**Java基础**

# HashMap

## 为什么浮在因子设置为0.75

<https://juejin.im/post/5e532459e51d4526cd1de801>

HashMap采用的结构为数组+链表+红黑树(1.8),利用hash算法将数据分配到各个桶中，如果两个值计算出的hash不一样，那么这个两个值一定不相同，如果一样，这两个值有可能相同，也有可能不相同，这就发生了hash碰撞，发生了hash碰撞，hashmap将第二个值加第一个值的链表中，如果发生碰撞的概率大，那么hashmap就会退化成一个链表，导致查询的效率低下。如果减小发生碰撞的概率，有两个方法:1.扩容 2.设计一个良好的hash算法

什么时候扩容，在hashmap中是的达到了临界值的时候就会扩容，临界值=负载因子\*容量，如果负载因子过小，那么会导致频繁的扩容，耗费内存空间，过大，就会导致hash碰撞的概率增加，会导致查询的速度降低，0.75不大也不小，而且\*2的幂是整数，所以选择0.75是合适的。

## 线程安全问题

<https://blog.csdn.net/rambomatrix/article/details/80788187>

# ConcurrentHashMap源码

Put元素

|  |
| --- |
| */\*\* Implementation for put and putIfAbsent \*/* **final** V putVal(K key, V value, **boolean** onlyIfAbsent) {  **if** (key == **null** || value == **null**) **throw new** NullPointerException();  **// key的哈希值的高16位和低16位异或，然后和当前容量&0x7fffffff**   **int hash = *spread*(key.hashCode());**  **int** binCount = 0;  **for** (Node<K,V>[] tab = **table**;;) {  Node<K,V> f; **int** n, i, fh;  **if** (tab == **null** || (n = tab.**length**) == 0)  tab = initTable();  **// i为下标**  **else if** ((f = *tabAt*(tab, i = (n - 1) & hash)) == **null**) {  **// 利用CAS添加元素**  **if** (*casTabAt*(tab, i, **null**,  **new** Node<K,V>(hash, key, value, **null**)))  **break**; *// no lock when adding to empty bin* }  **else if** ((fh = f.**hash**) == ***MOVED***)  tab = helpTransfer(tab, f);  **else** {  V oldVal = **null**;  **synchronized** (f) {  **if** (*tabAt*(tab, i) == f) {  **if** (fh >= 0) {  binCount = 1;  **for** (Node<K,V> e = f;; ++binCount) {  K ek;  **if** (e.**hash** == hash &&  ((ek = e.**key**) == key ||  (ek != **null** && key.equals(ek)))) {  oldVal = e.**val**;  **if** (!onlyIfAbsent)  e.**val** = value;  **break**;  }  Node<K,V> pred = e;  **if** ((e = e.**next**) == **null**) {  pred.**next** = **new** Node<K,V>(hash, key,  value, **null**);  **break**;  }  }  }  **else if** (f **instanceof** TreeBin) {  Node<K,V> p;  binCount = 2;  **if** ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,  value)) != **null**) {  oldVal = p.**val**;  **if** (!onlyIfAbsent)  p.**val** = value;  }  }  }  }  **if** (binCount != 0) {  **if** (binCount >= ***TREEIFY\_THRESHOLD***)  treeifyBin(tab, i);  **if** (oldVal != **null**)  **return** oldVal;  **break**;  }  }  }  addCount(1L, binCount);  **return null**; } |

初始化table

|  |
| --- |
| **private final** Node<K,V>[] initTable() {  Node<K,V>[] tab; **int** sc;  **while** ((tab = **table**) == **null** || tab.**length** == 0) {  // **sizeCtl小于0表示table正在初始化或者正在扩容，那么当前线程就要先让出执行的权利，以免干扰其他的线程初// 始化或者扩容的操作**  **if** ((sc = **sizeCtl**) < 0)  **Thread.*yield*();** *// lost initialization race; just spin* **else if** (***U***.compareAndSwapInt(**this**, ***SIZECTL***, sc, -1)) { **// 利用CAS修改sizeCtl的值**  **try** {  **if** ((tab = **table**) == **null** || tab.**length** == 0) {  **int** n = (sc > 0) ? sc : ***DEFAULT\_CAPACITY***;  @SuppressWarnings(**"unchecked"**)  Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n];  **table** = tab = nt;  sc = n - (n >>> 2);  }  } **finally** {  **sizeCtl** = sc;  }  **break**;  }  }  **return** tab; } |

1. 计算hashcode
2. 如果没有初始化table，就初始化(在初始化的时候会先判断sizeCtl的值是否小于0，如果小于0，则代表其他的线程正在扩容或者初始化，否则利用CAS将sizeCtl的值变为-1代表正在初始化)
3. 利用hashcod与当前容量取余求下标，利用CAS添加元素，如果添加成功直接跳出循环，否则继续循环

# 三种for循环区别

1. 如果不满足条件判断，那么**for循环和while循环就不会执行**，do while循环会执行至少一次。
2. for循环的变量再小括号中定义，只有循环内部才可以使用。（初始化语句不可以输出）。
3. While循环和do-while循环初始化语句本来就在外面，所以出来循环之后还可以继续使用。（初始化语句可以输出）

# equals和hashcode的作用

1.若重写了equals(Object obj)方法，则有必要重写hashCode()方法。

2.若两个对象equals(Object obj)返回true，则hashCode（）有必要也返回相同的int数。

3.若两个对象equals(Object obj)返回false，则hashCode（）不一定返回不同的int数。

4.若两个对象hashCode（）返回相同int数，则equals（Object obj）不一定返回true。

5.若两个对象hashCode（）返回不同int数，则equals（Object obj）一定返回false。

6.同一对象在执行期间若已经存储在集合中，则不能修改影响hashCode值的相关信息，否则会导致内存泄露问题。

一般来说涉及到对象之间的比较大小就需要重写equals方法，

# string和stringbuilder、stringbuffer的区别

**String是只读字符串，所引用的字符串不能被改变，一经定义，无法再增删改。**

**String 定义的字符串保存在常量池里面，进行+操作时不能直接在原有基础上拼接。**

**每次+操作 ： 隐式在堆上new了一个跟原字符串相同的StringBuilder对象，再调用append方法 拼接+后面的字符。**

String和Stringbuilder在单线程环境下使用；

StringBuffer在多线程环境下使用，可以保证线程同步；

Stringbuilder 和StringBuffer 实现方法类似，均表示可变字符序列，不过StringBuffer 用synchronized关键字修饰（保证线程同步）

# 容器

HashMap请见Java基础篇面试题

## ArrayLIst、LinkedLIst的区别

1. ArrayList的实现是基于数组来实现的，LinkedList的基于双向链表来实现。这两个数据结构的逻辑关系是不一样，当然物理存储的方式也会是不一样。

2. 对于随机访问，ArrayList优于LinkedList。

3. 对于插入和删除操作，LinkedList优于ArrayList

1. LinkedList比ArrayList更占内存，因为LinkedList的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

## HashSet

内部是用HashMap来实现的

## Hashmap

### ****谈一下HashMap的特性？****

1.HashMap存储键值对实现快速存取，允许为null。key值不可重复，若key值重复则覆盖。

2.非同步，线程不安全。

1. 底层是hash表，不保证有序(比如插入的顺序)

