AQS特性学习（二）

1.1可中断锁 lockInterruptibly()

```
1 public void lockInterruptibly() throws InterruptedException {
2  sync.acquireInterruptibly(1);
3 }
```

本质上：最终会调用AQS的 `acquireInterruptibly(1);`模板方法

```
1 public final void acquireInterruptibly(int arg) throws InterruptedExcepti
on {
2  //增加了对异常的状态的判断
3  //如果检测线程中断的状态改变的化，抛出中断异常后方法直接退出
4  if (Thread.interrupted())
```

```
5   throw new InterruptedException();
6   if (!tryAcquire(arg))
7   //同步状态获取失败，调用下面这个方法。
8   doAcquireInterruptibly(arg);
9  }
10
11
12
13
14  protected final boolean tryAcquire(int acquires) {
15      return nonfairTryAcquire(acquires); //调用 nonfairTryAcquire 此时 acquires==1 (想要获得锁)
16  }
17  nonfairTryAcquire（acquires:1）
18  final boolean nonfairTryAcquire(int acquires) {
19      //当前线程
20      final Thread current = Thread.currentThread();
21      int c = getState();//获得 当前锁的状态（1：有锁 0：无锁）
22      if (c == 0) { //当前锁状态 为无锁
23          if (compareAndSetState(0, acquires)) {  //尝试CAS获取锁 acquire为1
24              setExclusiveOwnerThread(current);   //将当前线程设置为当前独占线程
25              return true;        //返回true - tryAcquire() 返回true —— acquire获得同步状态退出
26          }
27      }
28      else if (current == getExclusiveOwnerThread()) {//当前锁状态为有锁 判断当前获得锁是否是当前线程（锁的重入）
29          int nextc = c + acquires;         //锁标记设置为 c+1 ：引用计数+1 锁的重入
30          if (nextc < 0) // overflow        //如果锁标记小于0  则抛出异常 最大锁计数超出
31              throw new Error("Maximum lock count exceeded");
32          setState(nextc);                      //锁标记大于0，则更新锁标记+1（引用计数）
33          return true;//返回true - tryAcquire() 返回true —— acquire获得同步状态退出
34                      //（当前锁就是本线程持有，只是进行了锁的重入）
35      }
36      return false; //当前或的锁的线程不是本线程，返回false：
37   //tryAcquire() 返回false 进入 doAcquireInterruptibly(arg);
38  }
```

本质：同步状态获取失败，调用doAcquireInterruptibly（）方法。

```
1  private void doAcquireInterruptibly(int arg)throws InterruptedException {
2   //跟 acquireQueued方法全部类似，就增加了响应中断的功能
3   final Node node = addWaiter(Node.EXCLUSIVE); //addWaiter在获取锁的内部包装线程为结点并尾插到同步队列里面去
4   //返回包装好的线程
5   boolean failed = true;
6   try {
7   for (;;) {
8   final Node p = node.predecessor();
9   if (p == head && tryAcquire(arg)) {
10   setHead(node);
11   p.next = null; // help GC
12   failed = false;
13   return;
14   }
15   if (shouldParkAfterFailedAcquire(p, node) &&
16   parkAndCheckInterrupt())
17   //即当当前线程前驱结点已经设置为-1，即后继结点（本结点）需要被阻塞
18   //然后执行parkAndCheckInterrupt，线程阻塞，只有当线程收到中断，或者线程被唤醒时才会返回
19   //中断的化返回true，被唤醒的化返回false
20   //返回true就会执行if条件里面的抛出中断异常及时响应中断，反之就直接再次自旋，或者一直被park着
21   //线程被阻阻塞是若检测到中断抛出中断异常退出
22   throw new InterruptedException();
23   }
24   } finally {
25   if (failed)
26   cancelAcquire(node);
27   }
28  }
```

解析：

> **与acquire方法逻辑几乎一致，唯一的区别是当 parkAndCheckInterrupt返回true时即线程阻塞时该线程被中断，代码抛出被中断异常。**

1.2超时等待获取锁（在中断获取锁的基础上增加超时功能)tryLock(long time, TimeUnit unit)

```
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

long time:时间
TimeUnit unit: 时间的单位

ReetrantLock 中的对lock接口的 tryLock(long time,TimeUnit unit)的覆写
public boolean tryLock(long timeout, TimeUnit unit)
 throws InterruptedException {
 return sync.tryAcquireNanos(1, unit.toNanos(timeout));
 }
```

