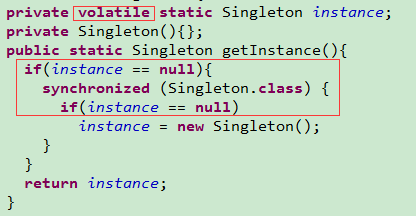
# 第一章 单例模式



注意：双重原因:如果线程一执行到第二个if判断时,切换到线程二,这时instance仍是null,则会出现两个instance.

**volatile**原因：instance = new Singleton();分解为三步：(1)分配对象的内存空间;(2)初始化对象;(3)设置instance指向刚分配的内存地址；2、3会重排，导致获取到未初始化的非null对象。**volatile屏蔽指令重排序的语义在JDK1.5才被完全恢复。**

# 第二章 生产者消费者

public class ArrayBlockingQueueDemo {

private final static ArrayBlockingQueue<Apple> queue= new ArrayBlockingQueue<>(1);

public static void main(String[] args){

new Thread(new Producer(queue)).start();

new Thread(new Producer(queue)).start();

new Thread(new Consumer(queue)).start();

new Thread(new Consumer(queue)).start();

}

}

class Apple {

public Apple(){

}

}

/\*\*

\* 生产者线程

\*/

class Producer implements Runnable{

private final ArrayBlockingQueue<Apple> mAbq;

Producer(ArrayBlockingQueue<Apple> arrayBlockingQueue){

this.mAbq = arrayBlockingQueue;

}

@Override

public void run() {

while (true) {

Produce();

}

}

private void Produce(){

try {

Apple apple = new Apple();

mAbq.put(apple);

System.out.println("生产:"+apple);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 消费者线程

\*/

class Consumer implements Runnable{

private ArrayBlockingQueue<Apple> mAbq;

Consumer(ArrayBlockingQueue<Apple> arrayBlockingQueue){

this.mAbq = arrayBlockingQueue;

}

@Override

public void run() {

while (true){

try {

TimeUnit.MILLISECONDS.sleep(1000);

comsume();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

private void comsume() throws InterruptedException {

Apple apple = mAbq.take();

System.out.println("消费Apple="+apple);

}

}

# 第四章 Java并发机制的底层实现

## 2.1 volatile的应用

Volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”。即当一个线程修改了一个被volatile修饰共享变量的值，新值总数可以被其他线程立即得知。Volatile不会引起线程上下文的切换和调度。

### volatile的特性

1) 可见性：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程从主内存中读取共享变量。

(1) 如果对声明了volatile的变量进行写操作，JVM会向处理器发送一条LOCK前缀的指令，将这个变量所在缓存行的数据写回到系统内存。

(2) 一个处理器的缓存回写到内存会导致其他处理器里缓存了该内存地址的数据无效。

2)原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

3)禁止指令重排序优化【1.5才完全恢复】。

## 2.2 synchronized的应用

JavaSE1.6对synchronized进行了各种优化，有些情况下他就并不那么重了。

### 1、Java中的每一个对象都可以作为锁

* 1. 对于普通同步方法，锁是当前实例对象
  2. 对于静态同步方法，锁是当前类的class对象
  3. 对于同步方法块，锁是synchronized括号里配置的对象

### 2、JVM实现代码块同步

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步。代码块同步是使用monitorenter和monitorexit指令实现的，而同步方法是依靠方法修饰符上的ACC\_SYNCHRONIZED来完成的，实质是对一个对象的监视器（monitor）进行获取。Monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处。任何对象都有一个monitor预知关联，当且一个monitor被持有后，它将处于锁定状态。线程指向到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁。

### 3、Java对象头

Synchronized用的锁是存在Java对象头里。如果对象是数组类型，则用3个字宽存储对象头，其他使用2字宽存储对象头。

Java对象头里的mark word里默认储存对象的hashcode、分代年龄和锁标记位。mark word 里存储的数据会随着锁标志位的变化而变化。锁标志位分为：轻量级锁00、重量级锁10、GC标记11、偏向锁01。

### 4、锁的升级与对比

在JavaSE1.6中，**引入了轻量级锁和偏向锁**。锁一共有4中状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁、重量级锁。锁可以升级但不能降级。

* 1. 偏向锁：偏向锁就是在**无竞争的情况下把整个同步都消除掉**，连CAS都不做了。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

【当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为“01”，即偏向模式。同时使用CAS操作把获取到这个锁的线程的ID记录在对象的mark word之中，如果CAS操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块，虚拟机都可以不再进行任何同步操作，只需简单测试一下对象头的mark word里是否存储着指向当前线程的偏向锁】

* 1. 轻量级锁：轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

代码进入同步块的时候，如果此同步对象没有被锁定，虚拟机首先将在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的 mark word复制到锁记录中。然后，虚拟机将使用CAS操作将对象的mark word更新为指向锁记录的指针。如果更新成功，那么这个线程就拥有了该对象的锁，并且对象的锁标记位转变为00.如果更新失败，虚拟机首先会检查对象的 mark word是够指向当前线程的栈帧，如果是说明当前线程已经拥有了这个对象的锁，否则说明这个锁对象已经被其他线程抢占，当前线程便尝试使用自旋来获取锁。自旋获取失败，这时轻量级锁就不再有效，要膨胀为重量级锁，锁标志变为“10”。

* 1. 轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。
  2. 锁的优缺点对比

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 使用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗 | 适用于一个线程访问同步块场景 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度 | 如果始终得不到锁竞争的线程，适用自旋会消耗CPU | 追求响应时间，同步块执行速度非常快 |
| 重量级锁 | 线程竞争不适用自旋，不会消耗CPU | 线程阻塞，响应时间缓慢 | 追求吞吐量  同步块指向速度较长 |

## 2.3 原子操作的实现原理

无锁执行者CAS的核心算法原理，然后分析Java执行CAS的实践者Unsafe类，该类中的方法都是native修饰的，因此我们会以说明方法作用为主介绍Unsafe类，最后再介绍并发包中的Atomic系统使用CAS原理实现的并发类。

### 1、Java中实现原子操作

在Java中可以通过锁和循环CAS的方式来实现原子操作。从Java1.5开始，JDK的并发包里提供了一些来支持原子操作，如AtomicBoolean、AtomicInteger和AtomicLong。

### CAS

1. **Compare and Swap, 翻译成比较并交换。**

java.util.concurrent包中借助CAS实现了区别于synchronouse同步锁的一种乐观锁。

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许java调用其他语言。而compareAndSwapInt就是借助C来调用CPU底层指令实现的。因为CAS是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题。

1. **执行函数：CAS(V,E,N)**

其包含3个参数：V表示要更新的变量,E表示预期值,N表示新值.

如果V值等于E值，则将V的值设为N。若V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。通俗的理解就是CAS操作需要我们提供一个期望值，当期望值与当前线程的变量值相同时，说明还没线程修改该值，当前线程可以进行修改，也就是执行CAS操作，但如果期望值与当前线程不符，则说明该值已被其他线程修改，此时不执行更新操作，但可以选择重新读取该变量再尝试再次修改该变量，也可以放弃操作，原理图如下

### 3、CAS实现原子操作的三大问题

**1) ABA问题**：

CAS需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是A，变成B，又变成了A**，那么使用CAS进行检查时会发现它的值没有发生变化，实际上变化了。

ABA问题的解决思路就是：使用版本号。在变量前面追加上版本号。

从JDK1.5开始，JDK的Atomic包里提供了一个类**AtomicStampedReference来解决ABA问题**。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

**2)循环时间长开销大**

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

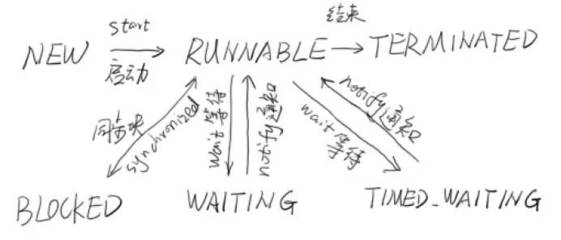
**3)只能保证一个共享变量的原子操作**

对多个共享变量操作时，循环CAS无法保证操作的原子性，这个时候可以用锁。从JDK1.5开始，JDK提供了**AtomicReference类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行CAS操作。

# 第三章 Java并发编程基础

## 5.1 线程的状态

Java线程在运行的生命周期中可能处于6种不同的状态：new、runnable、blocked、waiting、time\_waiting、terminated。



### 1. Deamon线程

当一个Java虚拟机中不存在非Deamon线程的时候，Java虚拟机将会退出。

### 2. 新建线程

1)实现 runnable接口

2)继承Thread类

3)实现Callable接口，实现call()

### 3. 启动和终止线程

启动线程：start()

中断线程interrupt() :线程并不会立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出，至于目标接到通知后如何处理，则完全由目标线程自行决定。

安全地终止线程：①使用中断；② 使用 Boolean 变量

### 4. 过期的 挂起（suspend）、继续执行（resume）和 stop

Suspend：线程不会释放任何资源，而是占有着资源进入睡眠状态，容易引起死锁。

Resume：被挂起的线程直到遇到resume操作，才能继续。

Stop：在终结一个线程时不会保证线程的资源正常释放。

### 5. 等待（wait）和通知（notify）

线程A调用 wait()，A就转为等待状态；

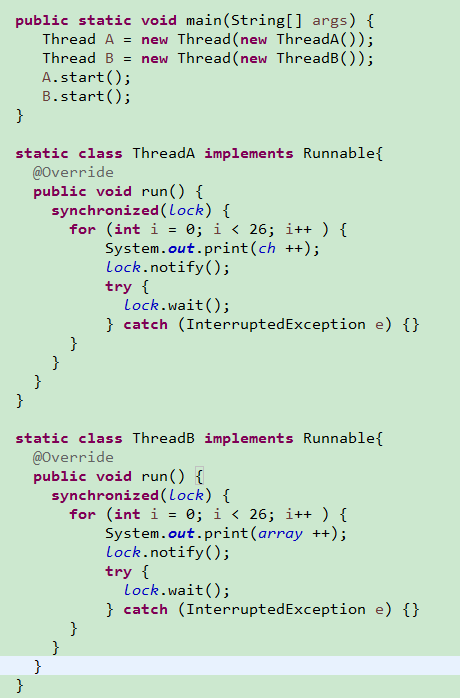
线程A 调用 notify，其他线程被唤醒争抢资源；

### 6. 等待线程结束（join）和谦让（yield）

Join：在线程A中调用 B.join() 表示祖师A线程，知道线程B 完成或超时。

Yield：调用Thread.yidld()，给予其它重要线程更多的机会。

### 7、线程交替打印例子



# 第六章 Java中的锁

CAS的操作在ReetrantLock的实现原理中可是随处可见。

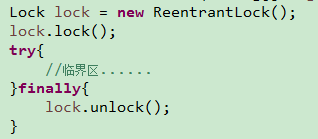
无锁操作：CAS

轻级的隐式锁：volatile

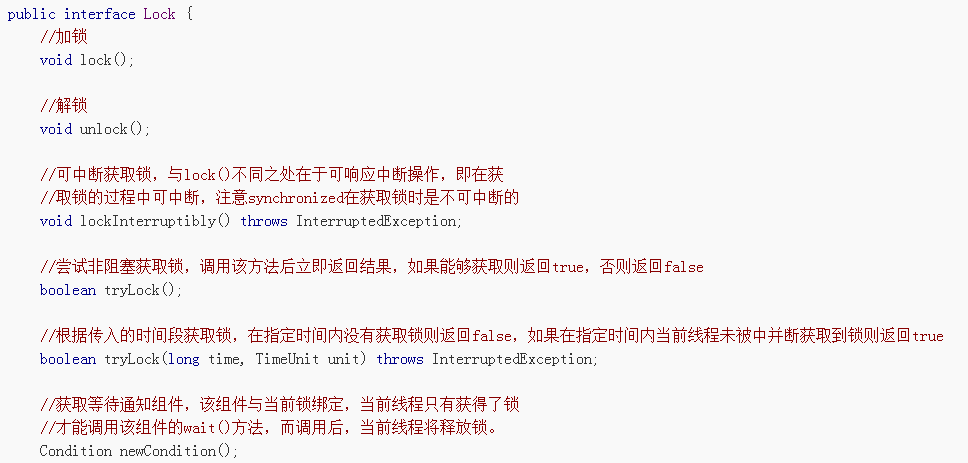
重量级的隐式锁：synchronized

显示锁：锁的持有和释放都必须手动编写

## 6.1 Lock接口



当前线程使用lock()方法与unlock()对临界区进行包围，其他线程由于无法持有锁将无法进入临界区直到当前线程释放锁，注意**unlock()操作必须在finally代码块中**，这样可以确保即使临界区执行抛出异常，线程最终也能正常释放锁。



在Java 1.5中，官方在concurrent并发包中加入了Lock接口，该接口中提供了lock()方法和unLock()方法对显式加锁和显式释放锁操作进行支持。Lock对象锁还提供了synchronized所不具备的其他同步特性，如尝试非阻塞地获取锁，能被中断锁的获取(synchronized在等待获取锁时是不可中的)，超时锁的获取，等待唤醒机制的多条件变量Condition等，这也使得Lock锁在使用上具有更大的灵活性。

## 6.2 队列同步器AbstractQueuedSynchronizer

队列同步器AbstractQueuedSynchronized 是用来构建锁或者其他同步组件的基础框架。它使用了一个int成员变量表示同步状态，并且提供了3个方法( getState()、setState(int newState)和compareAndSetState(int expect, int update))来操作同步状态，同时它们能保证状态的改变是安全的。

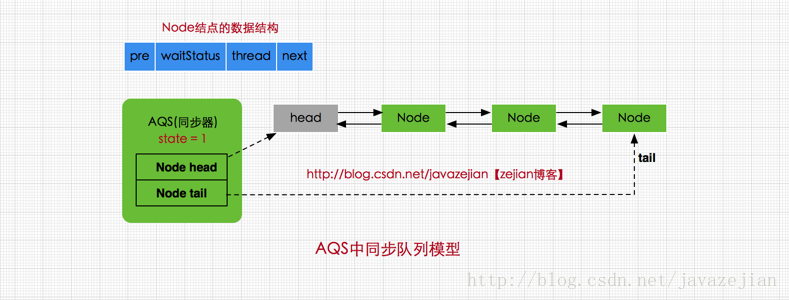
### 1. AQS工作原理概要

AbstractQueuedSynchronizer内部通过一个**int类型的成员变量state来控制同步状态**,当**state=0时，则说明没有任何线程占有共享资源的锁**，当**state=1**时，则说明有线程目前正在使用共享变量，其他线程必须**加入同步队列进行等待**。

AQS内部**通过内部类Node构成FIFO的同步队列**来完成线程获取锁的排队工作，同时利用**内部类ConditionObject构建等待队列**，当Condition调用await()方法后，线程将会加入等待队列中，而当Condition调用signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步队列等待，而另一种则是等待队列(可有多个)，通过Condition调用await()方法释放锁后，将加入等待队列。

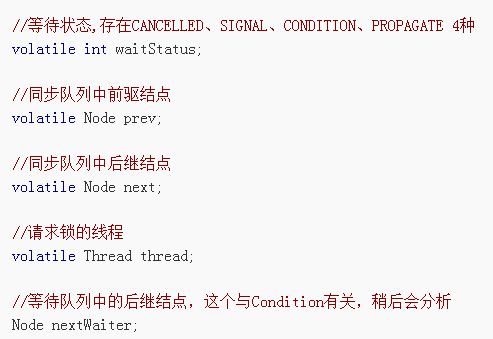
AQS中的同步队列模型，如下：





head和tail分别是AQS中的变量，**注意head为空结点，不存储信息**。而tail则是同步队列的队尾，同步队列采用的是**双向链表的结构**这样可方便队列进行结点增删操作。state变量则是代表同步状态，执行当线程调用lock方法进行加锁后，如果此时state的值为0，则说明当前线程可以获取到锁，同时将state设置为1，表示获取成功。**如果state已为1，也就是当前锁已被其他线程持有，那么当前执行线程将被封装为Node结点加入同步队列的尾部等待**。同步器提供了一个机遇CAS的设置为节点的方法：compareAndSetTail(Node expect, Node updte)。首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，后继节点将会在获取同步状态成功时将自己设置为首节点。由于只有一个线程能够成功获取到同步状态，因此设置头节点的方法并不需要使用CAS来保证。

