JUC

1 java.util.concurrent

Callable

ArrayBlockingQueue

ConcurrentHashMap

CopyOnWriteArrayList

CountDownLatch

CyclicBarrier

Exchanger

Executors

FutureTask

LinkedBlockingQueue

Semaphore

ThreadPoolExecutor

2 java.util.concurrent.locks

AbstractQueuedSynchronizer

Condition

Lock

LockSupport

ReentrantLock

ReadWriteLock

ReentrantReadWriteLock

3 java.util.concurrent.atomic

AtomicInteger

AtomicReference

4 synchronized关键字（底层也是cas+park实现的）

1 用户态和内核态

2 cas

cas会引起aba问题，解决方案:AtomicStampedReference

cas怎么解决原子性问题：lock

3 对象的内存布局

4 锁升级

5 volatile关键字

6 aqs

基于chl队列的state共享变量的锁模式，通过模版方法模式实现

AQS实现的排它锁：ReentrantLock

AQS实现的共享锁: ReadWriteLock\Semaphore\CountDownLatch\CyclicBarrier

```
* AtomicInteger
* 底层是CAS+volatile实现的
*
* CAS的缺点：
* 1 ABA问题，改进方式    加版本号   具体实现AtomicStampedReference
* 2 自旋时间过长导致消耗CPU资源,自适应自旋
* 3 只能保证一个共享变量的原子操作，可以使用AtomicReference
*
*
* 自旋锁使用场景：
* 如果业务逻辑执行时间很短，没有必要执行线程的切换（需要用户态->核心态转变），如果采用线程自旋的形式，会减少获取锁的成本。
*
* 如果竞争很激烈或者业务逻辑执行时间很长，采用自旋的形式不合适
* （参考sync锁升级过程,如果自旋的线程数=CPU核数/2，或者自旋次数超过10次就会升级为重量级锁）
*
* 注意此处AtomicInteger变量并没有使用volatile关键字
*
*
* 此场景并没有产生aba问题，应为每个线程都是执行+1操作
```

```
* 1.7 HashMap Entry ConcurrentHashMap Segment HashEntry
* 1.8 HashMap Node  ConcurrentHashMap Node
* AQS Node
*
* synchronized 和 Lock区别
* 1 存在层面   sync是Java关键字，是JVM底层实现的（c++ monitorObject对象） Lock是Java实现的，是JUC包中的一个接口
* 2 锁的状态   sync无法判断是否获得锁成功，Lock可以判断是否获取锁成功（isHeldByCurrentThread()）
* 3 锁的获取   sync中，A线程获得锁，B线程等待。如果A阻塞，则B一直等待。在Lock中，如果A获取锁，B可能会尝试获取锁，获取不到才会等待
* 4 锁的释放   sync中，代码执行完毕或者异常之后会自动释放。LOCK中，必须在finally中释放。
* 5 锁的类型   sync中，可重入、不可中断、非公平锁   LOCK，可重入、可中断、可公平、可不公平（sync不可中断的意思是：A线程获取锁，B线程等待，线程中断不能停止B的等待）
* 6 锁的性能   sync中，适用于竞争不激烈的场景。（经历过锁的优化之后，SYNC在竞争不激烈的情况下，性能优于Lock）. LOCK，适用于竞争激烈的场景
*
* firstWaiter lastWaiter
```

```
* AQS是抽象类，但是并没有抽象方法，因为如果是抽象类，所有抽象方法必须实现，但是一般情况下只需要实现部分方法。
* AQS底层采用的是 （state+CLH+CAS）实现的
*
* AQS类主要信息如下：
*  属性信息如下：
*      private transient volatile Node head;
*      private transient volatile Node tail;
*      private volatile int state;
*      private transient Thread exclusiveOwnerThread;
*  内部类信息如下：
*      static final class Node {
*          volatile int waitStatus;
*          volatile Node prev;
*          volatile Node next;
*          volatile Thread thread;
*          Node nextWaiter;
*      }
*
*      public class ConditionObject implements Condition {
*          private transient Node firstWaiter;
*          private transient Node lastWaiter;
*          await();
*          signal();
*          signalAll();
*      }
*  需要子类实现的方法如下：
*      1 tryAcquire(int arg); 获取排它锁 成功则返回true，失败则返回false
*      2 tryRelease(int arg); 释放排它锁 成功则返回true，失败则返回false
*      3 tryAcquireShared(int arg); 获取共享锁  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
*      4 tryReleaseShared(int arg); 释放共享锁 尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false
*      5 isHeldExclusively(); 是否线程独占  只有用到condition才需要去实现它
*
* 外部调用的时候使用的方法是(这些方法都是final的方法（模板方法模式）)：
* AbstractQueuedSynchronizer.acquire(int arg);
* AbstractQueuedSynchronizer.release();
* AbstractQueuedSynchronizer.tryAcquireNanos();
*
* AbstractQueuedSynchronizer.acquireShared();
* AbstractQueuedSynchronizer.releaseShared();
* AbstractQueuedSynchronizer.tryAcquireSharedNanos();
*
*
*
*
* CANCELLED(1)：表示当前结点已取消调度。当timeout或被中断（响应中断的情况下），会触发变更为此状态，进入该状态后的结点将不会再变化。
* SIGNAL(-1)：表示后继结点在等待当前结点唤醒。后继结点入队时，会将前继结点的状态更新为SIGNAL。
* CONDITION(-2)：表示结点等待在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(-3)：共享模式下，前继结点不仅会唤醒其后继结点，同时也可能会唤醒后继的后继结点。
* 0：新结点入队时的默认状态。
* 注意，负值表示结点处于有效等待状态，而正值表示结点已被取消。所以源码中很多地方用>0、<0来判断结点的状态是否正常。
*
```

```
*
* 需要说明的一点是：CLH队列中，head节点永远是一个哑巴节点，它不代表任何线程（即head节点中的thread永远是空），只有从次节点开始才代表了等待锁的线程。
* 也就是说当线程没有抢到锁被包装成Node节点扔进队列时，即使队列时空的，它也会排在第二个。
*
* AQS中的CAS操作针对5个变量，AQS类中的head,tail,state和Node类中的waitStatus,next
*
* Lock接口：
*  void lock();
*  void lockInterruptibly() throws InterruptedException;
*  boolean tryLock();
*  boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
*  void unlock();
*  Condition newCondition();
*
*  ReentrantLock implements Lock
*  主要内部类如下：
*      abstract static class Sync extends AbstractQueuedSynchronizer {
*          abstract void lock();
*      }
*      static final class NonfairSync extends Sync {
*          final void lock() {
*             if (compareAndSetState(0, 1))
*                 setExclusiveOwnerThread(Thread.currentThread());
*             else
*                 acquire(1);
*         }
*      }
*      static final class FairSync extends Sync {
*          final void lock() {
*             acquire(1);
*         }
*      }
*  主要方法如下：
*      1 构造器方法
*          public ReentrantLock() {
*              sync = new NonfairSync();
*          }
*      2 构造器方法
*          public ReentrantLock(boolean fair) {
*              sync = fair ? new FairSync() : new NonfairSync();
*          }
*      3 加锁方法
*          public void lock() {
*              sync.lock();
*          }
*      4 释放锁方法
*          public void unlock() {
*              sync.release(1);
*          }
*
* 思考AQS,ReentrantLock,Lock关系
*
* ReentrantLock是独占锁，但是有公平独占锁和非公平独占锁两种类型
* ReentrantReadWriteLock中的读锁是共享锁，也有公平共享锁和非公平共享锁两种类型
* 需要仔细体会下独占锁和共享锁    公平锁和非公平锁的关系
*
* ReadWriteLock 没有继承 Lock接口
```

