## Synchronized

一个对象有两个锁：对象锁 + 类锁

对象在堆内存中分为三块区域：对象头，实例数据，填充数据



实例数据：存放类的属性数据信息，父类属性信息

填充数据：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐

对象头：它实现synchronized的锁对象的基础，synchronized使用的锁对象是存储在Java对象头里的，jvm中采用2个字来存储对象头(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由Mark Word 和 Class Metadata Address 组成



以下是32位JVM的Mark Word

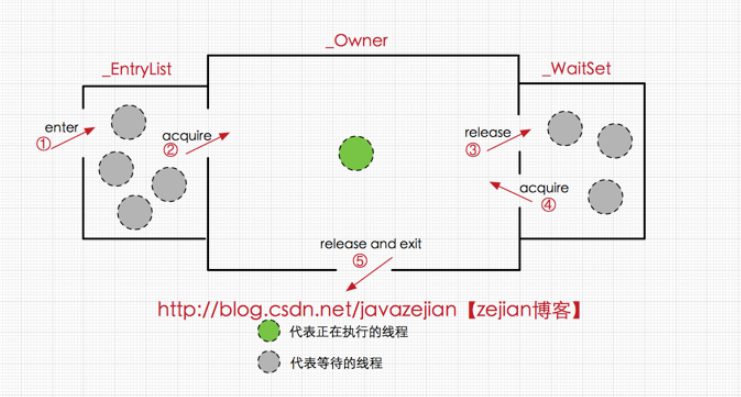


Mark Word是一个非固定的数据结构，它可以根据对象本身的状态(对象充当锁的状态)复用自己的存储空间

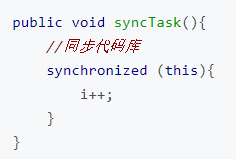


重量级锁：

每个对象都存在一个monitor 与之关联，在重量级锁中，指针指向与之关联的monitor，monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由ObjectMonitor实现的



在ObjectMonitor中有两个队列，\_WaitSet(等待池) 和 \_EntryList(锁池)，用来保存ObjectWaiter对象列表( 每个等待锁的线程都会被封装成ObjectWaiter对象)，\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 \_EntryList 集合，当线程获取到对象的monitor 后进入 \_Owner 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。这也是为什么任何对象都已作为锁的原因，也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因，Synchronized锁是非公平的



编译后的代码如右图所示，

monitorenter指令指向同步代码块的开始位置，monitorexit指令则指明同步代码块的结束位置，当执行monitorenter指令时，当前线程将试图获取 objectref(即对象锁) 所对应的 monitor 的持有权，当 objectref 的 monitor 的进入计数器为 0，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功。如果当前线程已经拥有 objectref 的 monitor 的持有权，那它可以重入这个 monitor (锁的重入性)，重入时计数器的值也会加 1。倘若其他线程已经拥有 objectref 的 monitor 的所有权，那当前线程将被阻塞，直到正在执行线程执行完毕，即monitorexit指令被执行，执行线程将释放 monitor(锁)并设置计数器值为0 ，其他线程将有机会持有 monitor 。第二个monitorexit是当程序抛出异常执行的

方法级(Synchronized修饰的是方法)的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED访问标志是否被设置，如果设置了，执行线程将先持有monitor，然后再执行方法，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor(如果方法被Synchronized修饰，则此方法会有一个ACC\_SYNCHRONIZED，当在执行此方法时，线程会尝试获取monitor，后面的操作就和同步代码的操作类似

早期(JDK1.6之前)Synchronized属于重量级锁，效率十分低下；因为监视器锁（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，而操作系统实现线程之间的切换时需要从用户态转换到核心态。互斥(Mutex)会导致线程挂起，而会让程序的运行级别从用户态切换到内核态，把线程挂起(挂起线程和恢复线程的操作都需要转入内核态中完成)

实际上一个线程对锁的占用时间非常短，很有可能当线程A快要释放锁的时候，线程B请求资源，发现被占用而被挂起，只要线程B在等待一下就可以获取资源，没必要挂起，所以可以让线程B执行一个忙循环(自旋)，这就是自旋锁

自旋等待不能代替阻塞，自旋等待虽然避免了线程切换的开销，但它是要占用处理器时间的， 所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了。自旋次数的默认值是10次，用户可以使用参数-XX:PreBlockSpin来更改，所以在写代码的过程中，尽量缩小同步的范围

在JDK 1.6中引入了自适应的自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。另一方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。

锁削除：在编译时，如果检测到共享数据不会存在锁竞争，而又对其加锁，编译时，会将此锁削除，如果多个线程调用StringBuilder的append的方法，但是本身StringBuilder就行线程安全的，所以如果对append额外加锁，锁会被移除

锁的状态总共有四种(（由低到高的次序)：无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁，但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级

无状态锁：就是没有锁操作

偏向锁：

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。

一个对象刚开始实例化的时候，没有任何线程来访问它的时候。它是可偏向的，它认为只可能有一个线程来访问它，所以当第一个线程来访问它的时候，它会偏向这个线程，此时，对象持有偏向锁。偏向第一个线程，这个线程在修改对象头MarkWord成为偏向锁的时候使用CAS操作(防止多个线程同时获取到锁)，并将对象头中的ThreadID改成自己的ID(在线程的的栈帧里创建lockRecord，在lockRecord里和锁对象的MarkWord里存储线程的id)，之后再次访问这个对象时，只需要对比ID，不需要再使用CAS在进行操作。一旦有第二个线程访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象时偏向状态，这时表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活，如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程(CAS)，如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁（偏向锁就是这个时候升级为轻量级锁的）。如果不存在使用了，则可以将对象回复成无锁状态，然后重新偏向。

关闭偏向锁，通过jvm的参数-XX:UseBiasedLocking=false,则默认会进入轻量级锁。

轻量级锁：

线程在执行同步块之前，JVM会先在当前线程的栈桢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，然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。（线程获得锁，会在线程的栈帧里创建lock record(锁记录变量)，让lock record的指针指向锁对象的对象头中的mark word.再让mark word指向lock record.这就是获取了锁。）

b线程在锁竞争时，发现锁已经被a线程占用，则b线程不进入内核态，让b线程自旋，执行空循环，等待a线程释放锁。如果，完成自旋策略还是发现a线程没有释放锁，或者让c线程占用了。则b线程试图将轻量级锁升级为重量级锁。



锁的可重入性：



由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。

线程中断：

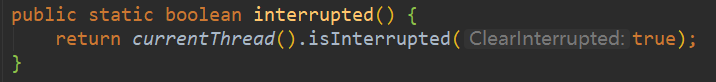
可以调用方法来结束正在执行的线程，如Thread.suspend，Thread.stop方法(已被弃用)，但是这种强制执行的方式会带来很多意外的操作，所以应该将这个操作交给程序，程序设置当前线程的状态为中断状态，但是是否中断这个线程由程序决定

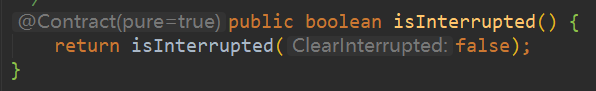
中断的方法：

public boolean isInterrupted()

public void interrupt()

public static boolean interrupted()





clearInterrupt表示是否清楚中断标记

线程的状态：NEW，RUNNABLE，BLOCKED，WAITING，TERMINATED

对于处于New和Terminated状态的线程，执行中断是毫无意义的，因为线程还未启动

对于处于Runnable状态的线程，当调用interrupt方法后，在会标记当前线程的处于中断状态，而并不会中断此线程，但是程序可以通过isInterrupted方法检测线程是否中断，中断后需要做哪些处理，这样就会非常灵活



线程处于BLOCKED状态说明该线程由于竞争某个对象的锁失败而被挂在了该对象的阻塞队列上了。那么此时发起中断操作不会对该线程产生任何影响，依然只是设置中断标志位。和Runnable类似，由程序自己判断

对于Waiting状态的线程，线程在运行的过程中由于缺少某些条件而被挂起在某个对象的等待队列上。当这些线程遇到中断操作的时候，会抛出一个InterruptedException异常，并清空中断标志位(为false)，提前退出Waiting状态。程序可以在try catch语句块中做处理

正在等待锁的线程，执行interrupt方法后，只会改变线程的状态，而不会抛出异常，此时对应的是线程的block状态

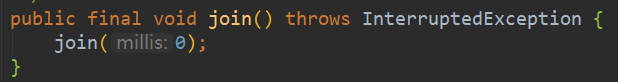
阻塞 = waiting + block



为什么处于waiting状态的线程被interrupt后，会抛出InterruptedException异常







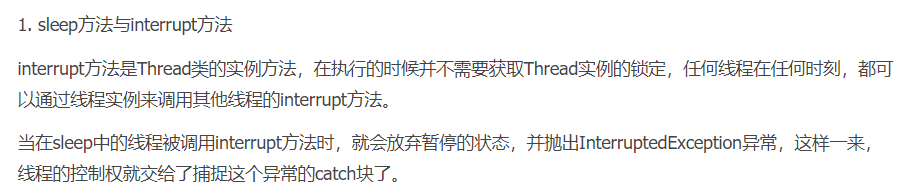
Join实际最终调用Object的Wait方法

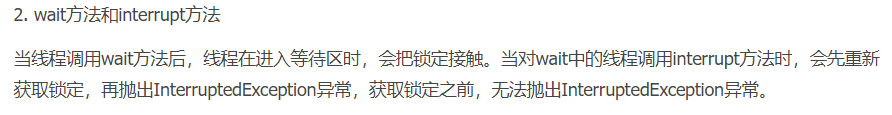
interrupt方法其实只是改变了中断状态而已。而sleep、wait和join这些方法的内部会不断的检查中断状态的值，从而自己抛出InterruptEdException。

这个异常是wait，sleep，join方法抛出的，而不是线程

对于wait中等待notify/notifyAll唤醒的线程，其实这个线程已经"暂停"执行，因为它正在某一对象的休息室中，这时如果它的中断状态被改变，那么它就会抛出异常。这个InterruptedException异常不是线程抛出的，而是wait方法，也就是对象的wait方法内部

会不断检查在此对象上休息的线程的状态，如果发现哪个线程的状态被置为已中断，则会抛出InterruptedException，意思就是这个线程不能再等待了，其意义就等同于唤醒它了。







Wait会释放锁，sleep不会释放

notify/notifyAl/wait方法，在使用时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常, 这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，

与sleep方法不同的是wait方法调用完成后，线程将被暂停，但wait方法将会释放当前持有的监视器锁(monitor)，直到有线程调用notify/notifyAll方法后方能继续执行，而sleep方法只让线程休眠并不释放锁。同时notify/notifyAll方法调用后，并不会马上释放监视器锁，而是在相应的synchronized(){}/synchronized方法执行结束后才自动释放锁。

线程阻塞和等待的区别：

等待是线程主动的，而阻塞是被动的

等待时线程以获取到锁，而阻塞是抢占锁

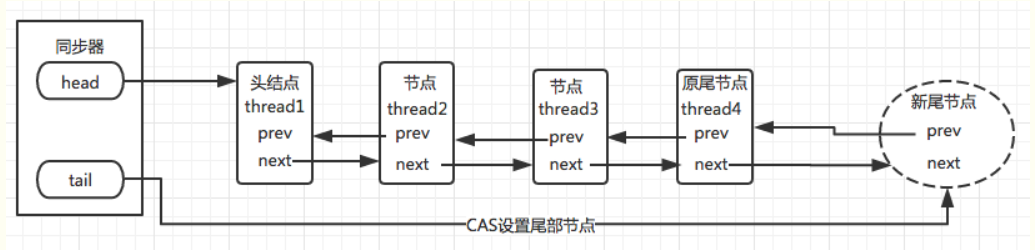
## Lock

AQS(AbstractQueuedSynchronizer)

AQS的核心思想是基于volatile int state这样的volatile变量，配合Unsafe工具对其原子性的操作来实现对当前锁状态进行修改。同步器内部依赖一个FIFO的双向队列来完成资源获取线程的排队工作。通过判断state的状态，锁是否被占用(state > 0)

AQS同步队列：内部的实现是依赖同步队列（一个FIFO的双向队列，其实就是数据结构双向链表）来完成同步状态的管理。当前线程获取同步状态失败时，同步器AQS会将当前线程和等待状态等信息构造成为一个节点（node）加入到同步队列，同时会阻塞当前线程； 当同步状态释放的时候，会把首节点中的线程唤醒，使首节点的线程再次尝试获取同步状态。AQS是独占锁和共享锁的实现的父类。

