### AQS类：

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

AQS类中定义了锁的共有属性，共有方法及模板方法，共有属性及共有方法如双向队列，状态位，park和unpark线程。模板方法中需要调用了一些需要在子类中实现的方法，这个可以看具体的子类实现。主要有下面几个：

**isHeldExclusively() //该线程是否正在独占资源。只有用到condition才需要去实现它。**

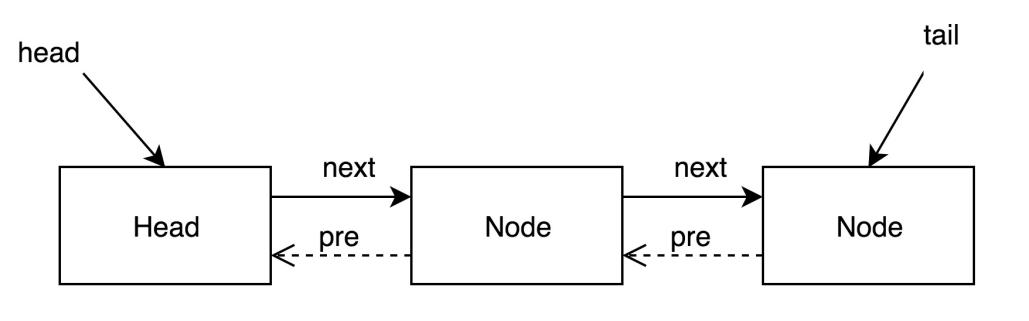
**tryAcquire(int) //独占方式。尝试获取资源，成功则返回true，失败则返回false。**

**tryRelease(int) //独占方式。尝试释放资源，成功则返回true，失败则返回false。**

**tryAcquireShared(int) //共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。**

**tryReleaseShared(int) //共享方式。尝试释放资源，成功则返回true，失败则返回false。**

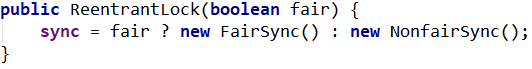
AQS使用一个FIFO队列表示排队等待锁的线程，队列头结点称作“哨兵节点”或者“哑结点”，它**不与任何线程关联**。其他的节点与等待线程关联，每个阶段维护一个等待状态waitStatus。如图

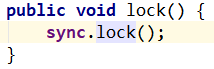


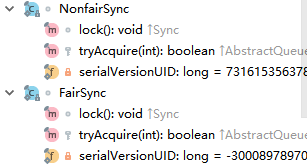
### ReentrantLock类：

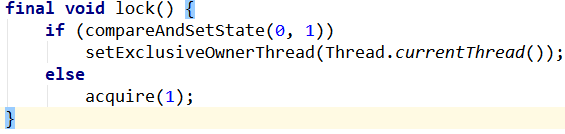
1596376234(1)

默认创建非公平锁

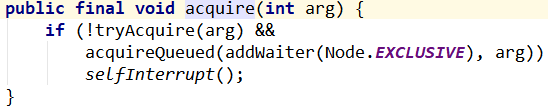




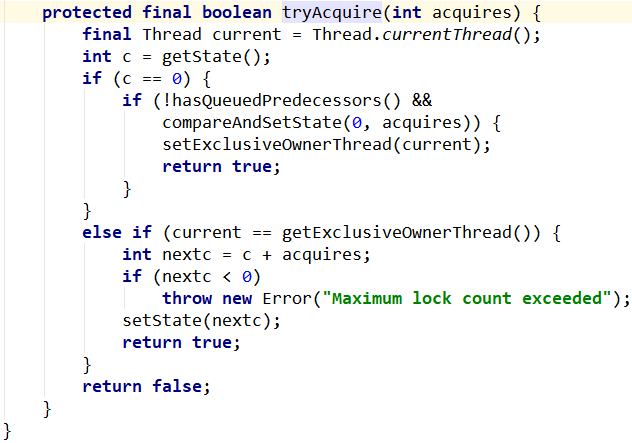




**公平锁的Lock方法直接调用acquire(1)方法，而不是先试图获取锁并改变状态位。**



这是父类AQS中实现的模板方法，其中tryAcquire()的具体方法在其子类中实现，下面看一下公平锁和非公平锁acquire的区别。这里的addWaiter方法是试图将抢占锁失败的线程包装成Node节点类放入双向等待队列中。AcquireQueued()方法在死循环中不断尝试获取锁。在下面详细说明。



