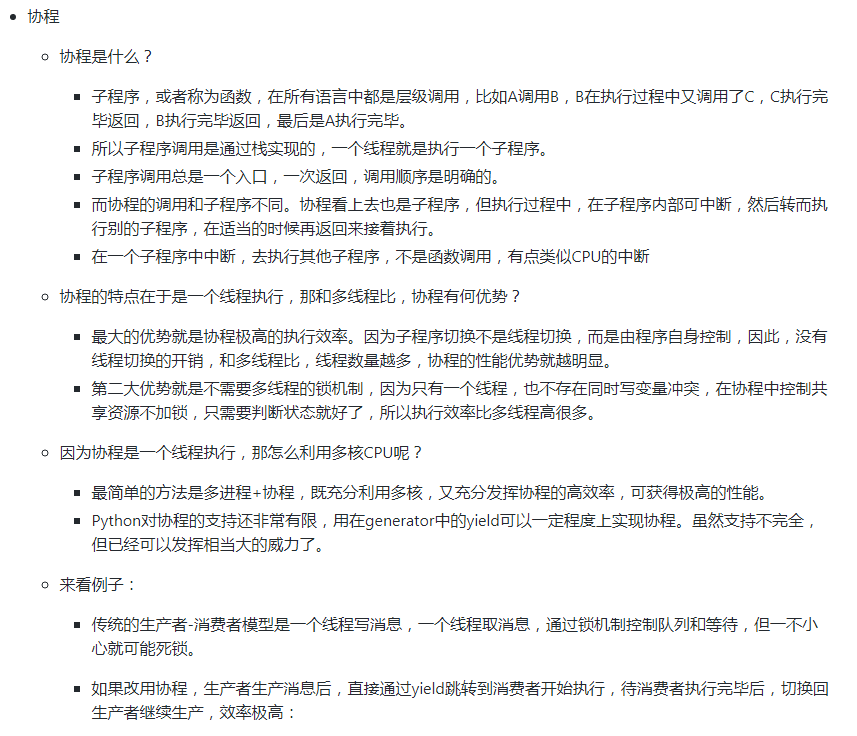
* 调度单位：进程是CPU资源分配的基本单位，线程是独立运行和独立调度的基本单位（CPU上真正运行的是线程）。
* 拥有资源：进程拥有自己的资源空间，一个进程包含若干个线程，线程与CPU资源分配无关，多个线程共享同一进程内的资源。
* 切换速度：线程的调度与切换比进程快很多。进程需要切换页表项，保持上下文信息，PCB，寄存器等，线程仅需要保持少量的寄存器与程序计数器即可。

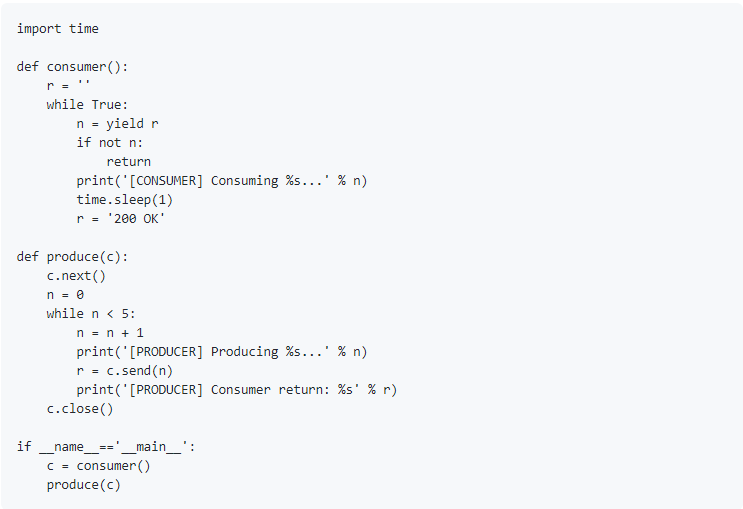
协程与线程的区别：

* 线程的切换由操作系统负责调度，协程由用户自己进行调度，因此减少了上下文切换，提高了效率。
* 线程的默认Stack大小是1M，而协程更轻量，接近1K。因此可以在相同的内存中开启更多的协程。
* 由于在同一个线程上，因此可以避免竞争关系而使用锁。
* 适用于被阻塞的，且需要大量并发的场景。但不适用于大量计算的多线程，遇到此种情况，更好实用线程去解决。