独占锁和共享锁：锁在一个时间点只能被一个线程占有。根据锁的获取机制，又分为“公平锁”和“非公平锁”。ReentrantLock和ReentrantReadWriteLock.Writelock是独占锁。共享锁：同一个时候能够被多个线程获取的锁，能被共享的锁。JUC包中ReentrantReadWriteLock.ReadLock

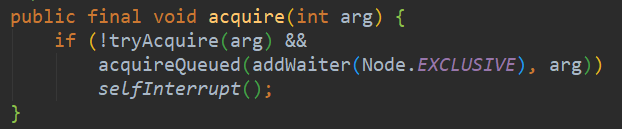


线程在获取(lock)同步状态(state=0)失败时，会将当前线程封装为一个NODE节点，并通过CAS机制存放到AQS队列尾节点，

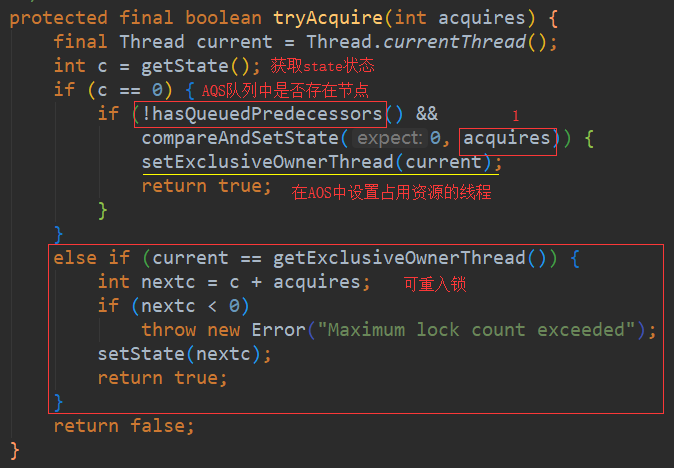
当线程释放(unlock)同步状态后，头节点在释放同步状态的时候，会唤醒后继节点，而后继节点将会在获取锁（同步状态）成功时候将自己设置为头节点。

ASQ是一个抽象类，在AQS中实现了很多方法，但是有些放入任然需要子类自己实现

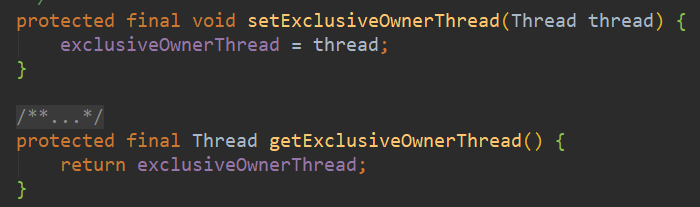
独占锁获取state的过程：



ReentrantLock的tryAcquire方法

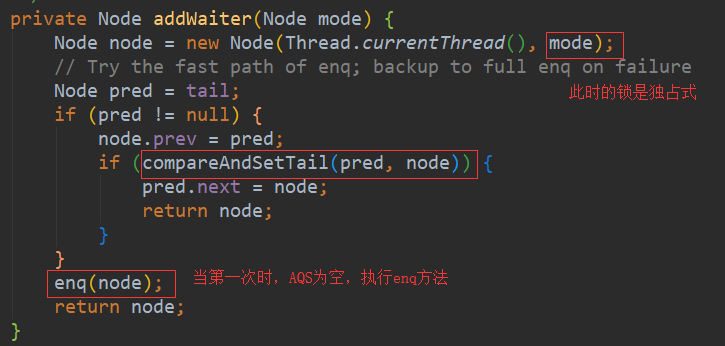


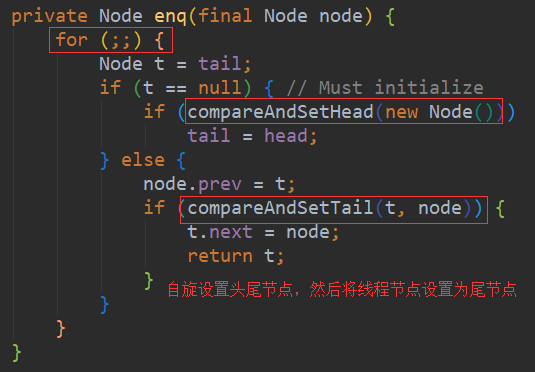
hasQueuedPredecessors判读队列中使用有比自己等待时间还长的线程



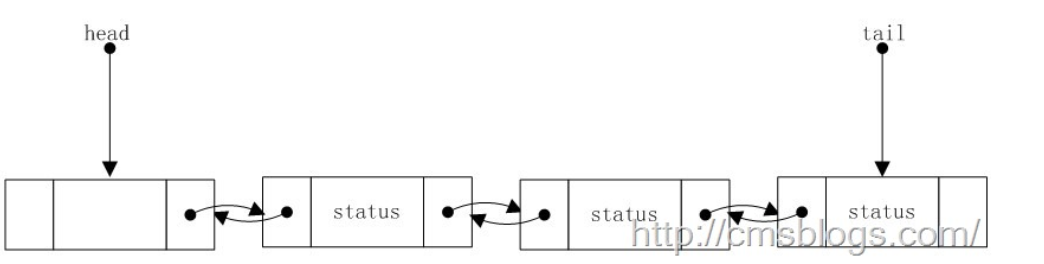
exclusiveOwnerThread：获取到state的线程

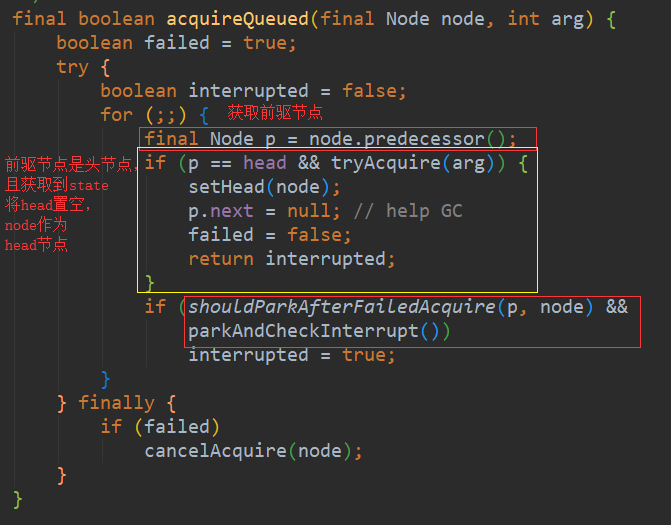
获取同步状态失败后，将线程添加到AQS队列





AQS的FIFO结构是，头结只是一个Node，尾节点为最后一个进AQS的线程Node





将Node添加到CLH队列后，自旋检测前驱是否为head节点，并尝试获取state

shouldParkAfterFailedAcquire：如果获取state失败，是否需要挂起

parkAndCheckInterrupt：挂起线程(Lock中通过park挂起线程)

只有shouldParkAfterFailedAcquire为true时，才会执行parkAndCheckInterrupt，短路校验

在Node中有一个volatile int waitStatus的来标记线程的状态，有如下4种值

CANCELLED：值为1，在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消该Node的结点，即结束状态，进入该状态后的结点将不会再变化。

SIGNAL：值为-1，被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。就会通知标识为SIGNAL状态的后继结点的线程执行。

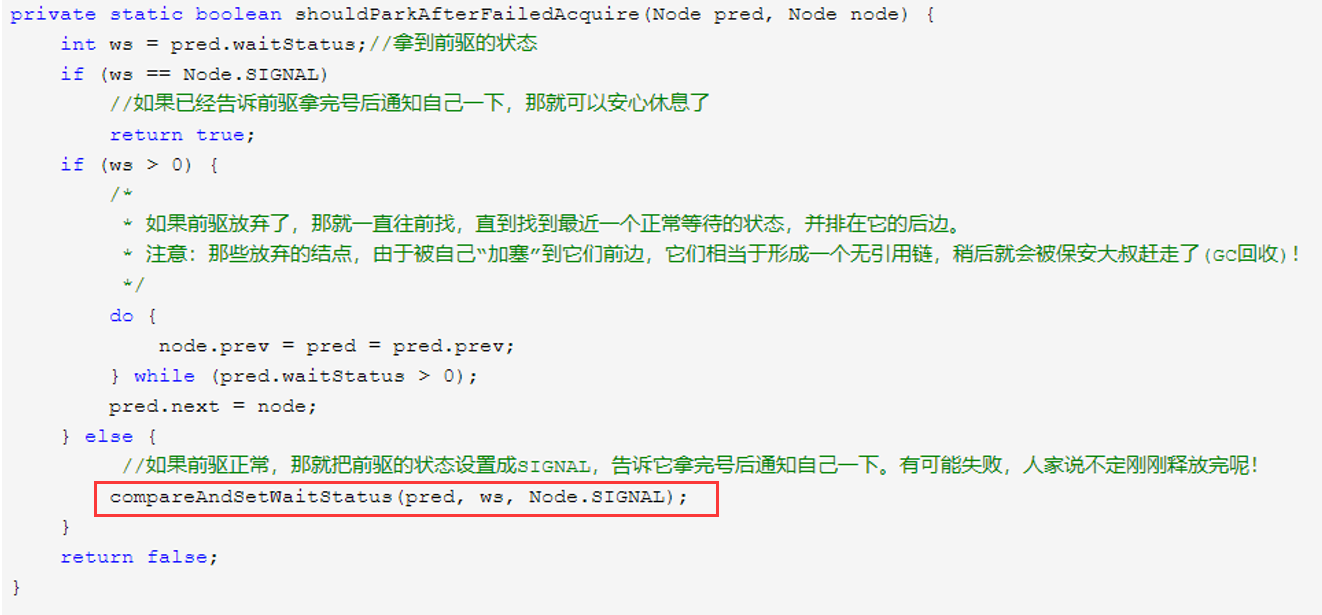
CONDITION：值为-2，与Condition相关，该标识的结点处于等待队列中，结点的线程等待在Condition上，当其他线程调用了Condition的signal()方法后，CONDITION状态的结点将从等待队列转移到同步队列中，等待获取同步锁。

PROPAGATE：值为-3，与共享模式相关，在共享模式中，该状态标识结点的线程处于可运行状态。

0状态：值为0，代表初始化状态。

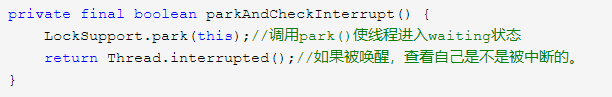
AQS在判断状态时，通过用waitStatus>0表示取消状态，而waitStatus<0表示有效状态。

shouldParkAfterFailedAcquire(Node, Node)



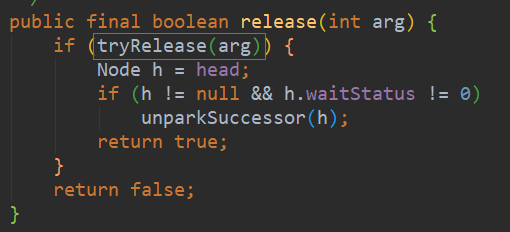
如果前驱不是SIGNAL，compareAndSetWaitStatus会将前驱节点的状态设置为SIGNAL，下次自旋时，shouldParkAfterFailedAcquire的返回结果为true，就会将当前节点的线程挂起，所以Lock的自旋次数为1，线程挂起(挂起后，会记录此时的操作步骤)

parkAndCheckInterrupt()