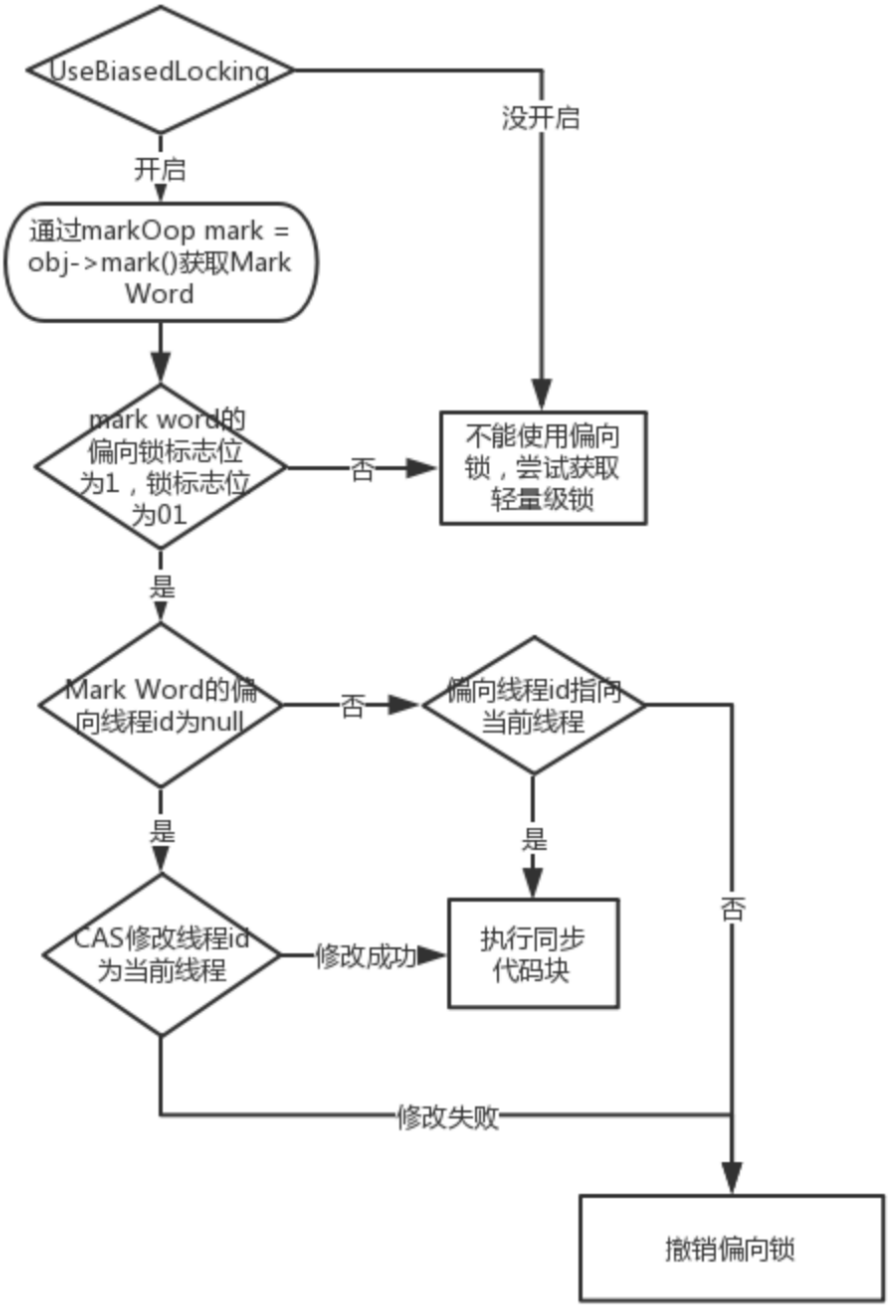
```
*  主要方法：
*       Lock readLock();
*       Lock writeLock();
*
*
* 之所以采用从后往前遍历，是因为我们处于多线程并发条件下，如果一个节点的next属性是null，并不能保证它是尾结点（
* 可能新加入的尾节点还没来得及执行prev.next=node）但是如果一个队列能够入队，则它的prev属性一定是有值的，所以反向查找是最准确的
*
* 锁的释放必须在finally中
*
*
* ReentrantReadWriteLock 实现了 ReadWriteLock
*
*
*
* 共享锁与独占锁的区别
* 1 独占锁模式下，只有独占锁的节点释放了之后，才会唤醒后续节点的线程。
* 2 共享锁模式下，当一个节点获取了共享锁，我们获取成功之后就可以唤醒后续节点线程，而不需要等待释放锁再唤醒线程。
*  共享锁可以被多个线程同时持有，一个线程获取到锁，后续节点有很大几率可以获取到锁。所以，在获取锁和释放锁的时候都会唤醒后续节点的线程。
*
* Semaphore#tryAcquireShared方法的返回值是一个int类型(独占锁返回的是boolean类型-代表获取锁成功或者失败)
* 0 代表获取锁成功，但是后续获取锁会失败
* 大于0，代表获取锁成功，后续获取锁大概率会成功
* 小于0，代表获取锁失败
*
* 共享锁--在构造方法中指定了state的值（代表了可以有多少个线程同时获取锁）
*
* 在独占锁中new Node()中,nextWaiter指向Node.EXCLUSIVE=null
* 在共享锁中new Node()中，nextWaiter指向Node.SHARED=new Node()--所有的节点的nextWaiter都指向同一个SHARED对象，可以用来判定下一个NODE是不是共享锁
*
* 在独占锁中setHead()--1 把head指针指向当前节点 2 当前节点的thread=null 3 当前节点的prev=null 4 元head节点的next=null(为了GC)
* 在共享锁中setHeadAndPropagate() --1 setHead()(包含了以上所有动作) 2 如果state还有剩余锁&&下一个节点是共享节点，调用releaseShared()方法
*
* 释放锁的逻辑
*
```

```
* 主要介绍java.util.concurrent.locks包下的类
*      AbstractQueuedSynchronizer
*
*     Condition
*
*     Lock
*          lock()  --一直等待锁
*          tryLock() -- 尝试获取锁，如果获取到返回true，否则返回false
*          tryLock(long, TimeUnit) -- 尝试在L时间内获取锁，如果获取到返回true，否则返回false
*          unlock() -- 解锁，必须放在finally中，一次lock对应一次unlock
```

