**Java基础**

# Java经典面试题

<https://blog.csdn.net/z784561257/article/details/93634062>

# HashMap

## 转换成红黑树的条件

## 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。

负载因子的值决定了HashMap中元素的数量达到一个什么临界值的时候就开始进行扩容，临界值=负载因子\*容量，当HashMap中元素的数量越来越多的时候，发生碰撞的概率就会变大，查询的效率就会降低，减少碰撞的概率有两种方式：扩容，选择好的Hash算法，可以将两者进行结合，在合适的时机进行扩容，选择好的Hash算法将元素均匀的分配到每个桶中。

负载因子=0.75是一个时间成本和空间成本的之间的很好的权衡。如果值过大，那么会节省空间，但是碰撞的概率变大了，查询的效率就变低了，如果过小，就会频繁的发生扩容，浪费空间。比如负载因子为1或者0.5的时候，为1的时候，扩容的次数少了，但是碰撞的概率大了，为0.5的时候，扩容的次数多了，浪费一半的存储空间。0.75介于0.5和1之间，刚好合适，而且容量为2的幂，乘以0.75刚好是个整数，所以选择0.75更加合适。

## 线程安全问题

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

## 谈谈JDK1.8下的HashMap在并发情况下链表成环的过程

jdk1.8下的hashmap采用的是尾插法，不会有链表成环的问题。jdk1.7下采用的头插法，会有链表成环的问题。

# 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

# List和Set的区别

List是可以存放重复的元素，是个有序的容器。

Set是个无序的容器，不可以存放重复的元素。

# 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

<https://juejin.im/post/5eaecf326fb9a043856f33d5>

## 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。

## 注意的

   我们知道，线程Thread对象中，每个线程对象内部都有一个的ThreadLocalMap对象。如果这个对象存储了多个大对象，则可能早出内存溢出OOM。为了防止这种情况发生，在ThreadLocal的源码中，有对应的策略，即调用 get（）、set（）、remove（） 方法，均会清除 ThreadLocal内部的 内存。

         ThreadLocal的内部是ThreadLocalMap。ThreadLocalMap内部是由一个Entry数组组成。Entry类的构造函数为 Entry（弱引用的ThreadLocal对象， Object value对象）。因为Entry的key是一个弱引用的ThreadLocal对象，所以在 垃圾回收 之前，将会清除此Entry对象的key。那么， ThreadLocalMap 中就会出现 key 为 null 的 Entry，就没有办法访问这些 key 为 null 的 Entry 的 value。这些 value 被Entry对象引用，所以value所占内存不会被释放。若在指定的线程任务里面，调用ThreadLocal对象的get（）、set（）、remove（）方法，可以避免出现内存泄露。