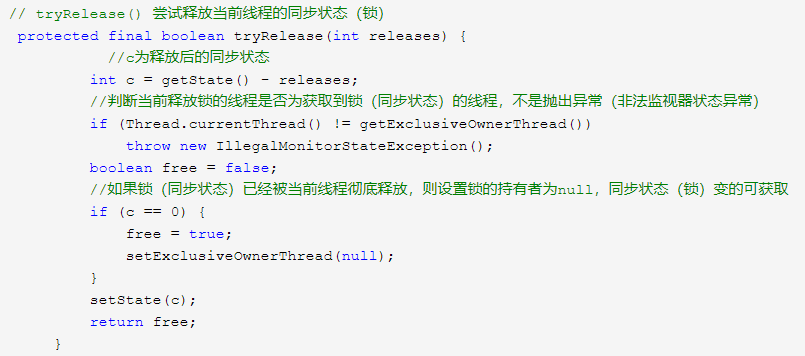
LockSupport.park是LockSupport阻塞原语的方法

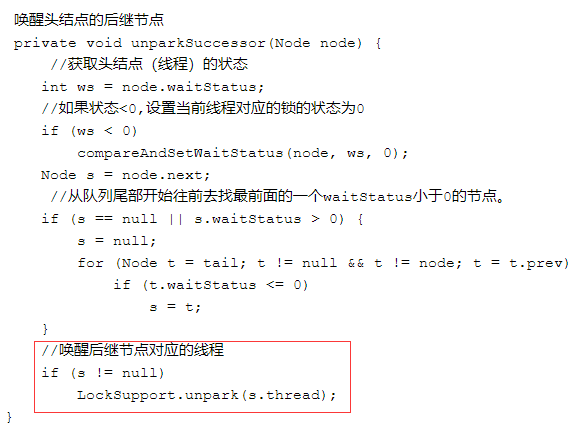
释放state的操作：



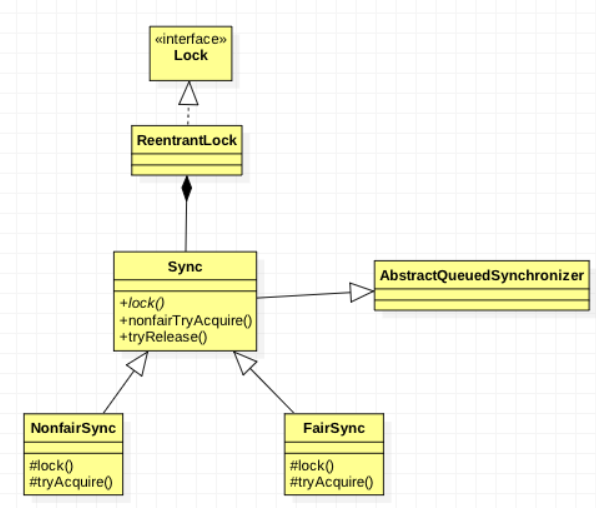
tryRelease方法释放state，unparkSuccessor修改头节点，并唤醒头结点的后继节点

这里的h.waitStatus为-1，在shouldParkAfterFailedAcquire自旋中修改的

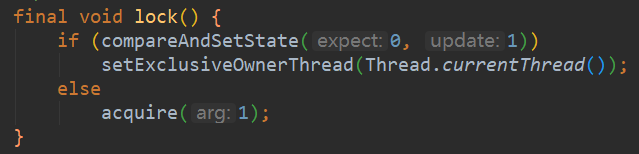




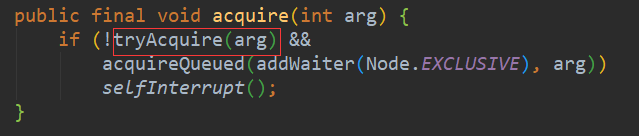
这就是ReentrantLock的公平锁获取和释放的AQS的源码



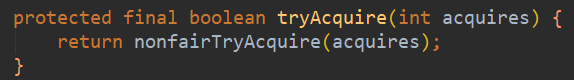
非公平锁的获取方式：

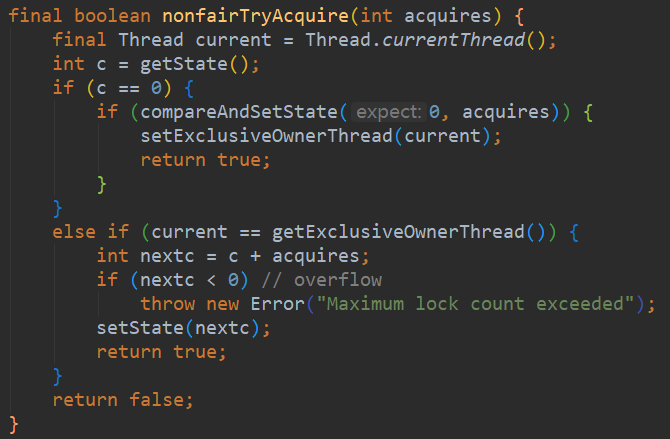


非公平锁在lock的时候先直接用cas判断state变量是否为0（尝试获取锁），成功的话更新成1，表示当前线程获取到了锁，不需要在排队，从而直接抢占的目的。而对于公平锁的lock方法是一开始就走AQS的双向队列排队获取锁。



tryAcquire方法有子类实现

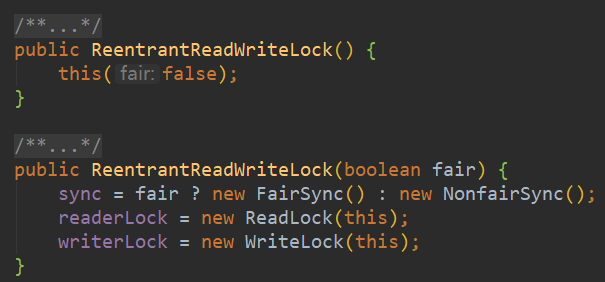




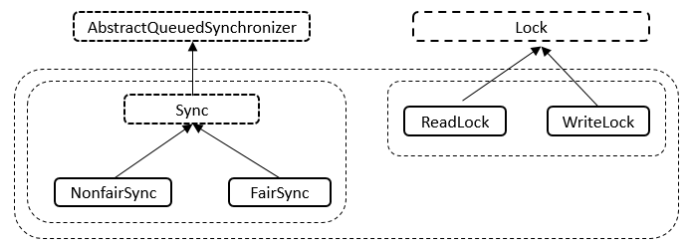
公平锁在获取的时候多了hasQueuePredecessors()方法，加入了当前节点是否有前驱节点的判断。如果返回true，表示有线程比当前线程等待的时间长，需要等待前驱线程获取并释放锁之后才能获取锁。返回false，表示当前的线程所在的节点为队列（CLH队列）的对头，可以直接获取锁。

公平锁和非公平锁的区别在获取锁的时候，公平锁有一个判断CLH队列中是否有节点的方法(hasQueuePredecessors)，当CLH中有节点时，新线程存放到CLH队尾，实现公平，而非公平锁在获取的时候采用抢占式获取，但是如果抢占失败，任然会被放到CLH队尾，这就和公平锁类似

共享锁：

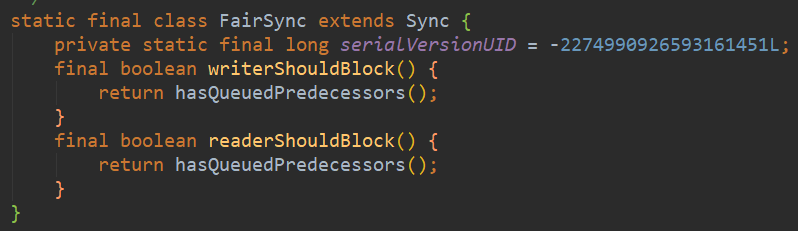


默认非公平

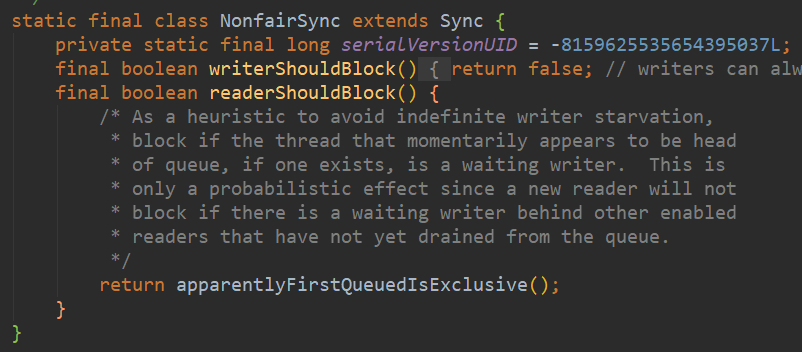


这里的Sync，FairSync，NonfairSync都是ReentrantReadWriteLock的内部类

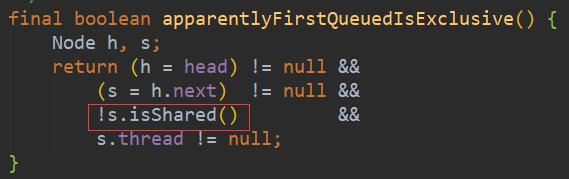
ReentrantReadWriteLock实现公平锁和非公平锁



公平锁在获取读写锁的时候都会去判断CLH队列(这里为什么在获取写锁的时候也要判断，因为可能写线程已经在等待)



非公平锁在获取写锁的时候直接采用抢占式，写锁会调用apparentFirstQueuedIsExcluisve

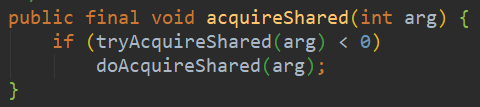
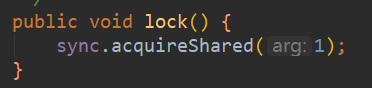


当前线程是写锁占用的线程时，返回true；否则返回false。也就说明，如果当前有一个写线程正在写，那么该读线程应该阻塞。



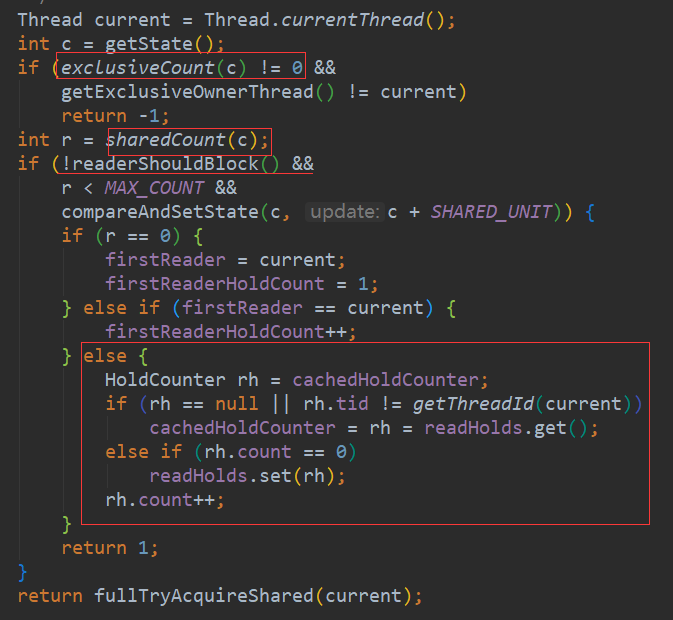
ReentrantReadWriteLock通过将state一分为二来判断锁的类型和占用锁的线程数

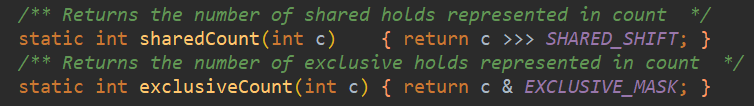
读锁：



tryAcquireShared获取读锁是否失败(当前资源被写锁占用)，doAcquireShared方法将该线程加入到等待队列

tryAcquireShared实现





exclusiveCount方法时根据传入的state计算独占锁(写锁)的数量，且当线程不是写线程，state的值不仅说明了当前锁的状态以及等待锁的数量

sharedCount获取读锁的个数(占用读锁的线程数，也叫重入次数)

readerShouldBlock读线程是否应该阻塞(公平/非公平锁)

获取读锁成功后记录重入次数

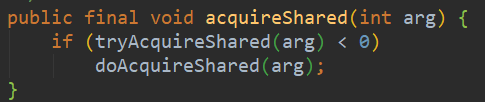
失败执行fullTryAcquireShared方法，就是循环处理CAS失败的情况



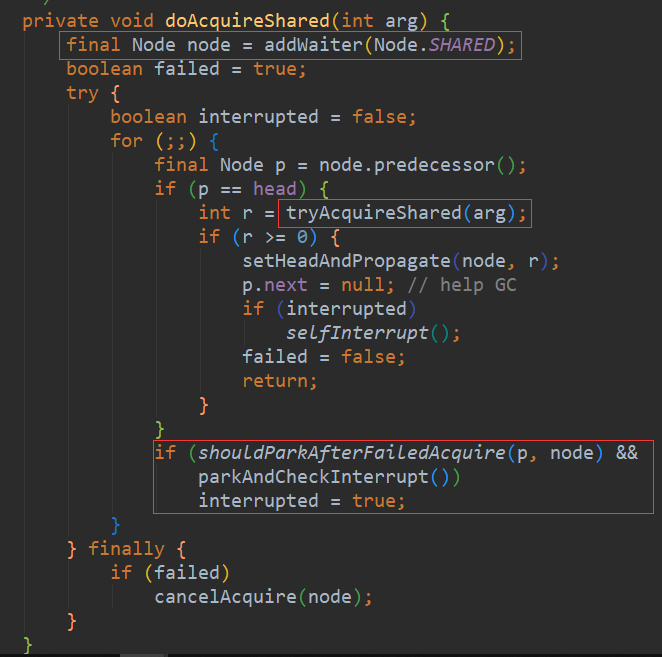
fullTryAcquireShared的主要方法，下面的是执行CAS成功后，返回结果为1

在tryAcquireShared中执行CAS成功后，会记录重入次数，失败后，会执行

fullTryAcquireShared方法，fullTryAcquireShared中循环执行CAS，如果成功返回结果为1在执行的过程中可能当前资源被写锁占用，则跳出循环，retrun -1