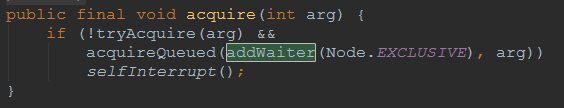
其中Node结点是AQS的内部类，是对每一个访问同步代码的线程的封装，其包含了需要同步的线程本身以及线程的状态，如**是否被阻塞，是否等待唤醒，是否已经被取消**等。每个Node结点内部关联其前继结点prev和后继结点next，这样可以方便线程释放锁后快速唤醒下一个在等待的线程。



其中**SHARED和EXCLUSIVE常量分别代表共享模式和独占模式**，所谓共享模式是一个锁允许多条线程同时操作，如信号量Semaphore采用的就是基于AQS的共享模式实现的，而独占模式则是同一个时间段只 能有一个线程对共享资源进行操作，多余的请求线程需要排队等待，如ReentranLock。

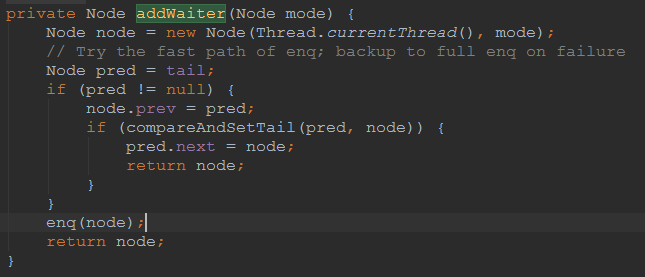
### 2. 独占式同步状态获取

1、通过调用aqs的acquire(int arg)方法获取同步状态。



完成同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的工作，主要是：①先调用实现者的tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态。②如果获取失败，则构造同步节点，并通过addWaiter(Node node)方法加入到同步队列的尾部；③调用acquireQueued(node, int)方法，以死循环的方式获取同步状态，并会阻塞节点的线程，直到前驱节点的出兑或阻塞线程被中断实现。

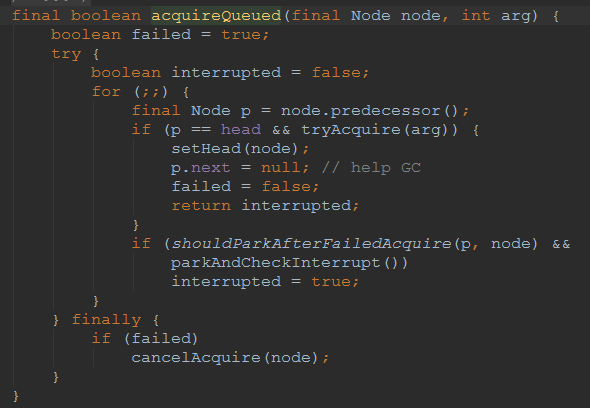
2、



通过调用compareAndSetTail(Node, Node)：线程安全添加到不同队列尾部；

Enq：在“死循环”中通过CAS将节点设置成为尾节点后，才能返回，并并发添加节点的请求通过CAS变得“串行化”。

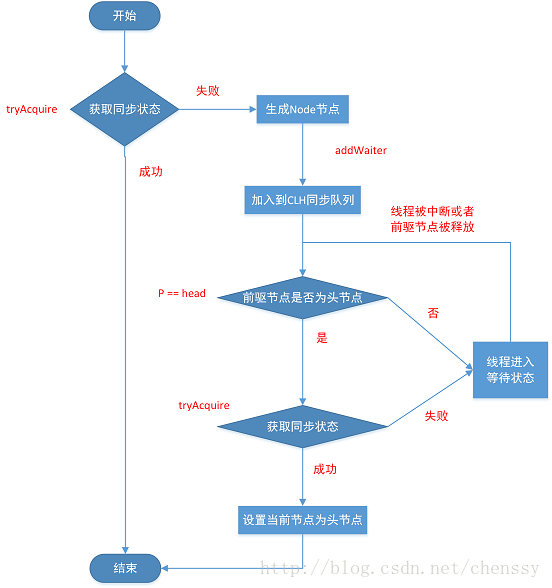
3、acquireQueued



节点进入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前去节点是头结点才能够尝试获取同步状态，因为：

① 头结点是成功获取到同步状态的节点，而头结点的线程释放了同步状态之后，将唤醒起后继节点，后继节点的线程呗唤醒后需要检查自己的前驱节点是否是头结点。

② 维护同步队列的FIFO原则。



### 3. 共享式同步状态获取

通过调用同步器的acquireShared(int arg)方法可以共享式地获取同步状态。在acquireShared(int arg)方法中，同步器调用 tryAcquireShared(int arg)方法尝试获取同步状态。当返回值大于等于0时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。即当前节点的前驱为头结点时，尝试获取同步状态，如果返回值大于等于0，表示该次获取同步状态成功并从自旋过程中退出。

protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 if (hasQueuedPredecessors())  
 return -1;  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

Semaphore 将aqs的同步状态用于保存当前可用许可的数量。tryAcquireShared首先计算剩余许可的数量，如果没有足够的许可，那么会返回一个值表示获取操作失败。如果还有剩余的许可，那么通过compareAndSetState以原子方式来降低许可的计数。

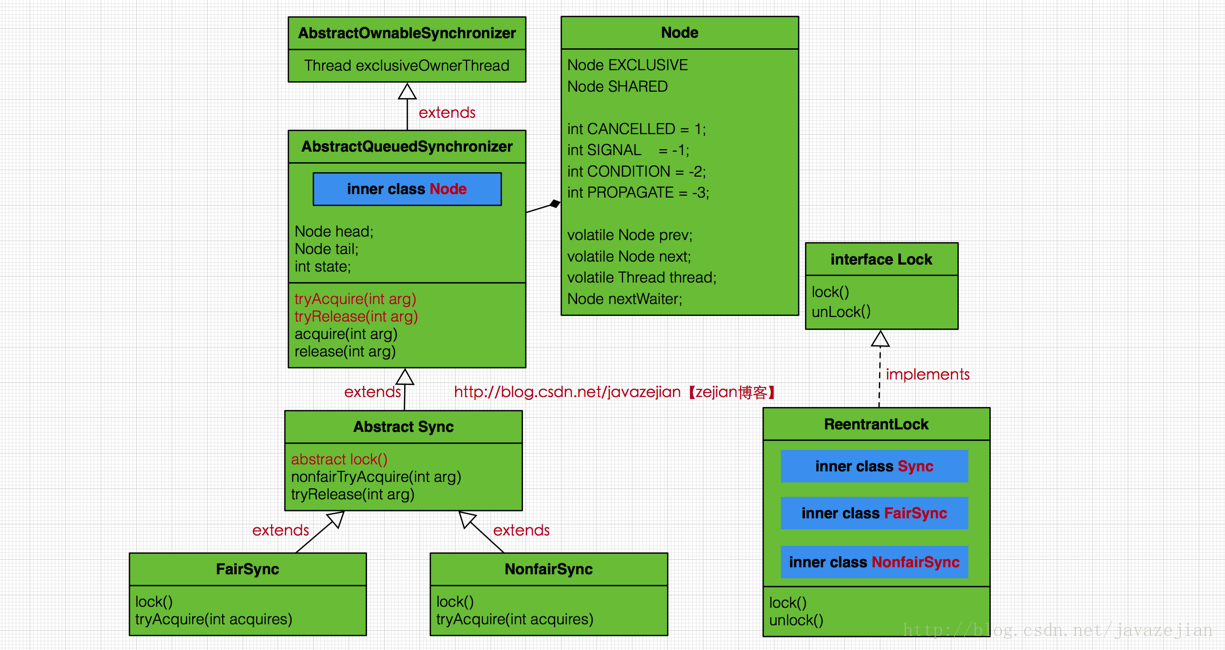
**CountDownLatch使用AQS像是：在同步状态中保存的是当前的计数值。CountDown方法调用release。从而导致计数值递减，并且当计数值为零时，解除所有等待线程的阻塞。Await调用acquire，当计数器为零时，acquire将立即返回，否则将阻塞。**

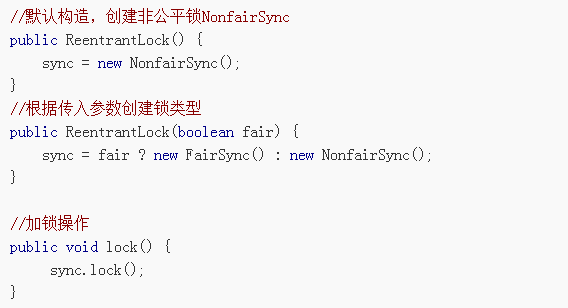
### 4. 独占式超时获取同步状态

增强版的方法：tryAcquireNanos(int arg,long nanos)。该方法为acquireInterruptibly方法的进一步增强，它除了响应中断外，还有超时控制。

针对超时控制，程序**首先记录上次唤醒时间**lastTime,当前唤醒时间now。如果获取同步状态失败，则需要计算出**需要休眠的时间间隔**nanosTimeout -= now - lastTime，如果nanosTimeout <= 0 表示已经超时了，返回false，如果大于spinForTimeoutThreshold（1000L）则需要休眠nanosTimeout ，如果nanosTimeout <= spinForTimeoutThreshold ，就不需要休眠了，直接进入快速自旋的过程。原因在于 spinForTimeoutThreshold 已经非常小了，非常短的时间等待无法做到十分精确，如果这时再次进行超时等待，相反会让nanosTimeout 的超时从整体上面表现得不是那么精确，所以在超时非常短的场景中，AQS会进行无条件的快速自旋。

## 6.3 ReentrantLock

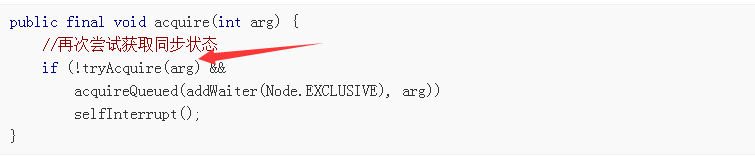




### 1. ReetrantLock中非公平锁



首先对同步状态执行CAS操作，尝试把state的状态从0设置为1。如果返回true则代表获取同步状态成功，也就是当前线程获取锁成，可操作临界资源。如果返回false，则表示已有线程持有该同步状态(其值为1)，获取锁失败，执行 acquire(1)方法，该方法是AQS中的方法，它对中断不敏感，即使线程获取同步状态失败，进入同步队列，后续对该线程执行中断操作也不会从同步队列中移出，方法如下

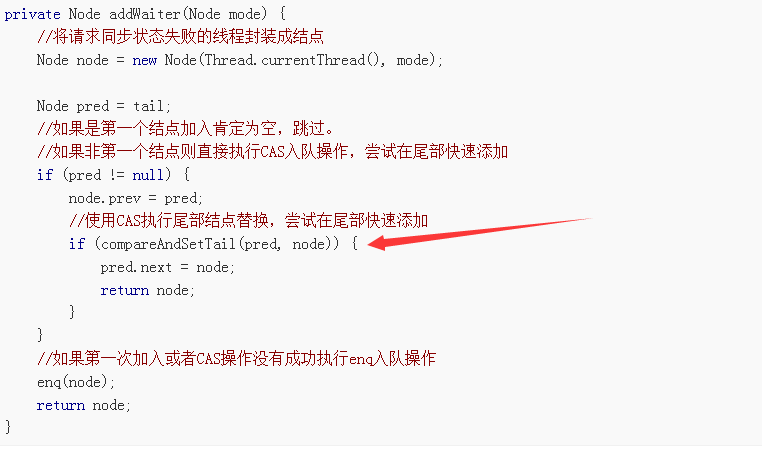


这里传入参数arg表示要获取同步状态后设置的值(即要设置state的值)，因为要获取锁，而status为0时是释放锁，1则是获取锁，所以这里一般传递参数为1，进入方法后首先会执行tryAcquire(arg)方法，在前面分析过该方法在AQS中并没有具体实现，而是交由子类实现，因此该方法是由ReetrantLock类内部实现的。



一是尝试**再次获取同步状态**，如果获取成功则将当前线程设置为OwnerThread，否则失败，二是**判断当前线程current是否为OwnerThread**，如果是则属于重入锁，state自增1，并获取锁成功，返回true，反之失败，返回false。

接着看之前的方法acquire(int arg)：如果tryAcquire(arg)返回true，acquireQueued自然不会执行，如果tryAcquire(arg)返回false，则会执行addWaiter(Node.EXCLUSIVE)进行入队操作,由于ReentrantLock属于独占锁，因此结点类型为Node.EXCLUSIVE，下面看看addWaiter方法具体实现：

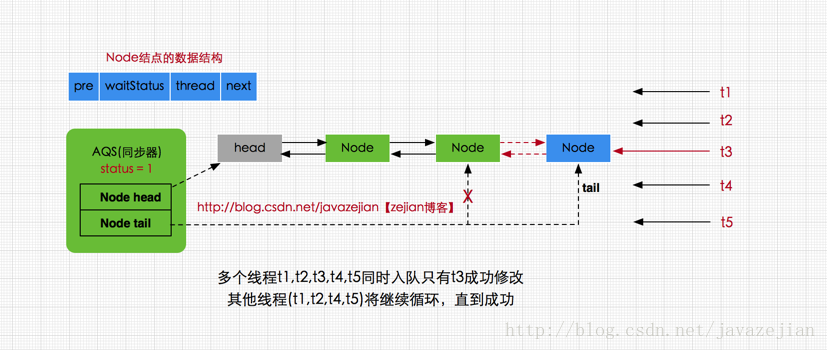


创建了一个Node.EXCLUSIVE类型Node结点用于封装线程及其相关信息，其中tail是AQS的成员变量，指向队尾。如果是第一个结点，则为tail肯定为空，那么将执行enq(node)操作，如果非第一个结点即tail指向不为null，直接尝试执行CAS操作加入队尾，如果CAS操作失败还是会执行enq(node)，继续看enq(node)：



这个方法使用一个**死循环进行CAS操作**，可以解决多线程并发问题。这里做了两件事，一是如果还没有初始同步队列则创建新结点并使用compareAndSetHead设置头结点，tail也指向head，二是队列已存在，则将新结点node添加到队尾。

注意这两个步骤都存在同一时间多个线程操作的可能，**如果有一个线程修改head和tail成功，那么其他线程将继续循环，直到修改成功，**这里使用CAS原子操作进行头结点设置和尾结点tail替换可以保证线程安全，**从这里也可以看出head结点本身不存在任何数据，**它只是作为一个牵头结点，而tail永远指向尾部结点(前提是队列不为null)。