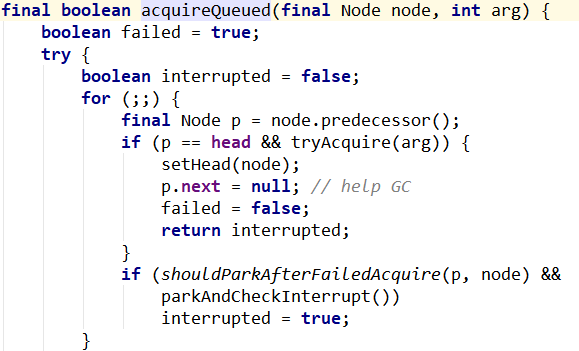
**在非公平锁中没有!hasQueuedPredecessors()方法，这个方法是先看多个线程排队的AQS队列中是否轮到该线程执行。如果是才能尝试获取锁。而非公平**

**锁直接试图抢占锁**。

**可重入锁：**

由上可知，在当前线程已经持有锁的情况下，可以直接再次获取锁，并且state值加1，在释放锁时会对锁进行减1操作。

在进行状态为设置时因为涉及到状态位的变化，并且是多线程共享一把锁，所以需要进行CAS原子操作。



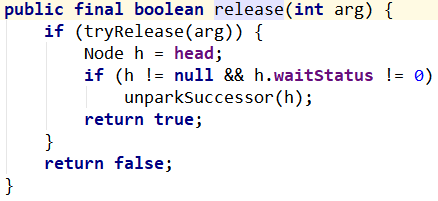
在该方法中不断排队尝试获取锁，shouldParkAfterFailedAcquire()方法会将等待队列中的已经执行的线程删除掉，**如果前一个节点是signal状态说明在本线程处理之前还有线程需要处理，则将当前线程挂起，防止一直不断尝试获取锁**。

**可中断锁：**

在lockInterruptibly()方法中，这里不是将interrupted设置为true而是在interrupted属性为true时直接抛出中断异常。

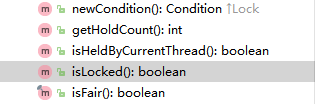
tryLock()方法并没有进入**循环获取锁的代码（即aquireQueue）**中，获取锁失败后直接返回。

tryLock(long,time)方法超时获取锁，规定时间内获取锁成功则返回true，否则返回false。



UnLock()方法比较简单，其调用release(1),tryRelease()方法释放锁，这里面猜想可知，如果state为0，将独占锁属性设置为null，否则只是将state-1,**在释放锁成功后，将头节点的下个节点唤醒。**

**其他常用方法：**

****

**参考链接：<https://www.cnblogs.com/weiqihome/p/9665718.html>**

**总结：ReentranLock获取锁的过程：**

**非公平锁：先尝试设置锁的状态位state，设置成功则设置独占线程属性，设置失败则进入acquire()方法再次试图获取锁，如果状态位0则先设置状态位并设置独占线程，否则看当前线程是否与独占线程相同，即当前线程是否正在重入锁，是的话则将state状态位+1,；如果获取锁（所谓获取锁就是设置状态位和设置独占线程）失败，则加入同步队列中。在同步队列中循环获取锁，但是只有头结点才是活跃结点，后面的线程会被挂起。**

**公平锁：直接进去acquire()方法（公平锁的重写方法）判断state状态位是否为0，为0，则先判断同步线程是否为空，为空才可以设置状态位并设置独占锁，为1，与公平锁一样试图重入锁。其他与公平锁一样。**

### ReentrantReadWriteLock类（与ReentrantLock大同小异对比起来看）：

读写锁说明：

线程进入读锁的前提条件：

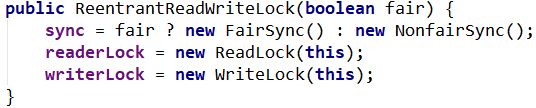
没有其他线程的写锁，可以有其他线程的写请求

没有写请求或者**有写请求，但调用线程和持有锁的线程是同一个。**

线程进入写锁的前提条件：

没有其他线程的读锁

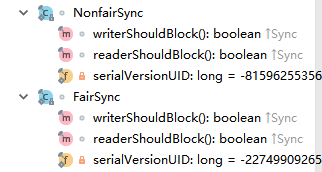
没有其他线程的写锁



1596384894(1)