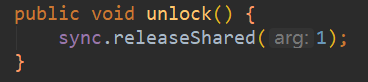


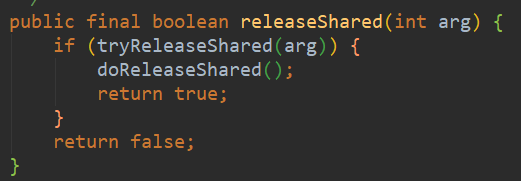
执行完tryAcquireShared，大于0表示获取锁成功，直接执行同步代码，如果失败(即在轮训获取资源时，资源被写锁占用，跳出轮训)，执行doAcquireShared



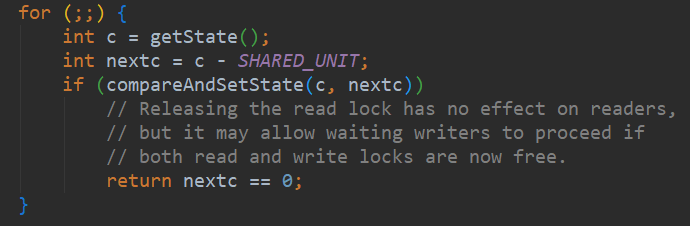
获取失败后，添加到CLH队列，轮训前驱一次，获取失败，挂起读线程

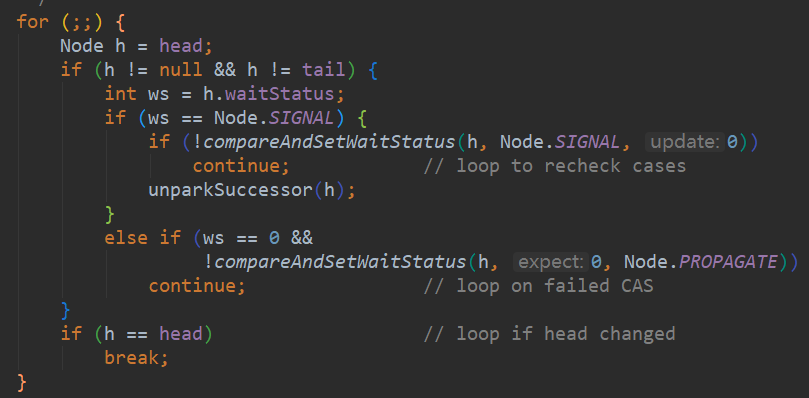
释放锁操作





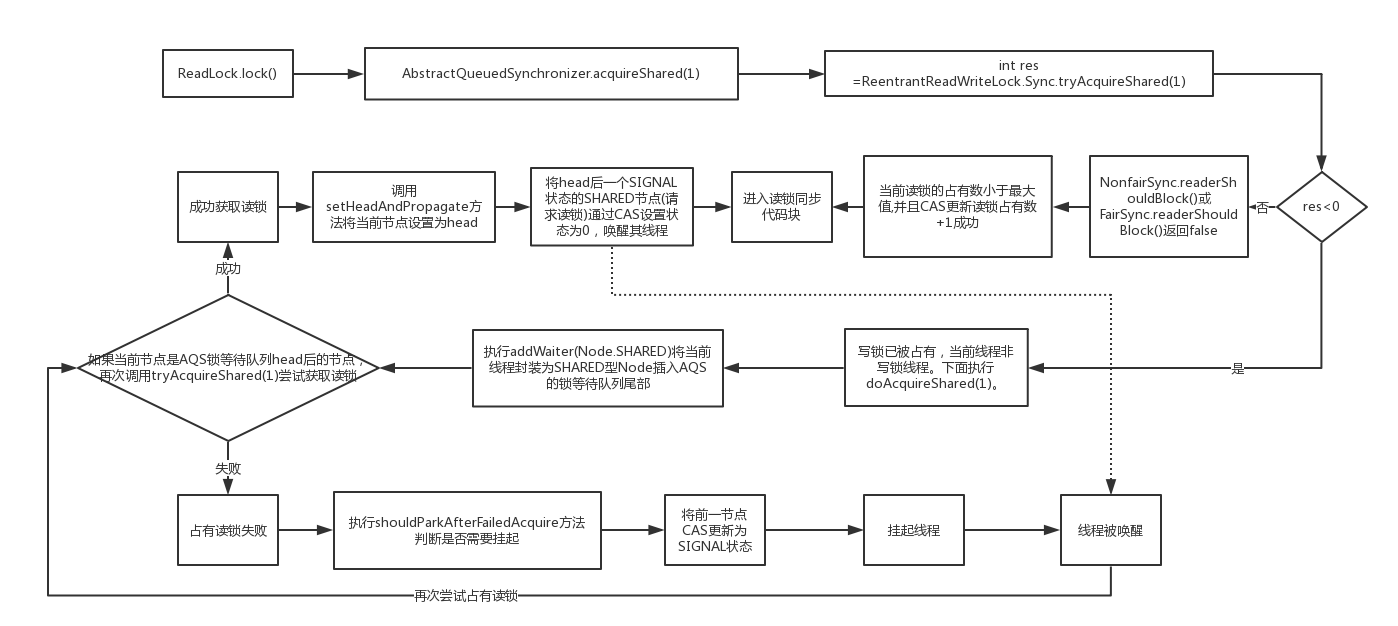
tryReleaseShared释放锁的操作

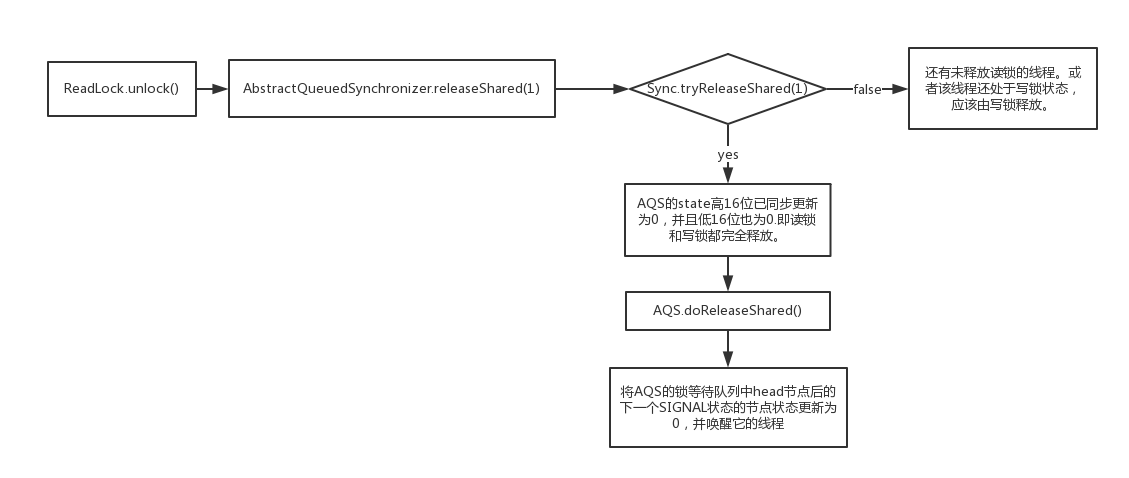




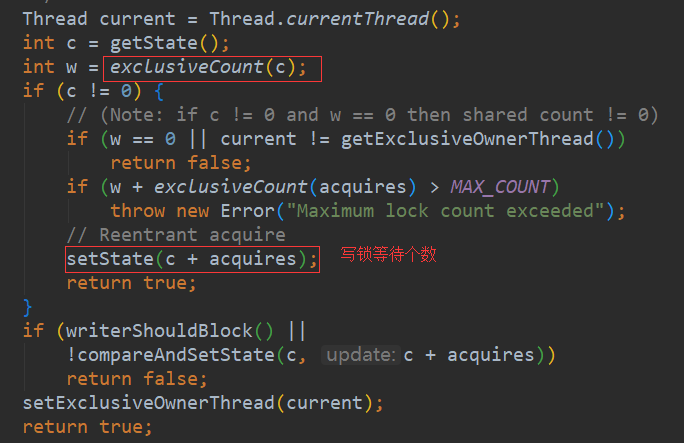
唤醒头节点的后继节点

读锁获取/释放过程：





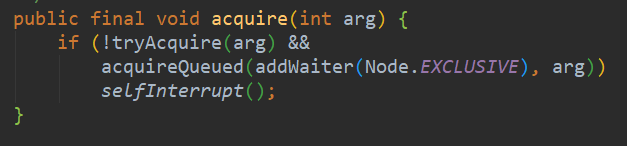
写锁：



写锁和ReentrantLock实现方式差不多，只是在tryAcquire有点不同

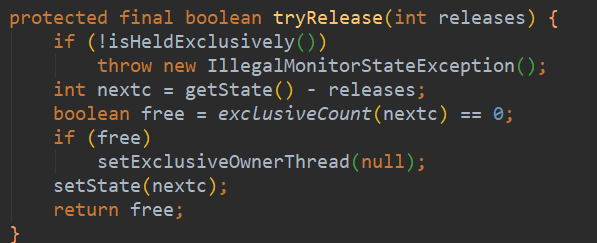
c != 0表示有锁占用，w==0表示锁为写锁

c == 0 执行CAS获取写锁



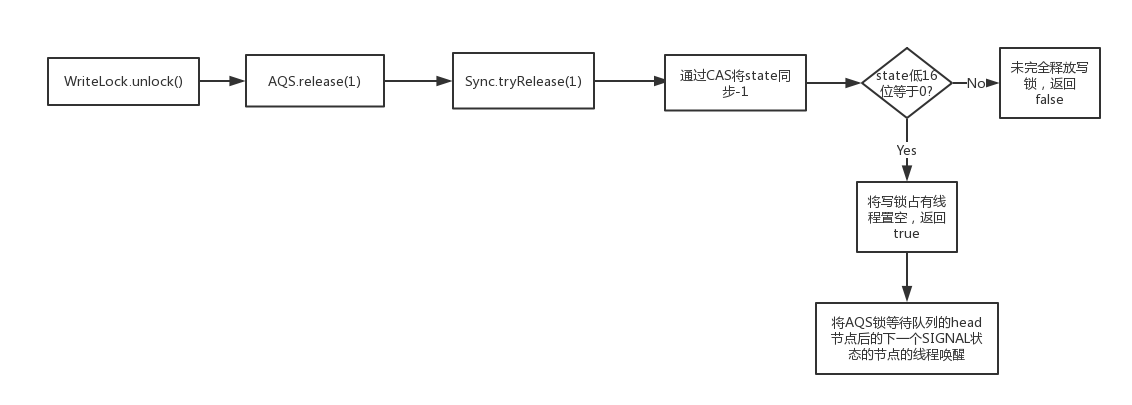
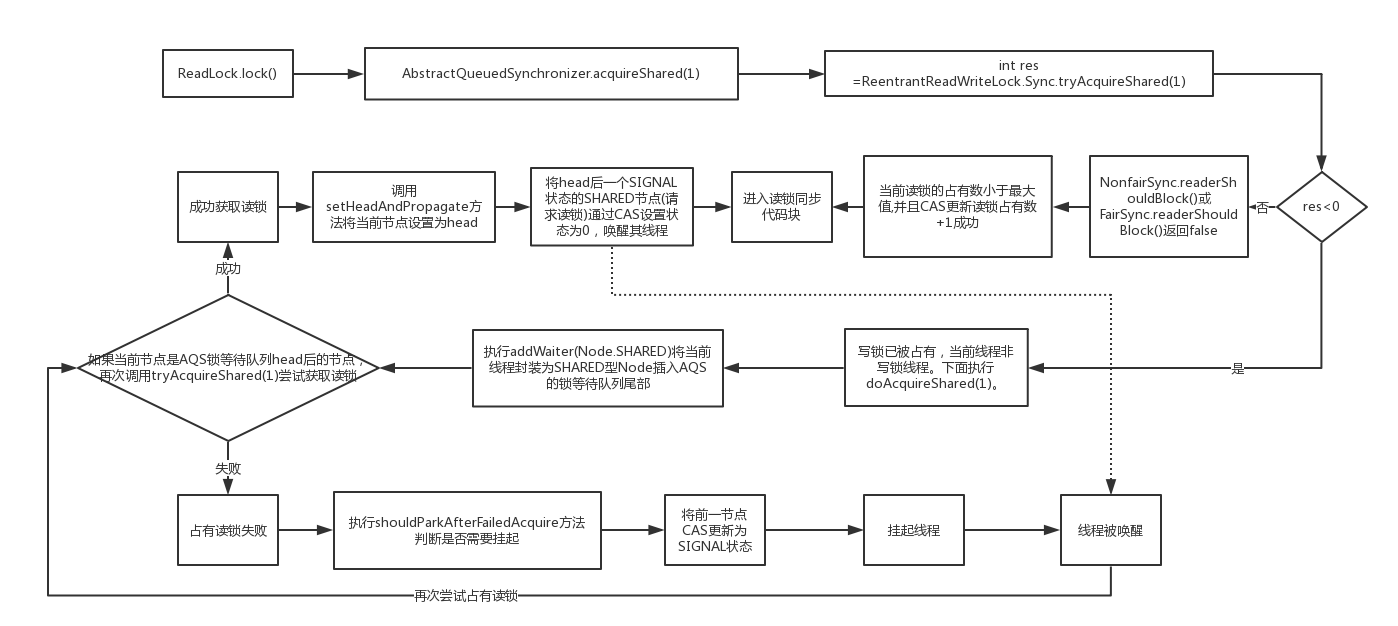
失败存放到CLH队列