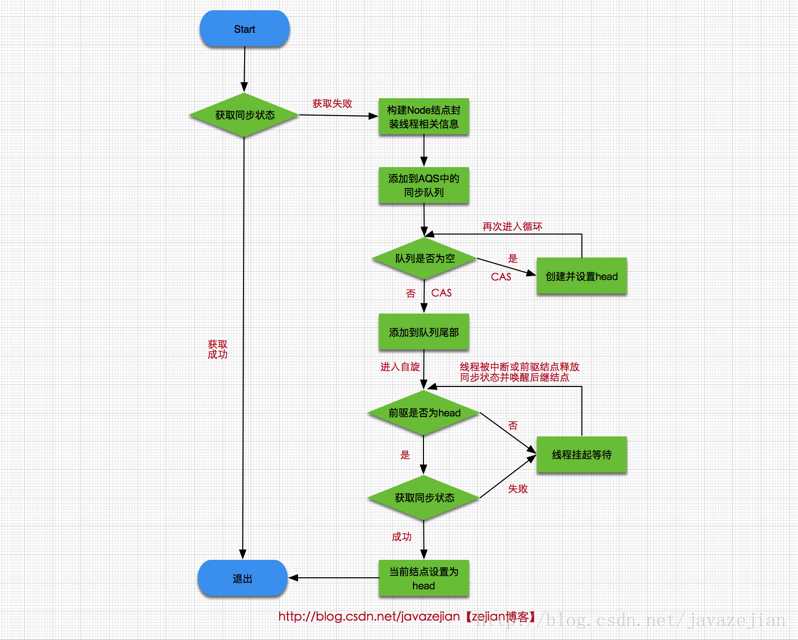


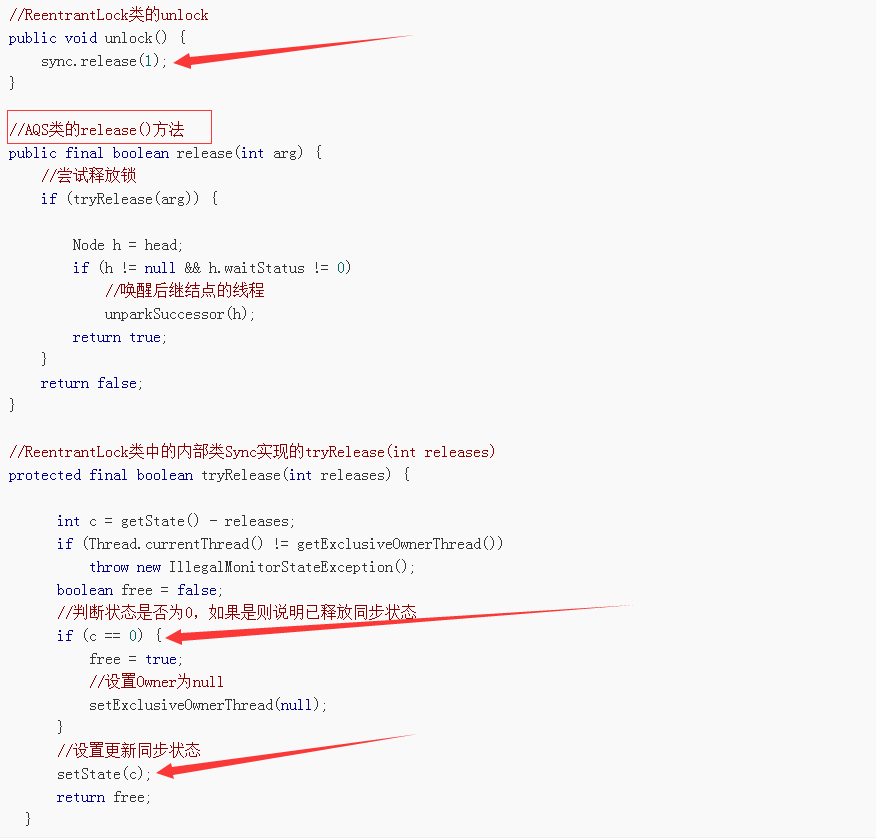
添加到同步队列后，结点就会进入一个自旋过程，即每个结点都在观察时机待条件满足获取同步状态，然后从同步队列退出并结束自旋，回到之前的acquire()方法，自旋过程是在acquireQueued(addWaiter(Node.EXCLUSIVE), arg))方法中执行的，代码如下



当前线程在自旋(死循环)中获取同步状态，当且仅当前驱结点为头结点才尝试获取同步状态，这符合FIFO的规则，即先进先出，其次head是当前获取同步状态的线程结点，只有当head释放同步状态唤醒后继结点，后继结点才有可能获取到同步状态，因此后继结点在其前继结点为head时，才进行尝试获取同步状态，其他时刻将被挂起。进入if语句后调用setHead(node)方法，将当前线程结点设置为head。

ReetrantLock内部间接通过AQS的FIFO的同步队列就完成了lock()操作，这里我们总结成逻辑流程图：





### 2. ReetrantLock中公平锁



唯一不同的位置为判断条件多了 hasQueuedPredecessors()方法，即加入了同步队列中当前节点是够有前驱节点的判断，如果该方法返回true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

### 3. 总结

非公平性锁可能是线程“饥饿”，为什么它又被设定成默认的实现呢？因为非公平性锁的开销更小，因为减少了上下文切换的次数。

公平性锁保证了锁的获取按照FIFO原则，则代价是进行大量的线程切换，非公平性锁虽然可能曹成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

## 6.4 读写锁ReentrantReadWriteLock

读写锁【jdk1.5】在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一堆锁，一个读锁和一个写锁，写操作对读操作保证可见性。把State状态作为一个读写锁的计数器，包括了重入的次数。

当写锁被获取到时，非当前写操作线程的读写操作都会被阻塞，写锁释放之后，所有操作继续执行。

### 1. 特性

公平性选择：支持非公平（默认）和公平的锁获取方式。

重入性：该锁支持重进入。读线程在获取了读锁之后，能够再次获取读锁。而写线程在获取了写锁之后，能够再次获取写锁，同时也可以获取读锁。

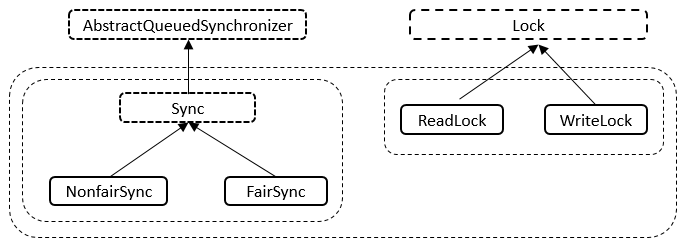
锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁。

### 2. 读写锁的接口

ReentrantReadWriteLock:

内部类：Sync、NonfairSync、FairSync、ReadLock、WriteLock

方法：Lock readLock();、Lock writeLock();



#### 2.1 Sync类

**1、类的内部类**

Sync类内部存在两个内部类，分别为HoldCounter和ThreadLocalHoldCounter，其中**HoldCounter主要与读锁配套使用**，其中，HoldCounter源码如下。【tid 1.7 1.8不一样】

|  |  |
| --- | --- |
| static final class HoldCounter {  // 计数  int count = 0;  // Use id, not reference, to avoid garbage retention  // final long tid = Thread.currentThread().getId(); 1.7  final long tid = getThreadId(Thread.currentThread()); } | 其中count表示某个读线程重入的次数，tid表示该线程的tid字段的值，该字段可以用来唯一标识一个线程。 |
| // 本地线程计数器  static final class ThreadLocalHoldCounter  extends ThreadLocal<HoldCounter> {  // 重写初始化方法，在没有进行set的情况下，获取的都是该HoldCounter值  public HoldCounter initialValue() {  return new HoldCounter();  }  } | |

**2、类的属性**

abstract static class Sync extends AbstractQueuedSynchronizer {

private static final long serialVersionUID = 6317671515068378041L;

static final int SHARED\_SHIFT = 16; // 高16位为读锁，低16位为写锁

static final int SHARED\_UNIT = (1 << SHARED\_SHIFT); // 读锁单位

static final int MAX\_COUNT = (1 << SHARED\_SHIFT) - 1; // 读锁最大数量

static final int EXCLUSIVE\_MASK = (1 << SHARED\_SHIFT) - 1; // 为了便于与运算去掉高16位

private transient ThreadLocalHoldCounter readHolds;

// 本地线程计数器, 保存当前线程持有的重入读锁的数目，在读锁重入次数为 0 时移除

private transient HoldCounter cachedHoldCounter; // 缓存的计数器

private transient Thread firstReader = null; // 第一个读线程

private transient int firstReaderHoldCount; // 第一个读线程的计数

}

把State状态作为一个读写锁的计数器，包括了重入的次数。state是32位的int值，所以把高位16位作为读锁的计数器，**低位的16位作为写锁的计数器**，所以读锁增加1就相当于增加了2\*16

**3、核心函数分析**

　　I. sharedCount函数

　　表示**占有读锁的线程数量**，源码如下

static int sharedCount(int c) { return c >>> SHARED\_SHIFT; }

　　说明：**直接将state无符号右移16位，就可以得到读锁的线程数量**，因为state的高16位表示读锁，对应的第十六位表示写锁数量。当读状态增加1时，即加0x00010000.

　　II. exclusiveCount函数

　　表示占有写锁的线程数量，源码如下

static int exclusiveCount(int c) { return c & EXCLUSIVE\_MASK; }

　　说明：直接**将状态state和（2^16 – 1，即0x0000FFFF）做与运算**，其等效于将state模上2^16。写锁数量由state的低十六位表示。

III. tryRelease函数

**protected** **final** **boolean** tryRelease(**int** releases) {

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

**int** nextc = getState() - releases;

**boolean** free = *exclusiveCount*(nextc) == 0;

**if** (free)

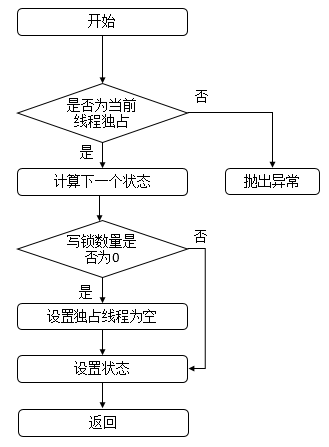
setExclusiveOwnerThread(**null**);

setState(nextc);

**return** free;

}

此函数用于释放写锁资源，首先会判断该线程**是否为独占线程**，若不为独占线程，则抛出异常，否则，**计算释放资源后的写锁的数量，若为0，表示成功释放**，资源不将被占用，否则，表示资源还被占用。其函数流程图如下。



**IV. tryAcquire函数**

protected final boolean tryAcquire(int acquires) {

Thread current = Thread.currentThread();// 获取当前线程

int c = getState(); // 获取状态

int w = exclusiveCount(c); // 写线程数量

if (c != 0) { // 状态不为0

// (Note: if c != 0 and w == 0 then shared count != 0)

if (w == 0 || current != getExclusiveOwnerThread()) // 写线程数量为0或者当前线程没有占有独占资源

return false;

if (w + exclusiveCount(acquires) > MAX\_COUNT) // 判断是否超过最高写线程数量

throw new Error("Maximum lock count exceeded");

// Reentrant acquire

// 设置AQS状态

setState(c + acquires);

return true;

}

if (writerShouldBlock() ||

!compareAndSetState(c, c + acquires)) // 写线程是否应该被阻塞

return false;

// 设置独占线程

setExclusiveOwnerThread(current);

return true;

}

此函数用于获取写锁，首先会获取state，判断是否为0，若为0，表示此时没有读锁线程，再判断写线程是否应该被阻塞，而在非公平策略下总是不会被阻塞，在公平策略下会进行判断（判断同步队列中是否有等待时间更长的线程，若存在，则需要被阻塞，否则，无需阻塞），之后在设置状态state，然后返回true。若state不为0，则表示此时存在读锁或写锁线程，若写锁线程数量为0或者当前线程为独占锁线程，则返回false，表示不成功，否则，判断写锁线程的重入次数是否大于了最大值，若是，则抛出异常，否则，设置状态state，返回true，表示成功。

**V. tryReleaseShared函数**

protected final boolean tryReleaseShared(int unused) {

// 获取当前线程

Thread current = Thread.currentThread();

if (firstReader == current) { // 当前线程为第一个读线程

// assert firstReaderHoldCount > 0;

if (firstReaderHoldCount == 1) // 读线程占用的资源数为1

firstReader = null;

else // 减少占用的资源

firstReaderHoldCount--;

} else { // 当前线程不为第一个读线程

// 获取缓存的计数器

HoldCounter rh = cachedHoldCounter;

if (rh == null || rh.tid != getThreadId(current)) // 计数器为空或者计数器的tid不为当前正在运行的线程的tid

// 获取当前线程对应的计数器

rh = readHolds.get();

// 获取计数

int count = rh.count;

if (count <= 1) { // 计数小于等于1

// 移除

readHolds.remove();

if (count <= 0) // 计数小于等于0，抛出异常

throw unmatchedUnlockException();

}

// 减少计数

--rh.count;

}

for (;;) { // 无限循环

// 获取状态

int c = getState();

// 获取状态

int nextc = c - SHARED\_UNIT;

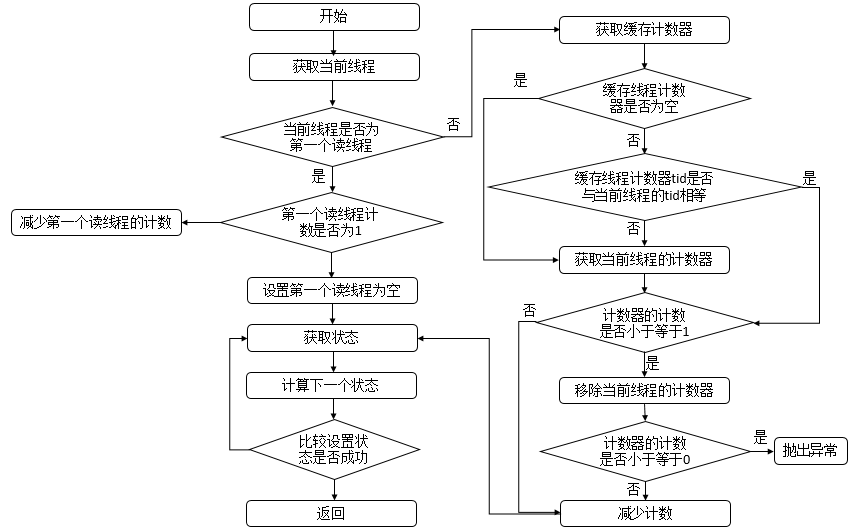
if (compareAndSetState(c, nextc)) // 比较并进行设置

return nextc == 0;

}

}

　　说明：**此函数表示读锁线程释放锁**。首先判断当前线程是否为第一个读线程firstReader，若是，则判断第一个读线程占有的资源数是否为1，若是，则设置第一个读线程firstReader为空，否则，将第一个读线程占有的资源数firstReaderHoldCount减1；若当前线程不是第一个读线程，那么首先会获取缓存计数器（上一个读锁线程对应的计数器 ），若计数器为空或者tid不等于当前线程的tid值，则获取当前线程的计数器，如果计数器的计数count小于等于1，则移除当前线程对应的计数器，如果计数器的计数count小于等于0，则抛出异常，之后再减少计数即可。无论何种情况，都会进入无限循环，该循环可以确保成功设置状态state。其流程图如下

　　VI. tryAcquireShared函数

// 共享模式下获取资源

protected final int tryAcquireShared(int unused) {

Thread current = Thread.currentThread();// 获取当前线程

int c = getState(); // 获取状态

if (exclusiveCount(c) != 0 && getExclusiveOwnerThread() != current)

**// 写线程数不为0并且占有资源的不是当前线程**

return -1;

int r = sharedCount(c); // 读锁数量

if (!readerShouldBlock() && r < MAX\_COUNT && compareAndSetState(c, c + SHARED\_UNIT)) {

**// 读线程不应该被阻塞、并且小于最大值、并且比较设置成功**

if (r == 0) { // 读锁数量为0

firstReader = current; // 设置第一个读线程

firstReaderHoldCount = 1; // 读线程占用的资源数为1

} else if (firstReader == current) { // 当前线程为第一个读线程

firstReaderHoldCount++; // 占用资源数加1

} else { **// 读锁数量不为0并且不为当前线程**

HoldCounter rh = cachedHoldCounter; // 获取计数器

if (rh == null || rh.tid != getThreadId(current))

**// 计数器为空或者计数器的tid不为当前正在运行的线程的tid**

cachedHoldCounter = rh = readHolds.get();// 获取当前线程对应的计数器

else if (rh.count == 0) // 计数为0

// 设置

readHolds.set(rh);

rh.count++;

}

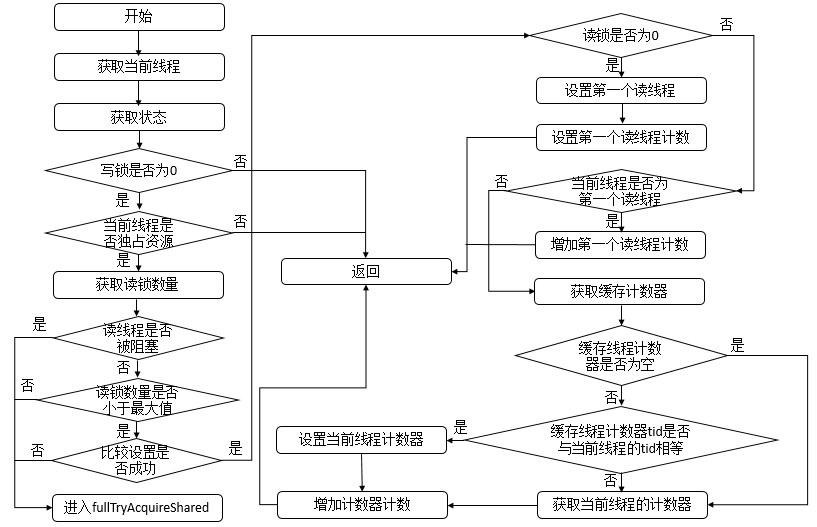
return 1;

}

return fullTryAcquireShared(current);

}

说明：此函数表示读锁线程获取读锁。首先判断写锁是否为0并且当前线程不占有独占锁，直接返回；否则，判断读线程是否需要被阻塞并且读锁数量是否小于最大值并且比较设置状态成功，若当前没有读锁，则设置第一个读线程firstReader和firstReaderHoldCount；若当前线程线程为第一个读线程，则增加firstReaderHoldCount；否则，将设置当前线程对应的HoldCounter对象的值。流程图如下。



### ****读写状态的设计****

读写锁的自定义同步器需要在同步状态（一个整型变量）上维护**多个读线程和一个写线**程的状态。需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写。

假设当前同步状态值为S，读状态为S>>>16（无符号右移16位）。写状态为 S & 0x0000FFFF（将高16位全部抹去）。当读状态增加1时，等于S+（1<<16），也就是S+0x00010000.