## 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

Java并发编程实战第169页。

## 死锁

每个线程都拥有其他线程需要的资源，并且等待其他线程已经拥有的资源，并且在获得所有需要的资源之前不放弃已经拥有的资源。

之所以短时间的持有锁，是为了降低锁的竞争程度，但却增加了在测试中找出潜在死锁风险的难度。

## 锁顺序死锁

两个线程试图以不同的顺序获取相同的锁，如果以相同的顺序获取锁，那么就不会出现锁顺序死锁问题。

## 动态的锁顺序死锁

## 在协作对象之间发生的死锁

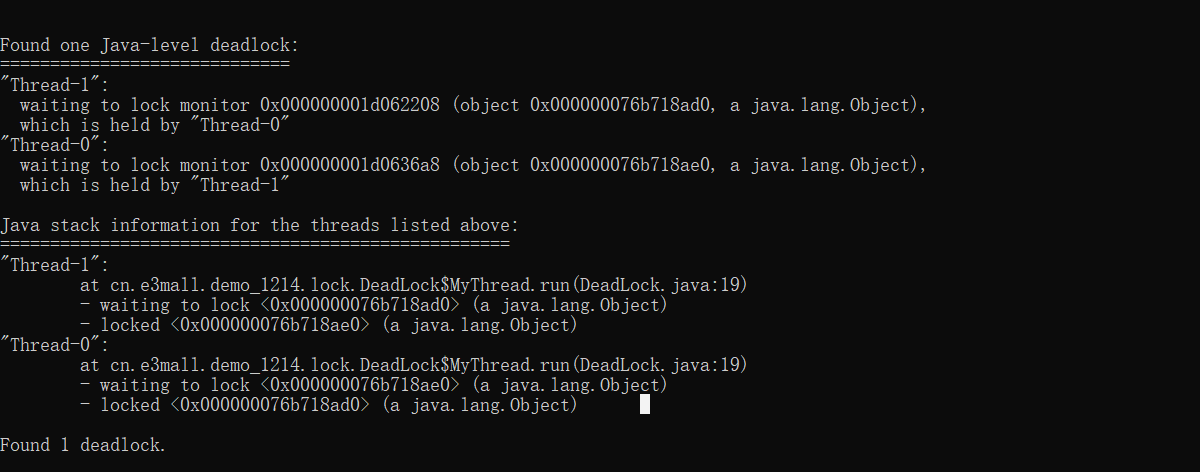
如果在持有锁时调用某个外部方法，那么将出现活跃性问题。在这个外部方法中可能会获取其他锁（这可能会产生死锁），或者阻塞时间过长，导致其他线程无法及时获取当前被持有的锁。

## 如何解决死锁

使用定时的锁，Lock类中tryLock功能，可以设置超时间，获取锁超时后，会返回失败信息，自动放弃锁。

通过线程转储信息来分析死锁。





## synchronized、lock、reentrantlock的区别、用法及原理

### 如何选择

在一些内置锁无法满足需求的情况下，ReentrantLock可以作为一种高级工具。当需要一些高级功能时才应该使用RetreentLock,这些功能包括：可定时的，可轮询的，可中断的锁获取操作，公平锁，以及非快结构的锁，否则，还是应该使用synchronized。

在JDK1.5中，内置锁与ReentrantLock相比还有一个优点：在线程转储中能给出在哪些调用帧中获得了哪些锁，并能够检测死锁和识别发生的死锁。

## 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。

# 自旋锁，轻量级锁，重量级锁，偏向锁

1.偏向锁

线程A请求对象F获取锁时，此时，F还没有被锁定过，锁标志为无锁状态，是否偏向状态为否，此时，A线程获取到锁，锁状态设置为偏向锁，是否偏向状态设置为是，并在Mark Word中设置线程A的ID，表示

线程A获取到偏向锁，存线程A的ID的意义在于，线程A再次请求锁时，检测到MarkWord中已经存在相同的ID，表示A已经获得了偏向锁，直接执行同步代码即可，偏向的意义就在此体现，如果获取失败，进行下一步

2.轻量级锁

线程B请求锁时，发现同步锁处于偏向状态，CAS获取锁，获取成功，将MarkWord中的线程ID替换为线程B的ID；获取失败，则撤销线程A的偏向锁，表示当前有一定的竞争，升级为轻量级锁，过程如下：

在线程B的栈中开辟一片空间P，操作一，保存指向MarkWord的指针；操作二，MarkWord中保存指向空间P的指针，如果上述两个操作都成功，表示线程B获取锁成功，MarkWord中的锁状态设置为轻量级锁