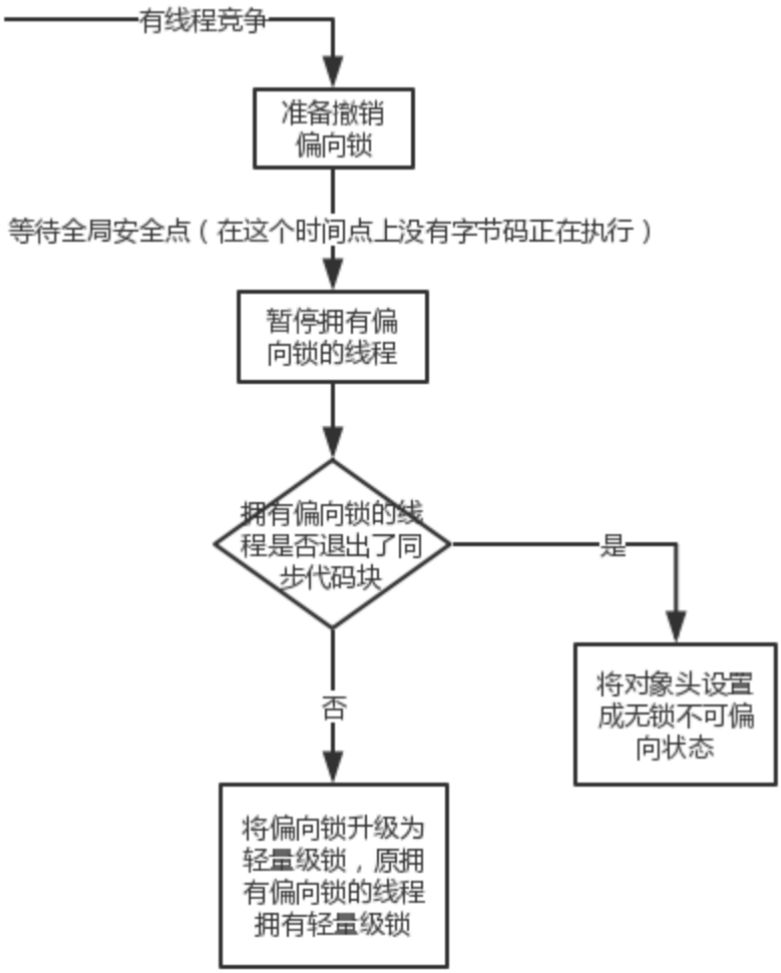
```
*          newCondition() -- 返回一个Condition对象(synchronized与Object.wait\notify
组合使用) LOCK与Condition.await\notify组合使用
*
*      LockSupport
*
*      ReentrantLock
*
*      ReadWriteLock
*
*      ReentrantReadWriteLock
```