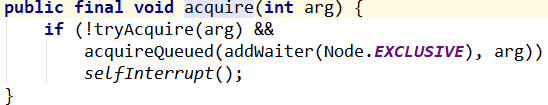
1596384918(1)

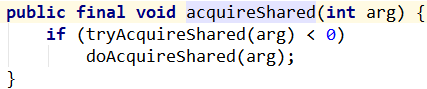
和ReentrantLock一样默认创建非公平锁。



公平锁和非公平锁的区别在于\*\*\*\*Block方法，其主要区别仍然是是否需要按顺序获取锁。不再赘述。

WriteLock和ReadLock必须要用同一把锁，不然做不到读写分离，写写分离。

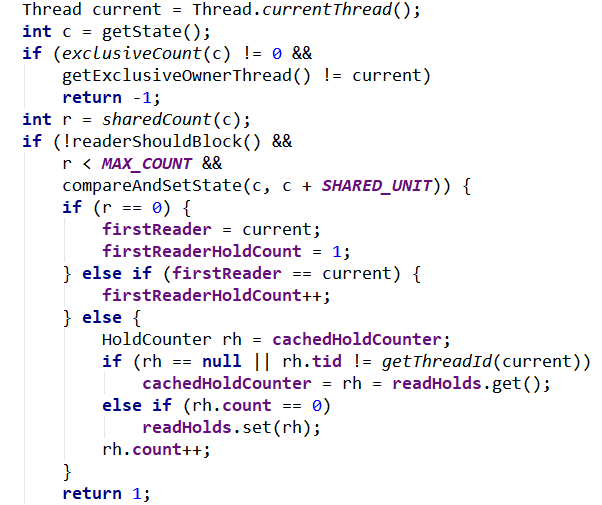




WriteLock的lock调用tryAcquire()方法，ReadLock的lock方法调用tryAcquireShared方法

**读锁和写锁在属性的区别主要有：读锁是可以共享而且重入的，因此读锁需要额外的属性来标识当前线程获取锁的数量。因为与当前线程有关，所以需要一个线程相关的ThreadLocal变量来存储该线程获取读锁的数量。**

**下面以获取读锁为例来演示：**

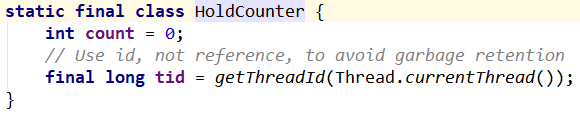
****

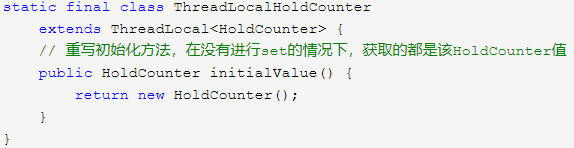
**1596417799(1)**

**c是32位二进制的变量，高16位标识读锁数量，低16为标识写锁数量。这里也可以看到公平锁和非公平锁的readerShouldBlock()在这里起的作用。**

**其实可以在当前线程绑定一个变量来存储读锁的数量，但是由于threadLocal.get()方法获取也需要时间，这里直接将当前线程的读锁数量放在一个变量firstReaderHoldCount中。在fullTryAcquireShared中逻辑差不多，死循环获取锁而已。TryLock(),unLock(),LockInterrupt()等方法以及写锁的各种方法逻辑都差不多。**

**这里还是将HoldCounter和**ThreadLocalHoldCounter**类的声明贴出来：**

****

****

**通过ThreadLocalHoldCounter获取与该线程的绑定的HoldCounter,每个线程的holdCounter保持的锁数量就是该线程获取的锁数量。**

**备注：每个线程类Thread类中有一个**ThreadLocal.ThreadLocalMap**类的属性，这个属性里面维持了一个数组，数组中的元素时一个Entry<ThreadLocal,object>,只要在该线程中就能获得该ThreadLocal对应的value。注意：ThreadLocalMap中的数组在放入元素时并不是顺序放入，而是根据该ThreadLcoal的偏移量和数组的长度的与操作找到的数组下表。若该位置已经有元素则向后位移一个位置，直到找到空位置。**

