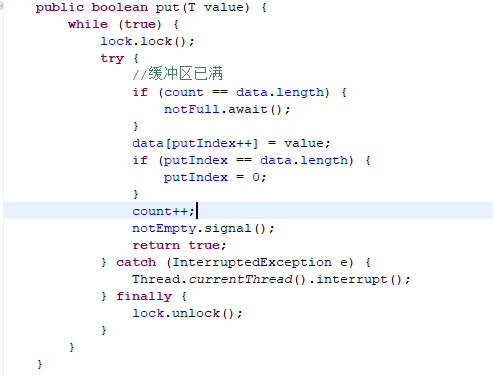
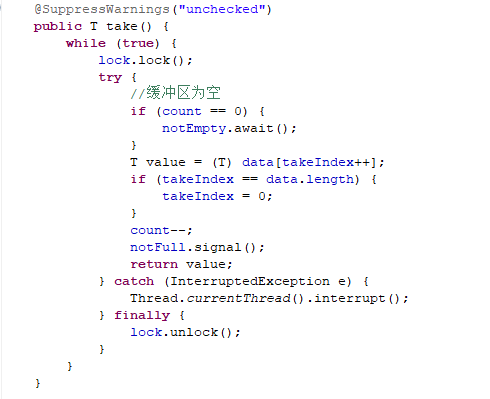
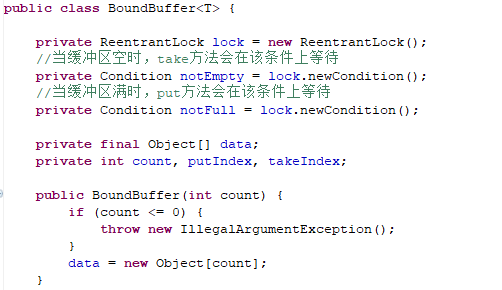
# Condition(jdk1.7.0\_67)

在Synchronizer声明的同步块中，线程间的通信可以使用监视器方法（wait，notify，notifyAll）实现。而在使用JUC下的Lock进行编程时，应该使用Condition代替监视器方法。Condition不仅实现了监视器方法的效果，而且实现了不响应中断等待，超时等待等方法。同样的，Condition需要依赖于独占锁，当未获取锁调用Condition的方法时，会抛出IllegalMonitorStateException异常。下面看一个使用Condition实现有界缓冲区的例子：

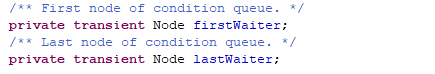


take获取缓冲区中的数据，当缓冲区为null时，在notEmpty条件上等待，直到其他线程添加元素后被唤醒，然后获取元素并唤醒在notFull条件上等待的线程。

put方法往缓冲区中添加元素，当缓冲区满时，在notFull条件上等待，直到其他线程获取元素后被唤醒，然后添加元素并唤醒在notEmpty条件上等待的线程。

# ConditionObject

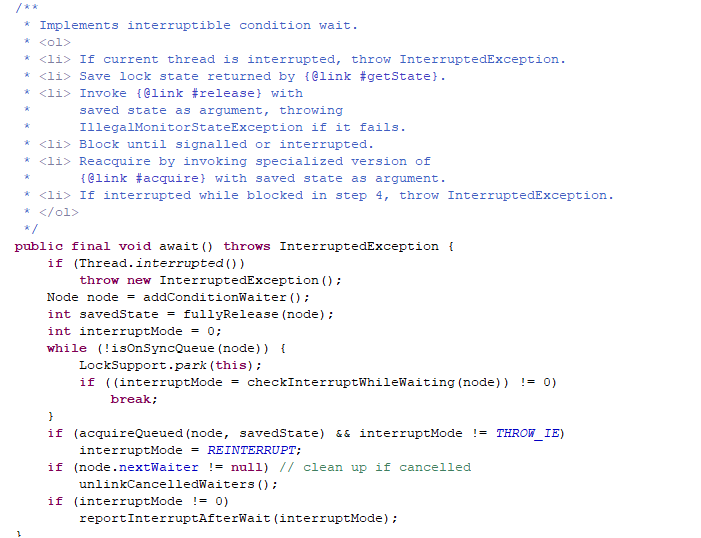
ConditionObject是在AbstractQueuedSynchronizer中的实现。ConditionObject内部维护了一个条件队列，条件队列中存储了在该条件上等待的线程，当调用await方法时，当前线程会被添加到队列的尾部，调用signal方法时，处于头部的线程将被唤醒，并从条件队列中移除，添加到同步队列中。