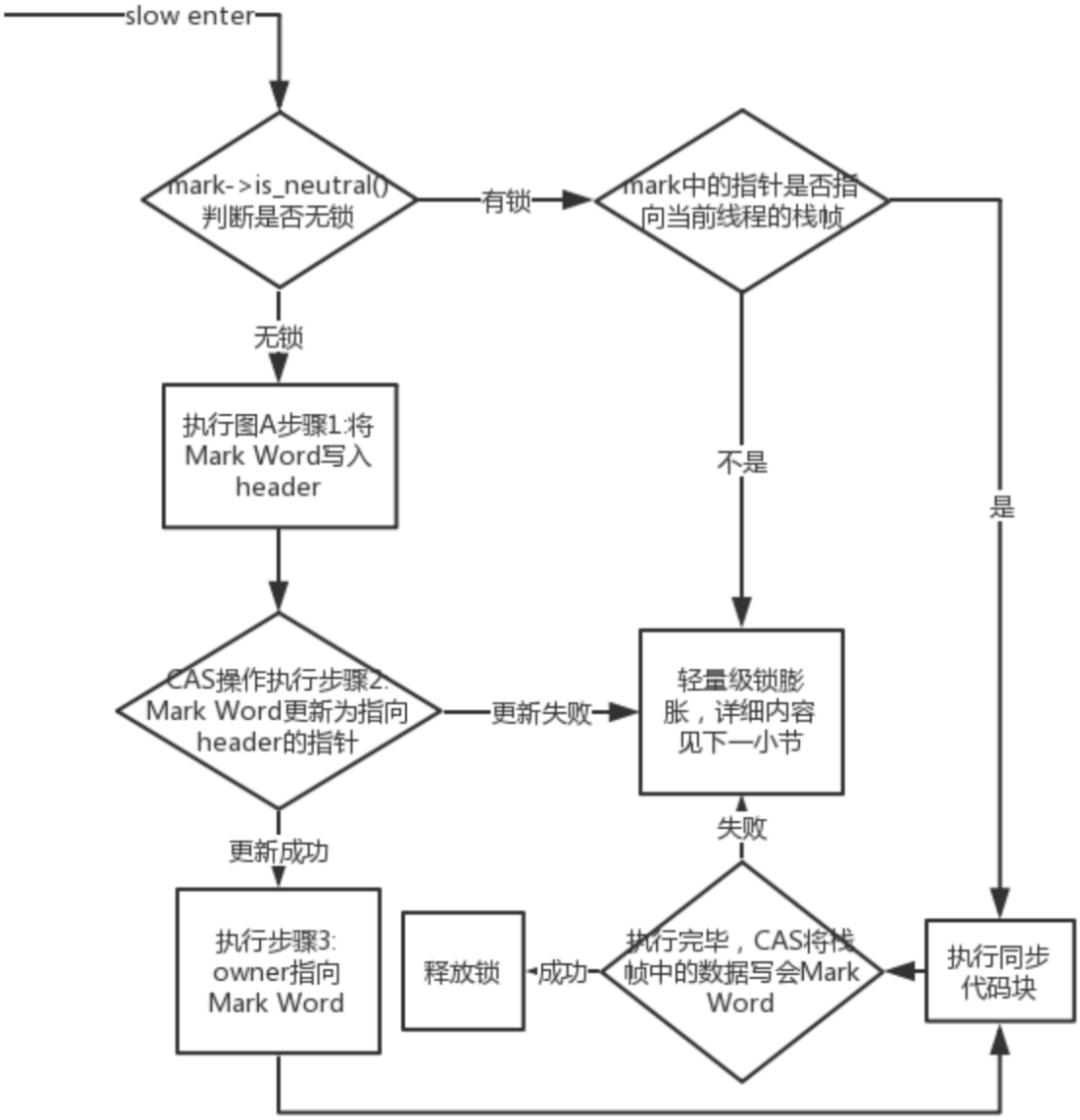
```
/**
 * synchronized关键字
 *
 * 重量级锁ObjectMonitor
 * （1）owner，指向持有ObjectMonitor的线程；
 * （2）WaitSet，wait状态的线程队列，等待被唤醒，也就是调用了wait；
 * （3）EntrySet，等待锁的线程队列，
 * （4）Recursions,重入次数
 *
 * 同步流程
 * （1）有两个线程，线程A、线程B将竞争锁访问同步代码块，先进入ObjectMonitor的EntrySet中等待
锁；
 * （2）当CPU调度线程A获取到锁则进入同步代码，ObjectMonitor owner属性指向线程A，线程B继续在
EntryList中等待；
 * （3）线程A在同步代码中执行wait，则线程进入WaitSet并释放锁，ObjectMonitor owner属性清
空；
 * （4）CPU调度使线程B获取到锁进入同步代码块，ObjectMonitor owner属性指向线程B，任务执行完
退出同步代码之前调用notifyAll，线程A被唤醒，从WaitSet转到EntryList中等待锁，线程B退出同步代
码块，ObjectMonitor owner属性清空；
 * （5）CPU调度使线程A获取同步锁，继续后续代码；
 *
 * 每一个JAVA对象都和一个ObjectMonitor对象相关联，关联关系存储在对象头中
 * 每一个试图进入代码块的线程都会被封装成ObjectWaiter对象，他们或者在EntryList中或者在
WaitSet中等待称为ObjectMonitor的owner
 *
 * synchronized 和 Lock区别
 * 1 存在层面  sync是Java关键字，是JVM底层实现的（c++ monitorObject对象） Lock是Java实现
的，是JUC包中的一个接口
 * 2 锁的状态  sync无法判断是否获得锁成功，Lock的排他锁，可以判断是否获取锁成功
（isHeldByCurrentThread()）
 * 3 锁的获取  sync中，A线程获得锁，B线程等待。如果A阻塞，则B一直等待。在Lock中，如果A获取
锁，B可能会尝试获取锁，获取不到才会等待
 * 4 锁的释放  sync中，代码执行完毕或者异常之后会自动释放。LOCK中，必须在finally中释放。
 * 5 锁的类型  sync中，可重入、不可中断、非公平锁  LOCK，可重入、可中断、可公平、可不公平
（sync不可中断的意思是：A线程获取锁，B线程等待，线程中断不能停止B的等待）
 * 6 锁的性能  sync中，适用于竞争不激烈的场景。（经历过锁的优化之后，SYNC在竞争不激烈的情况
下，性能优于Lock）. LOCK，适用于竞争激烈的场景
 *
 * 对象的内存布局：
 *  64位操作系统中：
 *      对象头
 *          8个字节的MarkWord（jvm启动时，采用了延迟开启偏向锁的策略，因为在jvm加载类时，明
确知道有锁的竞争.）
```