该方法本质当调用AQS的模板方法 tryAcquireNanos

```
public final boolean tryAcquireNanos(int arg, long nanosTimeout) throws I
nterruptedException {
  //检测中断
  if (Thread.interrupted())
  throw new InterruptedException();
  return tryAcquire(arg) ||
  doAcquireNanos(arg, nanosTimeout);
 }

//-------------------------------------------------------------
 protected final boolean tryAcquire(int acquires) {
     return nonfairTryAcquire(acquires); //调用 nonfairTryAcquire 此时 acq
uires==1 (想要获得锁)
 }
 nonfairTryAcquire（acquires:1）
 final boolean nonfairTryAcquire(int acquires) {
     //当前线程
     final Thread current = Thread.currentThread();
     int c = getState();//获得 当前锁的状态（1: 有锁 0: 无锁）
     if (c == 0) { //当前锁状态 为无锁
         if (compareAndSetState(0, acquires)) {  //尝试CAS获取锁 acquire为
1
             setExclusiveOwnerThread(current);   //将当前线程设置为当前独占
线程
             return true;        //返回true - tryAcquire() 返回true —— acq
uire获得同步状态退出
         }
     }
```

```
    else if (current == getExclusiveOwnerThread()) {//当前锁状态为有锁 判断当前获得锁是否是当前线程（锁的重入）
        int nextc = c + acquires;          //锁标记设置为 c+1 : 引用计数+1 锁的重入
        if (nextc < 0) // overflow         //如果锁标记小于0  则抛出异常 最大锁计数超出
            throw new Error("Maximum lock count exceeded");
        setState(nextc);                       //锁标记大于0，则更新锁标记+1（引用计数）
        return true;//返回true - tryAcquire() 返回true —— acquire获得同步状态退出
                    //（当前锁就是本线程持有，只是进行了锁的重入）
    }
    return false; //当前或的锁的线程不是本线程，返回false:
 //tryAcquire() 返回false 进入 doAcquireInterruptibly(arg);
}
```