### ****谈一下HashMap的底层原理是什么？****

基于hashing的原理，jdk8后采用数组+链表+红黑树的数据结构。我们通过put和get存储和获取对象。当我们给put()方法传递键和值时，先对键做一个hashCode()的计算来得到它在bucket数组中的位置来存储Entry对象。当获取对象时，通过get获取到bucket的位置，再通过键对象的equals()方法找到正确的键值对，然后在返回值对象。

### ****谈一下hashMap中put是如何实现的？****

**1.计算关于key的hashcode值（与Key.hashCode的高16位做异或运算）**

**2.如果散列表为空时，调用resize()初始化散列表**

**3.如果没有发生碰撞，直接添加元素到散列表中去**

**4.如果发生了碰撞(hashCode值相同)，进行三种判断**

**4.1:若key地址相同或者equals后内容相同，则替换旧值**

**4.2:如果是红黑树结构，就调用树的插入方法**

**4.3：链表结构，循环遍历直到链表中某个节点为空，尾插法进行插入，插入之后判断链表个数是否到达变成红黑树的阙值8；也可以遍历到有节点与插入元素的哈希值和内容相同，进行覆盖。**

**5.如果桶满了大于阀值，则resize进行扩容**

### ****谈一下hashMap中什么时候需要进行扩容，扩容resize()又是如何实现的？****

1.初始化数组table

2.当数组table的size达到阙值时即++size > load factor \* capacity 时，也是在putVal函数中

实现过程：(细讲)

1.通过判断旧数组的容量是否大于0来判断数组是否初始化过

否：进行初始化

判断是否调用无参构造器，

是:使用默认的大小和阙值

否:使用构造函数中初始化的容量，当然这个容量是经过tableSizefor计算后的2的次幂数

是，进行扩容，扩容成两倍(小于最大值的情况下)，之后在进行将元素重新进行与运算复制到新的散列表中

概括的讲：扩容需要重新分配一个新数组，新数组是老数组的2倍长，然后遍历整个老结构，把所有的元素挨个重新hash分配到新结构中去。

### 谈一下hashMap中get是如何实现的？

对key的hashCode进行hashing，与运算计算下标获取bucket位置，如果在桶的首位上就可以找到就直接返回，否则在树中找或者链表中遍历找，如果有hash冲突，则利用equals方法去遍历链表查找节点。

### 谈一下HashMap中hash函数是怎么实现的？还有哪些hash函数的实现方式？

对key的hashCode做hash操作，与高16位做异或运算

还有平方取中法，除留余数法，伪随机数法

### ****.为什么不直接将key作为哈希值而是与高16位做异或运算？****

因为数组位置的确定用的是与运算，仅仅最后四位有效，设计者将key的哈希值与高16为做异或运算使得在做&运算确定数组的插入位置时，此时的低位实际是高位与低位的结合，增加了随机性，减少了哈希碰撞的次数。

HashMap默认初始化长度为16，并且每次自动扩展或者是手动初始化容量时，必须是2的幂。

### .为什么是16？为什么必须是2的幂？如果输入值不是2的幂比如10会怎么样？

Index=hashcode&(length-1)

<https://blog.csdn.net/sidihuo/article/details/78489820>

<https://blog.csdn.net/eaphyy/article/details/84386313>

1.为了数据的均匀分布，减少哈希碰撞。因为确定数组位置是用的位运算，若数据不是2的次幂则会增加哈希碰撞的次数和浪费数组空间。(PS:其实若不考虑效率，求余也可以就不用位运算了也不用长度必需为2的幂次)

1. 输入数据若不是2的幂，HashMap通过一通位移运算和或运算得到的肯定是2的幂次数，并且是离那个数最近的数字。

### ****.谈一下当两个对象的hashCode相等时会怎么样？****

会产生哈希碰撞，若key值相同则替换旧值，不然链接到链表后面，链表长度超过阙值8就转为红黑树存储

### .如果两个键的hashcode相同，你如何获取值对象？

HashCode相同，通过equals比较内容获取值对象

### ****."如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？****

超过阙值会进行扩容操作，概括的讲就是扩容后的数组大小是原数组的2倍，将原来的元素重新hashing放入到新的散列表中去。

### .HashMap和HashTable的区别

相同点：都是存储key-value键值对的

不同点：

HashMap允许Key-value为null，hashTable不允许；

hashMap没有考虑同步，是线程不安全的。hashTable是线程安全的，给api套上了一层synchronized修饰;

HashMap继承于AbstractMap类，hashTable继承与Dictionary类。

迭代器(Iterator)。HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException。

容量的初始值和增加方式都不一样：HashMap默认的容量大小是16；增加容量时，每次将容量变为"原始容量x2"。Hashtable默认的容量大小是11；增加容量时，每次将容量变为"原始容量x2 + 1"；

添加key-value时的hash值算法不同：HashMap添加元素时，是使用自定义的哈希算法。Hashtable没有自定义哈希算法，而直接采用的key的hashCode()。

### ****请解释一下HashMap的参数loadFactor，它的作用是什么？****

loadFactor表示HashMap的拥挤程度，影响hash操作到同一个数组位置的概率。默认loadFactor等于0.75，当HashMap里面容纳的元素已经达到HashMap数组长度的75%时，表示HashMap太挤了，需要扩容，在HashMap的构造器中可以定制loadFactor。

### ****传统hashMap的缺点(为什么引入红黑树？)：****

JDK 1.8 以前 HashMap 的实现是 数组+链表，即使哈希函数取得再好，也很难达到元素百分百均匀分布。当 HashMap 中有大量的元素都存放到同一个桶中时，这个桶下有一条长长的链表，这个时候 HashMap 就相当于一个单链表，假如单链表有 n 个元素，遍历的时间复杂度就是 O(n)，完全失去了它的优势。针对这种情况，JDK 1.8 中引入了 红黑树（查找时间复杂度为 O(logn)）来优化这个问题。

### ****平时在使用HashMap时一般使用什么类型的元素作为Key？****

选择Integer，String这种不可变的类型，像对String的一切操作都是新建一个String对象，对新的对象进行拼接分割等，这些类已经很规范的覆写了hashCode()以及equals()方法。作为不可变类天生是线程安全的

### 1.7版本和1.8版本HashMap的区别



回答的时候:

1. 数组和链表的优缺点

数组: 数组存储区间是连续的，占用内存严重，故空间复杂的很大。但数组的二分查找时间复杂度小，为O(1)；数组的特点是：寻址容易，插入和删除困难；

链表: 链表存储区间离散，占用内存比较宽松，故空间复杂度很小，但时间复杂度很大，达O（N）。链表的特点是：寻址困难，插入和删除容易。

1. 先说出结构是什么(数组(存储的元素类型为entry:key、value、next)+链表)
2. 如何put元素
3. Put元素的时候，hash冲突的怎么办?(解决hash冲突的方法:开放地址法，链地址法，再hash法，建立一个公共溢出区。)
4. 如何get元素
5. Java8性能改善:



### 1.7版本和1.8版本ConcurrentHashMap的区别

#### 1.7



1. 结构(segment数组+HashEntry数组+链表)

Segment继承RetreentLock，每个Segment对象是一把锁，Segment数组的意义是把一个大的table划分成为一个个小的table来加锁，即利用了分段锁的技术来提高并发性。