firstWaiter指向条件队列的第一个节点，lastWaiter指向条件队列的最后一个节点。Node中的nextWaiter属性指向了后继节点。条件队列为为一个单项队列。

# 方法

## 2.1await



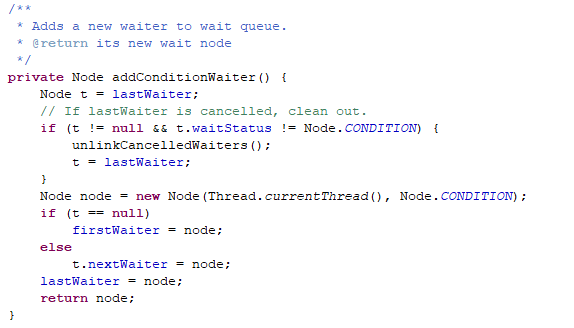
await执行流程如下。

1. 检查当前线程中断状态，响应中断。
2. 将当前线程添加到条件队列。
3. 释放锁并保存锁的状态。
4. 循环检查当前线程是否在同步队列中。处于同步队列中，执行（6）否则执行（5）。
5. 阻塞当前线程直到其他线程唤醒。唤醒之后重新检查中断并设置interruptMode，如果interruptMode!=0，跳出循环，否则继续（4）。
6. 当前线程处于同步队列，说明有其他线程唤醒了当前线程。获取锁，当前线程已经中断并且interruptMode!=THROW\_IE，设置interruptMode=REINTERRUPT。
7. 如果node.nextWaiter不为null，删除条件队列中已经取消的节点。
8. 根据interruptMode的值，处理线程中断。

下面挨个步骤来看一下。

## 2.2addConditionWaiter

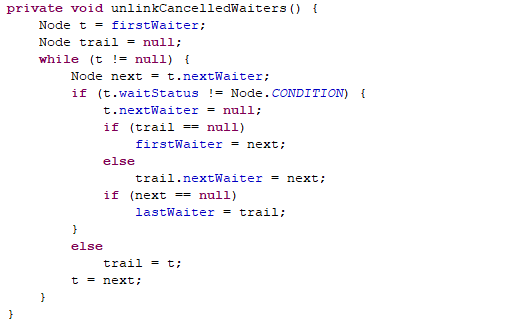
将当前线程添加到条件队列，2.1小节中的（2）。这里不会出现并发情况，因为只有获取到独占锁的线程才有资格执行Condition中的方法。

（1）将条件队列的最后节点封闭在栈上。

1. 检查t节点的状态，如果t!=null并且t.waitStatus!=CONDITION（这里只可能是CONDITION，CANCELLED两种状态），删除条件队列中已经取消的节点。重新设置t。
2. 新建节点。
3. 将节点添加到条件队列。