## 协程说一下？为什么会有协程？解决了什么问题

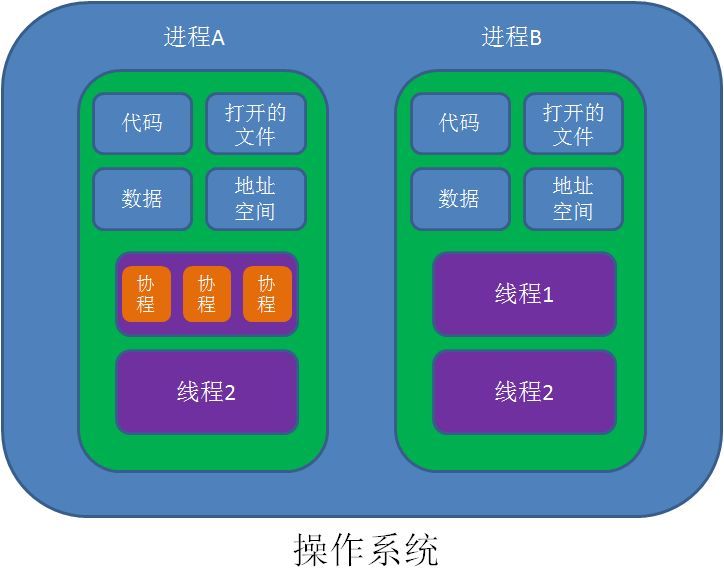






进程是操作系统资源分配的最小单位；线程为轻量级进程，是操作系统调度（CPU调度）执行的最小单位；协程是一种用户态的轻量级线程，一个线程可以拥有多个协程，协程调用完全由程序所控制，协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态。

协程，又称微线程，纤程。英文名Coroutine。一句话说明什么是线程：协程是一种用户态的轻量级线程。由程序员自己写程序来管理的轻量级线程叫做『用户空间线程』，具有对内核来说不可见的特性。



协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：

协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

协程的目的：在传统的J2EE系统中都是基于每个请求占用一个线程去完成完整的业务逻辑（包括事务）。所以系统的吞吐能力取决于每个线程的操作耗时。如果遇到很耗时的I/O行为，则整个系统的吞吐立刻下降，因为这个时候线程一直处于阻塞状态，如果线程很多的时候，会存在很多线程处于空闲状态（等待该线程执行完才能执行），造成了资源应用不彻底。

最常见的例子就是JDBC（它是同步阻塞的），这也是为什么很多人都说数据库是瓶颈的原因。这里的耗时其实是让CPU一直在等待I/O返回，说白了线程根本没有利用CPU去做运算，而是处于空转状态。而另外过多的线程，也会带来更多的ContextSwitch开销。

对于上述问题，现阶段行业里的比较流行的解决方案之一就是单线程加上异步回调。其代表派是node.js以及Java里的新秀Vert.x。协程的目的就是当出现长时间的I/O操作时，通过让出目前的协程调度，执行下一个任务的方式，来消除上下文切换上的开销。

协程的特点

* 线程的切换由操作系统负责调度，协程由用户自己进行调度，因此减少了上下文切换，提高了效率。
* 线程的默认Stack大小是1M，而协程更轻量，接近1K。因此可以在相同的内存中开启更多的协程。
* 由于在同一个线程上，因此可以避免竞争关系而使用锁。
* 适用于被阻塞的，且需要大量并发的场景。但不适用于大量计算的多线程，遇到此种情况，更好实用线程去解决。

协程的原理

当出现IO阻塞的时候，由协程的调度器进行调度，通过将数据流立刻yield掉（主动让出），并且记录当前栈上的数据，阻塞完后立刻再通过线程恢复栈，并把阻塞的结果放到这个线程上去跑，这样看上去好像跟写同步代码没有任何差别，这整个流程可以称为coroutine，而跑在由coroutine负责调度的线程称为Fiber。比如Golang里的 go关键字其实就是负责开启一个Fiber，让func逻辑跑在上面。由于协程的暂停完全由程序控制，发生在用户态上；而线程的阻塞状态是由操作系统内核来进行切换，发生在内核态上。因此，协程的开销远远小于线程的开销，也就没有了上下文切换上的开销。