1. Put操作:

Put操作时，先同过hash计算出Segment的位置，如果Segment没有初始化，即通过CAS进行操作，然后进行第二次hash操作，找到对应的hashentry的位置，这里会利用继承过来的锁的特性，在将数据插入指定的HashEntry的位置时，会通过tryLock方法尝试获取锁，如果获取成功，就直接插入，没有成功会通过自旋的方式继续调用tryLock方法获取锁，超过指定次数就挂起，等待被唤醒。

1. get操作

通过hash计算出Segment的位置，然后再hash定位到指定的HashEntry，遍历该HashEntry下的链表，成功就返回，不成功就返回null

1. Size

方案一:使用不加锁的方式取尝试多次计算ConcurrentHashMap的size，最多三次，比较前后两次计算的结果，结果一致就认为当前没有元素加入，计算的结果是准确的，计算出的值即为当前的size

方案二：如果第一种方案不符合，就会对每个Segment加上锁，然后计算ConcurrentHashMap的size返回。

#### 1.8

# 多线程

## 多线程有什么用

（1）**发挥多核CPU的优势**

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。单核CPU上所谓的”多线程”那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程”同时”运行罢了。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

（2**）防止阻塞**

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

（3）**便于建模**

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

## Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

（1）获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java

（2）打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈

## 生产者消费者模型的作用是什么

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

## 怎么检测一个线程是否持有对象监视器

我也是在网上看到一道多线程面试题才知道有方法可以判断某个线程是否持有对象监视器：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着“某条线程”指的是当前线程。

## ThreadLocal





ThreadLocal保存的变量只在线程本地共享，不同线程之间是不能共享的

Set方法：将变量值设置当前线程中。

Get方法：获取当前线程中的变量值

Remove方法：移除ThreadLocal中当前线程存储的值。

Set:

1. 获取当前的线程
2. 获取当前线程中的ThreadLocalMap对象
3. 如果为null，需要初始化
4. 不为null，以ThreadLocal对象作为key，存到ThreadLocalMap对象中

每个线程中都会有一份变量的副本，所以不同线程之间使用该变量的时候，互不影响，可以保证线程安全。

Get:

1. 获取当前先测回滚
2. 获取当前线程中的ThreadLocalMap对象
3. 以ThreadLocal作为key，从ThreadLocalMap中取获取。

## 创建线程的几种方式

1. 继承Thread类，重写run方法，调用start方法启动线程
2. 实现Runable接口，重写run方法
3. 实现Callable接口，重写call方法，可以获取返回值

第一种的优点:实现简单，如果获取当前线程直接使用this即可

缺点:Java中类只能继承一个类，继承了Thread，就不能继承其他的类

后两种的优点: 只是实现了接口，还可以继承其他的类，多个线程可以共享同一个target对象。

缺点: 编程稍微复杂，如果要启动线程，需要创建一个线程类，然后调用start方法启动，如果要获取当前的线程，必须要调用Thread.concurrentThread方法来获取。

## wait、sleep分别是谁的方法,区别?

Wait方法是Object中的，sleep方法是Thread中的。

区别:

调用wait方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法，线程才进入对象锁定池准备获取对象锁进入运行状态。

Sleep只是让当前的线程处于休眠状态让出cpu给其他的线程，但是不会放弃对象锁，等休眠的时间到了后自动回复运行状态。

## 线程间通信的方式

1. 使用volatile关键字

基于 volatile 关键字来实现线程间相互通信是使用共享内存的思想，大致意思就是多个线程同时监听一个变量，当这个变量发生变化的时候 ，线程能够感知并执行相应的业务。

1. 使用synchronized，Lock

3、使用Object类的wait()、notify、notifyAll方法

比如阻塞队列的实现，当队列满的时候，不能继续放入元素，需要等待其他线程从队列中获取元素使队列有空间可以继续存放元素；当队列为空的时候，不能继续获取元素，需要等待其他线程放入元素使队列变为非空，才能继续有元素可以获取；

1. 使用JUC工具类 CountDownLatch

CountDownLatch是基于AQS实现的，相当于也维护了线程 间共享的变量state。

4、基本LockSupport实现线程间的阻塞和唤醒。

## 介绍下什么是死锁？遇见过死锁吗，是怎么排查的

## 创建线程池的几种方式，线程池有什么好处

1. newCachedThreadPool

创建一个线程池，如果线程池中的线程数量过大，它可以有效的回收多余的线程，如果线程数不足，那么它可以创建新的线程。

1. newFixedThreadPool

创建固定数量的线程。

1. newScheduledThreadPool

该线程池支持定时周期性的任务运行

1. newSingleThreadExecutor

只有一个线程

好处：

如果每个请求来的时候，都创建一个线程取处理，线程的创建是个耗资源的操作，还需要jvm的操作系统的交互，请求处理完，再销毁，这种操作势必会延迟请求的处理，因为创建线程需要一定的时间，为了防止这种情况发生，可以使用线程池，重复的利用线程，而不是每次都取创建销毁线程，这不但节约资源还能提高效率，因为请求过来，线程已经存在了，不需要花费时间再去创建。总之利用线程池就是为了提高性能节约资源。

## 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：

假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

## synchronized reentrantlock的区别

**功能区别：**这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成

便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

**等待可中断**，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。

**公平锁**，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

**锁绑定多个条件**

一个ReentrantLock对象可以同时绑定对个对象。ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

**性能的区别：**在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

**Synchronized**进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

**ReentrantLock**

ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁

基于AQS这个基础框架来实现的，内部有共享变量state来表示获取锁的次数，当获取锁时，会先判断该变量是否等于0，等于0说明该锁没有被其他线程所获取，可以获取该锁，并且利用CAS来修改state变量，如果已经被其他线程持有，那么会判断锁的持有者和当前线程是否相等，如果相等，也是可以获取锁的，这就是锁的可重入性，可以被多次获取，如果锁的持有者不是当前线程，那么就会进入阻塞队列，在进入阻塞队列之前，会再次尝试获取锁，失败了，就直接添加到阻塞队列中去。

## CountDownLatch、CyclicBarrier的用法及原理

### CountDownLatch 的用法

|  |
| --- |
| **private static** CountDownLatch *countDownLatch* = **new** CountDownLatch(2);  **private static** ThreadPoolExecutor *threadPoolExecutor* = **new** ThreadPoolExecutor(10, 10,  0, TimeUnit.***MILLISECONDS***, **new** LinkedBlockingQueue<>(100));  **public static void** main(String[] args) {  Worker worker1 = **new** Worker(**"张三"**);  Worker worker2 = **new** Worker(**"李斯"**);   Boss boss = **new** Boss(**"王五"**);   *threadPoolExecutor*.execute(worker1);  *threadPoolExecutor*.execute(worker2);  *threadPoolExecutor*.execute(boss);   *threadPoolExecutor*.shutdown(); } **static class** Worker **implements** Runnable {  **private** String **name**;  **public** Worker(String name) {  **this**.**name** = name;  }  @Override  **public void** run() {  System.***out***.println(**name** + **"已经做完了工作"**);  *countDownLatch*.countDown();  } }  **static class** Boss **implements** Runnable {  **private** String **name**;  **public** Boss(String name) {  **this**.**name** = name;  }  @Override  **public void** run() {  System.***out***.println(**"等待员工完成工作"**);  **try** {  *countDownLatch*.await();  System.***out***.println(**name** + **"开始检查工作======================="**);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } } |

### 源码解读

#### countDown方法







1. 获取state的值
2. 如果state==0，就什么也不做
3. 否则，就将state-1，利用CAS设置state的值，如果设置成功，就返回，返回的值是根据当前state==0决定，如果等于0，那么 就会去唤醒阻塞队列中的线程，否则就什么也不做

将等待的线程放到阻塞队列中去



#### Await()方法





1. 判断state是否等于0
2. 如果state等于0，就直接往下执行，不需要阻塞
3. 如果不等于0，就会创建一个Node，将当前的线程对象封装进去，加到队列中
4. 在挂起线程前，会首先判断当前线程的前置节点是否是头结点，如果是头结点，就再去判断state的值，等于0，就将当前的节点设置为头结点，并且将thread设置为null
5. 如果前置节点不是头结点，继续判断钱直节点waitstatus，如果waitstatus==singnal，就调用LockSupport类中的park方法挂起，否则继续循环判断，此处使用的是自旋方式知道state==0

### CyclicBarrier

|  |
| --- |
| ***/\*\*  \* 场景:有三个运动员一起比赛，其中一个准备好了，必须等待其他两个运动员准备好，比赛才能开始  \*/* private static CyclicBarrier *cyclicBarrier* = new CyclicBarrier(3);  private static ThreadPoolExecutor *threadPoolExecutor* = new ThreadPoolExecutor(10, 10, 0, TimeUnit.*MILLISECONDS*,  new LinkedBlockingQueue<>(10));  public static void main(String[] args) {  Runner runner = new Runner("小明");  Runner runner2 = new Runner("大明");  Runner runner3 = new Runner("小大明");   *threadPoolExecutor*.execute(runner);  *threadPoolExecutor*.execute(runner2);  *threadPoolExecutor*.execute(runner3); }  static class Runner implements Runnable {   private String name;   public Runner(String name) {  this.name = name;  }   @Override  public void run() {  try {  System.*out*.println(name + "开始准备");  Thread.*sleep*(1000);  System.*out*.println(name + "准备好了");   *cyclicBarrier*.await();   System.*out*.println(name + "开始跑步");  } catch (InterruptedException | BrokenBarrierException e) {  e.printStackTrace();  }  } }** |

#### 源码剖析







1. 有线程中断就将CycleBarrier置为不可用
2. 将计数器-1
3. 如果等于0，执行构造器中传入的任务，如果构造器有传入任务
4. 换代:唤醒等待队列中的线程，重置计数器，重新new出个Generation，代表新的一代
5. 不等于零，阻塞直到被唤醒
6. 每个线程可以使用多个CycleBarrier，也就可以被多个CycleBarrier唤醒，如果被其他的CycleBarrier唤醒，那么g(局部变量)==generation(成员变量)，否则就是被自己的CycleBarrier唤醒的，就可以直接返回了。

## Volatile变量的作用和原理

### 简述内存模型

Java内存模型分为主内存和工作内存两大类。

1. **主内存：**多个线程共享的内存。方法区和对堆属于主内存。

方法区:方法区和堆一样，方法区是线程所共享的内存区域，它保存类的基本信息。比如类的字段、方法、常量池等。方法区的大小决定系统可以保存多少个类。如果系统定义太多个类，导致方法区溢出。虚拟机同样会抛出内存溢出的错误。

1. **工作内存:**每个线程独享的内存。虚拟机栈、程序计数器、本地方法栈属于线程工作内存。



**Java内存模型规定：所有变量都需要存储在主内存中，线程工作内存保存了变量在主内存中的副本，线程对变量的所有操作都在工作内存中进行，执行结束后在同步到主内存中去。**

这里必然有时间差，在这个时间差内，该线程对副本的操作，对于其他线程是不可见的，从而造成了可见性问题。



### 指令重排序

JMM对代码进行了编译优化，导致代码可能并不是按照代码编写的顺序执行的，而是经过JMM编译优化过的顺序执行。指令重排序对并发安全性有很大的影响，所以提供了一些happens-before规则定义一些禁止编译优化的场景。

### Volatile的作用

1. 保证共享变量的可见性：使用volatile修饰的变量，任何线程对其操作都是在主内存中进行的，不会产生副本，从而保证共享变量的可见性。
2. 防止局部指令重排序：happens-before中的volatile规则规定了一个线程先去写一个volatile变量，然后一个线程去读这个变量，那么这个写操作的结果一定对这个读的线程可见。

Volatile是通过内存屏障来防止指令重排序的。

硬件层面的内存屏障分为**Load Barrier**和**Store Barrier即读屏障和写屏障。**

* 对于Load Barrier来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从主内存加载数据。
* 对于Store Barrier来说，在指令后插入Store Barrier，能让写入缓存中的最新数据写入主内存，让其他线程可见。