**ThreadLocal造成内存泄漏的原因及解决办法：**

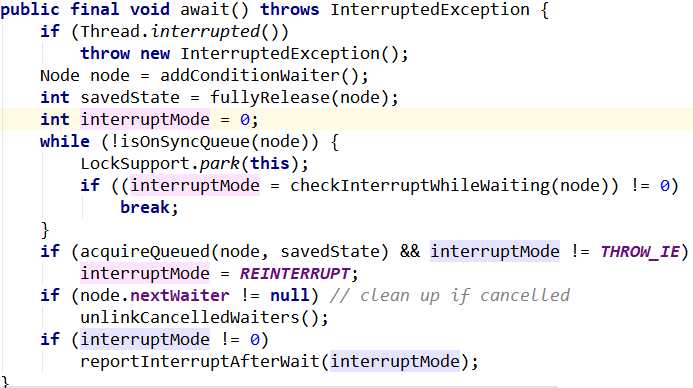
**ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用,而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，ThreadLocalMap 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。使用完 ThreadLocal方法后 最好手动调用remove()方法**

**参考网址：<https://www.cnblogs.com/xiaoxi/p/9140541.html>**

**总结：所谓读写分离的关键是：里面有个位数为32位的状态位，高16表示读锁，低16位表示写锁。写锁的时候回设置独占，读锁只设置状态位。**

### ConditionObject类：（只有两个方法被使用，其他方法都是被这两个方法调用）

***Await()方法：***



在ConditionObject中一般都是调用可中断锁，例如在ArrayBlockingList中使用ConditionObject就是调用的lockInterruptibly()方法。

**在使用await()方法前一定要加锁，否则await()方法中释放锁的时候会报错。同一个Lock实例化出来的所有ConditionObject共用一个同步队列，因为很明显他们共用一把锁来加锁和释放锁。每个ConditionObject有自己的等待队列。**

上述代码步骤如下：

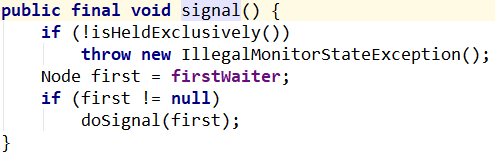
1）Node node = addConditionWaiter();

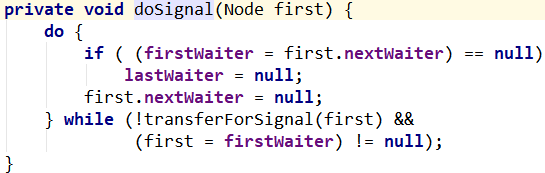
//将该线程包装成节点方法等待队列中,这里面会调用unlinkCancelledWaiters();方法将等待队列中**waitStatus**

不为condition的节点去除。

1. 释放自己拥有的锁，即将独占线程设为null,state设为0.释放失败将节点**waitStatus**值设为cancelled，并在unlinkCancelledWaiters()中被清除。
2. While循环中，当前线程不在同步队列中时，将该线程挂起，线程unpark后，查看中断状态，如果中断状态为true则，设置interruptMode为THROW\_IE或者REINTERRUPT若为THROW\_IE则在后续步骤中直接抛出异常，若为REINTERRUPT则调用Thread.*currentThread*().interrupt();这个方法是不会中断一个正在运行的线程。这一方法实际上完成的是，设置线程的中断标示位，在线程受到阻塞的地方（如调用sleep、wait、join等地方）抛出一个异常InterruptedException，并且中断状态也将被清除，这样线程就得以退出阻塞的状态（这里想要跳出循环一般需要其他线程调用signal方法将包含当前线程的node节点从等待队列中移除）
3. acquireQueued()方法就是死循环获取锁，从等待队列中进入同步队列中的节点不会再进入等待队列，除非再调用await()方法，这里一般会将等待队列中的第一个状态waitStaus为condition的节点放入同步队列中。**小细节**：这里面会将waitstatus从condition设置为0如果设置失败直接讲该节点移除，不会放入同步队列中，继续处理等待队列中的下一个节点。
4. 后面就是对中断的处理和对等待队列中无效节点的处理。

***Signal方法：***





TransferForSignal()方法中会将等待队列中的节点移到同步队列中，并视情况时候unpark线程。

***signalAll():***

将等待队列中所有的节点加入到同步队列中。并将其状态设置为SIGNAL

### LockSupport类：

参考网址：<https://blog.csdn.net/zyzzxycj/article/details/90268381>

参考网址：<https://www.cnblogs.com/minikobe/p/11937352.html>

主要是park,unpark方法，在先调用unpark方法时，调用park方法不会阻塞。

**与Object类的wait/notify机制相比，park/unpark有两个优点：**

1）以thread为操作对象更符合阻塞线程的直观定义

2）操作更精准，可以准确地唤醒某一个线程（notify随机唤醒一个线程，notifyAll唤醒所有等待的线程），增加了灵活性。

***park/unpark的设计原理核心是“许可”***：park是等待一个许可，unpark是为某线程提供一个许可。  
如果某线程A调用park，那么除非另外一个线程调用unpark(A)给A一个许可，否则线程A将阻塞在park操作上。