## 2.3unlinkCancelledWaiters

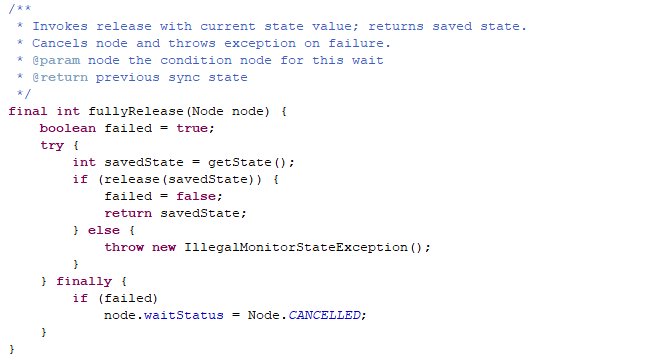
删除条件队列中已经取消的节点。



单向链表的删除操作。

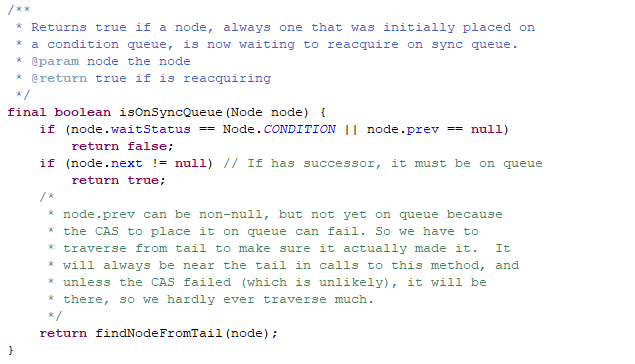
## 2.4fullyRelease

释放锁并返回锁的状态。2.1小节中的（3）。如果当前线程没有获取锁，该方法或抛出IllegalMonitorStateException异常。

 调用release释放锁，所以ConditionObject只能在独占模式下使用。释放成功，返回true，否则抛出IllegalMonitorStateException异常。如果释放失败将该节点设置为CANCELLED状态。会在其他线程调用await方法中移除取消的节点。

## 2.5isOnlySyncQueue

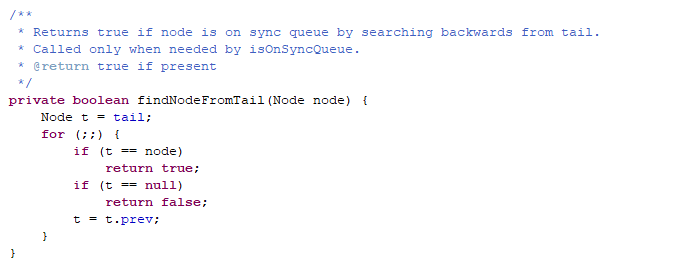
当前节点在同步队列中返回true，否则返回false。2.1小节中的（4）。



这里存在并发情况，因为在调用该方法前已经释放了锁，其他线程可能获取到锁执行signal方法，所以node.prev!=null时，并不能确定一定在同步队列中。signal方法会调用enq使节点入队，而enq在执行完node.prev=t之后执行compareAndSetTail可能会失败，这时候节点的状态为node.prev!=null但是该节点不再同步队列中。所以这里当node.prev!=null时，会调用findNodeFromTail方法遍历整个同步队列，查找当前节点。