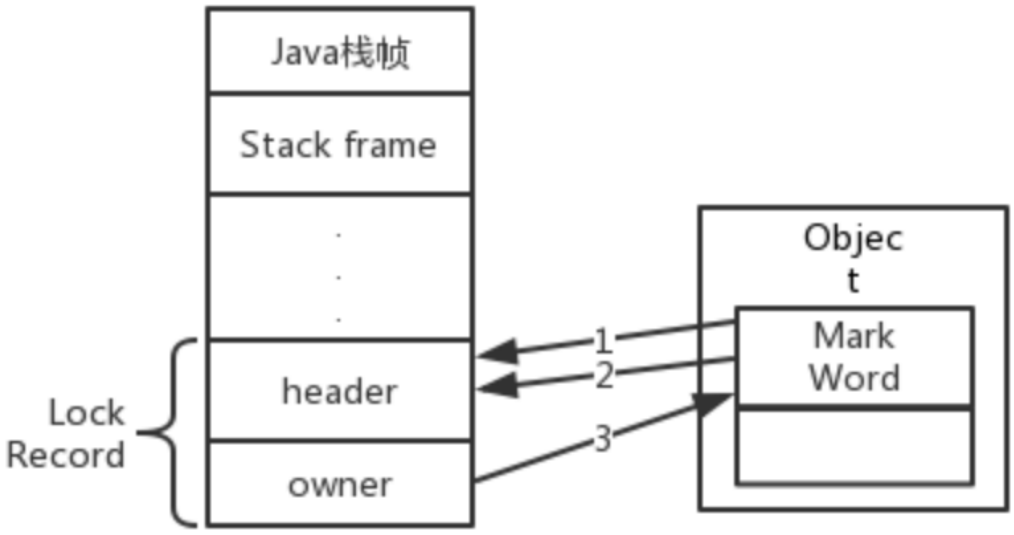
```
 *                  无锁              hashcode（跟实际内存位置有关） 4位的年龄信息 所以最大是16  1位的0 表明无偏向  01表明是无锁(其实001才是表明无锁)
 *                  偏向锁            ThreadId(偏向的线程ID)         4位的年龄信息 所以最大是16  1位的1 表明偏向锁  01表明是偏向锁（采用cas实现,UseBiasedLocking是否开启偏向锁）
 *                      如果UseBiasedLocking=false，则直接获取轻量级锁
 *                          如果ubl=true,则检查偏向锁标志是否是1，锁标志是否是01，如果不是该是什么锁，就尝试获取什么锁
 *                              如果是101，查看threadId是否是空，
 *                                  如果是空，尝试cas更改ThreadId
 *                                  如果非空，比较是否是当前线程
 *                                      如果是当前线程，获取锁成功。
 *                                      如果不是，则进行偏向锁的撤销。
 *                                          偏向锁撤销，等待全局安全点时，查看拥有锁的线程是否退出了同步代码块
 *                                              如果退出了，则设置成001
 *                                              如果没有退出，则升级为轻量级锁
 *                  轻量级锁        指向线程栈中的LockRecord的指针
                  00表明是轻量级锁（采用cas实现）
 *                      LockRecord中有header和owner两个结构
 *                      当前对象是否有锁
 *                          如果没锁，将当前对象的MarkWord写入当前线程的栈帧的LockRecord的header中（可能有多个线程写入同样的MarkWord）
 *                          cas操作把获取到锁的线程的LockRecord指针写入MarkWord中（可能有多个线程写入，但是只有一个会成功）
 *                              如果写入成功将LockRecord中的owner指向对象的MarkWord，然后执行同步代码块，执行完同步代码块，采用cas操作把记录的原来的MarkWord值写回，然后释放锁。
 *                  重量级锁        指向互斥量（重量级锁）的指针
                10重量级锁
 *                      有两个队列，同步队列和等待队列，这个是需要用户态和内核态的转变的，所以比较重。
 *                          先判定owner是否为空
 *                              如果为空，则设置owner=当前线程,recursions=1,进入同步代码块
 *                              如果非空，查看owner是否是当前线程
 *                                  如果是,recursions++,进入同步代码块
 *                                  如果不是,自旋的方式等待，自旋失败，通过cas进入entryset中
 *          4个字节的KlassPointer（默认开启了指针压缩）
 *          如果是数据，还有4个字节的数组长度
 *      实例数据
 *          int 4个字节
 *          long 8个字节
 *          reference 4个字节（默认开启了指针压缩）
 *      对其填充
 *          对象的大小必须是8字节的倍数
 *
 *
 * 重入:
 *  对于不同级别的锁都有重入策略，偏向锁:单线程独占，重入只用检查threadId等于该线程;
 *  轻量级锁：重入将栈帧中lock record的header设置为null，重入退出，只用弹出栈帧，直到最后一个重入退出CAS写回数据释放锁;
 *  重量级锁：重入_recursions++，重入退出_recursions--，_recursions=0时释放锁
 *
 * */
```


UseBiasedLocking
没开启
开启
通过markOop mark = obj->mark()获取Mark Word
mark word的偏向锁标志位为1，锁标志位为01
否
不能使用偏向锁，尝试获取轻量级锁
是
Mark Word的偏向线程id为null
否
偏向线程id指向当前线程
是
是
否
CAS修改线程id为当前线程
修改成功
执行同步代码块
修改失败
撤销偏向锁

偏向锁的撤销过程如下：


有线程竞争
准备撤销偏向锁
等待全局安全点（在这个时间点上没有字节码正在执行）
暂停拥有偏向锁的线程
拥有偏向锁的线程是否退出了同步代码块
是
将对象头设置成无锁不可偏向状态
否
将偏向锁升级为轻量级锁，原拥有偏向锁的线程拥有轻量级锁


slow enter
mark->is_neutral()
判断是否无锁
有锁
mark中的指针是否指向当前线程的栈帧
无锁
执行图A步骤1:将Mark Word写入header
不是
是
CAS操作执行步骤2:Mark Word更新为指向header的指针
更新失败
轻量级锁膨胀，详细内容见下一小节
失败
更新成功
执行步骤3:owner指向Mark Word
释放锁
成功
执行完毕，CAS将栈帧中的数据写会Mark Word
执行同步代码块