如果获取锁失败，进行下一步

3.自旋锁

自旋锁只是一种不断尝试获取轻量级锁的过程。如果重试一定次数后获取到锁，就执行同步代码，否则进行下一步

4.重量级锁

自旋锁重试之后如果抢锁依然失败，同步锁会升级至重量级锁，重量级锁就是java最原始的同步锁，在这个状态下，未抢到锁的线程都会被阻塞，等待被激活。

以上是JVM提高锁效率的方式在应用层，我们也可以采取一定的措施来提高锁效率

1，减少锁的持有时间

不需要同步执行的代码，不要放在同步代码块中。同步块中代码减少，锁的持续时间短，锁的性能会有所提高。

2，减小锁粒度

把资源分批使用不同的锁，不同批次的资源的操作互不影响。

比如ConturrentHashMap类，把map分成多段，每段一个锁，不在一段的数据可以同时修改。

3，锁分离

把关系不大的操作使用不同的锁，使这些操作互不影响。

比如LinkedBlockingQueue类，从队列头获取数据的take()方法和从队列末尾添加数据的put()方法分别使用不同的锁，两者互不影响。

4，锁粗化

指的是当虚拟机需要连续对同一把锁进行加锁和释放时，尽量改成只使用一次锁。

比如连续多个synchronized语句块，或循环中的synchronized语句块，用的是同一个对象作为锁，可以直接用一个synchronized语句块把他们都包含起来。