### ****读锁的读取与释放****

读锁能够被多个线程同时获取，**在没有其他写线程访问**（或者写状态为0）**时，读锁总会被成功地获取**，而所做的也只是线程安全的增加读状态。

如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态。

写线程数不为0并且占有资源的不是当前线程

### ****写锁的读取与释放****

如果当前线程在获取写锁时，**读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程**，则当前线程进入等待状态。

该方法除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取。原因在于：**读写锁要确保写锁的操作对读锁可见，如果允许在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。**

**写锁的释放：每次缩放均减少写状态。**当写状态为0时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续写线程可见。

### ****锁降级****

锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

只有一个线程能够获取到写锁，其他线程会被阻塞在读锁和写锁的lock()方法上，当前线程获取写锁完成数据准备之后，可以再获取读锁，随后释放写锁，完成锁降级。

锁降级中读锁的获取是否必要呢？答案是必要的，**主要是为了保证数据的可见性**，如果当前线程不获取读锁而是直接释放写锁没，假设此刻另一个线程（基座T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

读写锁不支持锁升级（把持读锁，获取写锁，最后释放读锁的过程）。目的是保证数据可见性。如果读锁已被多个线程获取。其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

### ****总结****

经过分析ReentrantReadWriteLock的源码，可知其可以实现多个线程同时读，此时，写线程会被阻塞。并且，写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样写入锁变成了读取锁。

## 6.5 Condition接口

### ****1. 接口方法****

public interface Condition {

/\*\*

\* 使当前线程进入等待状态直到被通知(signal)或中断

\* 当其他线程调用singal()或singalAll()方法时，该线程将被唤醒

\* 当其他线程调用interrupt()方法中断当前线程

\* await()相当于synchronized等待唤醒机制中的wait()方法

\*/

void await() throws InterruptedException;

//当前线程进入等待状态，直到被唤醒，该方法不响应中断要求

void awaitUninterruptibly();

//调用该方法，当前线程进入等待状态，直到被唤醒或被中断或超时

//其中nanosTimeout指的等待超时时间，单位纳秒

long awaitNanos(long nanosTimeout) throws InterruptedException;

//同awaitNanos，但可以指明时间单位

boolean await(long time, TimeUnit unit) throws InterruptedException;

//调用该方法当前线程进入等待状态，直到被唤醒、中断或到达某个时

//间期限(deadline),如果没到指定时间就被唤醒，返回true，其他情况返回false

boolean awaitUntil(Date deadline) throws InterruptedException;

//唤醒一个等待在Condition上的线程，该线程从等待方法返回前必须

//获取与Condition相关联的锁，功能与notify()相同

void signal();

//唤醒所有等待在Condition上的线程，该线程从等待方法返回前必须

//获取与Condition相关联的锁，功能与notifyAll()相同

void signalAll();

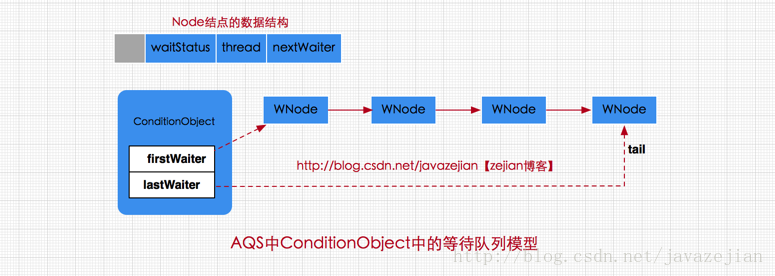
}

### ****2. 对比Object监视器****

|  |  |  |
| --- | --- | --- |
| 对比项 | Object 监视器方法 | Condition |
| 前置条件 | 获取对象锁 | 调用Lock.lock()获取锁  调用Lock.newCondition()获取condition对象 |
| 调用方法 | 直接调用  例如：object.wait() | 直接调用  例如：condition.await() |
| 等待队列个数 | 一个 | 多个 |
| 当前线程释放锁并进入等待状态 | 支持 | 支持 |
| 在等待状态中不响应中断 | 不支持 | 支持 |
| 当前线程释放锁并进入超时等待状态 | 支持 | 支持 |
| 当前线程释放锁并进入等待状态到将来的某个时间 | 不支持 | 支持 |
| 唤醒等待队列中的一个/全部线程 | 支持 | 支持 |

### 3.Condition实现分析

AQS中存在两种队列，一种是同步队列，一种是等待队列，而等待队列就相对于Condition而言的。**每个Condition都对应着一个等待队列**，也就是说如果一个锁上创建了多个Condition对象，那么也就存在多个等待队列。等待队列是一个FIFO的队列，在队列中每一个节点都包含了一个线程的引用，而该线程就是Condition对象上等待的线程。当一个线程调用了await()相关的方法，那么该线程将会释放锁，并构建一个Node节点封装当前线程的相关信息加入到等待队列中进行等待，直到被唤醒、中断、超时才从队列中移出。



在等待队列中使用的变量与同步队列是不同的，**Condtion中等待队列的结点只有直接指向的后继结点并没有指明前驱结点**，而且**使用的变量是nextWaiter而不是next**。firstWaiter指向等待队列的头结点，lastWaiter指向等待队列的尾结点，等待队列中结点的状态只有两种即CANCELLED和CONDITION，前者表示线程已结束需要从等待队列中移除，后者表示条件结点等待被唤醒。再次强调每个Codition对象对于一个等待队列，也就是说AQS中只能存在一个同步队列，但可拥有多个等待队列。下面从代码层面看看被调用await()方法(其他await()实现原理类似)的线程是如何加入等待队列的，而又是如何从等待队列中被唤醒的。

#### 3.1 等待

public final void await() throws InterruptedException {

//判断线程是否被中断

if (Thread.interrupted())

throw new InterruptedException();

//创建新结点加入等待队列并返回

Node node = addConditionWaiter();

//**释放当前线程锁即释放同步状态**

int savedState = fullyRelease(node);

int interruptMode = 0;

//判断结点是否同步队列(SyncQueue)中,即是否被唤醒

while (!isOnSyncQueue(node)) {

//挂起线程

LockSupport.park(this);

//判断是否被中断唤醒，如果是退出循环。

if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

break;

}

//被唤醒后执行自旋操作争取获得锁，同时判断线程是否被中断

if(acquireQueued(node,savedState) && interruptMode != THROW\_IE)

interruptMode = REINTERRUPT;

// clean up if cancelled

if (node.nextWaiter != null)

//清理等待队列中不为CONDITION状态的结点

unlinkCancelledWaiters();

if (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

执行addConditionWaiter()添加到等待队列。

private Node addConditionWaiter() {

Node t = lastWaiter;

// 判断是否为结束状态的结点并移除

if (t != null && t.waitStatus != Node.CONDITION) {

unlinkCancelledWaiters();

t = lastWaiter;

}

//创建新结点状态为CONDITION

Node node = new Node(Thread.currentThread(), Node.CONDITION);

//加入等待队列

if (t == null)

firstWaiter = node;

else

t.nextWaiter = node;

lastWaiter = node;

return node;

}

await()方法主要做了3件事，一是**同步队列的首节点调用addConditionWaiter()方法将当前线程封装成node结点加入等待队列**，二是调用**fullyRelease(node)方法释放同步状态并唤醒后继结点的线程，当前线程会进入等待状态**。三是调用isOnSyncQueue(node)方法判断结点是否在同步队列中，注意是个while循环，如果同步队列中没有该结点就直接挂起该线程，需要明白的是如果线程被唤醒后就调用acquireQueued(node, savedState)执行自旋操作争取锁，即当前线程结点从等待队列转移到同步队列并开始努力获取锁。

#### 3.2 通知

调用Condition的signal ()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移动到同步队列中。

public final void signal() {

//判断是否持有独占锁，如果不是抛出异常【当前线程必须是获取了锁的线程】

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

Node first = firstWaiter;

//唤醒等待队列第一个结点的线程

if (first != null)

doSignal(first);

}

这里signal()方法做了两件事，一是**判断当前线程是否持有独占锁**，没有就抛出异常，从这点也可以看出只有独占模式先采用等待队列，而共享模式下是没有等待队列的，也就没法使用Condition。二是唤醒等待队列的第一个结点，即执行doSignal(first)，首节点移动到同步队列并使用LockSupport唤醒节点中的线程。

private void doSignal(Node first) {

do {

//移除条件等待队列中的第一个结点，

//如果后继结点为null，那么说没有其他结点将尾结点也设置为null

if ( (firstWaiter = first.nextWaiter) == null)

lastWaiter = null;

first.nextWaiter = null;

//如果被通知节点没有进入到同步队列并且条件等待队列还有不为空的节点，则继续循环通知后续结点

} while (!transferForSignal(first) &&

(first = firstWaiter) != null);

}

//transferForSignal方法

final boolean transferForSignal(Node node) {

//尝试设置唤醒结点的waitStatus为0，即初始化状态

//如果设置失败，说明当期结点node的waitStatus已不为

//CONDITION状态，那么只能是结束状态了，因此返回false

//返回doSignal()方法中继续唤醒其他结点的线程，注意这里并

//不涉及并发问题，所以CAS操作失败只可能是预期值不为CONDITION，

//而不是多线程设置导致预期值变化，毕竟操作该方法的线程是持有锁的。

if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))

return false;

//加入同步队列并返回前驱结点p

Node p = enq(node);

int ws = p.waitStatus;

//判断前驱结点是否为结束结点(CANCELLED=1)或者在设置

//前驱节点状态为Node.SIGNAL状态失败时，唤醒被通知节点代表的线程

if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))

//唤醒node结点的线程

LockSupport.unpark(node.thread);

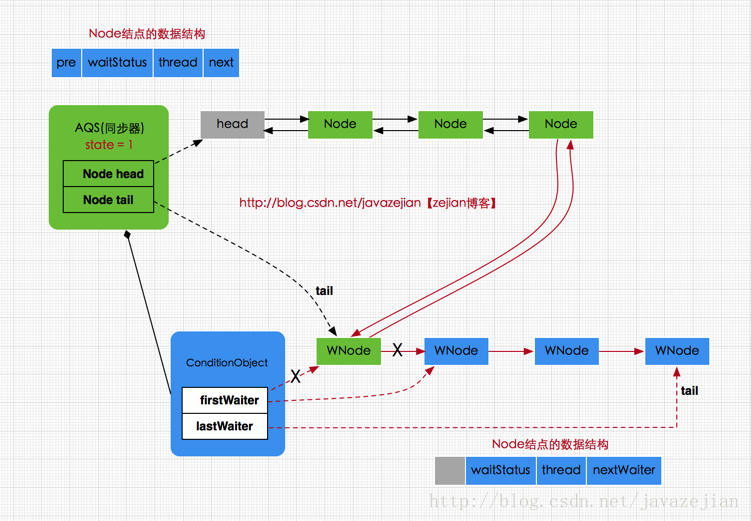
return true;

}

doSignal(first)方法中做了两件事，一是从条件等待队列移除被唤醒的节点，然后重新维护条件等待队列的firstWaiter和lastWaiter的指向。二是将从等待队列移除的结点加入同步队列(在transferForSignal()方法中完成的)，如果进入到同步队列失败并且条件等待队列还有不为空的节点，则继续循环唤醒后续其他结点的线程。

整个signal()的唤醒过程，即signal()被调用后，先判断当前线程是否持有独占锁，如果有，那么唤醒当前Condition对象中等待队列的第一个结点的线程，并通过调用同步器的enq方法，等待队列中的头结点线程安全地移动到同步队列中，如果加入同步队列失败，那么继续循环唤醒等待队列中的其他结点的线程，如果成功加入同步队列，那么如果其前驱结点是否已结束或者设置前驱节点状态为Node.SIGNAL状态失败，则通过LockSupport.unpark()唤醒被通知节点代表的线程，到此signal()任务完成。注意被唤醒后的线程，将从前面的await()方法中的while循环中退出，因为此时该线程的结点已在同步队列中，那么while (!isOnSyncQueue(node))将不在符合循环条件，进而调用AQS的acquireQueued()方法加入获取同步状态的竞争中。

Condition的signalAll()方法，相当于对等待队列中的每个节点均执行一次signal()方法，效果就是讲等待队列中所有节点全部移动待同步队列中，并唤醒每个节点的线程。



## 6.6 信号量-Semaphore

信号量维护了一个许可集，我们在初始化Semaphore时需要为这个许可集传入一个数量值，**该数量值代表同一时间能访问共享资源的线程数量**。线程可以通过acquire()方法获取到一个许可，然后对共享资源进行操作，注意如果许可集已分配完了，那么线程将进入等待状态，直到其他线程释放许可才有机会再获取许可，线程释放一个许可通过release()方法完成。

### ****1. 实现内部原理概要****

信号量Semaphore的类结构与ReetrantLock的类结构几乎如出一辙。Semaphore内部同样存在继承自AQS的内部类Sync以及继承自Sync的公平锁(FairSync)和非公平锁(NofairSync),从这点也足以说明Semaphore的内部实现原理也是基于AQS并发组件的， AQS是基础组件，**只负责核心并发操作，如加入或维护同步队列，控制同步状态**，等，而具体的**加锁和解锁操作交由子类完成**，因此子类Semaphore共享锁的获取与释放需要自己实现，这两个方法分别是获取锁的tryAcquireShared(int arg)方法和释放锁的tryReleaseShared(int arg)方法. Semaphore的内部类公平锁(FairSync)和非公平锁(NoFairSync)各自实现不同的获取锁方法即tryAcquireShared(int arg)，毕竟公平锁和非公平锁的获取稍后不同，而释放锁tryReleaseShared(int arg)的操作交由Sync实现，因为释放操作都是相同的

### **2. 非公平锁中的共享锁**

//默认创建公平锁，permits指定同一时间访问共享资源的线程数

public Semaphore(int permits) {

sync = new NonfairSync(permits);

}

在AQS中存在一个变量state，当我们创建Semaphore对象传入许可数值时，最终会赋值给state，state的数值代表同一个时刻可同时操作共享数据的线程数量，每当一个线程请求(如调用Semaphored的acquire()方法)获取同步状态成功，state的值将会减少1，直到state为0时，表示已没有可用的许可数，也就是对共享数据进行操作的线程数已达到最大值，其他后来线程将被阻塞，此时AQS内部会将线程封装成共享模式的Node结点，加入同步队列中等待并开启自旋操作。只有当持有对共享数据访问权限的线程执行完成任务并释放同步状态后，同步队列中的对于的结点线程才有可能获取同步状态并被唤醒执行同步操作，注意在同步队列中获取到同步状态的结点将被设置成head并清空相关线程数据(毕竟线程已在执行也就没有必要保存信息了)。

//Semaphore的acquire()

public void acquire() throws InterruptedException {

sync.acquireSharedInterruptibly(1);

}

**public** **void** acquireUninterruptibly(**int** permits) {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.acquireShared(permits);

}

/\*\*

\* 注意Sync类继承自AQS

\* AQS的acquireSharedInterruptibly()方法

\*/

public final void acquireSharedInterruptibly(int arg)

throws InterruptedException {

//判断是否中断请求

if (Thread.interrupted())

throw new InterruptedException();

//如果tryAcquireShared(arg)不小于0，则线程获取同步状态成功

if (tryAcquireShared(arg) < 0)

//未获取成功加入同步队列等待

doAcquireSharedInterruptibly(arg);