Java栈帧
Stack frame
header
owner
Lock Record
Objec t
Mark Word
1
2
3

图A

```
* ArrayBlockingQueue
* LinkedBlockingDeque
*      Executors.newFixedThreadPool()和Executors.newSingleThreadExecutor()底层均使用此阻塞队列,会造成内存溢出的情况
* SynchronousQueue
*      Executors.newCachedThreadPool()使用此阻塞队列，会大量创建线程
*
*      add(E e)  remove() 会报错
*      offer(E e)  poll()  返回
*      put(E e)   take()   会阻塞
```

```
/**
 * volatile关键字，轻量级的线程同步工具，
 * jmm(java memory model) java内存模型(可以画出内存模型)，规定需要满足三个条件
 * 1 可见性
 * 2 原子性
 * 3 有序性
 * 缓存行（64个字节）
 *
 *
 * volatile满足两种，可见性和有序性，但是不满足原子性(比较经典的是i++问题)
 * sync和lock也可以满足可见性，是释放锁之前会把变量刷回主存中
 * 重排序：指令重排和编译重排
 * 有序性：一个变量是boolean一个变量是int的两个线程问题（ais-as if serial happens-before）
 * volatile写happens-before在volatile读-实现了可见性
 * volatile通过memory barrier实现有序性
 *
 *      lock作用于主内存的变量，把一个变量标记为一个线程独占
 *      unlock
 *
 * •    主内存read>工作内存load>变量>use
 *
 * •    assign>变量store>工作内存write>主内存
 *
 * •    Object obj = new Object()底层字节码文件
 *
 * •    new
 *
 * •    dump
 *
 * •    invokespecial
 *
 * •    astore
 *
 * •    return
 *
 *
 *
 * 内存屏障和读写屏障
 *
 * volatile关键字的底层实现是：汇编代码会生成一个lock add 0指令
 * */
```

Condition接口的主要实现类是AQS的内部类`ConditionObject`，**每个Condition对象都包含一个等待队列。**该队列是Condition对象实现等待/通知的关键。AQS中同步队列与等待队列的关系如下：