5，弃用synchronized关键字

不使用synchronized关键字，可以自己编写代码实现类似偏向锁、自旋锁的功能，减少因为同步锁而带来的效率损耗。比如自己实现的自旋锁。

# 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

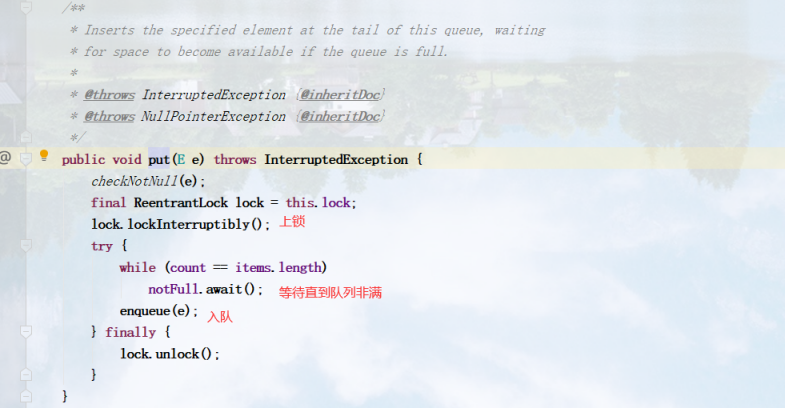
内部成员变量

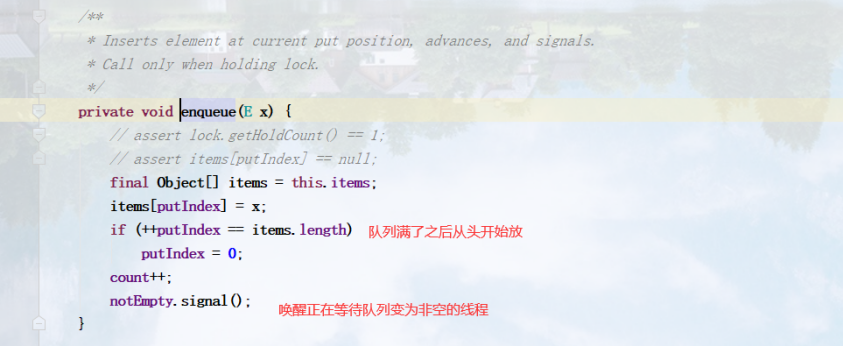
是通过Lock和condition实现的，内部的存储的数据结构为数组



### Put方法

不能传入null值

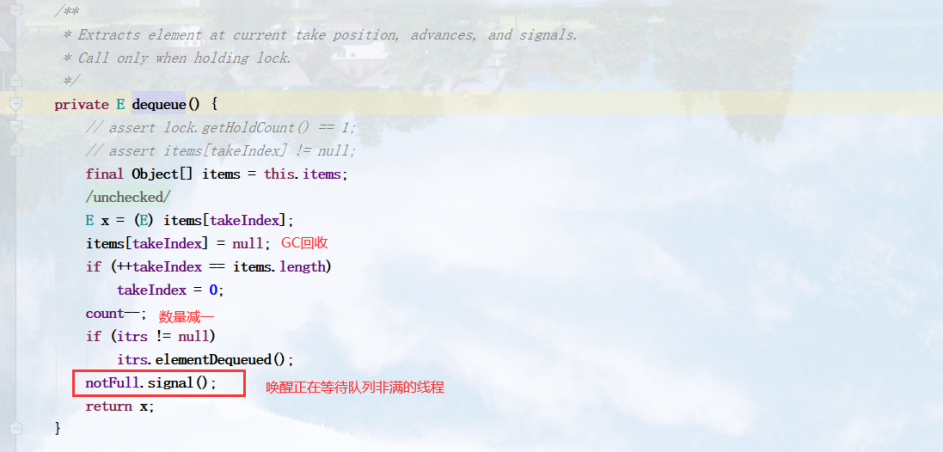




### take方法

会阻塞





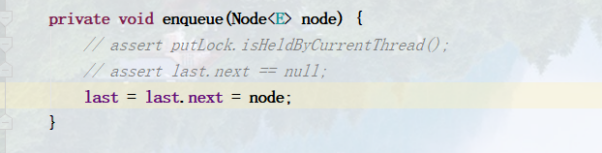
## LinkedBlockingQueue

### 内部结构



### put





### take

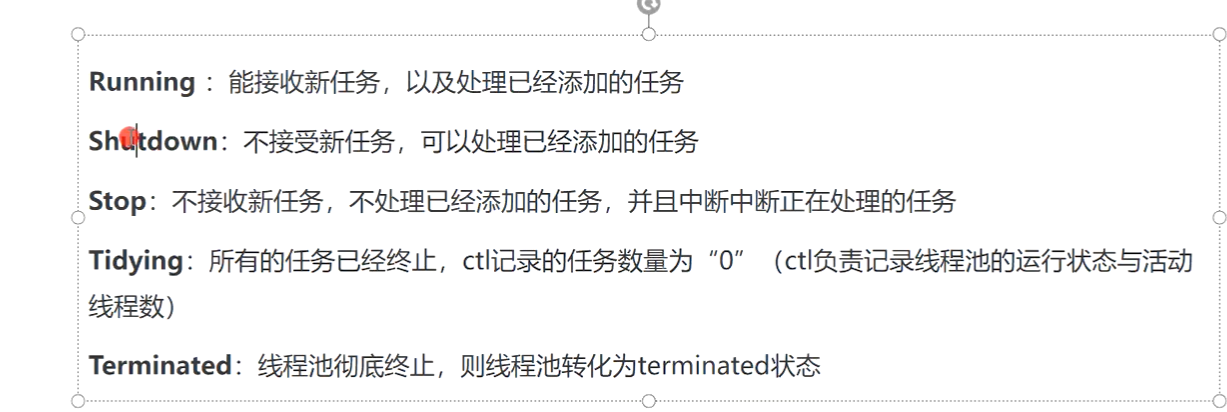