***unpark操作可以再park操作之前****。*  
**原因如下：在Linux系统下，是用的Posix线程库pthread中的mutex（互斥量），condition（条件变量）来实现的。**  
**mutex和condition保护了一个\_counter的变量，当park时，这个变量被设置为0，当unpark时，这个变量被设置为1。**

C++代码：

void Parker::park(bool isAbsolute, jlong time) {

if (Atomic::xchg(0, &\_counter) > 0) return;

void Parker::unpark() {

int s, status ;

status = pthread\_mutex\_lock(\_mutex);

assert (status == 0, "invariant") ;

s = \_counter;

\_counter = 1;//多次调用只是将\_counter重复赋值而已

这个“许可”是不能叠加的，“许可”是一次性的。  
比如线程B连续调用了三次unpark函数，当线程A调用park函数就使用掉这个“许可”，如果线程A再次调用park，则进入等待状态。

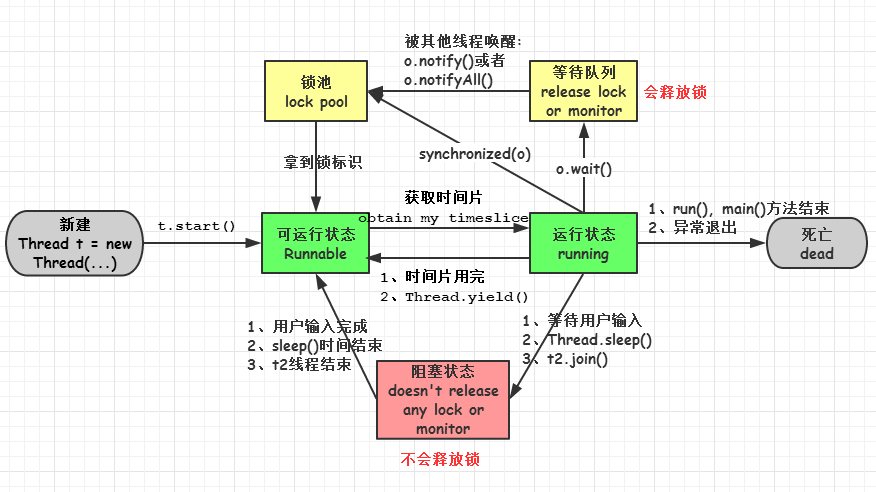
### Unsafe类：

### 参考网址：<https://blog.csdn.net/zyzzxycj/article/details/89877863>

参考网址：<https://www.cnblogs.com/minikobe/p/11937352.html>

***java本身的锁机制及线程的调度：***

既然涉及到锁的使用，就不可避免会对java本身的锁机制和Locks包下的锁机制进行对比。



1.新建(new)：新创建了一个线程对象。

2.可运行(runnable)：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。

3.运行(running)：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

4.阻塞(block)：处于运行状态中的线程由于某种原因，暂时放弃对 CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被 CPU 调用以进入到运行状态。

阻塞的情况分三种：  
(一). 等待阻塞：运行状态中的线程执行 wait()方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；  
(二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；  
(三). 其他阻塞: 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

1. 死亡(dead)：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

**Java 中用到的线程调度算法是什么？**

答：有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

***线程同步和调度方法：***

1. wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

（2）sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；

（3）notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

（4）notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

（5）yeield()方法使线程处于就绪状态。

***sleep()和yield()方法的区别:***

（1） sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

（2） 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

（3）sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

（4）sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

***如何停止一个正在运行的进程：***

1. 正常运行结束
2. 调用stop方法，但是不推荐这个方法，因为stop是过期作废的方法。
3. 使用interrupt方法中断线程（**interrupt**：用于中断线程。调用该方法的线程的状态为将被置为”中断”状态。注意：线程中断仅仅是置线程的中断状态位，不会停止线程。**interrupted**：是静态方法，查看当前中断信号是true还是false并且清除中断信号，**isInterrupted**：查看当前中断信号是true还是false）