释放锁



和ReentrantLock实现方式差不多，只是tryRelease中修改state的方式不同

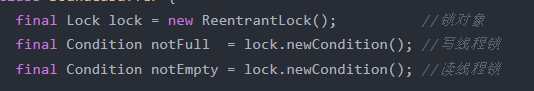
写锁获取/释放过程：



Lock是通过代码的方式实现自旋等操作，通过修改state的状态，来判断资源是否被占用，

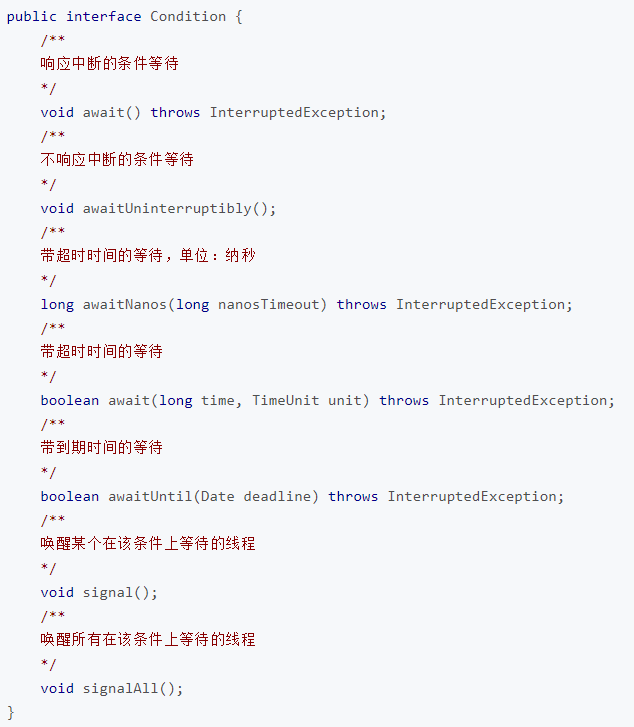
Lock中创建了一个CLH队列以及state，CLH用来存放等待线程，实现自旋，state代表共享变量，在并发量不高的情况下Synchronized的效率高于Lock

Condition



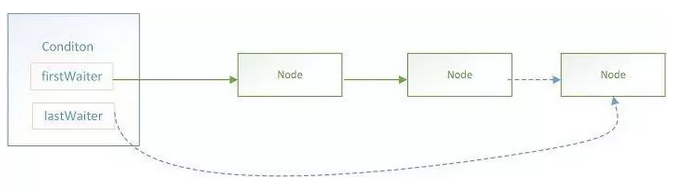
Condition它更强大的地方在于：能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition。Condition可以唤起指定类型的线程，而Object的notify()或notifyAll()不能明确指定唤起的线程。

为什么方法名不直接叫wait()/notify()/nofityAll()？因为Object的这几个方法是final的，不可重写！Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

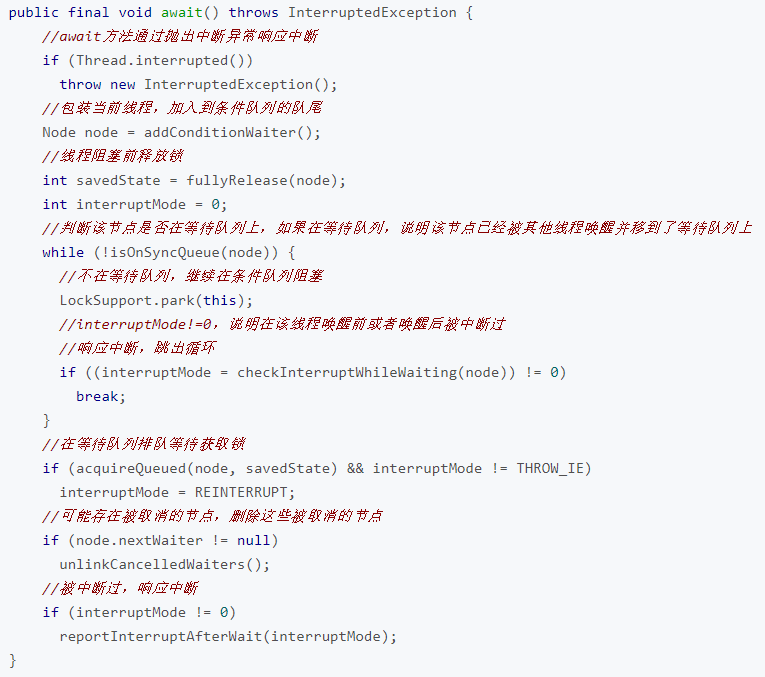


Condition也叫条件队列，AbstractQueuedSynchronizer实现了该接口(ConditionObject)。

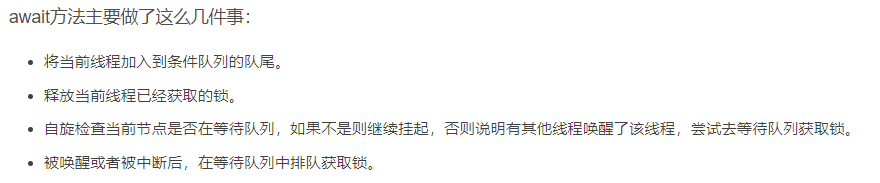
ConditionObject是通过firstWaiter和lastWaiter维护了一个条件队列。



await



将等待的线程封装为一个Node(waitStatus为CONDITION)，是否线程占用的锁且阻塞，并存放到一个条件队列中



Signal：



doSignal方法遍历条件队列，直到找到第一个未取消的节点。找到第一个未取消的节点后将该节点移到等待队列



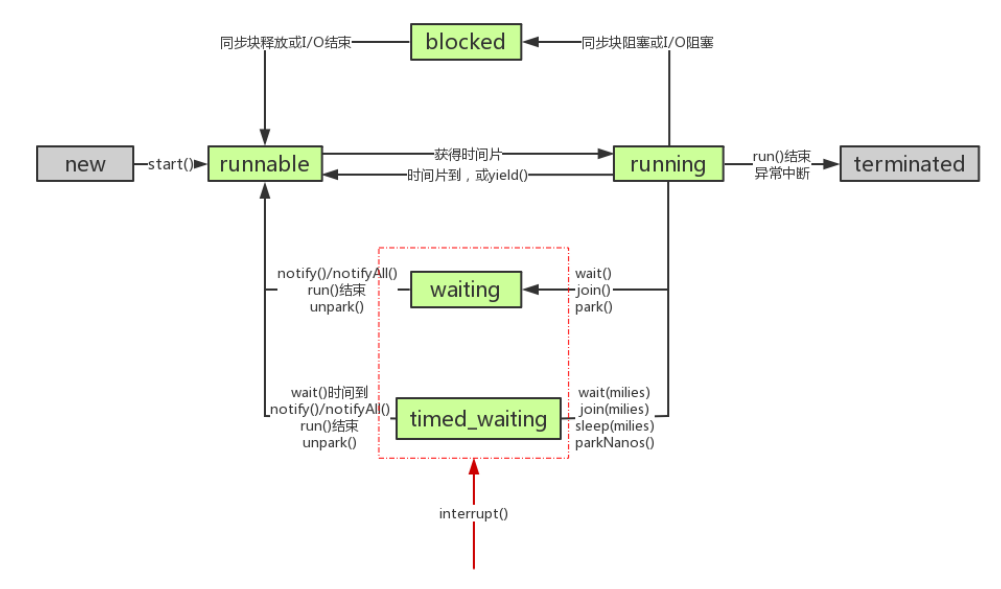
SIGNAL表示该Node是等待线程(等待获取锁)，CONDTION表示该Node是条件线程(等待被唤醒)

SignalAll就是遍历条件队列，调用transferForSignal方法，将节点从条件队列移到等待队列



lock优先考虑获取锁，待获取锁成功后，才响应中断。

lockInterruptibly 优先考虑响应中断，而不是响应锁的普通获取或重入获取。



指令重排序：Happen-Before

在执行指令的过程中，因为CPU和内存速度不匹配，在执行指令的时候可能会出现重排序，比如在new Object()操作后对int a进行初始化，创建对象需要分配内存，耗时，则CPU等待，所以CPU可以去执行int a操作，指令被重排序，这是2个指令之间没有联系可以重排序，但是指令之间相互联系就不能这样，所以需要规则来约束，这个规则叫Happen-Before

①程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作(但是还是会发生指令重排序，在多线程中不能正确执行)

②锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作

③volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作(两个线程，一个写，一个读，写先；将写操作视为原子操作)

④传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

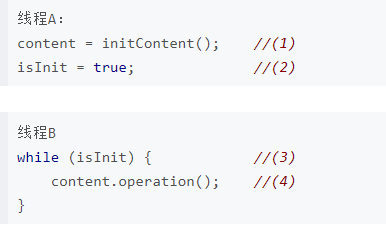
⑤线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

⑥线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

⑦线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

⑧对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

前4条重要，后4条显而易见



但是在如上情况下，线程A，B执行的不是同一段代码，所以会在满足Happen-Before的情况下发生指令重排序，但是，如果发送指令重排序(非常有可能，initContent操作耗时)，线程A中(2)先执行，(1)还未执行，线程B执行，就可能抛出异常[这也属于书写代码不规范导致的]，所以需要(2)在(1)之后执行，可以对isInit加volatile，作为指令重排序的分割点

Volatile实现原理：

Volatile被编译后，会多出一个lock前缀指令。lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

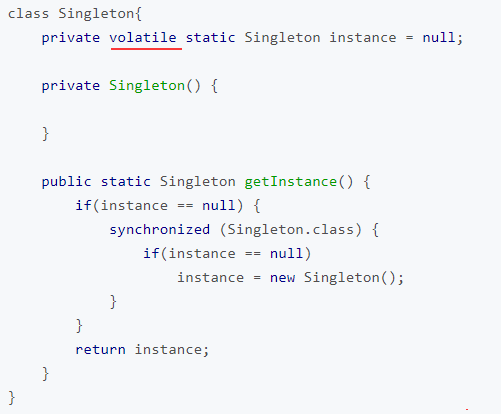
它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

它会强制将对缓存的修改操作立即写入主内存；

如果是写操作，它会导致其他CPU中对应的缓存行无效。

volatile能保证一致性，有序性，但是不能保证原子性(原子操作)

单例模式：



volatile修饰instance主要在于instance = new Singleton()

在JVM中这句话大概做了下面3件事情:

给instance分配内存

调用Singleton的构造函数来初始化成员变量

将instance对象指向分配的内存空间（执行完这步instance就为非null了）。

但是在JVM的即时编译器中存在指令重排序的优化。上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在3执行完毕、2未执行之前，被线程二抢占了，这时instance已经是非null了（但却没有初始化），所以线程二会直接返回instance，然后使用就会报错。

## 并发容器

虽然在java的集合中提供了线程安全的集合，但是这些线程安全的集合都是在方法前加Synchronized方法，效率低下

ConcurrentHashMap

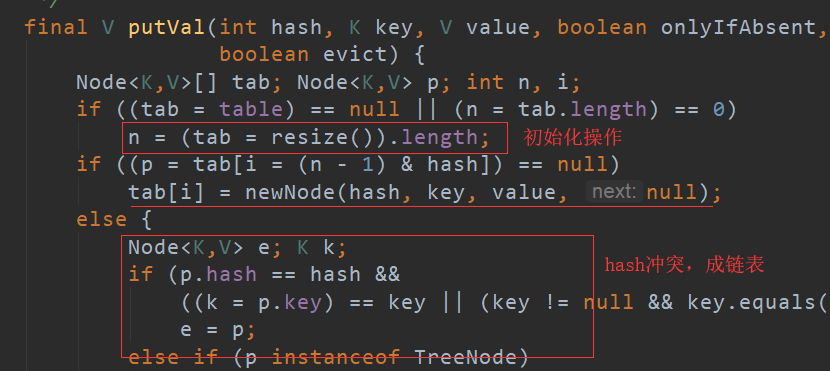
ConcurrentHashMap实现了HashTable的所有功能，线程安全，但却在检索元素时不需要锁定，因此效率更高。