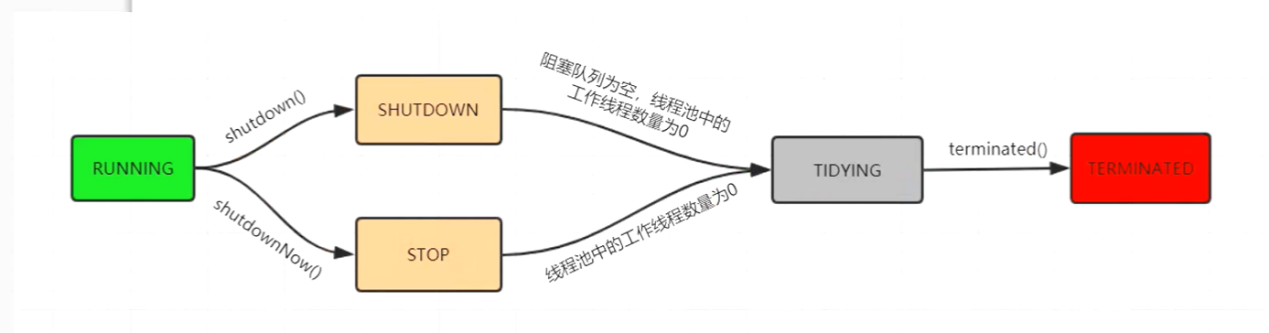
# 线程池

<https://juejin.im/post/5eae9c936fb9a0437a4d2e8d>



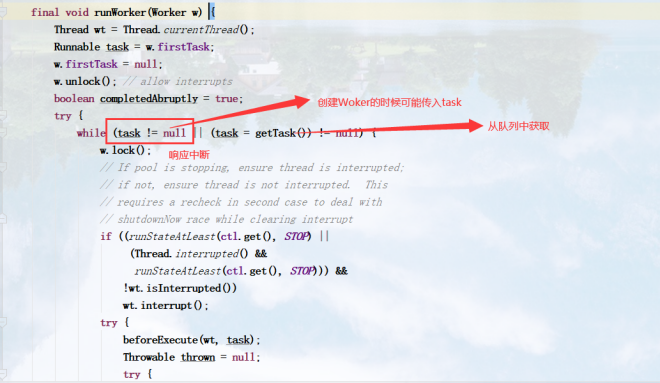
## 为什么要用线程池





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

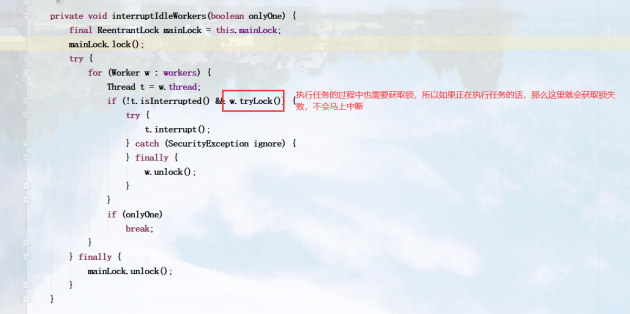
ThreadPoolExecutor.CallerRunsPolicy() 由向线程池提交任务的线程来执行该任务

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。

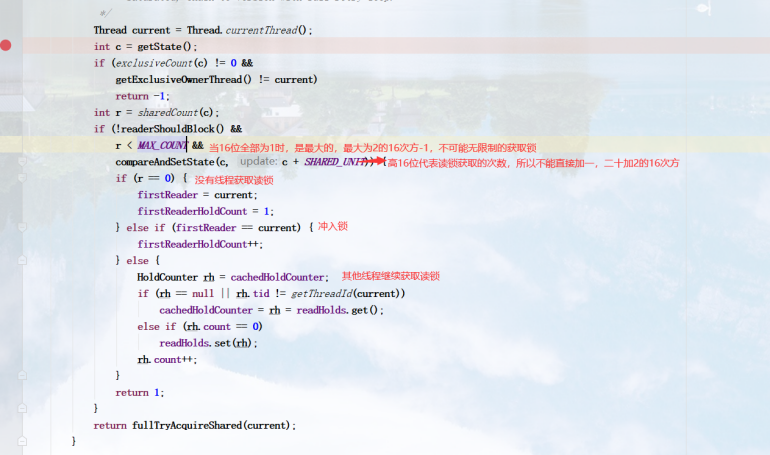
通俗一点理解就是浅拷贝出来的数据并不独立，如果被复制的对象改变了，那么浅拷贝的对象也会改变，深拷贝之后就会完全独立，与浅拷贝断绝关系。