协程的好处：

* 无需线程上下文切换的开销
* 无需原子操作锁定及同步的开销
* 方便切换控制流，简化编程模型

高并发+高扩展性+低成本：一个CPU支持上万的协程都不是问题。所以很适合用于高并发处理。

缺点：

* 无法利用多核资源：协程的本质是个单线程,它不能同时将 单个CPU 的多个核用上,协程需要和进程配合才能运行在多CPU上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是cpu密集型应用。
* 进行阻塞（Blocking）操作（如IO时）会阻塞掉整个程序

# 框架

## Spring

### Spring中用到了哪些设计模式，有什么好处？

<https://zhuanlan.zhihu.com/p/114244039>

* 简单工厂模式

实现方式：BeanFactory。Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

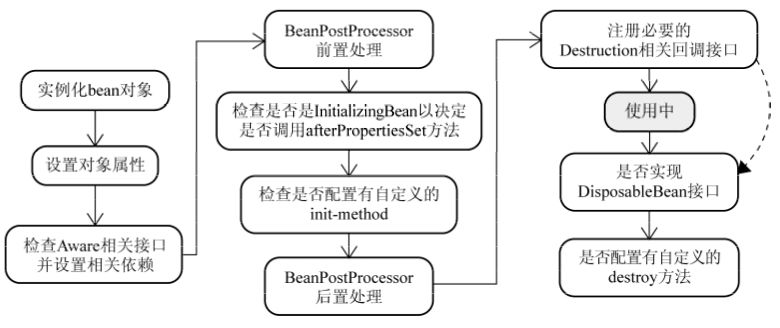
实质：由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

* 单例模式
* 代理模式
* 模板方法模式
* 观察者模式

### Bean的生命周期？

Spring Bean的生命周期分为四个阶段：

1. 实例化 Instantiation
2. 属性赋值 Populate
3. 初始化 Initialization
4. 销毁 Destruction



1. **实例化Bean**

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。 对于ApplicationContext容器，当容器启动结束后，便实例化所有的bean。 容器通过获取BeanDefinition对象中的信息进行实例化。并且这一步仅仅是简单的实例化，并未进行依赖注入。 实例化对象被包装在BeanWrapper对象中，BeanWrapper提供了设置对象属性的接口，从而避免了使用反射机制设置属性。

1. 设置对象属性（依赖注入）

实例化后的对象被封装在BeanWrapper对象中，并且此时对象仍然是一个原生的状态，并没有进行依赖注入。 紧接着，Spring根据BeanDefinition中的信息进行依赖注入。 并且通过BeanWrapper提供的设置属性的接口完成依赖注入。

1. 注入Aware接口

紧接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给bean。

1. BeanPostProcessor

当经过上述几个步骤后，bean对象已经被正确构造，但如果你想要对象被使用前再进行一些自定义的处理，就可以通过BeanPostProcessor接口实现。 该接口提供了两个函数：

* postProcessBeforeInitialzation( Object bean, String beanName ) 当前正在初始化的bean对象会被传递进来，我们就可以对这个bean作任何处理。 这个函数会先于InitialzationBean执行，因此称为前置处理。 所有Aware接口的注入就是在这一步完成的。
* postProcessAfterInitialzation( Object bean, String beanName ) 当前正在初始化的bean对象会被传递进来，我们就可以对这个bean作任何处理。 这个函数会在InitialzationBean完成后执行，因此称为后置处理。

1. InitializingBean

与init-method当BeanPostProcessor的前置处理完成后就会进入本阶段。 InitializingBean接口只有一个函数：

* afterPropertiesSet()这一阶段也可以在bean正式构造完成前增加我们自定义的逻辑，但它与前置处理不同，由于该函数并不会把当前bean对象传进来，因此在这一步没办法处理对象本身，只能增加一些额外的逻辑。 若要使用它，我们需要让bean实现该接口，并把要增加的逻辑写在该函数中。然后Spring会在前置处理完成后检测当前bean是否实现了该接口，并执行afterPropertiesSet函数。当然，Spring为了降低对客户代码的侵入性，给bean的配置提供了init-method属性，该属性指定了在这一阶段需要执行的函数名。Spring便会在初始化阶段执行我们设置的函数。init-method本质上仍然使用了InitializingBean接口。