<https://juejin.im/post/5e7ac6096fb9a07ce31f1230>

# 什么是乐观锁和悲观锁

（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

（2）悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

# 什么是自旋

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

# try{}里有一个return语句，那么紧跟在这个try后的finally{}里的code会不会被执行，什么时候被执行，在return前还是后?

return前被执行，有程序为证

# 阻塞队列

## ArrayBlockingQueue

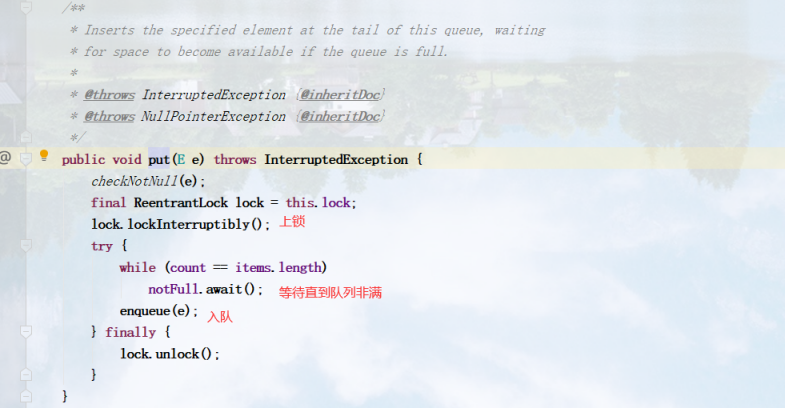
内部成员变量

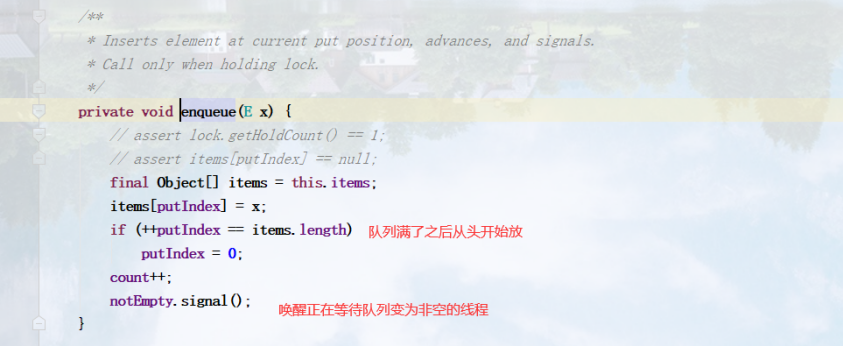
是通过Lock和condition实现的，内部的存储的数据结构为数组



### Put方法

不能传入null值

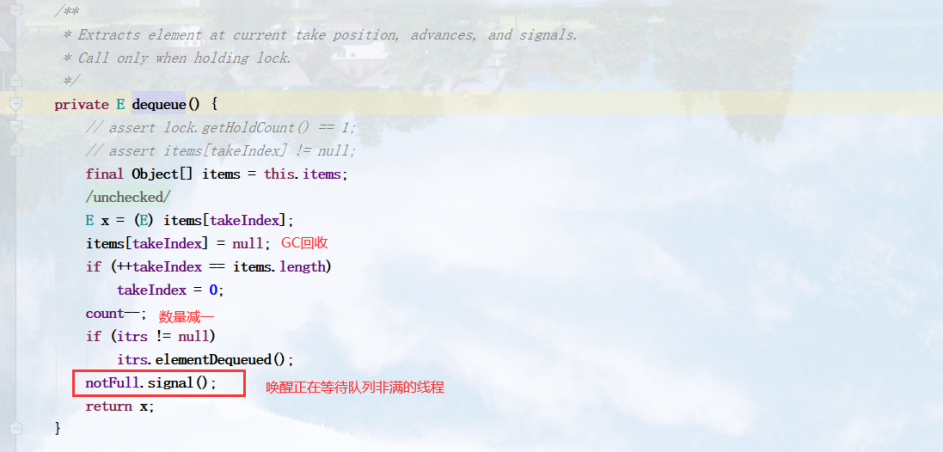




### take方法

会阻塞





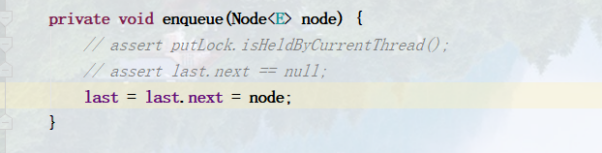
## LinkedBlockingQueue

### 内部结构



### put





### take

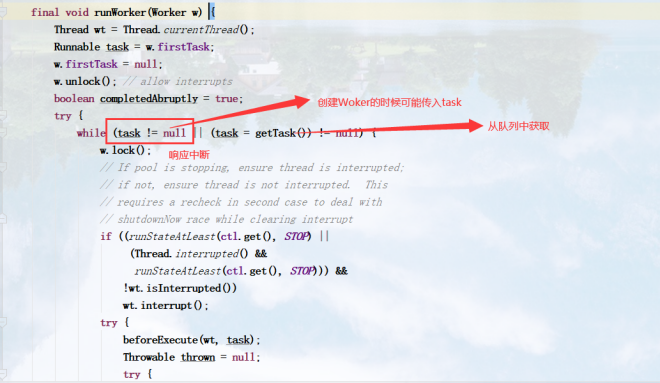


# ThreadPoolExecutor

## 为什么要用线程池

1. 减少资源的消耗:减少了每次创建线程、销毁线程的消耗
2. 提高相应速度:每次请求到来时，由于线程的创建已经完成，故可以直接执行任务，因此提高了响应速度。
3. 提高线程的可管理性:线程是一种稀缺资源，若不加以限制，不仅会占用大量资源，而且会影响系统的稳定性。因此，线程池可以对线程的创建与停止、线程数量等等因素加以控制，使得线程在一种可控的范围内运行，不仅保证系统稳定运行，而且方便性能调优。

在运行任何任务之前，将获取锁以防止任务执行时其他池中断，然后确保除非池正在停止，否则此线程不会设置其中断。



## 原理

（1）如果没有空闲的线程执行该任务且当前运行的线程数少于corePoolSize，则添加新的线程执行该任务。

（2）如果没有空闲的线程执行该任务且当前的线程数等于corePoolSize同时阻塞队列未满，则将任务入队列，而不添加新的线程。

（3）如果没有空闲的线程执行该任务且阻塞队列已满同时池中的线程数小于maximumPoolSize，则创建新的线程执行任务。

（4）如果没有空闲的线程执行该任务且阻塞队列已满同时池中的线程数等于maximumPoolSize，则根据构造函数中的handler指定的策略来拒绝新的任务。

keepAliveTime：表示空闲线程的存活时间。

TimeUnitunit：表示keepAliveTime的单位。

为了解释keepAliveTime的作用，我们在上述情况下做一种假设。假设线程池这个单位已经招了些临时工，但新任务没有继续增加，所以随着每个员工忙完手头的工作，都来workQueue领取新的任务（看看这个单位的员工多自觉啊）。随着各个员工齐心协力，任务越来越少，员工数没变，那么就必定有闲着没事干的员工。这样的话领导不乐意啦，但是又不能轻易fire没事干的员工，因为随时可能有新任务来，于是领导想了个办法，设定了keepAliveTime，当空闲的员工在keepAliveTime这段时间还没有找到事情干，就被辞退啦，毕竟地主家也没有余粮啊！当然辞退到corePoolSize个员工时就不再辞退了，领导也不想当光杆司令啊！

handler：表示当workQueue已满，且池中的线程数达到maximumPoolSize时，线程池拒绝添加新任务时采取的策略。

## 线程池的拒绝策略

ThreadPoolExecutor.AbortPolicy() 抛出RejectedExecutionException异常

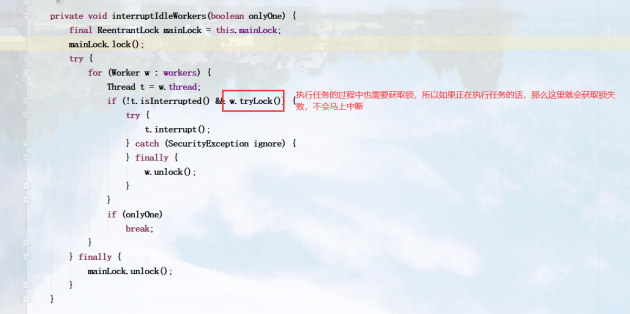
ThreadPoolExecutor.CallerRunsPolicy() 由向线程池提交任务的线程来执行该任务

ThreadPoolExecutor.DiscardOldestPolicy() 抛弃最旧的任务（最先提交而没有得到执行的任务）

ThreadPoolExecutor.DiscardPolicy() 抛弃当前的任务

## 关闭线程池的几种状态

### shutdown方法



1、调用之后不允许继续往线程池内继续添加线程;

2、线程池的状态变为SHUTDOWN状态;

3、所有在调用shutdown()方法之前提交到ExecutorSrvice的任务都会执行;

4、一旦所有线程结束执行当前任务，ExecutorService才会真正关闭。

### shutdownNow()

1、该方法返回尚未执行的 task 的 List;

2、线程池的状态变为STOP状态;