# 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



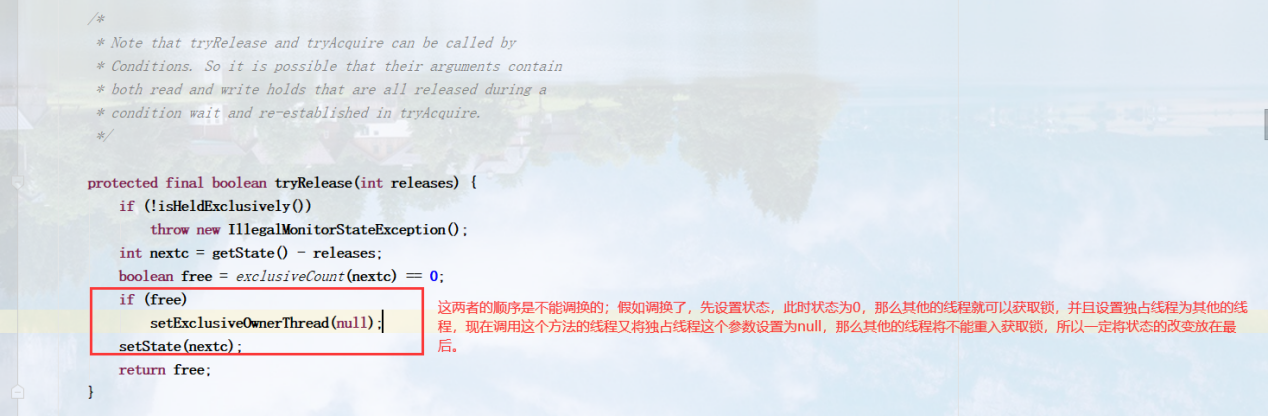
1. 获取当前的线程
2. 获取当前的同步状态
3. 判断写锁的数量!=0并且当前锁的持有者也不是当前线程，就直接返回-1，表示获取锁失败
4. 获取读锁的数量
5. 判断是否被阻塞(如果同步队列中有元素，并且该元素中持有的线程是独占模式，就需要被阻塞)并且读锁的数量是否超过了最大数量并且利用CAS去修改state的值，如果以上三个提交都成立，说明获取锁成功:(1)判断当前读锁是否被持有，没有被持有，那么该线程就是第一个获取读锁的线程，将当前线程的地址保存到firstreader中，并且将数量设置为1
6. 如果读锁已经被持有并且是被当前线程持有，直接将数量加1(3)如果以上提交都不成立，那么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。

## 有没有比读写锁更快的锁？

<https://blog.csdn.net/qq_31706095/article/details/106054883>

StampedLock横空出世。

StampedLock其实是对读写锁的一种改进，它支持在读同时进行一个写操作,也就是说，它的性能将会比读写锁更快。

更通俗的讲就是在读锁没有释放的时候是可以获取到一个写锁，获取到写锁之后，读锁阻塞，这一点和读写锁一致，唯一的区别在于读写锁不支持在没有释放读锁的时候获取写锁。

### StampedLock三种模式

悲观读：与读写锁的读写类似，允许多个线程获取悲观读锁

写锁：与读写锁的写锁类似，写锁和悲观读是互斥的。

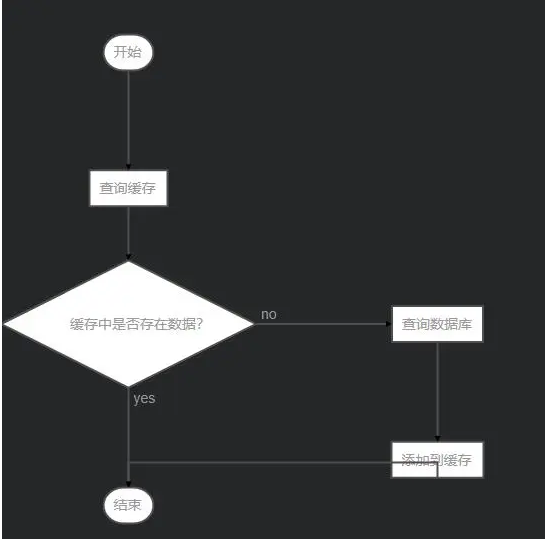
乐观读：无锁机制，类似于数据库中的乐观锁，它支持在不是放写锁的时候是可以获取到一个写锁的，这点和读写锁不同。