}

Semaphore的acquire()方法也是可中断的。**首先进行线程中断的判断**，如果没有中断，那么先尝试调用实现者Sync的tryAcquireShared(arg)【首先获取state，减去1，直到信号量是否已小于0或者CAS执行成功返回int信号量】方法**获取同步状态是否小于0**。如果获取成功【大于0】，那么方法执行结束，如果获取失败，则创建一个共享模式（Node.SHARED）的结点并加入同步队列，加入完成后，当前线程进入自旋状态，首先判断前驱结点是否为head：如果是，那么尝试获取同步状态并返回信号量r值，如果r大于0，则说明获取同步状态成功，将当前线程设置为head，并通知后续结点继续获取同步状态。如果前驱结点不为head或前驱结点为head并尝试获取同步状态失败，那么判断前驱结点的waitStatus值是否为SIGNAL【等待被唤醒状态】并调整同步队列中的node结点状态，如果返回true，那么将当前线程挂起并返回是否中断线程的flag。

#### 不可中断的获取锁

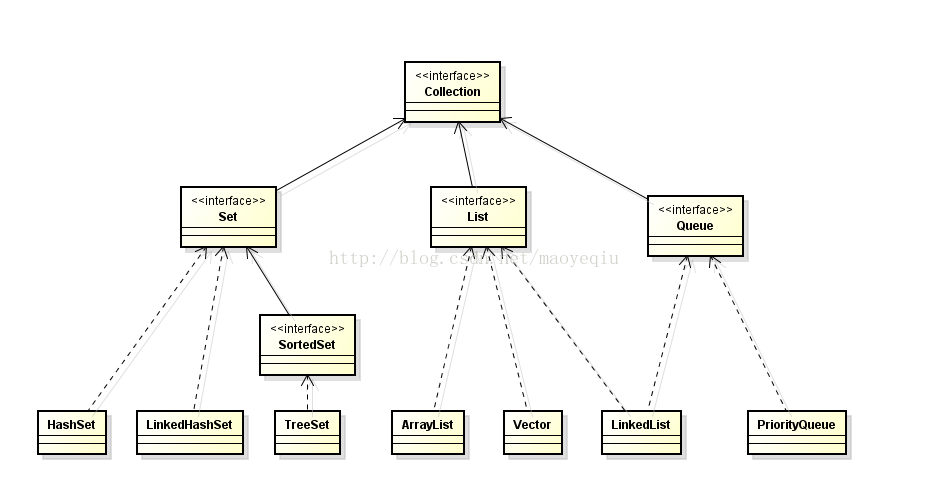
与带中断请求方法不同的是少了线程中断的判断以及异常抛出，其他操作都一样。

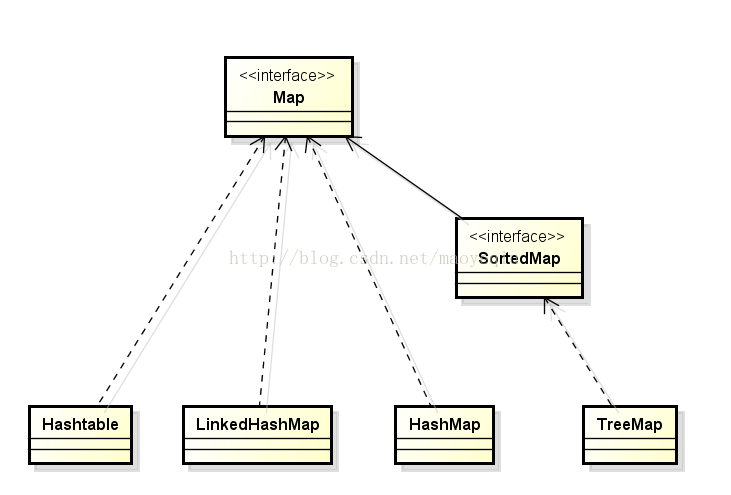
#### 释放锁

Semaphore调用了AQS中的releaseShared(int arg)方法。通过tryReleaseShared(arg)方法【释放同步状态，更新state的值，这里是无锁操作，即for死循环和CAS操作来保证线程安全问题】尝试释放同步状态，如果释放成功，那么将调用doReleaseShared()唤醒同步队列中后继结点。

doReleaseShared()方法中通过调用unparkSuccessor(h)方法唤醒head的后继结点对应的线程。注意这里把head的状态设置为Node.PROPAGATE是【在共享模式中使用表示获得的同步状态会被传播】为了保证唤醒传递。

# 第七章 Java并发容器和框架





## HashSet:

HashSet继承AbstractSet类，实现Set、Cloneable、Serializable接口。 Set接口是一种不包括重复元素的Collection，它维持它自己的内部排序。

1. \* 默认构造函数
2. \* 初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。
3. \*/
4. **public** HashSet() {
5. map = **new** HashMap<>();
6. }

LinkedHashSet:

TreeSet:

ArrayList:

LinkedList:

HashMap:

LinkedHashMap:

HashTable:

TreeMap:

## 7.1 HashMap【1.7】

### 1. HashMap的数据结构

HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap底层就是一个数组结构，数组中的每一项又是一个链表。**Entry就是数组中的元素**，每个 Map.Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。**HashMap允许存放null键和null值**。

### 2. HashMap的存取实现

**static** **class** Entry<K,V> **implements** Map.Entry<K,V> {

**final** K key;

V value;

Entry<K,V> next;

**final** **int** hash;  }

HashMap中put元素的时候，首先根据该 key 的hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，那么在这个位置上的元素将以链表的形式存放，**新加入的放在链头**，最先加入的放在链尾。新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部。

### 3. HashMap的resize（rehash）

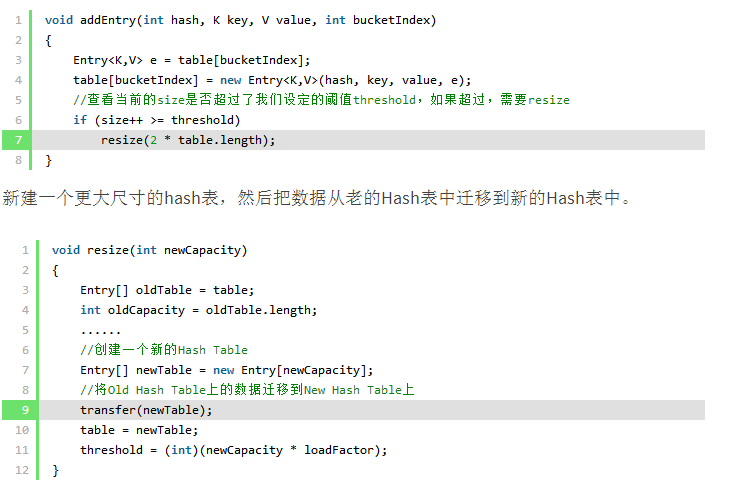
当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作。

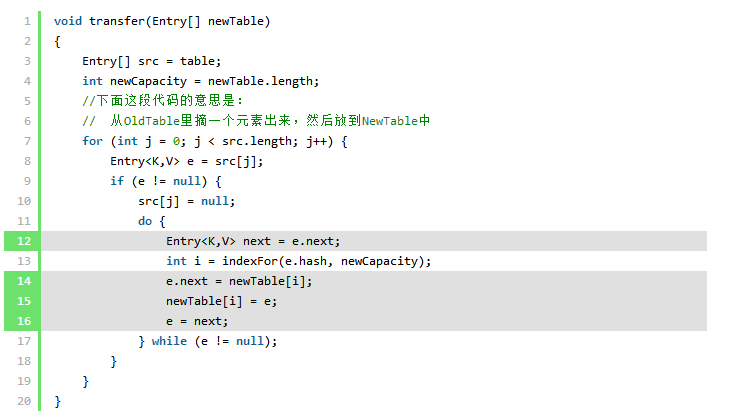
### 4. HashMap的死循环

多线程同时put时，如果同时触发了rehash操作，会导致HashMap中的链表中出现循环节点，进而使得后面get的时候，会死循环。



检查容量是否超标

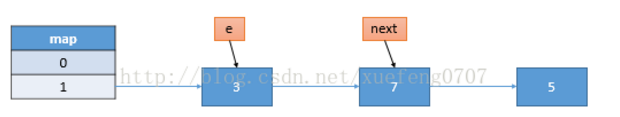




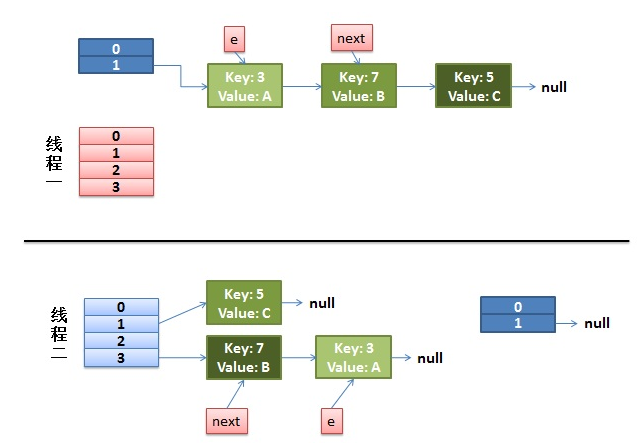
(1)线程1、2都进入transfer第三行,此时map为：



(2)线程1执行到【int i = indexFor(e.hash, newCapacity);】，线程2执行到【src[j] = null;】，此时map为：



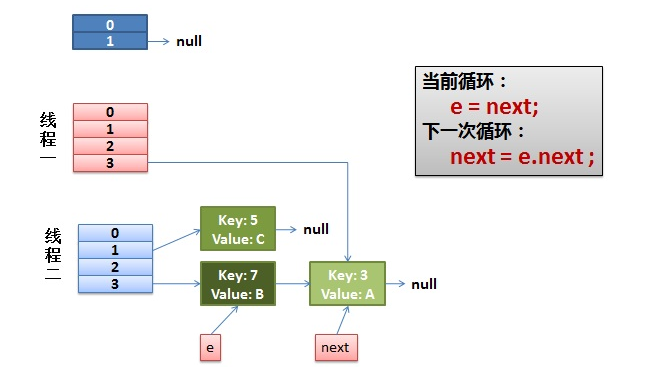
(3) 直接把Thread2执行完毕, 此时map的内容为：



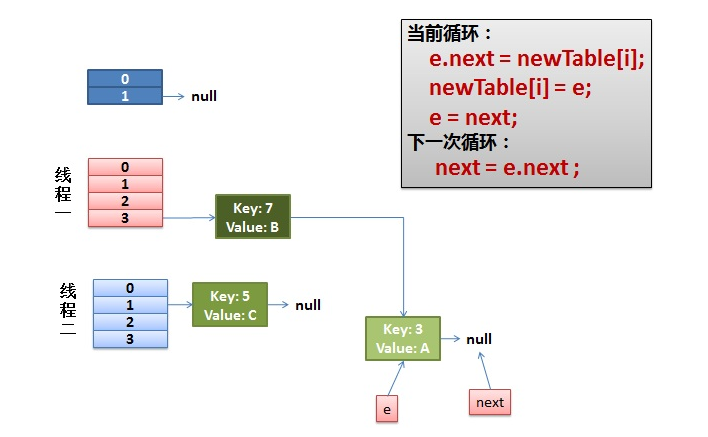
**注意：**因为Thread1的 **e 指向了key(3)，而next指向了key(7)**，其在线程二rehash后，指向了线程二重组后的链表。我们可以看到**链表的顺序被反转后**。

(4)切换回Thread1，先是执行 newTalbe[i] = e;然后是e = next，导致了**e指向了key(7)**，

而下一次循环的next = e.next导致了**next指向了key(3)**, 此时map的内容为：



(5) Thread1第2次循环后，把key(7)摘下来，放到newTable[i]的第一个，然后把e和next往下移。

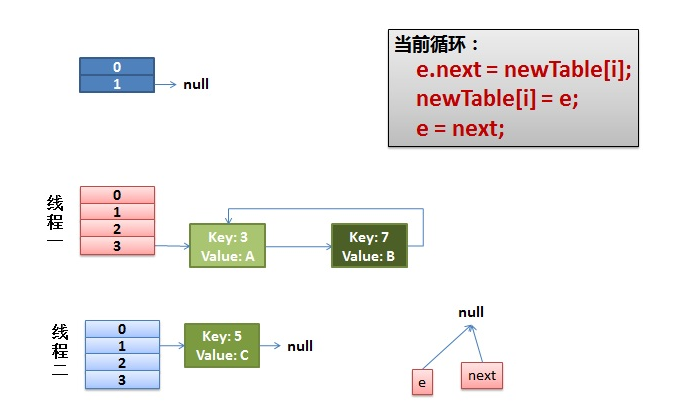


(5)环形链接出现。

e.next = newTable[i] 导致 key(3).next 指向了 key(7)

注意：此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了。

于是，当我们的线程一调用到，HashTable.get(11)时，悲剧就出现了——Infinite Loop。



## 7.2 HashMap【1.8】

1.8中hashmap**声明两对指针，维护两个连链表**，依次在末端添加新的元素。因此不会因为多线程put导致死循环，但是依然有其他的弊端，比如数据丢失等等。因此多线程情况下还是建议使用concurrenthashmap。

HashMap采用的是**数组+链表+红黑树**的形式。数组是可以扩容的，链表也是转化为红黑树的。用户可以设置的参数：初始总容量默认16，默认的加载因子0.75。初始的数组个数默认是16 容量X加载因子=阈值。一旦目前容量超过该阈值，则执行扩容操作。

HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。

什么时候扩容？

1 当前容量超过阈值

2 **当链表中元素个数超过默认设定（8个）**，**当数组的大小还未超过64的时候，此时进行数组的扩容，如果超过则将链表转化成红黑树**

当**数组大小已经超过64并且链表中的元素个数超过默认设定（8个）时，将链表转化为红黑树**。

### 1. Put过程

* 根据key计算出hash值
* hash值&（数组长度-1）得到所在数组的index
  + 如果该index位置的Node元素不存在，则直接创建一个新的Node
  + 如果该index位置的Node元素是TreeNode类型即红黑树类型了，则直接按照红黑树的插入方式进行插入
  + 如果该index位置的Node元素是非TreeNode类型则，则按照链表的形式进行插入操作

链表插入操作完成后，判断是否超过阈值TREEIFY\_THRESHOLD（默认是8），超过则要么数组扩容要么链表转化成红黑树

* 判断当前总容量是否超出阈值，如果超出则执行扩容

### 2. 扩容过程

按照**2倍**扩容的方式。这里为啥是2倍？因为2倍的话，更加容易计算他们所在的桶，并且各自不会相互干扰。桶0中的元素会被分到桶0和桶4中。

扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。

### 3. get过程

根据key计算hash值：计算出hashCode值，hash=hashCode^(hashCode>>>16);

hash值&（数组长度-1）得到所在数组的index

如果要找的key就是上述数组index位置的元素，直接返回该元素的值

如果该数组index位置元素是TreeNode类型，则按照红黑树的查询方式来进行查找O(logn)

如果该数组index位置元素非TreeNode类型，则按照链表的方式来进行遍历查询O(n)

### 4. 性能

随着size的变大，JDK1.7的花费时间是增长的趋势，而JDK1.8是明显的降低趋势，并且呈现对数增长稳定。当一个**链表太长的时候，HashMap会动态的将它替换成一个红黑树**，这样的话会将时间复杂度从O(n)降为O(logn)。

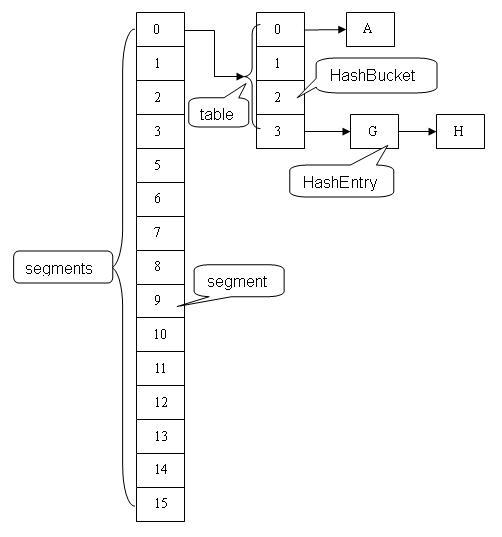
## 7.3 HashTable

HashTable是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占。如线程1使用put进行元素添加，线程2不但不能使用put添加元素，也不能使用get获取元素。hashTable 不允许key和value为null。

ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

## 7.4 ConcurrentHashMap【1.7】



### 1. 结构

ConcurrentHashMap采用了非常精妙的"分段锁"策略，ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成，主干是个Segment数组。**Segment继承了ReentrantLock**，所以它就是一种可重入锁（ReentrantLock)。HashEntry则用于存储键值对数据。

在ConcurrentHashMap里包含一个Segment数组。Segment的结构和HashMap类似，是一种数组和链表结构。一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，Segment里维护了一个HashEntry数组。当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁。（就按默认的ConcurrentLeve为16来讲，理论上就允许16个线程并发执行）。

### 2.实现原理

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

ConcurrentHashMap内部分为很多个Segment，Segment继承了ReentrantLock，表明每个segment都可以当做一个锁。对于一个key，需要经过三次（）hash操作，才能最终定位这个元素的位置，这三次hash分别为：

* 对于一个key，先进行一次hash操作，得到hash值h1，也即h1 = hash1(key)；
* 将得到的h1的高几位进行第二次hash，得到hash值h2，也即h2 = hash2(h1高几位)，通过**h2能够确定该元素的放在哪个Segment**；
* 将得到的h1进行第三次hash，得到hash值h3，也即h3 = hash3(h1)，通过h3能够**确定该元素放置在哪个HashEntry**。

ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。HashEntry代表每个hash链中的一个节点，其结构如下所示：

static final class HashEntry<K,V> {

final K key;

final int hash;

volatile V value;

volatile HashEntry<K,V> next;

}

#### 2.1 初始化Segment

Segment(float lf, int threshold, HashEntry<K,V>[] tab) {

this.loadFactor = lf;//负载因子

this.threshold = threshold;//阈值

this.table = tab;//主干数组即HashEntry数组

}

#### 2.2 初始化ConcurrentHashMap

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {

… …

//MAX\_SEGMENTS 为1<<16=65536，也就是最大并发数为65536

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

//2的sshif次方等于ssize，例:ssize=16,sshift=4;ssize=32,sshif=5

int sshift = 0;

int ssize = 1;

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

this.segmentShift = 32 - sshift;

this.segmentMask = ssize - 1;

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

//计算cap的大小，即Segment中HashEntry的数组长度，cap也一定为2的n次方.

int c = initialCapacity / ssize;

if (c \* ssize < initialCapacity)

++c;

int cap = MIN\_SEGMENT\_TABLE\_CAPACITY;

while (cap < c)

cap <<= 1;

… …

}

传入的参数有initialCapacity，loadFactor，concurrencyLevel这三个。

* initialCapacity表示新创建的这个ConcurrentHashMap的初始容量，也就是上面的结构图中的**Entry数量**。默认值为static final int DEFAULT\_INITIAL\_CAPACITY = 16;
* loadFactor表示**负载因子**，就是当ConcurrentHashMap中的元素个数大于loadFactor \* 最大容量时就需要rehash，扩容。默认值为static final float DEFAULT\_LOAD\_FACTOR = 0.75f;
* concurrencyLevel表示**并发级别**，这个值用**来确定Segment的个数**。
* **Segment数组的大小ssize**是由concurrentLevel来决定的，但是却不一定等于concurrentLevel，**ssize一定是大于或等于concurrentLevel的最小的2的N次幂。**比如：默认情况下concurrentLevel是16，则ssize为16；若concurrentLevel为14，ssize为16；若concurrentLevel为17，则ssize为32。为什么Segment的数组大小一定是2的次幂？其实主要是便于通过按位与的散列算法来定位Segment的index。
* segmentMask：段掩码，假如**ssize**为16，则段掩码为16-1=15；segments长度为32，段掩码为32-1=31。这样得到的所有bit位都为1，可以更好地保证散列的均匀性

segmentShift：2的sshift次方等于ssize，segmentShift=32-sshift。若segments长度为16，

segmentShift=32-4=28。这里之所以用32是因为ConcurrentHashMap里的hash()方法输

出的最大数是32位的。无符号右移segmentShift，则意味着只保留高几位（其余位是

没用的），然后与段掩码segmentMask位运算来定位Segment。

* Cap就是**segment里HashEntry数组的长度**。计算每个Segment平均应该放置多少个元素，**这个值c是向上取整的值**。比如初始容量为15，Segment个数为4，则每个Segment平均需要放置4个元素。Cap最小是2或者大于等于C的最小的2的N次幂。
* Segment的容量threshold=(int)cap\*loadFactor,默认情况下，并发级别concurrentLevel=16，ssize=16, 初始容量initialCapacity=16，负载因子loadFactor=0.75，通过运算cap=1, threshold = 0。

#### 2.3 定位segment

在插入和获取数据元素的时候，必须先通过散列算法定位到segment。首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再散列。之所以进行在散列，目的是减少散列冲突，使元素能够均匀地分布在不同的Segment上，从而提高容器的存取效率。如果不适用再散列，散列冲突会非常严重，只要地位一样，无论高位是什么数，起散列值总是一样。使用在散列，可以把每一位的数据都散列开了，通过这种再散列能让数字的每一位都参加到散列运算中，从而减少散列冲突。

### 3 put操作

**public** V put(K key, V value) {

Segment<K,V> s;

**if** (value == **null**)

**throw** **new** NullPointerException();

**int** hash = hash(key);

**int** j = (hash >>> segmentShift) & segmentMask;

**if** ((s = (Segment<K,V>)***UNSAFE***.getObject (segments, (j << ***SSHIFT***) + ***SBASE***)) == **null**)

s = ensureSegment(j);

**return** s.put(key, hash, value, **false**);

}

操作步骤如下：

* 判断value是否为null，如果为null，直接抛出异常。
* key通过一次hash运算得到一个hash值。
* 将得到hash值向右按位移动segmentShift位，然后再与segmentMask做&运算得到segment的索引j。默认segmentShift=28，segmentMask=15，hash值向右移动28位就变成这个样子：0000 0000 0000 0000 0000 0000 0000 xxxx，然后再用这个值与segmentMask做&运算，也就是**取最后四位的值**。**这个值确定Segment的索引**。即hash值的最高sshift位。
* 使用**Unsafe的方式**从Segment数组中获取该索引对应的Segment对象。
* 调用Segment的put方法【put操作是要加锁的】向这个Segment对象中put值，这个put操作也基本是一样的步骤（通过&运算获取HashEntry的索引，然后set）。

### 4. get操作

**public** V get(Object key) {

Segment<K,V> s;

HashEntry<K,V>[] tab;

**int** h = hash(key);

**long** u = (((h >>> segmentShift) & segmentMask) << ***SSHIFT***) + ***SBASE***;

**if** ((s = (Segment<K,V>)***UNSAFE***.getObjectVolatile(segments, u)) != **null** &&

(tab = s.table) != **null**) {

**for** (HashEntry<K,V> e = (HashEntry<K,V>) ***UNSAFE***.getObjectVolatile

(tab, ((**long**)(((tab.length - 1) & h)) << ***TSHIFT***) + ***TBASE***);

e != **null**; e = e.next) {

K k;

**if** ((k = e.key) == key || (e.hash == h && key.equals(k)))

**return** e.value;

}

}

**return** **null**;

}

操作步骤为：

* 和put操作一样，先通过key进行**两次hash确定应该去哪个Segment**中取数据。
* 使用**Unsafe获取对应的Segment**，然后再进行一次&运算得到HashEntry链表的位置，然后从链表头开始遍历整个链表（因为Hash可能会有碰撞，所以用一个链表保存），如果找到对应的key，则返回对应的value值，如果链表遍历完都没有找到对应的key，则说明Map中不包含该key，返回null。get操作是不需要加锁的（如果value为null，会调用readValueUnderLock，只有这个步骤会加锁），通过前面提到的volatile和final来确保数据安全。

### 5. size操作

size操作与put和get操作最大的区别在于，**size操作需要遍历所有的Segment才能算出整个Map的大小**，而put和get都只关心一个Segment。假设我们当前遍历的Segment为SA，那么在遍历SA过程中其他的Segment比如SB可能会被修改，于是这一次运算出来的size值可能并不是Map当前的真正大小。牛逼的作者还有一个更好的Idea：**先给3次机会，不lock所有的Segment**，遍历所有Segment，累加各个Segment的大小得到整个Map的大小，如果**某相邻的两次计算获取的所有Segment的更新的次数**（每个Segment都有一个modCount变量，这个变量在Segment中的Entry被修改时会加一，通过这个值可以得到每个Segment的更新操作的次数）**是一样的**，说明计算过程中没有更新操作，则直接返回这个值。如果这三次不加锁的计算过程中Map的更新次数有变化，则之后的计算先对所有的Segment加锁，再遍历所有Segment计算Map大小，最后再解锁所有Segment。

**for** (;;) {

**if** (retries++ == ***RETRIES\_BEFORE\_LOCK***) { //2

**for** (**int** j = 0; j < segments.length; ++j)

ensureSegment(j).lock(); // force creation

}

sum = 0L;

size = 0;

overflow = **false**;

**for** (**int** j = 0; j < segments.length; ++j) {

Segment<K,V> seg = *segmentAt*(segments, j);

**if** (seg != **null**) {

sum += seg.modCount;

**int** c = seg.count;

**if** (c < 0 || (size += c) < 0)

overflow = **true**;

}

}

**if** (sum == last)

**break**;

last = sum;

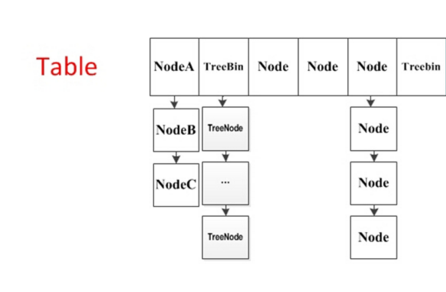
}

**containsValue操作采用了和size操作一样的想法。**

### 6. 注意事项

* ConcurrentHashMap中的key和value值都不能为null，HashMap中key可以为null，HashTable中key不能为null。
* ConcurrentHashMap是线程安全的类并不能保证使用了ConcurrentHashMap的操作都是线程安全的！
* ConcurrentHashMap的get操作不需要加锁，put操作需要加锁

## 7.5 ConcurrentHashMap【1.8】



1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。

主要设计上的变化有以下几点:

* JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，锁住node来实现减小锁粒度。如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。
* 设计了MOVED状态 当resize的中过程中 线程2还在put数据，线程2会帮助resize。
* 使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。
* sizeCtl的不同值来代表不同含义，起到了控制的作用。
* 并发控制使用Synchronized和CAS来操作。至于为什么JDK8中使用synchronized而不是ReentrantLock，可能是JDK8中对synchronized有了足够的优化吧。
* 在其他方面也有一些小的改进，比如新增字段 transient volatile CounterCell[] counterCells; 可方便的计算hashmap中所有元素的个数，性能大大优于jdk1.7中的size()方法。

### 1. 属性

下面看一下基本属性：

|  |
| --- |
| // node数组最大容量  private static final int MAXIMUM\_CAPACITY = 1 << 30;  // 默认初始值，必须是2的幕数  private static final int DEFAULT\_CAPACITY = 16;  // 负载因子  private static final float LOAD\_FACTOR = 0.75f;  // 链表转红黑树阀值,> 8 链表转换为红黑树  static final int TREEIFY\_THRESHOLD = 8;  //树转链表阀值，小于等于6  static final int UNTREEIFY\_THRESHOLD = 6;  **private transient volatile int sizeCtl;** |

### 2. 重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为null，**初始化发生在第一次插入操作**，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。
2. **nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。
3. **sizeCtl** ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。
   * **-1 代表table正在初始化**
   * **-N 表示有N-1个线程正在进行扩容操作**
   * 其余情况：  
     **1、如果table未初始化，表示table需要初始化的大小**。  
     **2、如果table初始化完成，表示table的容量**，默认是table大小的0.75倍，居然用这个公式算0.75（n - (n >>> 2)）。
4. **Node**：保存key，value及key的hash值的数据结构。

class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

volatile V val;

volatile Node<K,V> next;

... 省略部分代码

}

**其中value和next都用volatile修饰，保证并发的可见性。**

1. **ForwardingNode**：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点为null或则已经被移动。

ConcurrentHashMap在构造函数中只会初始化sizeCtl值，并不会直接初始化table，而是延缓到第一次put操作。

### 3. 实例初始化

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为**100，最终会调整成256**，确保**table的大小总是2的幂次方**.

tableSizeFor(initialCapacity + a + 1));

private static final int tableSizeFor(int c) {

int n = c - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

int resule = (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

return resule;

}

### 4. table初始化

table初始化操作**会延缓到第一次put行为**。但是put是可以并发执行的，Doug Lea是如何实现table只初始化一次的？sizeCtl默认为0，执行第一次put操作的线程会执行Unsafe.compareAndSwapInt方法修改sizeCtl为-1，有且只有一个线程能够修改成功，其它线程通过Thread.yield()让出CPU时间片等待table初始化完成。如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片。

### 5. put操作

put操作采用CAS+synchronized实现并发插入或更新操作，具体实现如下。

* **hash算法**

int hash = spread(key.hashCode());

* **table中定位索引位置，n是table的大小**

int index = (n - 1) & hash

* **获取table中对应索引的元素f。**

Doug Lea采用Unsafe.getObjectVolatile来获取。虽然table是volatile修饰的，但不能保证线程每次都拿到table中的最新元素，Unsafe.getObjectVolatile可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。

* 如果f为null，说明table中这个位置第一次插入元素，利用**Unsafe.compareAndSwapObject**方法插入Node节点。
  + 如果CAS成功，说明Node节点已经插入，随后addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
  + 如果CAS失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。
* **如果f的hash值为-1，**说明当前f是ForwardingNode节点，**意味有其它线程正在扩容，**则一起进行扩容操作。
* 其余情况把新的Node节点**按链表或红黑树的方式插入到合适的位置**，这个过程采用同步内置锁synchronized实现并。
* 在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。
  + 如果f.hash >= 0，说明f是链表结构的头结点，遍历链表，如果找到对应的node节点，则修改value，否则在链表尾部加入节点。
  + 如果f是TreeBin类型节点，说明f是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
  + 如果链表中节点数binCount >= TREEIFY\_THRESHOLD(默认是8)，则把链表转化为红黑树结构。

### 6. table扩容

当table容量不足的时候，即table的元素数量达到容量阈值sizeCtl，需要对table进行扩容。整个扩容分为两部分：

* 构建一个nextTable，大小为table的两倍。
* 把table的数据复制到nextTable中。

ConcurrentHashMap是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。先看第一步，构建nextTable，毫无疑问，这个过程只能只有单个线程进行nextTable的初始化，通过Unsafe.compareAndSwapInt修改sizeCtl值，保证只有一个线程能够初始化nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的1.5。

节点从table移动到nextTable，大体思想是遍历、复制的过程。

遍历过所有的节点以后就完成了复制工作，把table指向nextTable，并更新sizeCtl为新数组大小的0.75倍 ，扩容完成。

### 7. 红黑树构造

注意：如果链表结构中元素超过**TREEIFY\_THRESHOLD阈值，默认为8个**，则把链表转化为红黑树，提高遍历查询效率。

可以看出，生成树节点的代码块是同步的（synchronized），进入同步代码块之后，再次验证table中index位置元素是否被修改过。主要根据Node节点的hash值大小构建二叉树。

### 8. get操作

判断table是否为空，如果为空，直接返回null。

计算key的hash值，并获取指定table中指定位置的Node节点，通过遍历链表或则树结构找到对应的节点，返回value值。

### 9. 总结

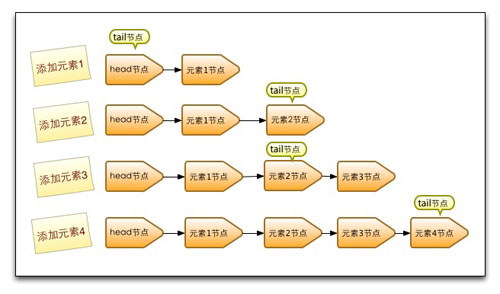
ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

## 7.6 ConcurrentLinkedQueue

ConcurrentLinkedQueue 是非阻塞队列。ConcurrentLinkedQueue 由head节点和tail节点组成，每个节点Node由节点元素item和指向下一个节点next的引用组成，节点和节点之间就是通过这个next关联起来，从而组成一张链表结构的队列。默认情况下head节点存储的元素为空，tail节点并不总是尾节点。