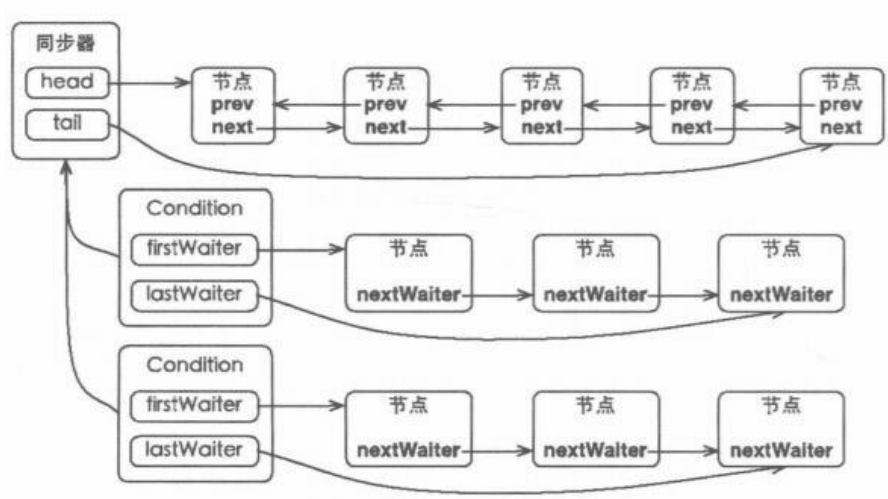


图 5-10　同步队列与等待队列

在Object的监视器模型上，一个对象拥有一个同步队列与一个等待队列，而AQS拥有一个同步队列和多个等待队列。

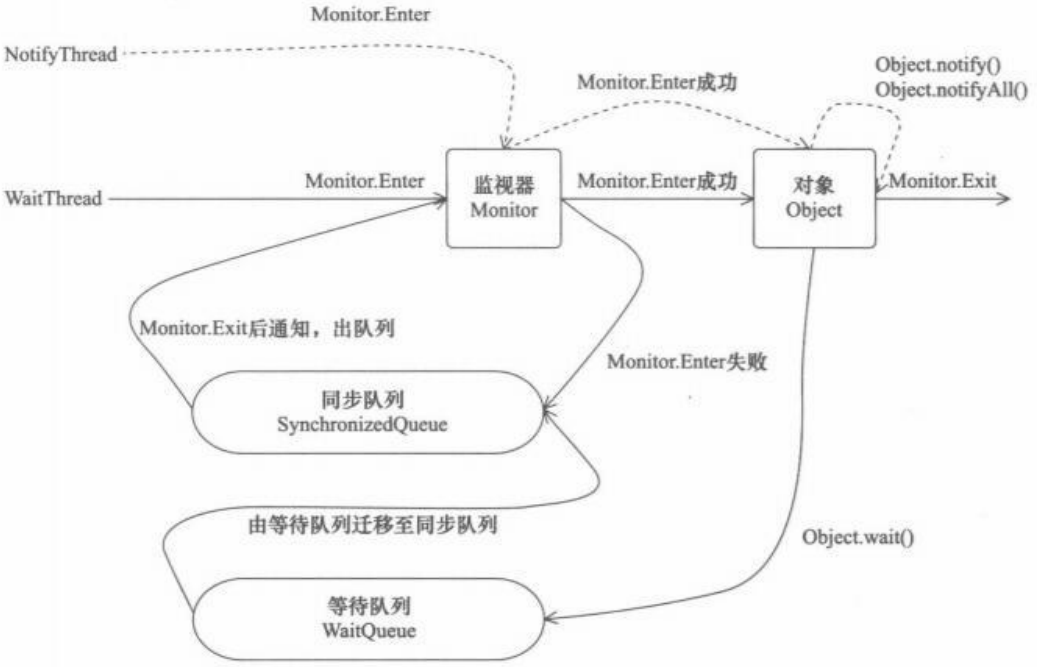


图 4-3　WaitNotify.java 运行过程

调用condition的await方法，将会使当前线程进入等待队列并释放锁(先加入等待队列再释放锁)，同时线程状态转为等待状态。

从同步队列和阻塞队列的角度看，调用await方法时，相当于**同步队列的首节点移到condition的等待队列中**

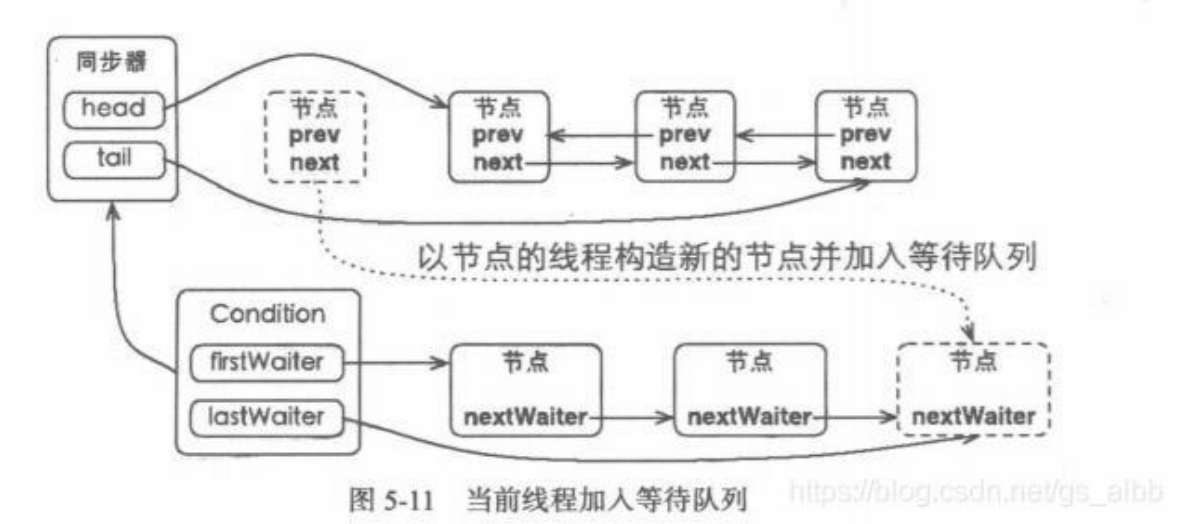


图 5-11 当前线程加入等待队列

调用condition的signal方法时，将会把等待队列的首节点移到等待队列的尾部，然后唤醒该节点。 被唤醒，并不代表就会从await方法返回，也不代表该节点的线程能获取到锁，它一样需要加入到锁的竞争acquireQueued方法中去，只有成功竞争到锁，才能从await方法返回。