### 基本语法

## 什么是读写锁

允许多个线程同时读，但是只允许一个线程写，在线程获取到写锁的时候，其他写操作和读操作都会处于阻塞状态，读锁和写锁也是互斥的，所以在读的时候是不允许写的，那如何实现一个读写锁呢？

读写锁比传统的synchronized速度要快很多，原因就是在于读写锁支持读并发，而synchronized要求所有操作都是串行化，举个例子，我需要查询某个用户的基本信息，这些信息很少发生变化，所以我们会将这部分信息存放到缓存中，我们的查询操作为：

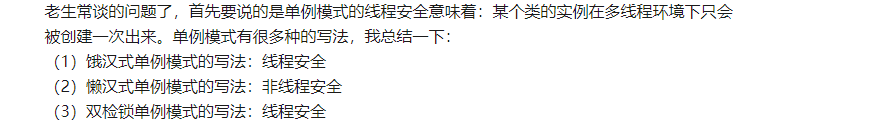


按照上面流程图，如果使用synchronized的时候，查询缓存都会阻塞，但是使用读写锁，查询缓存时并发的，查询数据库是阻塞的，所以，读写锁在读多写少的情况下，性能明显要优于synchronized。

人类的文明在进步，java也在进步，对知识的渴望也在不断的增加，所以我们就不断的在想这么一个问题，读写锁的读和写是互斥，那我们能不能做到读和写支持并发呢？

# 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。
* 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：进程请求和保持条件：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。

# 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。

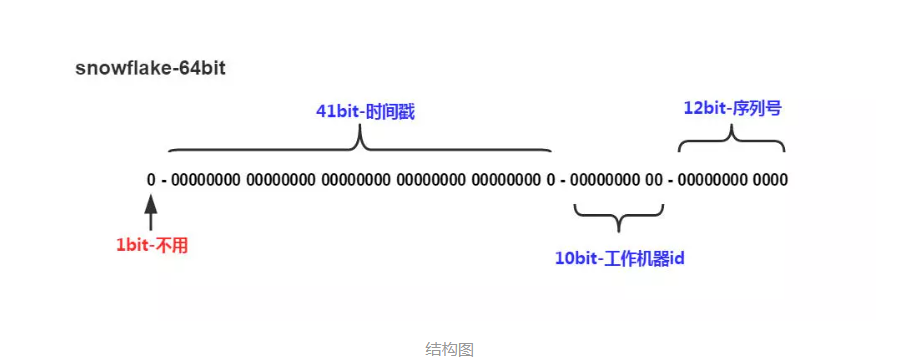
s1=s1+1会出错，s1+1是int型，不能将int赋值给s1。需要显示转换，s1=(int)(s1+1)，而s1+=1不会出错，至于原因，有人说和编译器的机制有关，需要看编译原理，话说编译原理什么的最讨厌了，就这样吧。

# CAS操作的原理

<https://blog.csdn.net/lixinkuan328/article/details/94319775>

# 雪花算法

SnowFlake算法生成id的结果是一个64bit大小的整数，它的结构如下图：



**1bit**，不用，因为二进制中最高位是符号位，1表示负数，0表示正数。生成的id一般都是用整数，所以最高位固定为0。

**41bit-时间戳**，用来记录时间戳，毫秒级。  
- 41位可以表示IMG_256个数字，  
- 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 IMG_257，减1是因为可表示的数值范围是从0开始算的，而不是1。  
- 也就是说41位可以表示IMG_258个毫秒的值，转化成单位年则是IMG_259年

**10bit-工作机器id**，用来记录工作机器id。  
- 可以部署在IMG_260个节点，包括5位datacenterId和5位workerId  
- 5位（bit）可以表示的最大正整数是IMG_261，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId

**12bit-序列号**，序列号，用来记录同毫秒内产生的不同id。  
- 12位（bit）可以表示的最大正整数是IMG_262，即可以用0、1、2、3、....4094这4095个数字，来表示同一机器同一时间截（毫秒)内产生的4095个ID序号。

由于在Java中64bit的整数是long类型，所以在Java中SnowFlake算法生成的id就是long来存储的。