为什么hashMap的value可以为null，而hashTable，ConcurrentHashMap的value不能为null？

因为在get获取null时，不知道是K本身对应的V就是null，还是K根本未找到映射，在HashMap中在get之前有一个contains(key)操作，检测K是否存在，而加支持并发的HashTabella中，contains(key)和get是两个操作，可能contains为true，单后面被其他线程移除了，导致get获取到null，所以在添加的时候就不支持添加V为null

HashMap非线程不安全的原因：

当两个线程同时进行put操作时，线程A，B的Entry的hash相同，会导致hash冲突，形成链表，但是在多线程环境下，可能只有线程A或B的Entry会存放到Map中，因为HashMap是不同步的



由于hash相同，I = (n -1) & hash的结果一样，最终会导致tab[i]只会存储一个线程的Entry，而不是成链

当容量不足时，扩容，会新开辟一个原来2倍的数组，然后将原来的Entry赋值到新的数组中，但是在赋值过程会重新计算Entry的位置，不是原封不动的复制过去，在赋值过程中会重新计算hash(index = HashCode（Key） & （Length - 1）)，这步操作交rehash操作



在网上有在多线程环境下，resize可能会导致节点成环，在JDK1.8之后是不会的，通过分析内存模型得出的，但是在多线程环境下，还是会导致其他很多和预期不一样的结果，所以HashMap只能在方法内部使用，多个线程之间不要共享，只有线程共享的数据才会产生意外的结果，如果是在方法里，是不会出现这样的情况，执行方法时，开辟栈空间，线程私有

当容量不足时，扩容，会新开辟一个原来2倍的数组，然后将原来的Entry赋值到新的数组中，但是在赋值过程会重新计算Entry的位置，不是原封不动的复制过去，在赋值过程中会重新计算hash(index = HashCode（Key） & （Length - 1）)

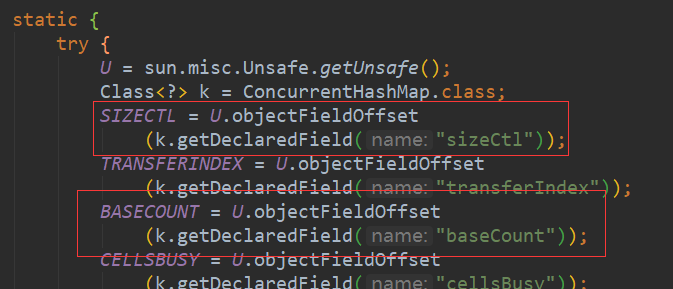
HashMap为什么不直接采用数组+红黑数的存储方式，而采用数组+链表+红黑树，链表查询时间复杂度为O(n)，而红黑数的时间复制度为O(logN)，但是在添加的时候，链表比红黑色快(当链表长度超过8位，链表转化为红黑树)

private static final long SIZECTL; 阀值

private transient volatile int sizeCtl;

private static final long BASECOUNT; 容量

private transient volatile long baseCount;

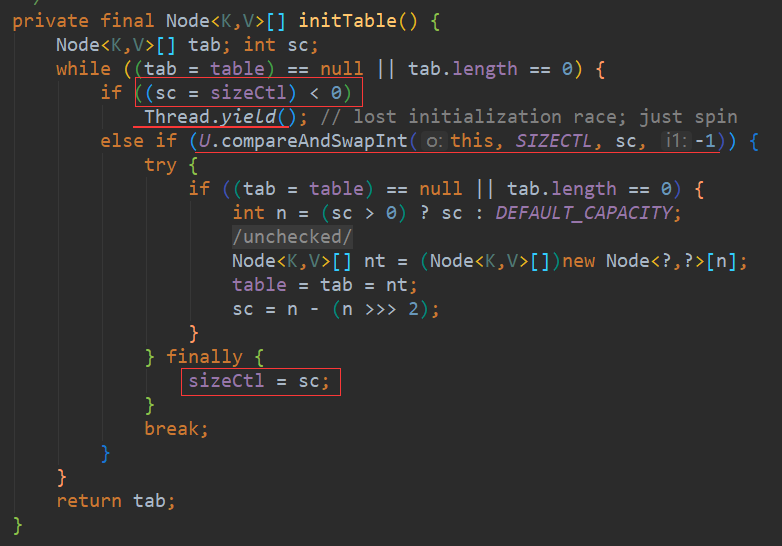


SIZECTL表示的是sizeCtl的偏移量(应该就是内存中地址)

sizeCtl为-1时表示table正在初始化，-N表示有N-1个线程正在进行扩容操作，其余情况表示table的阀值

ConcurrentHashMap要做到线程安全，无非是在添加，扩容，初始化的时候做到线程安全即可，ConcurrentHashMap是通过Synchronized+CAS实现线程安全的

初始化操作：JDK1.8以后，都是put的时候才会初始化table



第一个线程执行CAS，将sizeCtl的值变为-1(SIZECTL对应sizeCtl)

Thread.yield() 将线程运行状态变为就绪状态

扩容：

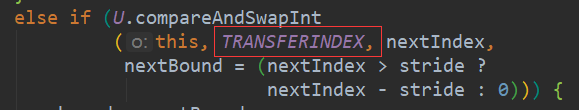
ConcurrentHashMap的扩容机制：先创建一个原来2倍容量的数组，然后多个线程将原来的数组往新数组里赋值，加快了速度。



这里的nextTab就是新table，nextTab可能为null，但是只要在容量不足的时候传递过来的nextTab为null，所有线程只要一个线程能创建nextTab(具体在addCount中分析)



ForwardingNode的Hash值为-1(表示整个table中扩容)，后面跟着nextTab



设置当前线程操作的Node，TRANSFERINDEX(transferIndex)线程共享



当线程将Node赋值到nextTab后，将原来table对应的Node，设置为ForwardingNode，帮助扩容

扩容比较复杂

Put操作：

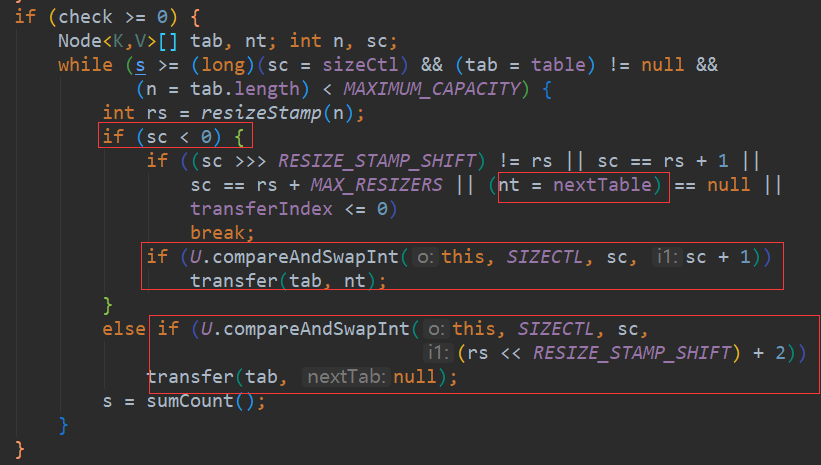


Put操作完成后，会对baseCount进行+1操作(addCount)



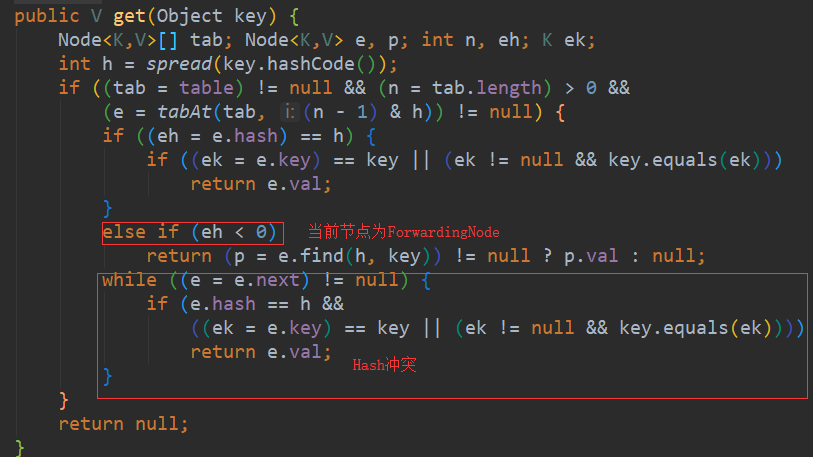
这里是对baseCount进行+1操作，如果失败会采用LongAdder进行操作，LongAdder的效率比AtomicLong快，但是更占内存

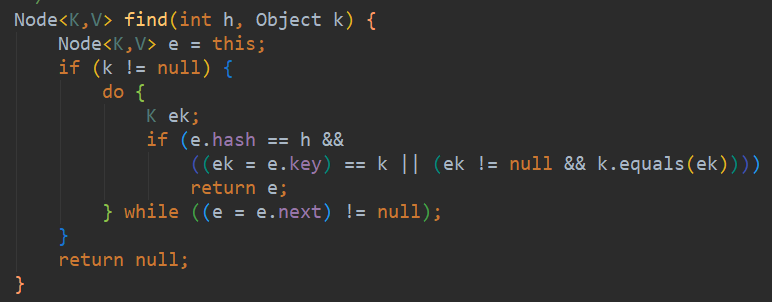
addCount时扩容操作



最终只有一个线程在执行transfer的时候传入的nextTab为null

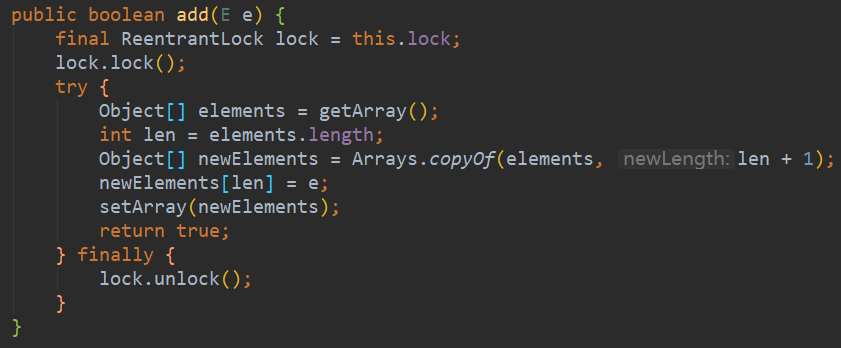
Get方法：





CopyOnWriteArrayList(COW)

当在添加元素的时，会新创建一个数组，然后将原来的值复制到新数组中，实现读写分离，所以在初始化CopyOnWriteArrayList的实现需要指定大小(默认为0)并赋值



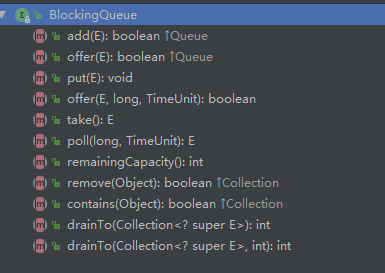
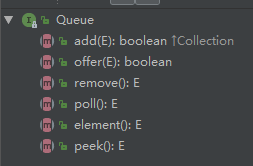
这里任然要加锁，不然会导致副本重复

但是当向CopyOnWriteArrayList中写入大数据时，可能会触发Full GC，而且会导致数据不一致问题，当将数据写入到新数组中，但还未执行setArray，导致在读取的时候数据不一致

根据CopyOnWrite的思想，也可以对Map进行CopyOnWrite，但是一般使用ConcurrentHashMap

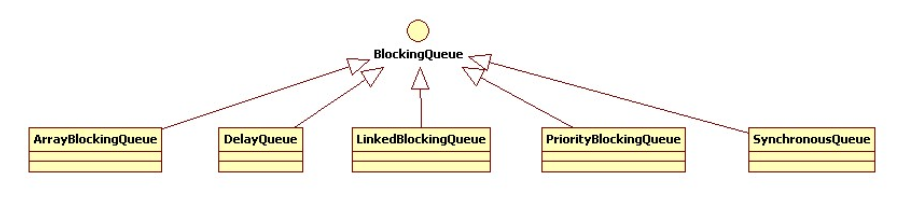
BlockingQueue：阻塞队列

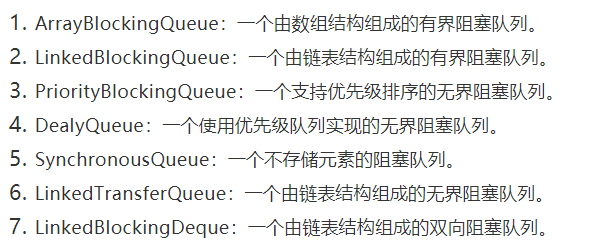
在生产者消费者中，需要通过线程通信的机制来唤醒或阻塞线程，非常麻烦，可以采用BlockingQueue作为生产者消费者的共享容器，当BlockingQueue为空时，读线程会被阻塞，当BlockingQueue满时，会阻塞写线程，不用我们控制线程的阻塞和唤醒