1. DisposableBean和destroy-method

和init-method一样，通过给destroy-method指定函数，就可以在bean销毁前执行指定的逻辑。

## SpringMVC

<https://zhuanlan.zhihu.com/p/114244039>

* 简单工厂模式

实现方式：BeanFactory。Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

实质：由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

* 单例模式
* 代理模式
* 模板方法模式
* 观察者模式

## Mybatis

# 设计模式

## 动态代理模式 JDK与CGLIB字节码生成技术

# 场景题

## 如何应对高并发？

所谓高并发指的是：在同时或极短时间内，有大量的请求到达服务端，每个请求都需要服务端耗费资源进行处理，并做出相应的反馈。

从服务端视角看高并发：服务端处理请求需要耗费服务端的资源，比如能同时开启的进程数、能同时运行的线程数、网络连接数、cpu、I/O、内存等等，由于服务端资源是有限的，那么服务端能同时处理的请求也是有限的。高并发问题的本质就是：资源的有限性。

解决的思路：增加资源去响应服务，或者是减少资源使用的时间。

高并发带来的问题：服务端的处理和响应会越来越慢，甚至会丢弃部分请求不予处理，更严重的会导致服务端崩溃。

高并发处理的基本思路：

* 增加资源供给，比如：更大的网络带宽，使用更高配置的服务器，使用高性能的Web服务器，使用高性能的数据库
* 请求分流，比如：使用集群,分布式的系统架构，数据库分库
* 使用缓存队列，将请求保存在缓存队列中，然后按序请求。
* 应用优化，比如：使用更高效的编程语言,优化处理业务逻辑的算法,优化访问数据库的SQL

再细分：

* Web服务器层面：
* 对Web服务器进行配置优化，比如：调整内存数量、线程数量等
* 对Web服务器进行集群，提供多个能提供相同服务的Web服务器，以实现负载均衡
* Web应用层面，常见的手段有：
* 优化处理业务逻辑的算法
* 合理高效的利用缓存，部分业务可以考虑内存数据库，或者是进行纯内存处理
* 优化访问数据库的Sql，可以考虑利用存储过程等数据库的能力
* 合理使用多线程，使用线程池，加快业务处理
* 数据库层面，常见的手段有：
* 合理选择数据库的引擎，比如Mysql的InnoDB与MyISAM引擎进行配置优化
* 可以考虑使用存储过程来处理复杂的数据逻辑
* 数据库集群，进行读写分离，分库、分表，降低单库、单表的数据量
* 合理设计数据库的表结构、索引等