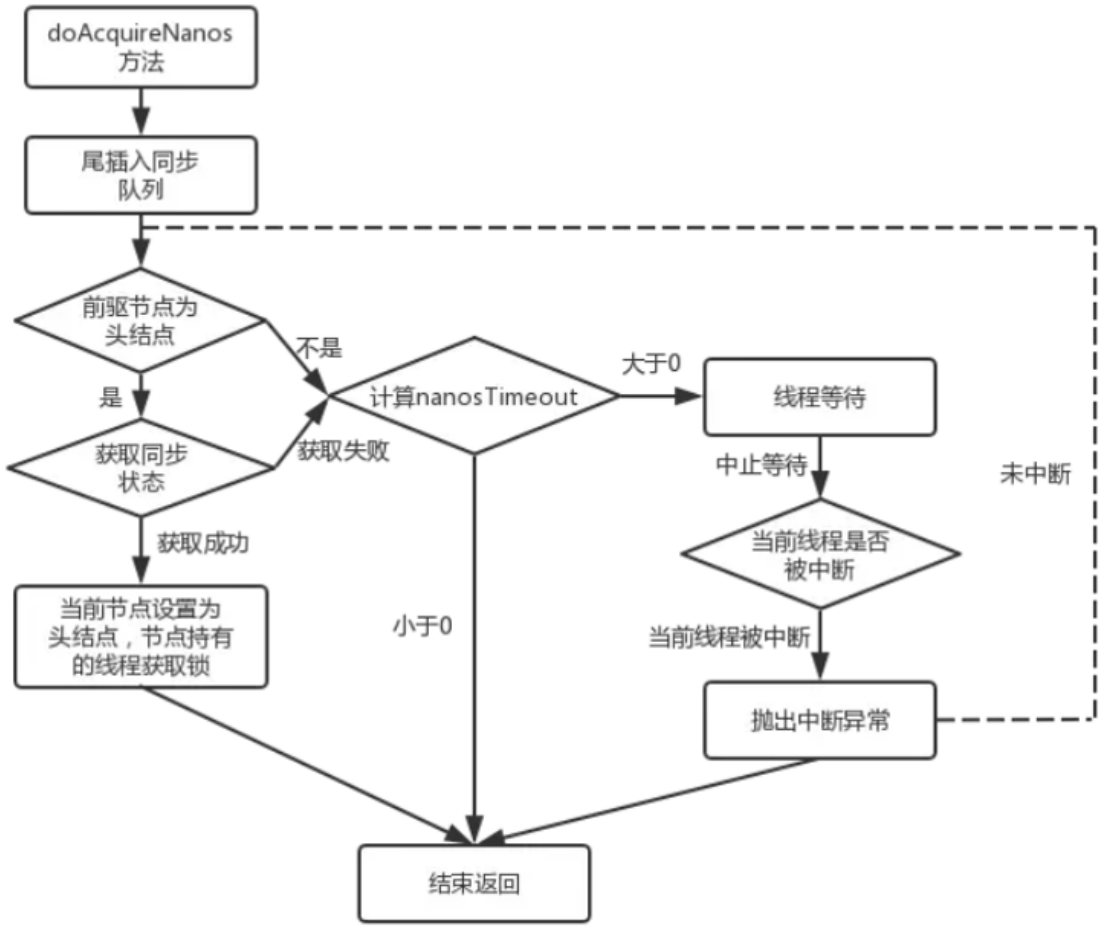
本质功能实现 doAcquireNanos(int arg, long nanosTimeout)

```
private boolean doAcquireNanos(int arg, long nanosTimeout)
 throws InterruptedException {
 //1.传入时间小于0，方法直接退出，线程获取锁失败
 if (nanosTimeout <= 0L)
 return false;
 //2.根据超时时间算出截时间
 final long deadline = System.nanoTime() + nanosTimeout;
 final Node node = addWaiter(Node.EXCLUSIVE);//3.addWaiter在获取锁的内部包装线程为结点并尾插到同步队列里面去
 //返回包装，并尾插好的的线程
  boolean failed = true;
  try {
  for (;;) {
  final Node p = node.predecessor();
  //4.当前线程获得锁出队列
  if (p == head && tryAcquire(arg)) {
  setHead(node);
  p.next = null; // help GC
  failed = false;
  return true;
  }
  //5.1再次计算超时时间 截至时间-当前时间值
  nanosTimeout = deadline - System.nanoTime();
  //5.2判断是否已经超值，线程已经需要退出了，
```

```
 if (nanosTimeout <= 0L)
 return false;
 //5.3线程阻塞等待
 if (shouldParkAfterFailedAcquire(p, node) &&
 nanosTimeout > spinForTimeoutThreshold)
 //5.4在超时时间内仍未被唤醒，线程退出
 //如果在这个时间内没有获得锁，则线程阻塞
 LockSupport.parkNanos(this, nanosTimeout);

 //5.5线程被中断抛出中断异常
 if (Thread.interrupted())
 throw new InterruptedException();
 }
 } finally {
 if (failed)
 cancelAcquire(node);
 }
}



--------------------parkNanos

public static void parkNanos(Object blocker, long nanos) {
 //截至时间-当前时间值：nanos
 if (nanos > 0) {
 Thread t = Thread.currentThread(); //获得当前线程
 setBlocker(t, blocker);
 UNSAFE.park(false, nanos);
 setBlocker(t, null);
 }
}
```

逻辑流程图：

tryLock(long time, TimeUnit unit) p ---->该方法本质当调用AQS的模板方法tryAcquireNanos->doAcquireNanos



逻辑总结：

> **程序逻辑同独占锁可响应中断式获取基本一致，唯一的不同在于获取锁失败后，对超时时间的处理上，**
> **在第1步会 先计算出按照现在时间和超时时间计算出理论上的截止时间，比如当前时间是8h10min,超时时间是10min，那么根 据deadline = System.nanoTime() + nanosTimeout计算出刚好达到超时时间时的系统时间就是8h 10min+10min = 8h 20min。**
> **然后根据deadline - System.nanoTime()就可以判断是否已经超时了，比如，当前系统时间是8h 30min 很明显已经超过了理论上的系统时间8h 20min，deadline - System.nanoTime()计算出来就是一个负数，自然而然 会在3.2步中的If判断之间返回false。**
> **如果还没有超时即3.2步中的if判断为true时就会继续执行3.3步**
> **通过 LockSupport.parkNanos使得当前线程阻塞，**
> **同时在3.4步增加了对中断的检测，**
> **若检测出被中断直接抛出被中断异常。**

特性总结：

1.独占式：获取锁（aquire）与释放（release）

2.独占式锁的特性：

响应中断lockInterruptibly() ----acquireInterruptibly()

超时获取锁 tryLcok(long time,TimeUnit unit) - doAcquireNanos(arg,nanosTimeout)