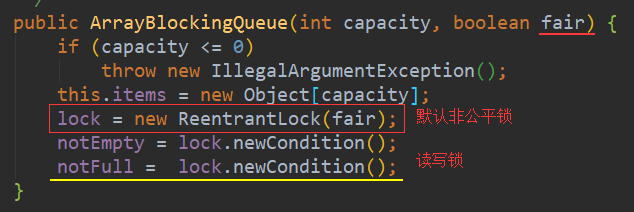


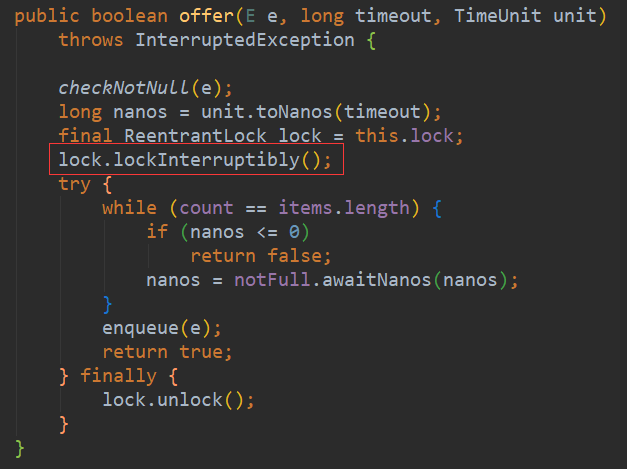
BlockingQueue中实现了非阻塞队列的方法，一般使用阻塞队列的时候，使用的是put/take方法





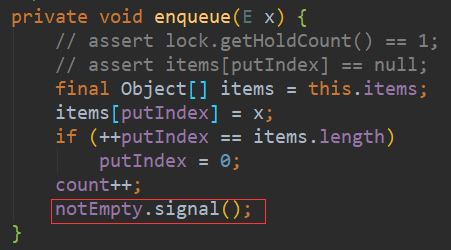
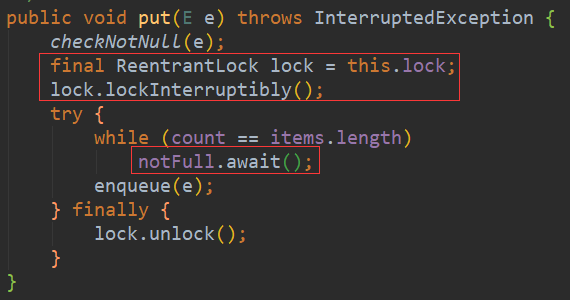
ArrayBlockingQueue：(先进先出)

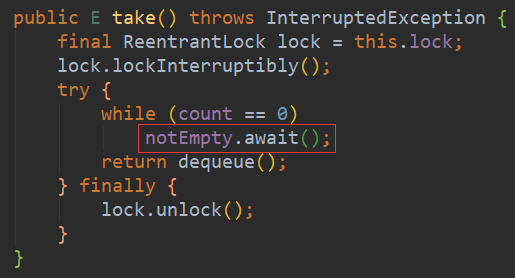
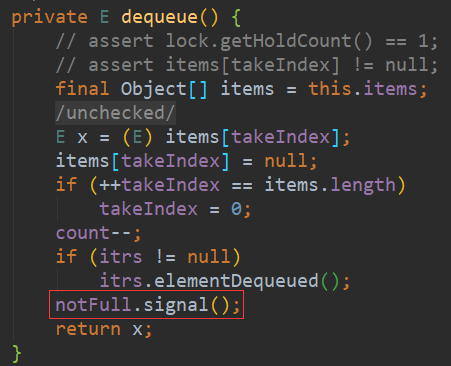




Offer(int)是在线程获取到锁后开始计时的

非同步方法都是先获取锁，先执行



Array实现先进先出操作：count统计容量，putIndex记录添加下标，takeIndex记录获取下标

Count不用是volatile的，应该采用lock，线程安全

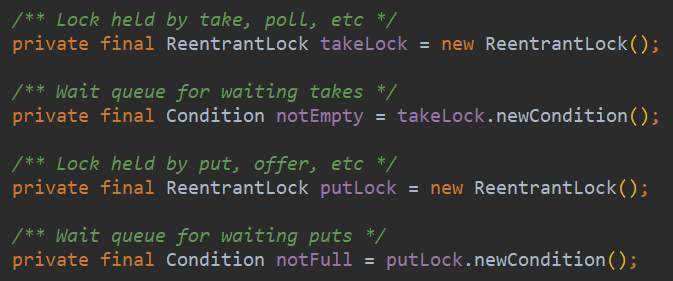
删除元素的操作：take，poll，remove

Poll直接删除第一个元素，队列为空，返回null

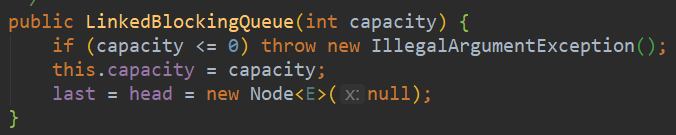
Take直接删除第一个元素，队列为空，阻塞

Remove根据出入的Obj删除，并将前面的元素往后移

LinkedBlockingQueue：

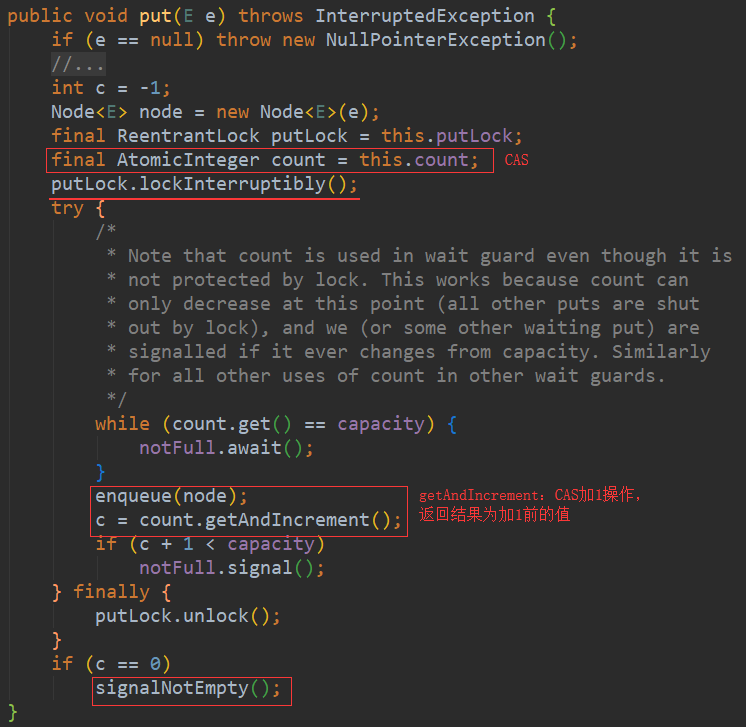


实现读写锁分离



LinkedBlockingQueue有默认的构造函数，capacity的为Integer.MAX\_VALUE，但是当写线程大于读线程速度时，会导致OOM，所以一般指定capacity

Put

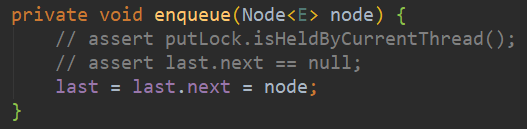
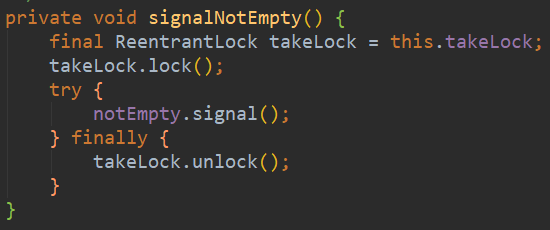


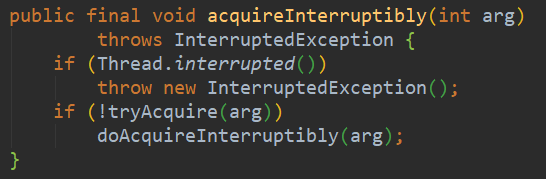
这里put后，唤起的生产者，而不像ArrayListBlocking那样唤起的是消费者，因为Linked采用的是读写锁分离，两者不影响，生产者可生产，说明队列不满，可以唤起其他被阻塞的生产者，这也是为什么Linked吞吐量大的原因

C==0才唤起消费者的原因：c既然大于0说明存在消费者在消费，c=0时，消费者都被阻塞，但是此时唤醒了生产者，生产者生产，所以可以唤醒消费(关键：生产者消费者并未共用同一把锁)

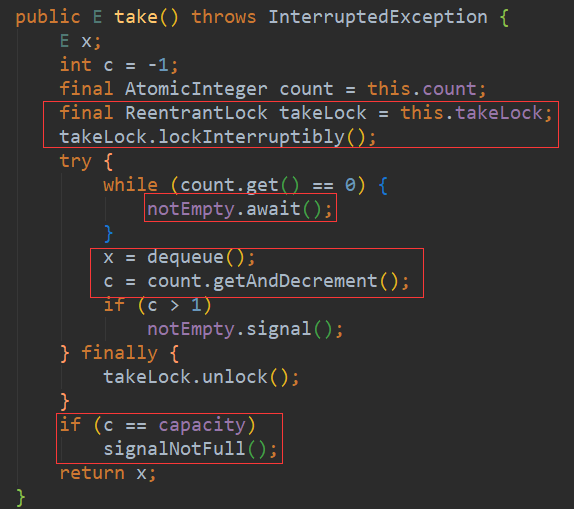
消费消费后，唤醒其他消费者

生产生产后，唤醒其他生产者

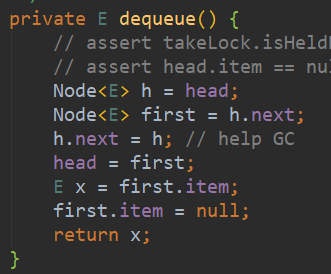
 

有些方法会抛出InterruptedException异常，使用它们获取锁的方法是lockInterruptibly，当线程interrupted后调用lockInterruptibly方法就会抛异常，lockInterruptibly内部实现的

Take：



实现方式差不多



队列大小用capacity表示，元素个数用AtomicInteger

ArrayBlockingQueue和LinkedBlockingQueue的区别：

Array有界，初始化时必须指定大小，Linked有/无界，无界(Integer.MAX\_VALUE)

Array只有一个锁，Linked读写锁分离，所以Linked的并发性高，但是每次Put，Take都生成一个Node，占用内存

由于Array只有一个锁，Linked两个锁，Array的count直接采用int i，而Linked采用的是AtomicInteger

对notEmpty和notFull的唤醒操作均使用的是signal()而不是signalAll()。

DelayQueue：

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

使用场景：

　　DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个超时未响应的连接队列

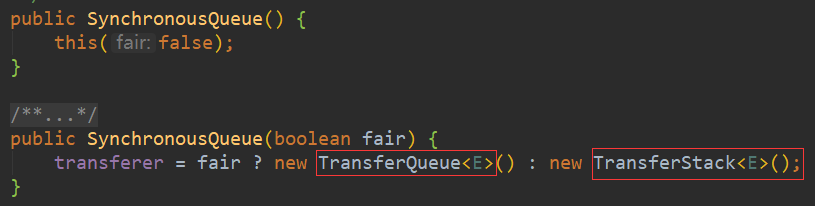
PriorityBlockingQueue：

基于优先级的阻塞队列（优先级的判断通过构造函数传入的Compator对象来决定），但需要注意的是PriorityBlockingQueue并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。在实现PriorityBlockingQueue时，内部控制线程同步的锁采用的是公平锁。