### 1. 入队列【CAS添加到队列尾部直至成功，永远返回true】

入队列就是讲入队节点添加到队列的尾部。



入队主要做两件事情，第一是**将入队节点设置成当前队列尾节点的下一个节点**。第二是**更新tail节点**，如果tail节点的**next节点不为空，则将入队节点设置成tail节点**，如果**tail节点的next节点为空，则将入队节点设置成tail的next节点**，所以**tail节点不总是尾节点。**

从源代码角度来看整个入队过程主要做二件事情。第一是**定位出尾节点**，第二是使用**CAS**算法能将入队节点**设置成尾节点的next节点**，如不成功则重试。

**第一步定位尾节点**。判断tail是否有next节点，这个队列刚初始化，p节点和p的next节点都等于空，需要返回head节点。

**if** (p == q){

p = (t != (t = tail)) ? t : head;

}

**第二步设置入队节点为尾节点**。p.casNext(**null**, newNode)方法用于将入队节点设置为当前队列尾节点的next节点，p如果是null表示p是当前队列的尾节点，如果不为null表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

**HOPS的设计意图**

使用hops变量来控制并减少tail节点的更新频率，并不是每次节点入队后都将tail节点更新为尾节点，而是当tail节点和尾节点的距离大于等于常量hops的值（默认等于1）时才更新tail节点。

**入队方法返回永远是 true。**

### 2. 出队列

出队列的就是从队列里返回一个节点元素，并清空该节点对元素的引用。首先**获取头节点**的元素，然后判断头节点元素**是否为空**，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果**不为空，则使用CAS**的方式将头节点的引用**设置成null**，如果CAS成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了head节点，导致元素发生了变化，需要重新获取头节点。

并不是每次出队时都更新head节点，当head节点里有元素时，直接弹出head节点里的元素，而不会更新head节点。只有当head节点里没有元素时，出队操作才会更新head节点。这种做法也是通过hops变量来减少使用CAS更新head节点的消耗，从而提高出队效率。

## 27.7 阻塞队列

阻塞的插入方法：当队列满时，队列会阻塞插入元素的线程，直到队列不满。

阻塞的移除方法：当队列空时，队列会阻塞移除元素的线程，直到队列非空。

阻塞队列常用于生产者和消费者的场景。

### BlockingQueue接口方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法/处理方式 | 抛出异常 | 返回特殊值 | 一直阻塞 | 超时退出 |
| 插入方法 | add(e) | Offer(e) | Put(e) | Offer(e,time,unit) |
| 移除方法 | Remove() | Poll() | Take() | Poll(time,unit) |
| 检查方法 | Element() | Peek() | 不可用 | 不可用 |

* 抛出异常:当队列满时，再往队列里插入元素，抛出IllegalStateException异常

当队列空时；从队列里获取元素会抛出NoSuchElementException异常；

* 返回特殊值:当队列插入元素时，会返回元素是否插入成功，成功返回true；

如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。

* 一直阻塞:一直阻塞，直到队列可用或者相应中断退出。
* 超时退出:队列会阻塞生产者线程一段时间；如果超过了指定时间，生产者线程退出。

**如果是无界阻塞队列，队列不可能会出现满的情况，所以使用put和offer方法永远不被阻塞，而是offer方法时，返回的永远是true。**

### 2. ArrayBlockingQueue

#### 2.1 基本使用

ArrayBlockingQueue 是一个用**数组**实现的**有界**阻塞队列，其内部按**先进先出**的原则对元素进行排序，其中**put方法**和**take方法**为添加和删除的阻塞方法，下面我们通过ArrayBlockingQueue队列实现一个生产者消费者的案例，Consumer 消费者和 Producer 生产者，通过ArrayBlockingQueue 队列获取和添加元素，其中消费者调用了take()方法获取元素当队列没有元素就阻塞，生产者调用put()方法添加元素，当队列满时就阻塞，通过这种方式便实现生产者消费者模式。比直接使用等待唤醒机制或者Condition条件队列来得更加简单。

public class ArrayBlockingQueueDemo {

private final static ArrayBlockingQueue<Apple> queue= new ArrayBlockingQueue<>(1);

public static void main(String[] args){

new Thread(new Producer(queue)).start();

new Thread(new Producer(queue)).start();

new Thread(new Consumer(queue)).start();

new Thread(new Consumer(queue)).start();

}

}

class Apple {

public Apple(){

}

}

/\*\*

\* 生产者线程

\*/

class Producer implements Runnable{

private final ArrayBlockingQueue<Apple> mAbq;

Producer(ArrayBlockingQueue<Apple> arrayBlockingQueue){

this.mAbq = arrayBlockingQueue;

}

@Override

public void run() {

while (true) {

Produce();

}

}

private void Produce(){

try {

Apple apple = new Apple();

mAbq.put(apple);

System.out.println("生产:"+apple);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 消费者线程

\*/

class Consumer implements Runnable{

private ArrayBlockingQueue<Apple> mAbq;

Consumer(ArrayBlockingQueue<Apple> arrayBlockingQueue){

this.mAbq = arrayBlockingQueue;

}

@Override

public void run() {

while (true){

try {

TimeUnit.MILLISECONDS.sleep(1000);

comsume();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

private void comsume() throws InterruptedException {

Apple apple = mAbq.take();

System.out.println("消费Apple="+apple);

}

}

ArrayBlockingQueue内部的阻塞队列是通过重入锁ReenterLock和Condition条件队列实现的，所以ArrayBlockingQueue中的元素存在公平访问与非公平访问的区别，对于公平访问队列，被阻塞的线程可以按照阻塞的先后顺序访问队列，即先阻塞的线程先访问队列。而非公平队列，当队列可用时，阻塞的线程将进入争夺访问资源的竞争中，也就是说谁先抢到谁就执行，没有固定的先后顺序。创建公平与非公平阻塞队列代码如下：

//默认非公平阻塞队列

ArrayBlockingQueue queue = new ArrayBlockingQueue(2);

//公平阻塞队列

ArrayBlockingQueue queue1 = new ArrayBlockingQueue(2,true);

//构造方法源码

public ArrayBlockingQueue(int capacity) {

this(capacity, false);

}

public ArrayBlockingQueue(int capacity, boolean fair) {

if (capacity <= 0)

throw new IllegalArgumentException();

this.items = new Object[capacity];

lock = new ReentrantLock(fair);

notEmpty = lock.newCondition();

notFull = lock.newCondition();

}

#### 2.2 原理概要

ArrayBlockingQueue的内部是通过一个可重入锁ReentrantLock和两个Condition条件对象来实现阻塞。

### 3. LinkedBlockingQueue

**Offer()方法**做了两件事，第一件事是判断队列是否满，满了就直接释放锁，没满就将节点封装成Node入队，然后再次判断队列添加完成后是否已满，不满就继续唤醒在条件对象notFull上的添加线程。第二件事是，判断是否需要唤醒等到在notEmpty条件对象上的消费线程。

为什么添加完成后是继续唤醒在条件对象notFull上的添加线程而不是像ArrayBlockingQueue那样直接唤醒notEmpty条件对象上的消费线程？而又为什么要当if (c == 0)时才去唤醒消费线程呢？

唤醒添加线程的原因，在添加新元素完成后，会判断队列是否已满，不满就继续唤醒在条件对象notFull上的添加线程，这点与前面分析的ArrayBlockingQueue很不相同，在ArrayBlockingQueue内部完成添加操作后，会直接唤醒消费线程对元素进行获取，这是因为**ArrayBlockingQueue只用了一个ReenterLock同时对添加线程和消费线程进行控制，这样如果在添加完成后再次唤醒添加线程的话，消费线程可能永远无法执行**，而对于LinkedBlockingQueue来说就不一样了，其内部对添加线程和消费线程分别使用了各自的ReenterLock锁对并发进行控制，也就是说添加线程和消费线程是不会互斥的，所以添加锁只要管好自己的添加线程即可，添加线程自己直接唤醒自己的其他添加线程，如果没有等待的添加线程，直接结束了。如果有就直到队列元素已满才结束挂起，当然offer方法并不会挂起，而是直接结束，只有put方法才会当队列满时才执行挂起操作。注意消费线程的执行过程也是如此。这也是为什么LinkedBlockingQueue的吞吐量要相对大些的原因。

为什么要判断if (c == 0)时才去唤醒消费线程呢，这是因为消费线程一旦被唤醒是一直在消费的（前提是有数据），所以c值是一直在变化的，c值是添加完元素前队列的大小，此时c只可能是0或c>0，如果是c=0，那么说明之前消费线程已停止，条件对象上可能存在等待的消费线程，添加完数据后应该是c+1，那么有数据就直接唤醒等待消费线程，如果没有就结束啦，等待下一次的消费操作。如果c>0那么消费线程就不会被唤醒，只能等待下一个消费操作（poll、take、remove）的调用，那为什么不是条件c>0才去唤醒呢？我们要明白的是消费线程一旦被唤醒会和添加线程一样，一直不断唤醒其他消费线程，如果添加前c>0，那么很可能上一次调用的消费线程后，数据并没有被消费完，条件队列上也就不存在等待的消费线程了，所以c>0唤醒消费线程得意义不是很大，当然如果添加线程一直添加元素，那么一直c>0，消费线程执行的换就要等待下一次调用消费操作了（poll、take、remove）。

## 7.8 Fork/Join框架

Fork/Join框架是Java7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小人物，最终汇合每个小任务结果后得到大任务结果的框架。

### 1. 工作窃取算法

工作窃取算法（work-stealing）算法是指某个线程从其让队列里窃取任务来执行。干完活的线程去其他线程的队列里窃取一个任务来执行。

通常使用双端队列，被窃取任务线程永远从双端队列的头部那任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

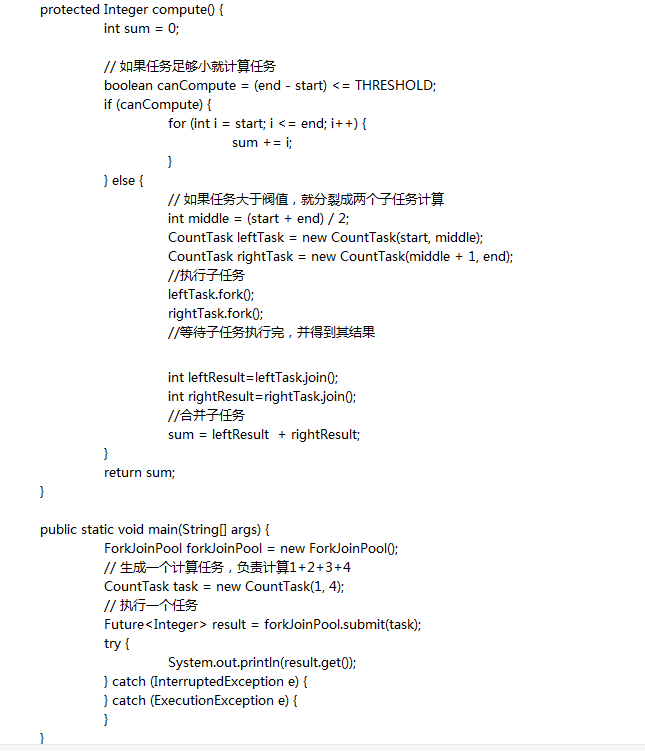
### 2. Fork/join框架设计

步骤一：分割任务。Fork类来把大任务分割成子任务，知道分割出的子任务足够小

步骤二：执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程获取任务执行，执行完的结果都同一放在一个队列中，启动一个线程从队列里拿数据合并。

使用步骤：① 创建一个forkjoin任务，需要继承 RecursiveTask（有返回结果）或者RecursiveAction（没有返回结果），重写compute方法；② ForkJoinTask通过ForkJoinPool来执行。

### 3. 使用Fork/join



4. 实现原理

ForkJoinPool由ForkJoinTask数组（存放任务）和ForkJoinWorkerThread数组（执行任务）组成。

执行任务：ForkJoinTask.fork() 🡪ForkJoinWorkerThread.pushTask()异步执行

🡪ForkJoinPool.signalWork()唤醒/创建线程执行

等待任务执行完：ForkJoinTask.join()🡪dojoin()返回结果

1. Normal:完成；②cancelled:取消；③signal:信号；④exceptional:异常

# 第八章 13个原子操作类

### 1、Java中实现原子操作

在Java中可以通过锁和循环CAS的方式来实现原子操作。从Java1.5开始，JDK的并发包里提供了一些来支持原子操作，如AtomicBoolean、AtomicInteger和AtomicLong。

### 2、CAS实现原子操作的三大问题

**1) ABA问题**：

CAS需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是A，变成B，又变成了A**，那么使用CAS进行检查时会发现它的值没有发生变化，实际上变化了。

ABA问题的解决思路就是：使用版本号。在变量前面追加上版本号。

从JDK1.5开始，JDK的Atomic包里提供了一个类**AtomicStampedReference来解决ABA问题**。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

**2)循环时间长开销大**

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

**3)只能保证一个共享变量的原子操作**

对多个共享变量操作时，循环CAS无法保证操作的原子性，这个时候可以用锁。从JDK1.5开始，JDK提供了**AtomicReference类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行CAS操作。

### 3. CAS原理

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

compareAndSet**利用JNI来完成CPU指令的操作**。

*public final boolean compareAndSet(int expect, int update) {     
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
    }*

CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用。

而compareAndSwapInt就是借助C来调用CPU底层指令实现的。下面从分析比较常用的CPU（intel x86）来解释CAS的实现原理。

程序会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。如果程序是在多处理器上运行，就为cmpxchg指令加上lock前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略lock前缀。

lock前缀的说明如下：

①确保对内存的读-改-写操作原子执行。

②禁止该指令与之前和之后的读和写指令重排序。

1. 把写缓冲区中的所有数据刷新到内存中。

# 第九章 并发工具类

## 9.1 等待多线程完成的CountDownLatch

CountDownLatch允许一个或多个线程等待其他线程完成操作。Join用于让当前执行线程等待join线程执行结束。其实现原理是不停检查join线程是否存活，如果线程存活则让当前线程永远等待。

CountDownLatch通过构造函数传入一个初始计数值，调用者可以通过调用CounDownLatch对象的**cutDown()方法，来使计数减1**；如果调用对象上的**await()方法，那么调用者就会一直阻塞在这里，直到别人通过cutDown方法，将计数减到0，才可以继续执行**。

|  |
| --- |
|  |

public CountDownLatch(int count) {  };

//参数count为计数值

public void await() throws InterruptedException { };

//调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };

//和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public void countDown() { }; //将count值减1

## 9.2 同步屏障CyclicBarrier

CyclicBarrier让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

CyclicBarrier的默认构造方法会传int型参数，表示屏障拦截的线程数量，每个线程调用await方法表示已经到达了屏障，当前线程被阻塞。还提供了一个更高级的构造函数

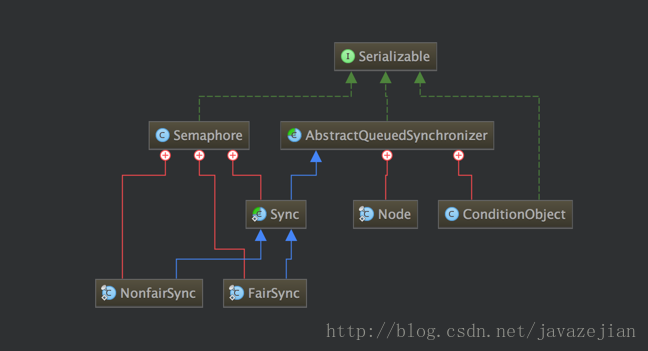
**public** CyclicBarrier(**int** parties, Runnable barrierAction)

用于在线程到达屏障时，优先执行barrierAction。

**CyclicBarrier和 CountDownLatch的区别**：CountDownLatch的计数器只能使用一次，而CyclicBarrier 的计数器可以使用reset方法重置。同时CyclicBarrier提供了获取阻塞的线程数量的方法和阻塞线程是否被中断的方法。