3、阻止所有正在等待启动的任务, 并且停止当前正在执行的任务。

# 浅拷贝和深拷贝

深复制和浅复制最根本的区别在于是否是真正获取了一个对象的复制实体，而不是引用。

深拷贝和浅拷贝是只针对Object和Array这样的引用数据类型的。

浅复制 —-只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。

深复制 —-在计算机中开辟了一块新的内存地址用于存放复制的对象。

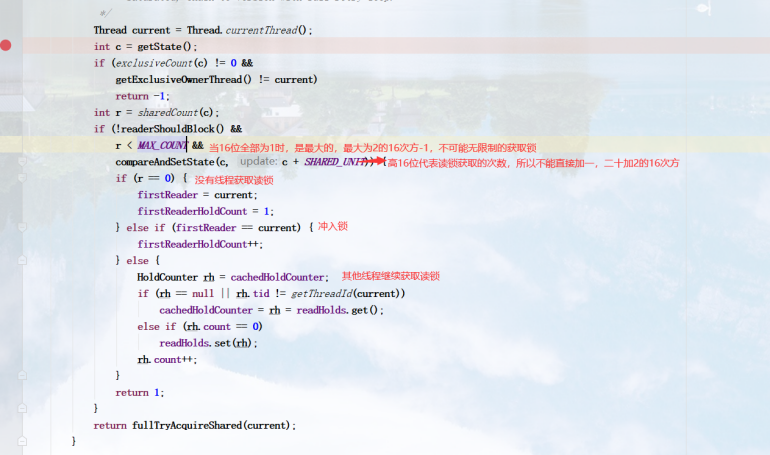
通俗一点理解就是浅拷贝出来的数据并不独立，如果被复制的对象改变了，那么浅拷贝的对象也会改变，深拷贝之后就会完全独立，与浅拷贝断绝关系。

# 多线程之间需要等待协调，才能完成某种工作，问怎么设计这种协调方案？如：子线程循环10次，接着主线程循环100，接着又回到子线程循环10次，接着再回到主线程又循环100，如此循环50次。

|  |
| --- |
| *\*/* **public class** WaitNotifyTest {  */\*\*  \* 多线程之间需要等待协调，才能完成某种工作，问怎么设计这种协调方案？  \* 如：子线程循环10次，接着主线程循环100，接着又回到子线程循环10次，接着再回到主线程又循环100，如此循环50次。  \*/* **private static** Object *object* = **new** Object();   **private static final int *num*** = 100;   **public static void** main(String[] args) {  Thread t = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** m = 0; m < ***num***; m++) {  **synchronized** (*object*) {  **try** {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*() + **"第"** + (m + 1) + **"次:"** + i);  }  System.***out***.println(Thread.*currentThread*() + **"第"** + (m + 1) + **"次运行完毕"**);   *object*.notifyAll();  *object*.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }, **"子线程"**);  t.start();   **for** (**int** m = 0; m < ***num***; m++) {  **synchronized** (*object*) {  **try** {  *object*.wait();   **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*() + **"第"** + (m + 1) + **"次:"** + i);  }  System.***out***.println(Thread.*currentThread*() + **"第"** + (m + 1) + **"次运行完毕"**);   *object*.notifyAll();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  } } |

# 读写锁

读读共享；读写互斥；写写互斥



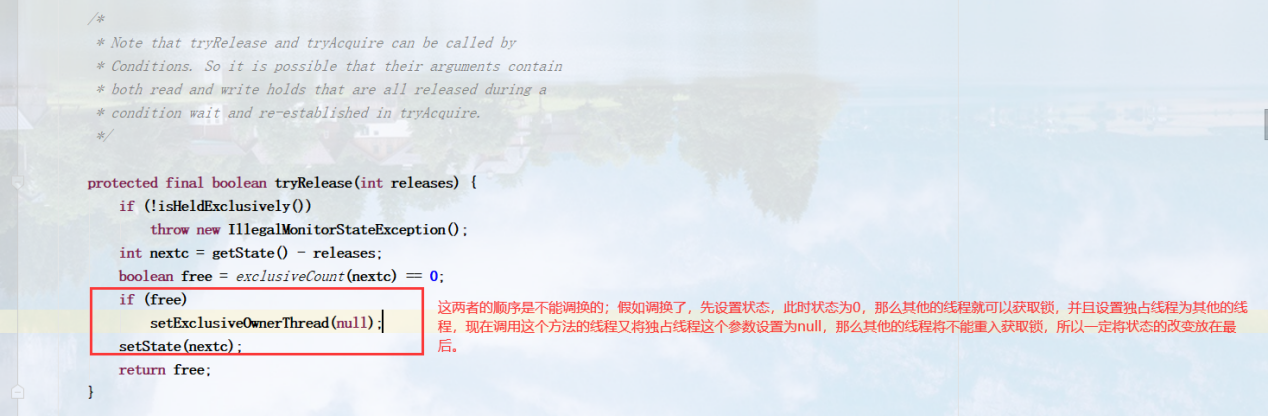
1. 获取当前的线程
2. 获取当前的同步状态
3. 判断写锁的数量!=0并且当前锁的持有者也不是当前线程，就直接返回-1，表示获取锁失败
4. 获取读锁的数量
5. 判断是否被阻塞(如果同步队列中有元素，并且该元素中持有的线程是独占模式，就需要被阻塞)并且读锁的数量是否超过了最大数量并且利用CAS去修改state的值，如果以上三个提交都成立，说明获取锁成功:(1)判断当前读锁是否被持有，没有被持有，那么该线程就是第一个获取读锁的线程，将当前线程的地址保存到firstreader中，并且将数量设置为1
6. 如果读锁已经被持有并且是被当前线程持有，直接将数量加1(3)如果以上提交都不成立，那么就是有其他的线程来获取读锁，cachedHoldCounter保存的是上一个获取锁的线程，如果为null或者该变量持有的线程不是当前线程，那么需要重新创建HoldCounter，readHolds是一个继承了ThreadLocal，已经被初始化，直接get就行，之后将数量+1
7. 如果获取锁失败，直接进入fullTryAcquireShared方法:



以下这段代码主要解决是否是重入获取锁



## 写锁的释放

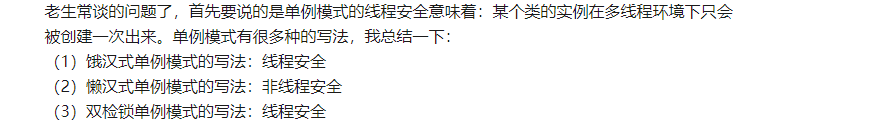


## 获取锁的几种可能的情况

1. A线程获取读锁，B线程获取写锁将被阻塞，C线程获取读锁也将被阻塞。这样可以防止B线程出现饥饿的情况（比如A获取读锁，B获取写锁被阻塞，但是后面又十几个线程来获取读锁，假设可以获取，那么写锁将一直被阻塞，可能发生饥饿）
2. A线程获取读锁，C线程获取读锁可以获取，B线程获取写锁将被阻塞，C线程获取读锁可以获取，因为在之前曾经获取过。
3. A线程获取写锁，B线程获取读锁将被阻塞，此时A线程释放写锁，还没有去还清B线程，C线程获取读锁，B线程在同步队列中，由于等待获取读锁，所以C线程无需阻塞，直接取尝试获取，读读之间是共享的。

# 单例模式





# swtich是否能作用在byte上，是否能作用在long上，是否能作用在string上?

switch语句中的表达式只能是整数类型，即必须是int、char或者枚举类型数据。不能是boolean或浮点型，甚至其他类型的整数数据(byte,short及long)。

# 死锁

## 什么是死锁

死锁是线程彼此持有对方需要的资源，但是又不愿意放弃自己持有的资源

## 产生死锁的必要条件

* 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。
* 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。
* 环路等待条件：在发生死锁时，必然存在一个进程--资源的环形链。