## 如何提高系统吞吐量？

系统吞度量要素：

QPS（TPS）：每秒钟request/事务 数量

并发数： 系统同时处理的request/事务数

响应时间： 一般取平均响应时间

QPS（TPS）= 并发数/平均响应时间 或者 并发数 = QPS\*平均响应时间

方法：

* 增加并发数

1.比如增加tomcat并发的线程数，开喝服务器性能匹配的线程数，可以更多满足服务请求。

2.增加数据库的连接数，预建立合适数量的TCP连接数

3.后端服务尽量无状态话，可以更好支持横向扩容，满足更大流量要求

4.调用链路上的各个系统和服务尽量不要单点，要从头到尾都是能力对等的，不能让其中某一点成为瓶颈。

5.RPC调用的尽量使用线程池，预先建立合适的连接数。

* 减少平均响应时间

1.请求尽量越前结束，越好，这样压力就不要穿透到后面的系统上，可以在各个层上加上缓存

2.流量消峰。放行适当的流量，处理不了的请求直接返回错误或者其他提示。和水坝道理很类似

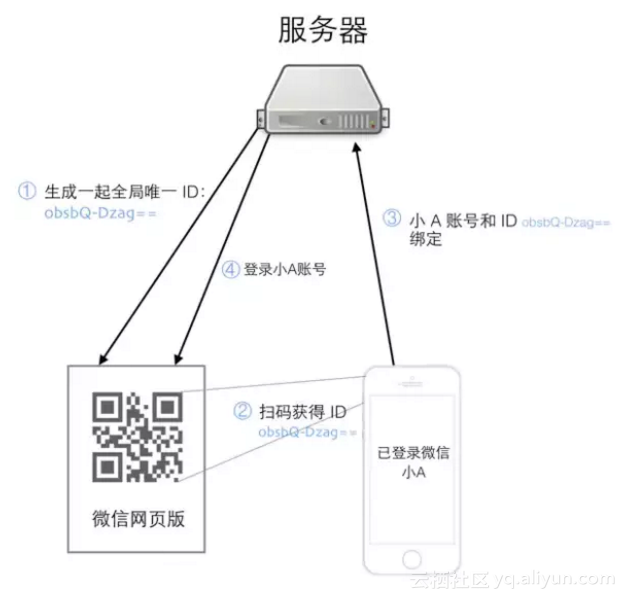
3.减少调用链

4.优化程序

5.减少网络开销，适当使用长连接

6.优化数据库，建立索引

## 扫码登录如何实现，微信支付怎么实现



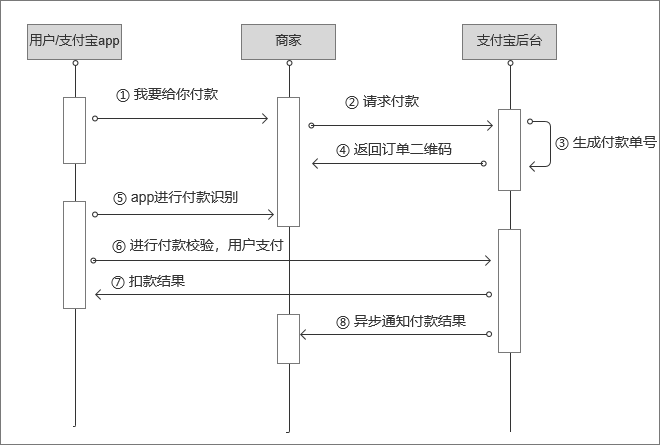
①：用户 A 访问微信网页版，微信服务器为这个会话生成一个全局唯一的 ID，上面的 URL 中 obsbQ-Dzag== 就是这个 ID，此时系统并不知道访问者是谁。

②：用户A打开自己的手机微信并扫描这个二维码，并提示用户是否确认登录。

③：手机上的微信是登录状态，用户点击确认登录后，手机上的微信客户端将微信账号和这个扫描得到的 ID 一起提交到服务器

④：服务器将这个 ID 和用户 A 的微信号绑定在一起，并通知网页版微信，这个 ID 对应的微信号为用户 A，网页版微信加载用户 A 的微信信息，至此，扫码登录全部流程完成

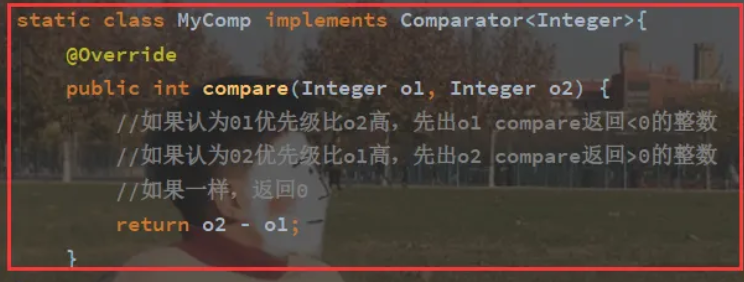
扫码登录看起来神奇，主要是因为微信 APP 扫自家的码会做一些普通二维码软件不会做的额外的操作，那就是将当前已登录的微信和扫出来的 ID 提交到微信服务器，类似的应用还有扫码支付、扫码加公众号等功能。



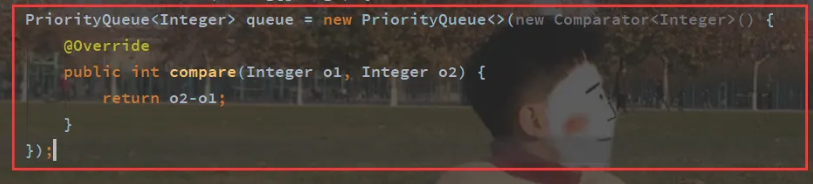
# 算法笔记

## 重写比较器接口

成员内部类



匿名内部类



Lambda表达式：lambda表达式可以使用类的成员变量。