## 9.3 控制并发线程数的Semaphore

信号量(Semaphore)，维护了一个许可集，我们在初始化Semaphore时需要为这个许可集传入一个数量值，该数量值代表同一时间能访问共享资源的线程数量。线程可以通过**acquire()**方法获取到一个许可，然后对共享资源进行操作，注意如果许可集已分配完了，那么线程将进入等待状态，直到其他线程释放许可才有机会再获取许可，线程释放一个许可通过**release()**方法完成。



Semaphore内部同样存在继承自AQS的内部类Sync以及继承自Sync的公平锁(FairSync)和非公平锁(NofairSync)。Semaphore的内部类公平锁(FairSync)和非公平锁(NoFairSync)各自实现不同的获取锁方法即tryAcquireShared(int arg)，而释放锁tryReleaseShared(int arg)的操作交由Sync实现。

Reentrank获取/释放锁都返回true或false。获取锁，状态值和0比较，不为0时，比较是否为当前线程，状态值加1。释放锁，状态值和0比较。Semaphore释放锁，返回true和false。获取锁返回剩余许可数量。如果state值代表的许可数已为0，则请求线程将被加入到同步队列并阻塞。

调用Semaphore的acquire()方法后将会调用到AQS的acquireSharedInterruptibly()，也就是说Semaphore的acquire()方法也是可中断的。在acquireSharedInterruptibly()方法内部先进行了线程中断的判断，如果没有中断，那么先尝试调用tryAcquireShared(arg)方法获取同步状态，如果获取失败调用doAcquireSharedInterruptibly(arg);方法加入同步队列等待。这里的tryAcquireShared(arg)是个**模板方法**，AQS内部没有提供具体实现，由子类实现，也就是有Semaphore内部自己实现。

非公平锁：

protected int tryAcquireShared(int acquires) {

return nonfairTryAcquireShared(acquires);

}

abstract static class Sync extends AbstractQueuedSynchronizer {

final int nonfairTryAcquireShared(int acquires) {

//使用死循环

for (;;) {

int available = getState();

int remaining = available - acquires;

//判断信号量是否已小于0或者CAS执行是否成功

if (remaining < 0 || compareAndSetState(available, remaining))

return remaining;

}

}

}

先**获取state的值**，并**执行减法操作**，得到remaining值，如果remaining**不小于0**，那么线程**获取同步状态成功**，可访问共享资源，并**更新state的值**；如果remaining**小于0**，那么线程获取同步状态失败，将被**加入同步队列**(通过doAcquireSharedInterruptibly(arg))，注意Semaphore的acquire()可能存在并发操作，因此nonfairTryAcquireShared()方法体内部采用**无锁(CAS)**并发的操作保证对state值修改的安全性。如果尝试获取同步状态失败，那么将会执行**doAcquireSharedInterruptibly**(int arg)方法. 由于当前线程没有获取同步状态，因此创建一个**共享模式（Node.SHARED）的结点**并通过addWaiter(Node.SHARED)**加入同步队列**，加入完成后，当前线程进入自旋状态，首先判断前驱结点是否为head，如果是，那么尝试获取同步状态并返回r值，如果r大于0，则说明获取同步状态成功，将当前线程设置为head并传播，传播指的是，同步状态剩余的许可数值不为0，通知后续结点继续获取同步状态，到此方法将会return结束，获取到同步状态的线程将会执行原定的任务。但如果前驱结点不为head或前驱结点为head并尝试获取同步状态失败，那么调用shouldParkAfterFailedAcquire(p, node)方法判断前驱结点的waitStatus值是否为SIGNAL并调整同步队列中的node结点状态，如果返回true，那么执行parkAndCheckInterrupt()方法，将当前线程挂起并返回是否中断线程的flag。

到此，加入同步队列的整个过程完成。这里小结一下，在AQS中存在一个变量state，当我们创建Semaphore对象**传入许可数值**时，**最终会赋值给state**，state的数值代表同一个时刻可同时操作共享数据的线程数量，每当一个线程请求(如调用Semaphored的acquire()方法)获取同步状态成功，state的值将会减少1，直到state为0时，表示已没有可用的许可数，也就是对共享数据进行操作的线程数已达到最大值，其他后来线程将被阻塞，此时AQS内部会将线程封装成共享模式的Node结点，加入同步队列中等待并开启自旋操作。只有当持有对共享数据访问权限的线程执行完成任务并释放同步状态后，同步队列中的对于的结点线程才有可能获取同步状态并被唤醒执行同步操作，注意在同步队列中获取到同步状态的结点将被设置成head并清空相关线程数据(毕竟线程已在执行也就没有必要保存信息了)，AQS通过这种方式便实现共享锁。

Semaphore， AQS中通过state值来控制对共享资源访问的线程数，每当线程请求同步状态成功，state值将会减1，如果超过限制数量的线程将被封装共享模式的Node结点加入同步队列等待，直到其他执行线程释放同步状态，才有机会获得执行权，而每个线程执行完成任务释放同步状态后，state值将会增加1，这就是共享锁的基本实现模型。至于公平锁与非公平锁的不同之处在于公平锁会在线程请求同步状态前，判断同步队列是否存在Node，如果存在就将请求线程封装成Node结点加入同步队列，从而保证每个线程获取同步状态都是先到先得的顺序执行的。非公平锁则是通过竞争的方式获取，不管同步队列是否存在Node结点，只有通过竞争获取就可以获取线程执行权。

# 第十章 线程池-Executor框架

Eexecutor作为灵活且强大的**异步执行框架**，其支持多种不同类型的任务执行策略，提供了一种标准的方法将任务的提交过程和执行过程解耦开发，基于生产者-消费者模式，其提交任务的线程相当于生产者，执行任务的线程相当于消费者，并用Runnable来表示任务，Executor的实现还提供了对生命周期的支持，以及统计信息收集，应用程序管理机制和性能监视等机制。

## 10.1 线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程：

1. 线程池判断**核心线程池**里的线程都在执行任务。如果不是，创建一个新的工作线程执行任务，如果是，进入2);
2. 线程池判断**工作队列**是否已满。没满，存储到这个工作队列中，否则，进入3);
3. 线程池判断**线程池的线程**是否都在执行任务，若满，则采用拒绝策略来处理这个任务。

ThreadPoolExecutor执行execute方法分为以下4种情况：

1. 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务【获取全局锁】；
2. 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。
3. 如果不能将任务加入BlockingQueue(队列已满)，创建新的线程来处理任务；
4. 如果当前运行的线程超过maximumPoolSize，任务将被拒绝。

## 10.2 线程池的使用

### 1. 创建线程池

通过ThreadPoolExecutor来创建一个线程池：

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,

TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler )

* corePoolSize：线程池的基本大小；
* maximumPoolSize：线程池的最大数量；
* keepAliveTime：多余线程活动保持的时间
* workQueue：任务队列，用于保存等待执行的任务的阻塞队列，可以选择：
  + ArrayBlockingQueue：基于数组的有界阻塞队列，FIFO。
  + LinkedBlockingQueue：基于链表的阻塞队列，Executors.newFixedThreadPool()
  + SynchronousQueue：不存储元素的阻塞队列。Executors.newCachedThreadPool()
  + PriorityBlockingQueue：具有优先级的无线阻塞队列。
* Handler：拒绝策略。
  + AbortPolicy：直接抛出异常。
  + CallerRunsPolicy：只用调用者所在线程来运行任务。
  + DiscardOldestPolicy：丢弃队列里最近的一个任务，并指向当前任务。
  + DiscardPolicy：不处理，丢弃掉。

### 2. 提交任务

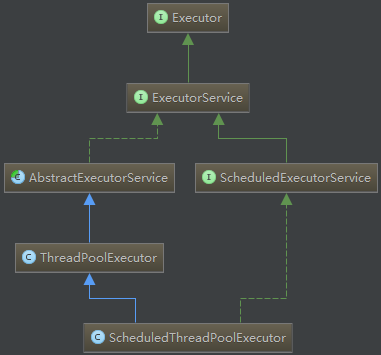
* + execute()：提交不需要返回值的任务
  + submit()：提交需要返回值的任务，返回一个future类型的对象，通过get()获取返回值。

### 3. 关闭线程池

遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。

* + shutdown()：只是将线程池的状态设置为shutdown状态，然后中断所有没有正在执行任务的线程。
  + shutdownNow()：首先将线程池的状态设置为STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表。

## 10.1 Executor的UML图



**Executor：**一个接口，其定义了一个接收Runnable对象的方法executor(Runnable command)。

**ExecutorService：**是一个比Executor使用更广泛的子类接口，其提供了生命周期管理的方法，以及可跟踪一个或多个异步任务执行状况返回Future的方法

**AbstractExecutorService：**ExecutorService执行方法的默认实现

**ScheduledExecutorService：**一个可定时调度任务的**接口**

**ScheduledThreadPoolExecutor：**ScheduledExecutorService的实现，一个可定时调度任务的线程池。

**ThreadPoolExecutor：**线程池，可以通过调用Executors以下静态工厂方法来创建线程池并返回一个ExecutorService对象。

## 10.2 ThreadPoolExecutor构造函数

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory, RejectedExecutionHandler handler)

//后两个参数为可选参数

参数说明：

**corePoolSize：**核心线程数，如果运行的线程少于corePoolSize，则创建新线程来执行新任务，即使线程池中的其他线程是空闲的。

**maximumPoolSize:** 最大线程数，可允许创建的线程数。

**keepAliveTime:** 如果线程数多于corePoolSize,多余的线程的存活时间。

**Unit:** keepAliveTime参数的时间单位。

**workQueue:** 保存任务的阻塞队列，被提交但尚未被执行的任务。

threadFactory:使用ThreadFactory创建新线程，默认使用defaultThreadFactory创建线程

handle:定义处理被拒绝任务的策略，默认使用ThreadPoolExecutor.AbortPolicy。

## **10.3 .Executors**

提供了一系列静态工厂方法用于创建各种线程池。

**newFixedThreadPool**:创建可重用且**固定线程数**的线程池。

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

**newSingleThreadExecutor:**创建一个单线程的Executor

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

**newScheduledThreadPool:**创建一个可延迟执行或定期执行的线程池。模拟心跳机制

public ScheduledThreadPoolExecutor(int corePoolSize) {

super(corePoolSize, Integer.MAX\_VALUE, 0, TimeUnit.NANOSECONDS,

new DelayedWorkQueue());

}

**newCachedThreadPool:**创建可缓存的线程池，如果线程池中的线程在60秒未被使用就将被移除。

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.***MAX\_VALUE***, 60L, TimeUnit. ***SECONDS***,

new SynchronousQueue<Runnable>()));

}

## 10.4 Executor的生命周期

ExecutorService提供了管理Eecutor生命周期的方法，ExecutorService的生命周期包括了：运行、关闭和终止三种状态。

ExecutorService在初始化创建时处于运行状态。

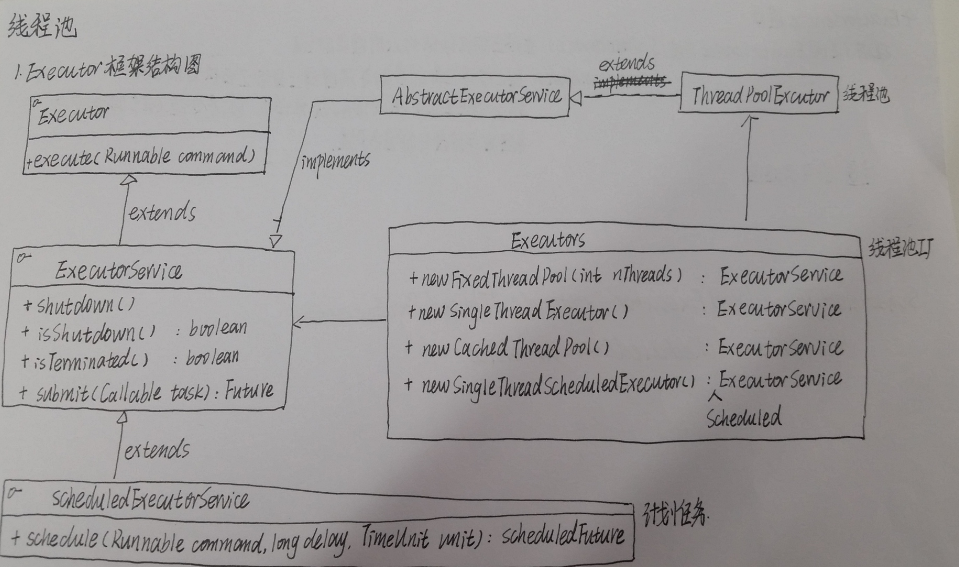
shutdown方法等待提交的任务执行完成并不再接受新任务，在完成全部提交的任务后关闭

shutdownNow方法将强制终止所有运行中的任务并不再允许提交新任务

排队有三种通用策略：

**1.直接提交队列。**默认选项是SynchronousQueue，它将任务直接提交给线程而不保持它们。提交的任务不会被真实的保存，总是加你个新任务提交给线程执行，如果没有空闲的进程，则尝试创建新的进程。达到最大值，则拒绝。  
 **2.无界队列。**使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。 这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）  
    **3. 有界队列**。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。

## **10.5 Executor框架**



* Executor：是一个接口，将任务的提交和任务的执行分离开来；
* ThreadPoolExecutor：线程池的核心实现类，用来执行被提交的任务；
* ScheduledThreadPoolExecutor：实现类，在给定的延迟后运行命令，或者定时执行；
* Future接口和实现Future接口的FutureTask类，代表异步计算的结果。

### 1. ThreadPoolExecutor

ThreadPoolExecutor通常使用工厂类Executors来创建3种类型的ThreadPoolExecutor：

* newFixedThreadPool:创建可重用且固定线程数的线程池。

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

* newSingleThreadExecutor:创建一个单线程的Executor

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

* newCachedThreadPool:创建可缓存的线程池，如果线程池中的线程在60秒未被使用就将被移除。

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit. SECONDS,

new SynchronousQueue<Runnable>()));

}

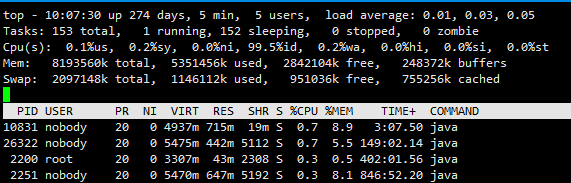
### 2. ScheduledThreadPoolExecutor

通常使用工厂类Executors来创建2种类型的ScheduledThreadPoolExecutor：

* newScheduledThreadPool:创建一个可延迟执行或定期执行的线程池。模拟心跳机制
* public ScheduledThreadPoolExecutor(int corePoolSize) {
* super(corePoolSize, Integer.MAX\_VALUE, 0, TimeUnit.NANOSECONDS,
* new DelayedWorkQueue());
* }
* 下面提供了四种预定义的处理程序策略：  
         1. 在默认的 ThreadPoolExecutor.AbortPolicy 中，处理程序遭到拒绝将抛出运行时RejectedExecutionException。  
         2. 在 ThreadPoolExecutor.CallerRunsPolicy中，线程调用运行该任务的 execute 本身。  
         此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。  
         3. 在ThreadPoolExecutor.DiscardPolicy中，不能执行的任务将被删除。  
         4. 在ThreadPoolExecutor.DiscardOldestPolicy 中，如果执行程序尚未关闭，

# 第十一章 线上问题定位

1. Top 查看每个进程的情况：关注command为Java的%CPU



1. 使用top的交互命令数字 1 查看每个CPU的性能数据。

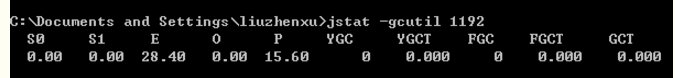
如果某个CPU的利用率到达100%，可能写了一个死循环。

1. 使用top的交互命令H查看每个线程的性能数据。

* 某个线程CPU利用率一直100%，则说明是这个线程有可能有死循环。
* 某个线程一直在TOP10，这个线程可能有性能问题。
* CPU利用率高的几个线程在不断变化，说明并不是由某一个线程导致的。

对于第一种情况，也可能是GC造成的：

* jstat –gcutil 线程pid 1000 5



S0、S1 代表两个Survivor区；

E 代表 Eden 区；

O（Old）代表老年代；

P（Permanent）代表永久代；

YGC（Young GC）代表Minor GC；

YGCT代表Minor GC耗时；

FGC（Full GC）代表Full GC耗时；

GCT代表Minor & Full GC共计耗时。

* 还可dump下来： jstack pid > /home/lhelper/dump17