## 解决死锁的基本方法

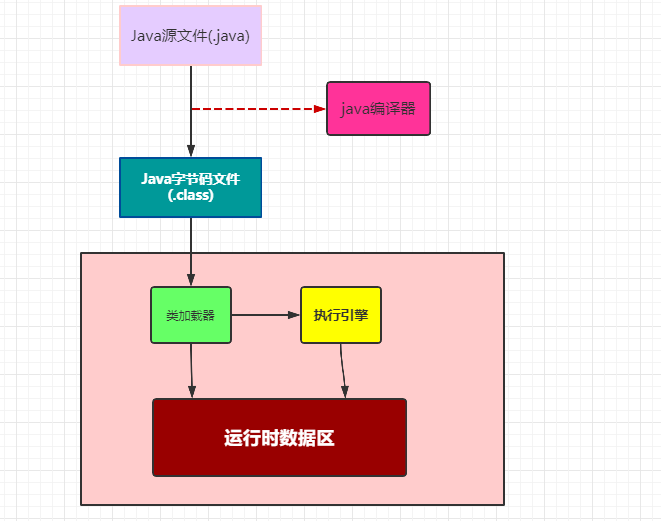
1. 以确定的顺序获取死锁
2. 超时放弃

## 解除死锁

* 剥夺资源：从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态；
* 撤消进程：可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态.消除为止；所谓代价是指优先级、运行代价、进程的重要性和价值等。

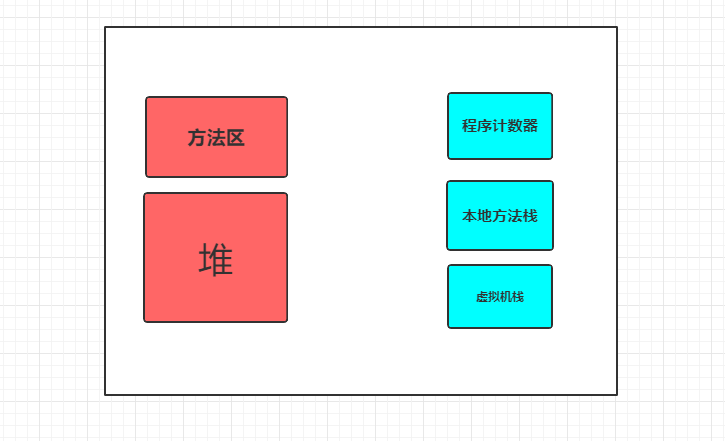
# JVM

## JVM的内存机制



如上图所示，首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后缀)，然后由JVM中的类加载器加载各个类的字节码文件，加载完毕之后，交由JVM执行引擎执行。在整个程序执行过程中，JVM会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一般被称作为Runtime Data Area（运行时数据区），也就是我们常说的JVM内存。因此，在Java中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

### 运行时数据区包括哪几部分



#### 程序计数器

程序计数器（Program Counter Register），也有称作为PC寄存器。想必学过汇编语言的朋友对程序计数器这个概念并不陌生，在汇编语言中，程序计数器是指CPU中的寄存器，它保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。

　　虽然JVM中的程序计数器并不像汇编语言中的程序计数器一样是物理概念上的CPU寄存器，但是JVM中的程序计数器的功能跟汇编语言中的程序计数器的功能在逻辑上是等同的，也就是说是用来指示 执行哪条指令的。

　　由于在JVM中，多线程是通过线程轮流切换来获得CPU执行时间的，因此，在任一具体时刻，一个CPU的内核只会执行一条线程中的指令，因此，为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。因此，可以这么说，程序计数器是每个线程所私有的。

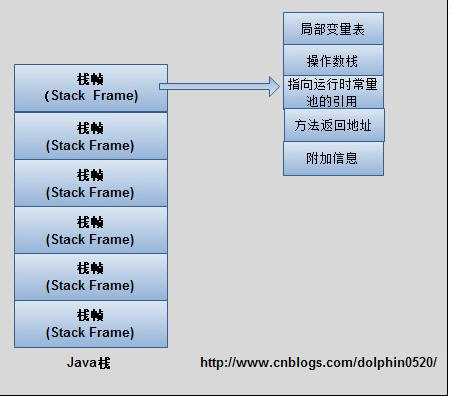
　　在JVM规范中规定，如果线程执行的是非native方法，则程序计数器中保存的是当前需要执行的指令的地址；如果线程执行的是native方法，则程序计数器中的值是undefined。

　　由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的。

#### Java栈

　Java栈也称作虚拟机栈（Java Vitual Machine Stack），也就是我们常常所说的栈，跟C语言的数据段中的栈类似。事实上，Java栈是Java方法执行的内存模型。为什么这么说呢？下面就来解释一下其中的原因。

　　Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池（运行时常量池的概念在方法区部分会谈到）的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。讲到这里，大家就应该会明白为什么 在 使用 递归方法的时候容易导致栈内存溢出的现象了以及为什么栈区的空间不用程序员去管理了（当然在Java中，程序员基本不用关系到内存分配和释放的事情，因为Java有自己的垃圾回收机制），这部分空间的分配和释放都是由系统自动实施的。对于所有的程序设计语言来说，栈这部分空间对程序员来说是不透明的。下图表示了一个Java栈的模型：



局部变量表，顾名思义，想必不用解释大家应该明白它的作用了吧。就是用来存储方法中的局部变量（包括在方法中声明的非静态变量以及函数形参）。对于基本数据类型的变量，则直接存储它的值，对于引用类型的变量，则存的是指向对象的引用。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

　　操作数栈，想必学过数据结构中的栈的朋友想必对表达式求值问题不会陌生，栈最典型的一个应用就是用来对表达式求值。想想一个线程执行方法的过程中，实际上就是不断执行语句的过程，而归根到底就是进行计算的过程。因此可以这么说，程序中的所有计算过程都是在借助于操作数栈来完成的。

　　指向运行时常量池的引用，因为在方法执行的过程中有可能需要用到类中的常量，所以必须要有一个引用指向运行时常量。

　　方法返回地址，当一个方法执行完毕之后，要返回之前调用它的地方，因此在栈帧中必须保存一个方法返回地址。

　　由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰。

#### 本地方法栈

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和Java栈合二为一。

#### 堆

　在C语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。那么在Java中是怎么样的呢？

　　Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

#### 方法区

　方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

　　在Class文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池，用来存储编译期间生成的字面量和符号引用。

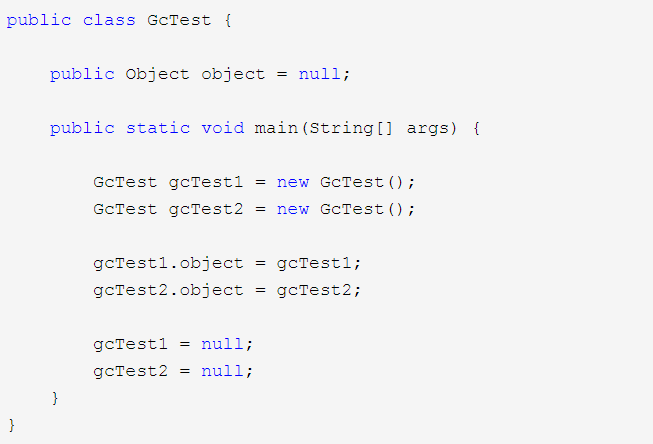
　　在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法。