**注意**：在某处抛出中断异常，也是结束线程的一种方式。这种方式实际上式控制线程的执行流程来结束线程，它将线程执行完毕了。例如：

public void run() {

try {

while (!Thread.currentThread().isInterrupted()&& more work to do) {

do more work

}

} catch (InterruptedException e) {

//线程在wait或sleep期间被中断了

} finally {

//线程结束前做一些清理工作

}

}

public void run() {

while (!Thread.currentThread().isInterrupted()&& more work to do) {

try {

...

sleep(delay);//在这里抛出异常后，直接走catch和finally代码块。

} catch (InterruptedException e) {

Thread.currentThread().interrupt();//重新设置中断标示

}

}

}

由此可知，通过抛出异常的方式来改变线程的执行流程是一种方式。或者信号量的方式，例如：

class Example2 extends Thread {

volatile boolean stop = false;// 线程中断信号量，这里需注意一点的是需将共享变量定义成volatile 类型或将对它的一切访问封入同步的块/方法（synchronized blocks/methods）中

public static void main(String args[]) throws Exception {

Example2 thread = new Example2();

thread.start();

Thread.sleep(3000);

thread.stop = true;

Thread.sleep(3000);

System.out.println("Stopping application...");

}

public void run() {

// 每隔一秒检测一下中断信号量

while (!stop) {

System.out.println("Thread is running...");

long time = System.currentTimeMillis();

/\*

\* 使用while循环模拟 sleep 方法，这里不要使用sleep，这样while检测stop条

\* 件在sleep或者wait()时就不会执行，无法跳出阻塞。

\* stop信号量也就失去了意义。

\*/

while ((System.currentTimeMillis() - time < 1000)) {}

}

System.out.println("Thread exiting under request...");

}

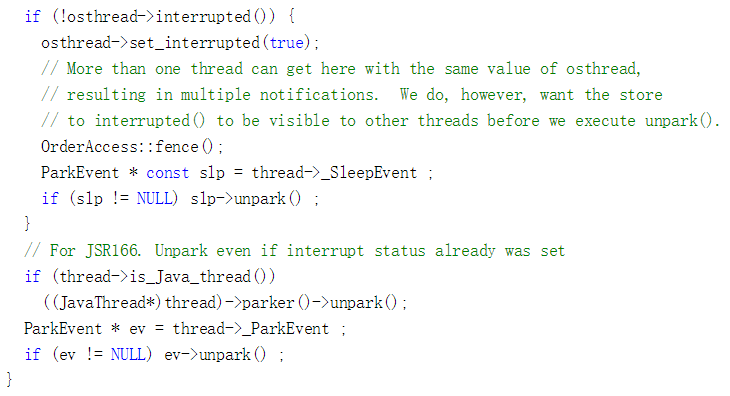
}

但是在线程处于阻塞状态则无法往下进行，另外，在调用interrupt方法时， 如果线程调用thread.sleep、thread.join、thread.wait、condition.await、以及可中断的通道上的 I/O 操作方法后可进入阻塞状态，会抛出异常。

***可中断通道：***

实现此InterruptibleChannel接口的通道是可中断的：如果某个线程在可中断通道上因调用某个阻塞的 I/O 操作（常见的操作一般有这些：serverSocketChannel. accept()、socketChannel.connect、socketChannel.open、socketChannel.read、socketChannel.write、fileChannel.read、fileChannel.write）而进入阻塞状态，而另一个线程又调用了该阻塞线程的 interrupt 方法，这将导致该通道被关闭，并且已阻塞线程接将会收到ClosedByInterruptException，并且设置已阻塞线程的中断状态。另外，如果已设置某个线程的中断状态并且它在通道上调用某个阻塞的 I/O 操作，则该通道将关闭并且该线程立即接收到 ClosedByInterruptException；并仍然设置其中断状态。

**Object.wait, Thread.sleep方法，**\_ParkEvent变量用于synchronized同步块和Object.wait()，\_SleepEvent变量用于Thread.sleep()。调用sleep或者wait方法会唤醒线程。而parker()->unpark()则更一般，只要调用了java中unsafe类的park()方法，最终都会调用unpark()唤醒线程。

****

### 阻塞队列（有时间了解一下）：

**底层使用的是ReentrantLock和condition**

**阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析**

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列

1.ArrayBlockingQueue按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列

2.LinkedBlockingQueue 是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为 Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序