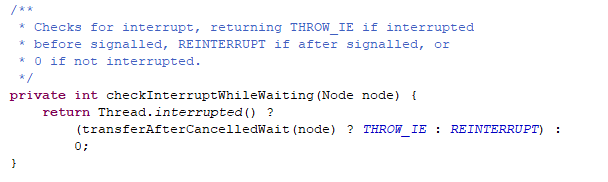
## 2.6findNodeFromTail

如果节点在同步队列中返回true，否则返回false。



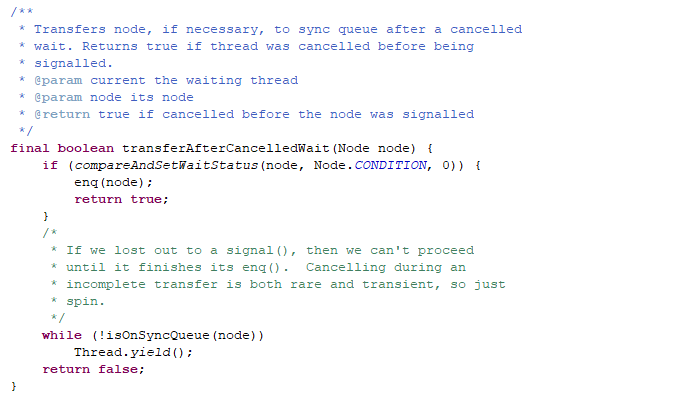
## 2.7checkInterruptWhileWaiting

当线程唤醒后重新检查线程中断状态。如果在其他线程在调用signal之前中断返回THRW\_IE，在调用之后中断返回REINTERRUPT，如果没有中断返回0。2.1小节中的（5）。



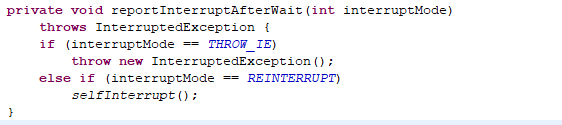
## 2.8transferAfterCancelledWait

设置节点状态并转移到同步队列。在设置节点状态时如果失败，说明有其他线程调用了signal方法并更新了节点状态。那么直到signal将节点添加到同步队列完成后，才会从该方法返回，否则将一直让出执行权。



## 2.9reportInterruptAfterWait

根据interruptMode值选择抛出中断异常还是重新设置中断。

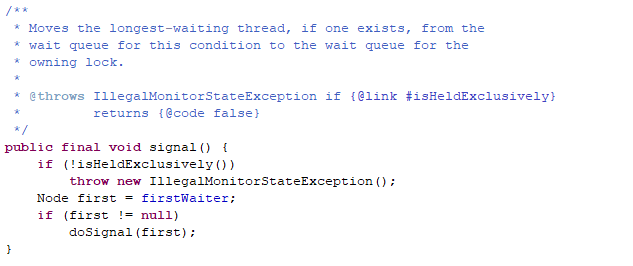


如果interruptMode==THROW\_IE抛出异常，如果

interruptMode==REINTERRUPT，重新设置中断。

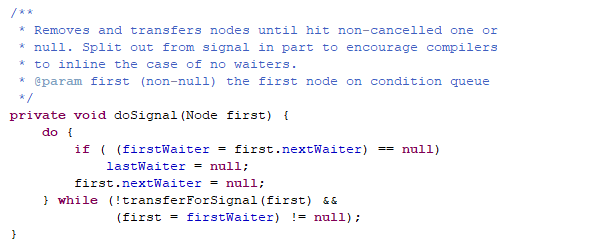
## 2.10signal

移除等待时间最长的线程节点，即头节点，并将该线程添加到同步队列。然后唤醒该线程。调用该方法之前首先获取对应的锁，否则抛出IllegalMonitorStateException异常。