　　在JVM规范中，没有强制要求方法区必须实现垃圾回收。很多人习惯将方法区称为“永久代”，是因为HotSpot虚拟机以永久代来实现方法区，从而JVM的垃圾收集器可以像管理堆区一样管理这部分区域，从而不需要专门为这部分设计垃圾回收机制。不过自从JDK7之后，Hotspot虚拟机便将运行时常量池从永久代移除了。

## 垃圾回收机制

在java中是通过引用来和对象进行关联的，也就是说如果要操作对象，必须通过引用来进行。那么很显然一个简单的办法就是通过引用计数来判断一个对象是否可以被回收。不失一般性，如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到，那么这个对象就成为可被回收的对象了。这种方式成为引用计数法。

　　这样的方法简单粗暴，而且效率很高。效率高必然会暴露一些问题，如果某些对象呗循环引用，即使你把对象赋值为null，这种算法照样不能回收。看下下面的代码



虽然gcTest1，gcTest2是null，他们指向的对象已经不会被访问到了，但是由于它们互相引用对方，导致它们的引用计数都不为0，那么垃圾收集器就永远不会回收它们。

　　上面的问题已经暴露出来了，下面看看jvm是怎么解决这个问题的。为了解决这个问题，在Java中采取了可达性分析法。该方法的基本思想是通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。在《深入理解jvm》讲解的很仔细，笔者就简单介绍下GC Roots的概念，想深入了解的可以去读下笔者介绍的这本书。

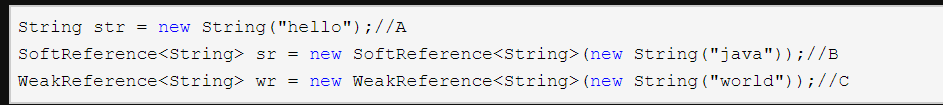
　　以下三类对象在jvm中作为GC roots，来判断一个对象是否可以被回收 (通常来说我们只要知道虚拟机栈和静态引用就够了)

 　　1、虚拟机栈(JVM stack)中引用的对象(准确的说是虚拟机栈中的栈帧(frames)) 。我们知道，每个方法执行的时候，jvm都会创建一个相应的栈帧(栈帧中包括操作数栈、局部变量表、运行时常量池的引用)，栈帧中包含这在方法内部使用的所有对象的引用(当然还有其他的基本类型数据)，当方法执行完后，该栈帧会从虚拟机栈中弹出，这样一来，临时创建的对象的引用也就不存在了，或者说没有任何gc roots指向这些临时对象，这些对象在下一次GC时便会被回收掉

　　2、方法区中类静态属性引用的对象 。静态属性是该类型(class)的属性，不单独属于任何实例，因此该属性自然会作为gc roots。只要这个class存在，该引用指向的对象也会一直存在。class 也是会被回收的，在面后说明

 　　3、本地方法栈(Native Stack)引用的对象

 　　下面介绍下关于软引用（softReference）和弱引用（weakReference）的对象垃圾回收对他们做的处理



上面的几个对象中回收情况如下，B在内存不足的情况下会将String对象判定为可回收对象，C无论什么情况下String对象都会被判定为可回收对象。也就是说软引用会在内存溢出（OOM）的时候回收，而弱引用无论什么情况都会在下一轮回收的时候回收掉。

　　一般jvm会对这些对象回收

　　1、显示地将某个引用赋值为null或者将已经指向某个对象的引用指向新的对象。

　　2、局部引用所指向的对象。

　　3、上面说的弱引用（weakReference）。

### 垃圾收集算法

#### Mark-Sweep（标记-清除）算法

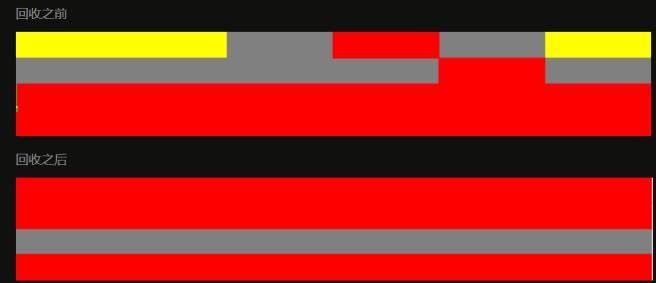
这是最基础的垃圾回收算法，之所以说它是最基础的是因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。图解来自网络，很好的说明了标记-清楚算法的处理前和处理后的内存分布。



很容易看出这样的操作是有弊端的，这样讲标记的对象的清楚后，内存块就变的零零散散，如果现在有一个对象占用的内存很大，这个时候必须要在执行一遍垃圾回收，为这个大的对象腾出空间。

#### Copying（复制）算法

为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。



复制算法会提前空出一般的内存，在垃圾回收的时候将存活的对象移动的另外一半内存，这样内存的移动消耗太大，虽然内存不是零散的，但是代价太高。

#### Mark-Compact（标记-整理）算法

为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。具体过程如下图所示：



#### Generational Collection（分代收集）算法

　分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，并不是回收所有，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。可以调用System.gc()方法查看回收情况。

　　目前大部分垃圾收集器对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

　　而由于老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。

　　注意，在堆区之外还有一个代就是永久代（Permanet Generation），它用来存储class类、常量、方法描述等。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

### 典型的垃圾收集器

　　下面都是些概率性的东西，笔者看得也似懂非懂，直接搬过来分享给大家

## **1.Serial/Serial Old**

　　Serial/Serial Old收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Copying算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

## **2.ParNew**

　　ParNew收集器是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

## **3.Parallel Scavenge**

　　Parallel Scavenge收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其采用的是Copying算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

## **4.Parallel Old**

　　Parallel Old是Parallel Scavenge收集器的老年代版本（并行收集器），使用多线程和Mark-Compact算法。

## **5.CMS**

　　CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

## **6.G1**

　　G1收集器是当今收集器技术发展最前沿的成果，它是一款面向服务端应用的收集器，它能充分利用多CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

### 总结和补充

　对象的内存分配，往大方向上讲就是在堆上分配，对象主要分配在新生代的Eden Space和From Space，少数情况下会直接分配在老年代。如果新生代的Eden Space和From Space的空间不足，则会发起一次GC，如果进行了GC之后，Eden Space和From Space能够容纳该对象就放在Eden Space和From Space。在GC的过程中，会将Eden Space和From  Space中的存活对象移动到To Space，然后将Eden Space和From Space进行清理。如果在清理的过程中，To Space无法足够来存储某个对象，就会将该对象移动到老年代中。在进行了GC之后，使用的便是Eden space和To Space了，下次GC时会将存活对象复制到From Space，如此反复循环。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。

　　一般来说，大对象会被直接分配到老年代，所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组，比如：

　　byte[] data = new byte[4\*1024\*1024]

　　这种一般会直接在老年代分配存储空间。

　　当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM的相关参数。

## 类加载

需要看一下深入理解java虚拟机

# short s1=1;s1=s1+1；有什么错?short s1=1；s1+=1；有什么错?

面试题都是很变态的，要做好受虐的准备。

s1=s1+1会出错，s1+1是int型，不能将int赋值给s1。需要显示转换，s1=(int)(s1+1)，而s1+=1不会出错，至于原因，有人说和编译器的机制有关，需要看编译原理，话说编译原理什么的最讨厌了，就这样吧。