（ArrayBlockingQueue 和LinkedBlockingQueue区别：1：底层实现不同

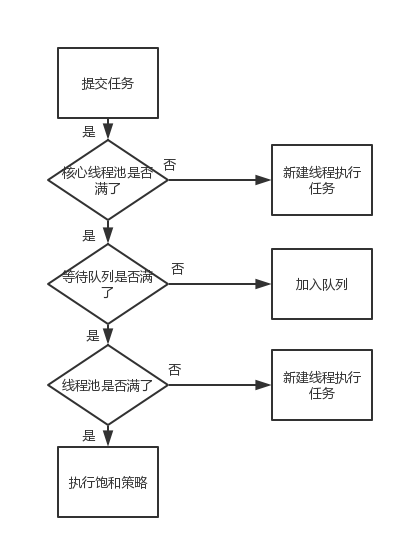
ArrayBlockingQueue 底层使用数组来维护队列，LinkedBlockingQueue 底层使用链表来维护队列，在添加和删除队列中的元素的时候，会创建和销毁节点对象，在高并发和大量数据的时候，GC压力很大。2.锁的方式不同，LinkedBlockingQueue  **获取数据和添加数据使用不同的锁对象**。但是这样会增加额外的复杂性）

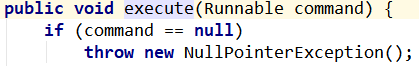
1. PriorityBlockingQueue是一个支持优先级的无界阻塞队列。默认情况下元素采取自然顺序升序排列
2. DelayQueue是一个支持**延时获取元素**的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素（比较有用，有时间看一下源码）
3. SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。**吞入量较高**
4. LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队列的两端插入和移出元素

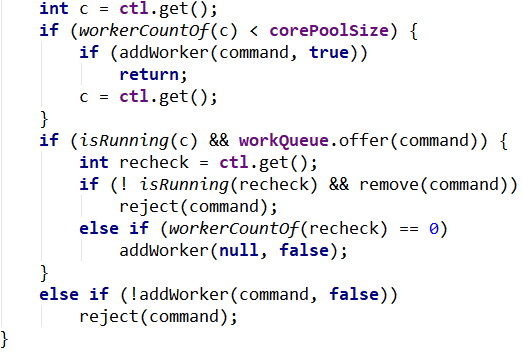
参考网址：https://www.jianshu.com/p/32665a52eba1

### ThreadPoolExecutor类：

任务提交流程大体流程：





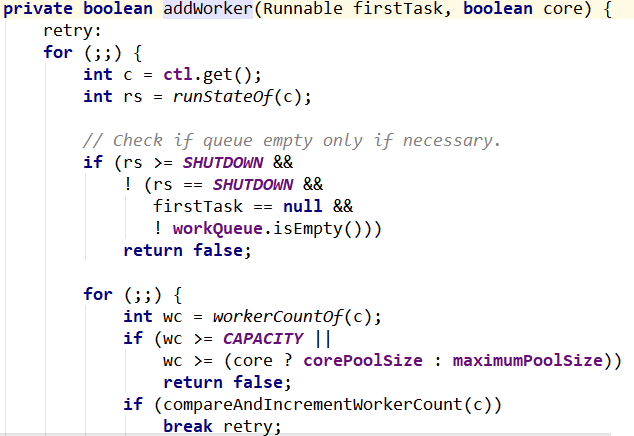


**3**

**2**

**1**

1. WorkerCountof(c)<corePoolSize表示当前线程数小于核心线程数，这是添加新的线程到workers中，并执行任务，后面具体分析addWorker()方法。
2. 当前线程数少于核心线程数，isRunning()判断是否处于running状态，workQueue判断是否能将任务放入队列，在运行状态且**放入队列**的情况下，再次进行状态核查，看时候处于运行状态，如果不是将任务移除，并执行拒绝策略。
3. 在放入队列失败，即队列满了的情况下执行addWorker()方法，增加线程数量但是不能超过最大线程数。如果已经达到最大执行拒绝策略。