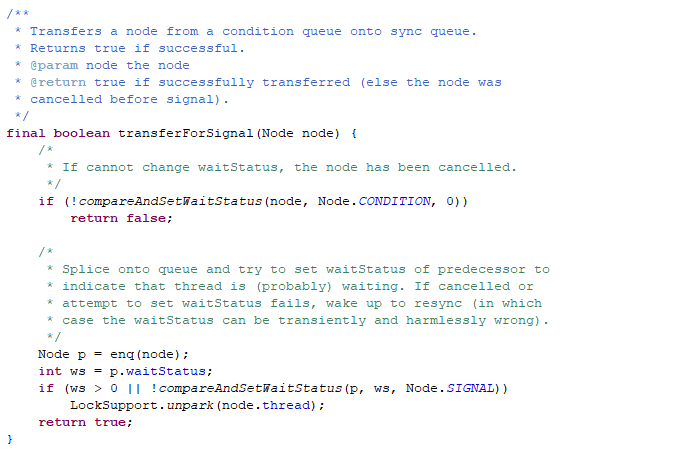
## 2.11doSignal

移除和转移节点直到遇到一个非取消节点或者null。



## 2.12transferForSignal

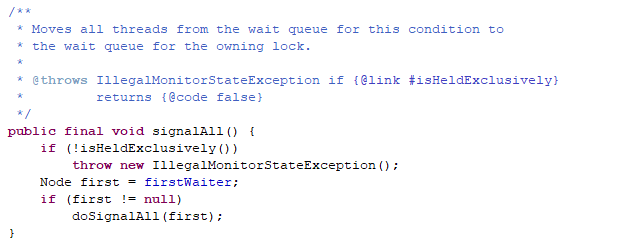
将节点从条件队列转移到同步队列。



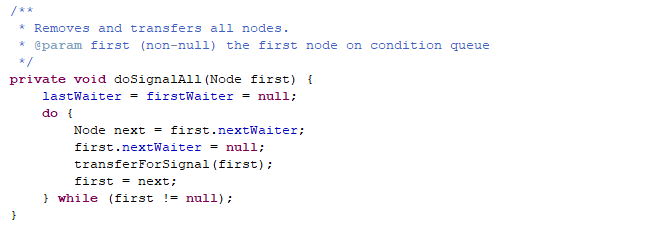
1. 设置节点状态，如果更新状态失败，说明节点被取消，返回false。
2. 将节点入队。
3. 如果节点状态>0或者节点状态<=0并且设置节点状态失败，那么唤醒线程。
4. 返回true。

## 2.13signalAll

移除条件队列中所有的线程，并添加到同步队列，执行该方法前仍要先获取锁。

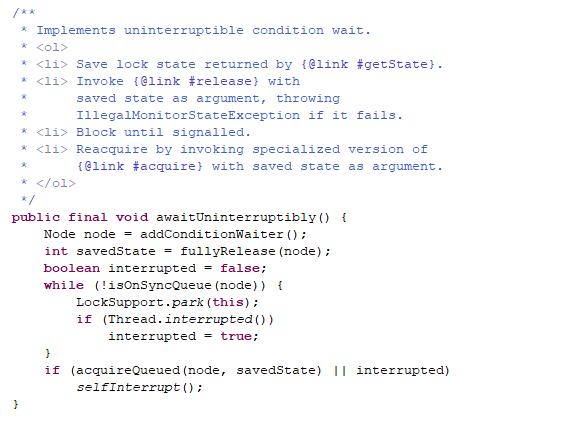


## 2.14doSignalAll



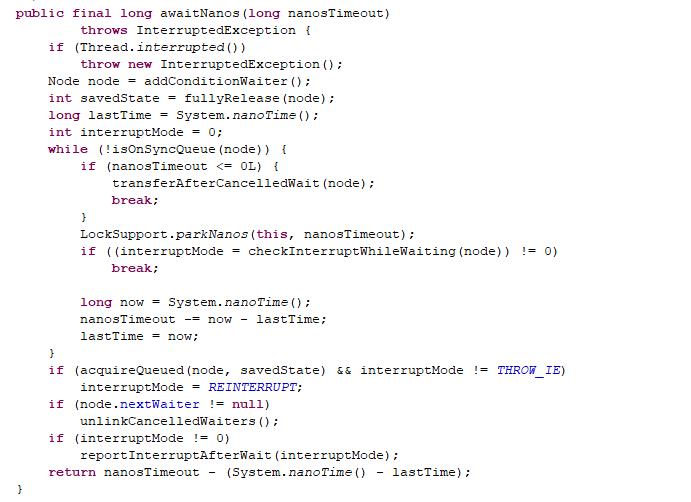
遍历条件队列的每一个元素，并执行transferForSignal方法。

## 2.15awaitUninterruptibly



不响应中断的等待实现。

## 2.16awaitNanos



超时等待，与AbstractQueuedSynchronizer超时获取锁的实现使用了相同机制，这里不再赘述。还有awaitUntil等实现基本类似。