SynchronousQueue：

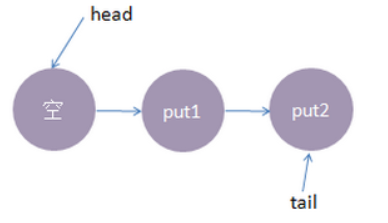
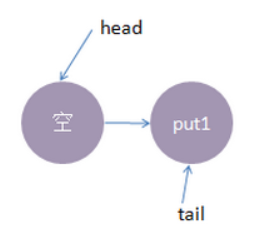
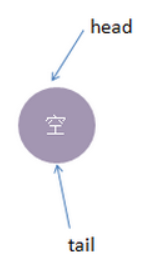
SynchronousQueue内部没有缓冲队列，当生产者put操作后，必须等消费者take后生产者才能继续生产，内部没有队列。调用take时，需要等待有生产者put，生产者和消费者互相等待对方，握手，然后一起离开。

SynchronousQueue通过CAS保存线程安全，默认才非公平锁



公平锁：

TransferQueue这个内部队列，它有一个head和tail指针，用于指向当前正在等待匹配的线程节点

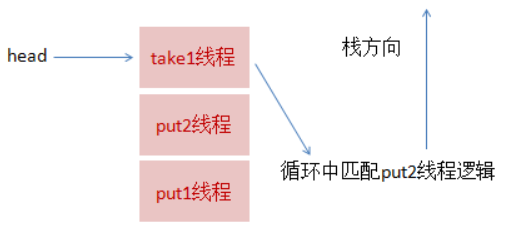


put1线程入队列，自旋一小会后睡眠等待，put2线程入队列，自旋一小会后睡眠等待

当线程来take时，从head开始，公平

非公平：

非公平模式底层的实现使用的是TransferStack，一个栈，实现中用head指针指向栈顶

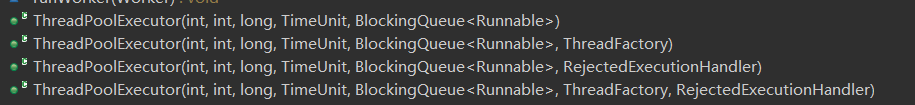


Put1线程，put2线程入栈后，take1线程读取，take1会入栈，然后take1线程循环执行匹配put2线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1线程，发生并发冲突，take1匹配put1

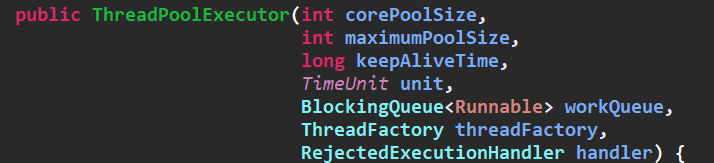
SynchronousQueue主要put必须要take后，put线程才会结束

## 线程池

线程的创建和销毁都会占用大量资源，所以采用线程池来获取线程



ThreadPoolExecutor是线程池的一个核心类，提供了4个构造器，前三个构造器实际最终都是调用第四个构造器



corePoolSize：核心池大小，创建线程池后，默认线程池中没有任何线程，当有任务来了的时候就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会将到达的线程存放到缓存队列中

maximumPoolSize：线程池最大线程数，即线程池中最多能创建多少个线程

keepAliveTime：线程没有任务执行时在保持多久的时间会终止，默认情况下只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用[只有线程数大于核心线程数时才会挂掉线程]，但是如果调用了allowCoreThreadTimeOut(true)[允许核心线程TimeOut]时，在线程池中的线程数不大于corePoolSize时，keepAliveTime也会起作用

unit：keepAliveTime的时间单位

workQueue：阻塞队列，储存等待执行的线程ArrayBlockingQueue/LinkedBlockingQueue/

SynchronousQueue用的较多，PriorityBlockingQueue/DelayedWorkQueue用得少

threadFactory：线程工厂，主要用来创建线程(可以指定，也可以采用默认的DefaultThreadFactory)

handler：表示当拒绝处理任务时的策略(线程数达到maximumPoolSize时，有新线程请求将拒绝)

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。 默认，用者可以捕获这个异常，然后根据需求编写自己的处理代码。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

可以通过setCorePoolSize()和setMaximumPoolSize()对线程池进行动态调整

工作流程：

默认情况下，创建完线程池后并不会立即创建线程, 而是等到有任务提交时才会创建线程来进行处理。（除非调用prestartCoreThread或prestartAllCoreThreads方法）

当线程数小于核心线程数时，每提交一个任务就创建一个线程来执行，即使当前有线程处于空闲状态，直到当前线程数达到核心线程数。

当前线程数达到核心线程数时，如果这个时候还提交任务，这些任务会被放到队列里，等到线程处理完了手头的任务后，会来队列中取任务处理。

当前线程数达到核心线程数并且队列也满了，如果这个时候还提交任务，则会继续创建线程来处理，直到线程数达到最大线程数。

当前线程数达到最大线程数并且队列也满了，如果这个时候还提交任务，则会触发饱和策略。

如果某个线程的控线时间超过了keepAliveTime，那么将被标记为可回收的，并且当前线程池的当前大小超过了核心线程数时，这个线程将被终止。

线程池的关闭：当线程池不使用时，如果不关闭，GC是不会回收的，这样就会导致内存泄漏

shutdown：不会立即终止，等待所有线程执行完毕(包含任务缓存队列里的)才关闭

shutdownNow：立即终止，包含正在执行的任务

设置allowCoreThreadTimeOut为true，在过期时间后，核心线程会被释放，则线程池被GC回收

ThreadPoolExecutor比较重要的几个方法

execute()：核心方法，通过此方法可以向线程池提交一个任务，交由线程池执行(执行一个Runnable任务)

submit()：也是用来向线程池提交任务额，但是和execute不同，它能够返回任务执行的结果(Callable任务)

shutdown()和shutdownNow()是用来关闭线程池的。

表示线程池状态

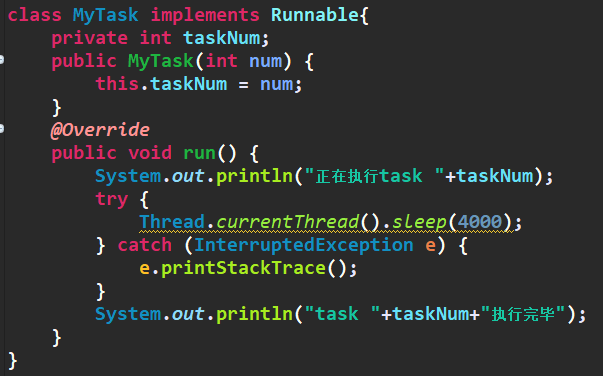
volatile int runState; //当前线程池的状态，volatile变量用来保证线程之间的可见性

static final int RUNNING = 0; //创建线程池后，初始时

static final int SHUTDOWN = 1; //调用shutdown方法后，线程池不接受新的任务，但是让所有等待的线程执行完任务

static final int STOP = 2; //调用shutdownNow方法后，线程池不接受新的任务，并且中断所有正在执行的任务

static final int TERMINATED = 3; //调用shutdown/shutdownNow方法后，等待所有线程结束(包含缓存中)后，线程池的状态



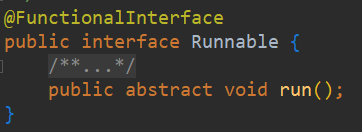
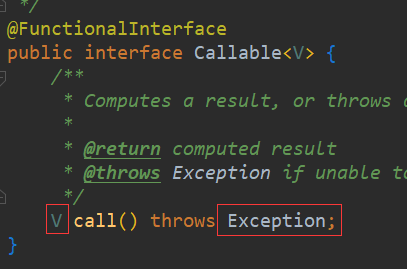


以封装好的线程池：可以调用Executors的方法来创建线程池



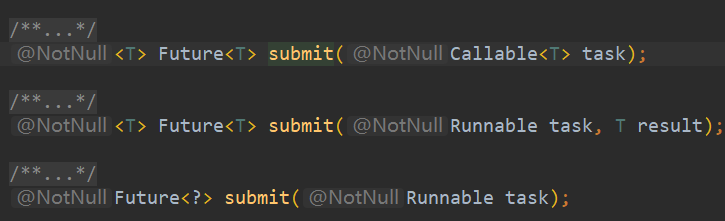
Callable/Future：

在使用Runnable或Thread创建线程时，不能或许线程执行完返回的结果，可以通过Callable和Future实现



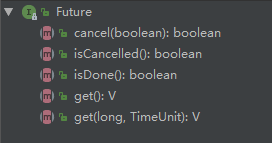
Call方法有返回值，可能会抛出异常，

Callabll一般情况下是配合ExecutorService来使用的



一般情况下我们使用第一个submit方法和第三个submit方法，第二个submit方法很少使用。ThreadPoolExecutor线程池就是ExecutorService的实现方法

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。



cancel方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务。如果任务已完成，直接返回false

isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。

isDone方法表示任务是否已经完成，若任务完成，则返回true；

get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；

Future提供：判断任务是否完成；能够中断任务；能够获取任务执行结果。

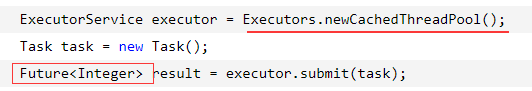
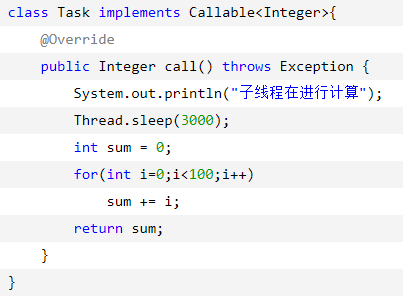
Callable产生结果，Future获取结果，Future的获取方式无非就是CAS机制

FutureTask：Future接口的实现类

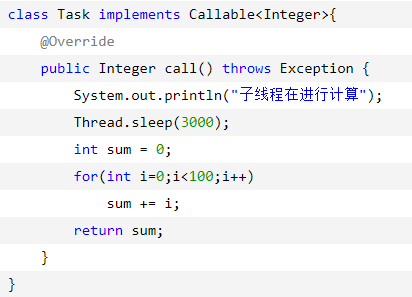
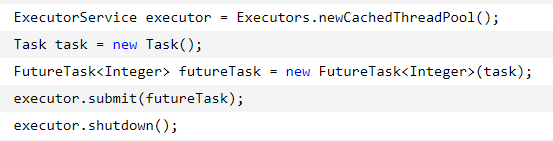




Callable创建线程，然后将Callable交给ExecutorService执行，返回一个Future，根据需求执行future的方法

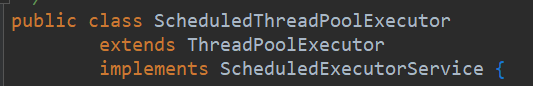


线程池就是ExecutorService的实现类，执行线程池中线程可以调用execute(Runnable任务)和submit方法(Callable任务，也可以是Runnable任务)，这里返回的result就是FutureTask

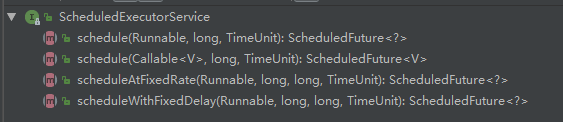
 

其实这方式和第一种类似，线程池底层就是这样实现的

ScheduledThreadPoolExecutor：定时任务

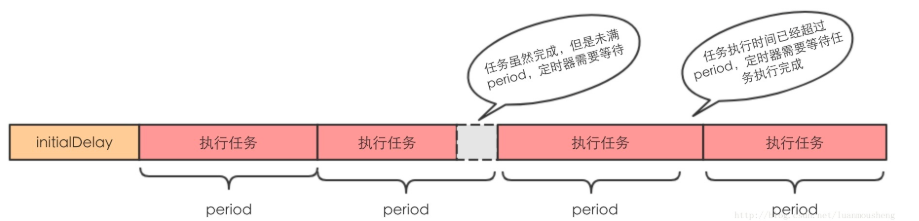


继承线程池，实现接口



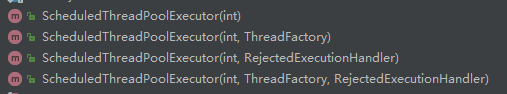
Schedule在指定时间后开始执行任务，一个是Runnable，一个是Callable

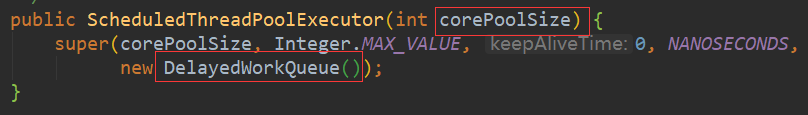
scheduleAtFixedRate：定时延后开始执行任务，以后每隔period时长再次执行该任务



scheduleWithFixedDelay：定时延后开始执行任务，以后任务执行完成后等待delay时长，再次执行任务







Int为核心线程数

无论哪种方式执行定时任务，最终都会调用delayedExecute方法



定时任务就是将任务存放到线程池中，然后到达时后执行线程

MQ：

MQ和Redis类似，在服务器上安装，然后通过jar操作命令

Executor框架：就是线程池相关