### CountDownLatch类：

### 1597049038(1)

### （调用内部类:Sync来具体实现功能）

### 此类主要有两个方法countDown()和await()方法。CountDownLatch创建时会给state赋一个初始值，**CountDown()方法的每一次调用都会将state减1**，如下：

### 1597049284(1)

### 1597049326(1)

2

### 当state为0时返回true，调用2中的代码使得因为调用await()方法处于挂起状态的线程恢复。Await方法如下：

### 1597049928(1)

### 线程恢复之后继续执行该循环，并跳出循环从而执行线程中的代码。

### Semaphore类：

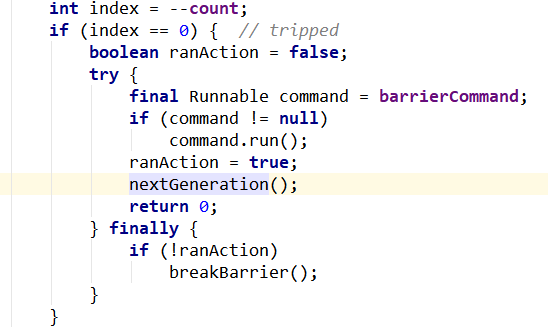
(与countdownLatch调用内部类:Sync)

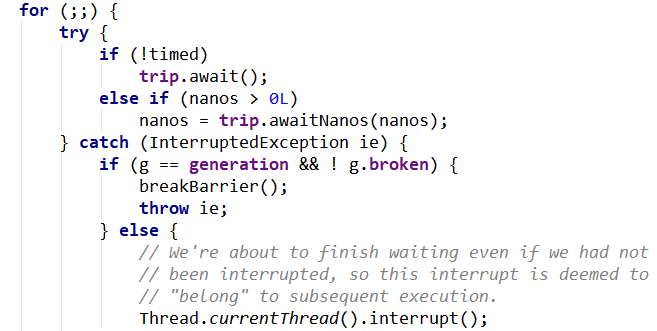
主要有acquire()和release()两个方法。Semaphore类在实例化时传入一个数count表示state能达到的最大值，acquire()方法默认每次将state加1，当state等于count时加入同步队列（acquire(n)消费n个）。Release()方法每次将state减1，并激活同步队列中的线程来控制并发线程数量。所以Semaphore类是一个控制线程并发数的类。具体算法直接看源码。

### CyclicBarrier类：

（ReentrantLock和ConditionObject的结合）

跟CountdownLatch一样的套路，首先是构造函数，传入一个int（以count表示），和一个Runnable。主要是await()方法。每次调用await()都将count值减1，当count值为0时，调用传入的Runnable线程并重置count值而且将阻塞的线程全部唤醒继续执行任务（还是得争抢锁）：（ReentrantLock和ConditionObject）





* CountDownLatch：一个或者多个线程，等待其他多个线程完成某件事情之后才能执行；
* CyclicBarrier：多个线程互相等待，直到到达同一个同步点，再继续**一起执行**

#### **sleep() 和 wait() 有什么区别？**

两者都可以暂停线程的执行

* 类的不同：sleep() 是 Thread线程类的静态方法，wait() 是 Object类的方法。
* 是否释放锁：sleep() 不释放锁；wait() 释放锁。
* 用途不同：Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
* 用法不同：wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会自动苏醒。

### 多线程中 synchronized 锁升级的原理是什么？

### synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级

ThreadPoolExecutor构造函数重要参数分析

ThreadPoolExecutor 3 个最重要的参数：

corePoolSize ：核心线程数，线程数定义了最小可以同时运行的线程数量。

maximumPoolSize ：线程池中允许存在的工作线程的最大数量

workQueue：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

ThreadPoolExecutor其他常见参数:

keepAliveTime：线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁；

unit ：keepAliveTime 参数的时间单位。

threadFactory：为线程池提供创建新线程的线程工厂

handler ：线程池任务队列超过 maxinumPoolSize 之后的拒绝策

### epoll是[Linux内核](https://baike.baidu.com/item/Linux%E5%86%85%E6%A0%B8" \t "https://baike.baidu.com/item/epoll/_blank)为处理大批量[文件描述符](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6/9809582" \t "https://baike.baidu.com/item/epoll/_blank)而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在**大量[并发连接](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91%E8%BF%9E%E6%8E%A5/3763280" \t "https://baike.baidu.com/item/epoll/_blank)中只有少量活跃的情况下的系统[CPU](https://baike.baidu.com/item/CPU/120556" \t "https://baike.baidu.com/item/epoll/_blank)利用率**。另一点原因就是**获取事件的时候，它无须遍历整个被侦听的描述符集**，只要遍历那些被**内核IO事件异步唤醒而加入Ready队列的描述符集合就行了**。epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率