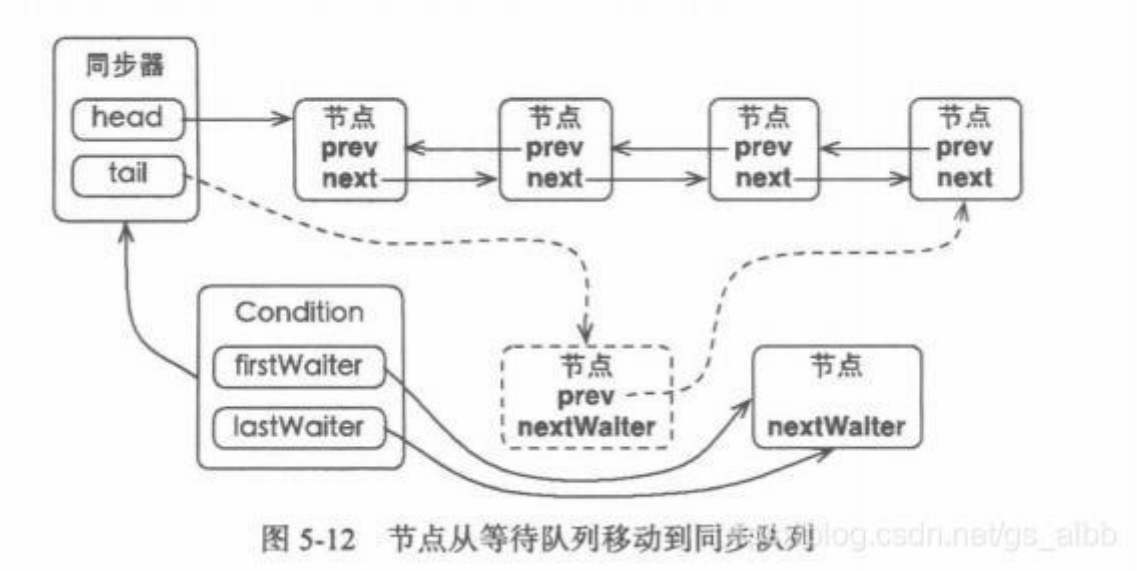
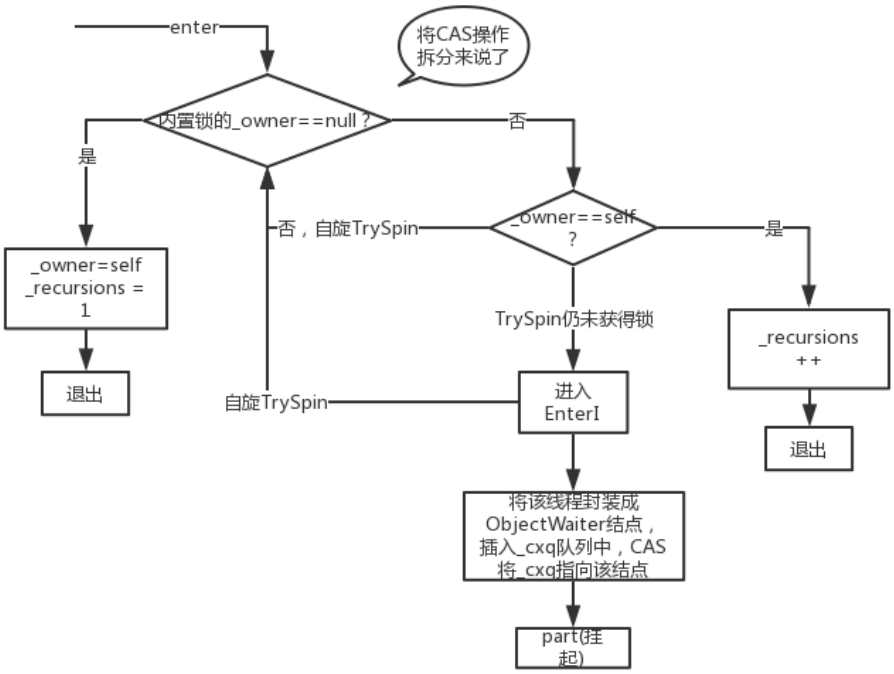
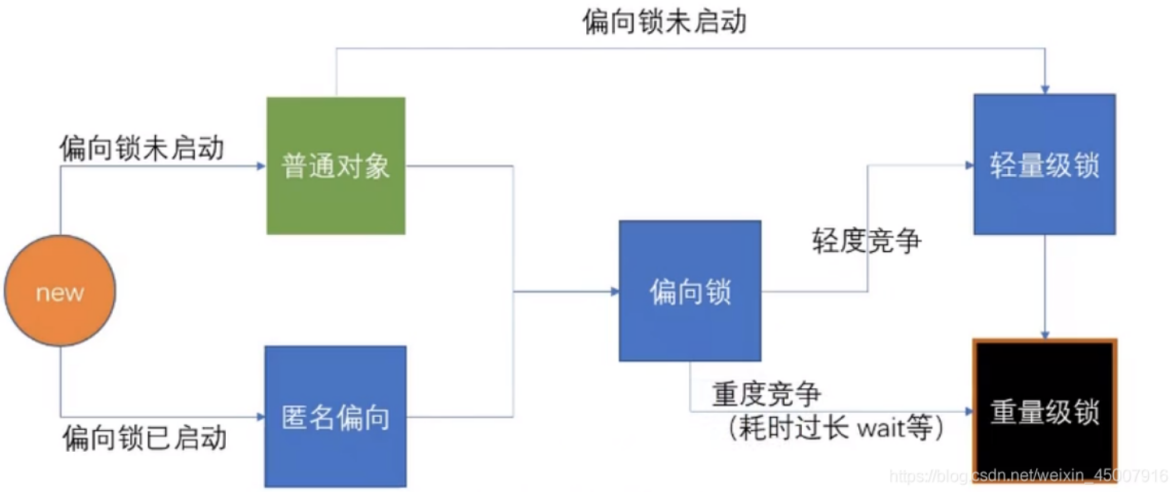


图 5-12 节点从等待队列移动到同步队列

SYNC重量级锁的获取过程





Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
无锁态（new）	unused	hashCode（如果有调用）	unused	分代年龄	0	0	1

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位	
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0	1

锁状态	62位	2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0	0
重量级锁	指向互斥量（重量级锁）的指针	1	0
GC标记信息	CMS过程用到的标记信息	1	1

讲解SYNC

https://blog.csdn.net/zwjyyy1203/article/details/106217887

讲解aqs

https://segmentfault.com/a/1190000016058789

AQS非公平锁底层原理代码分析：

```
调用端代码样例
Lock lock = new ReentrantLock();
lock.lock();
try {
    //do buss
} finally {
    lock.unlock();
}
```

```
//构造方法中指定了是非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}
```

```
//lock.lock调用非公平锁的lock方法
final void lock() {
    //并没有检查当前队列是否有排队情况，直接尝试获取锁，此处显示出非公平锁,此处并没有采用重试，失败之后尝试acquire入队列
    if (compareAndSetState(0, 1))
        //如果CAS把state变成1，表示获取锁成功，把当前线程设置成独占线程
        setExclusiveOwnerThread(Thread.currentThread());
    else
        //调用AQS的acquire方法
        acquire(1);
}
```

```
//获取锁的逻辑-主要包含三个方法
//tryAcquire-aqs子类实现的方法
//addWaiter-把当前线程封装成Node对象放入队列,注意此处的Node类型是Node.EXCLUSIVE
//acquireQueued-更新前节点的状态
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        //如果是被别的线程中断唤醒，获取到锁之后，会自己打断
        selfInterrupt();
}
```

```
//非公平锁中tryAcquire方法的具体实现
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
```

```
    int c = getState();
    if (c == 0) {
        //如果当前锁没有被占用，通过CAS操作尝试获取锁,此处也体现出非公平锁,此处并没有采用重试，失败之后返回false
        if (compareAndSetState(0, acquires)) {
            //设置排他线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //线程重入的情况
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //非上述两种情况，返回false，后续进入CLH队列
    return false;
}
```

```
//把Node节点放入队列
private Node addWaiter(Node mode) {
    //构建新Node
    /*Node(Thread thread, Node mode) {
     *       this.nextWaiter = mode;
     *       this.thread = thread;
     *}
     */
    Node node = new Node(Thread.currentThread(), mode);
    // 尝试快速入队
    Node pred = tail;
    if (pred != null) {
        //当前线程的节点的prev值变成尾结点，考虑如果有多个线程同时进入，会有多个线程的节点的prev值会变成尾结点
        node.prev = pred;
        //如果tail节点不是空的，即队列中有值，采用CAS操作把AQS的尾节点变成当前线程的节点，此处只会有一个线程会成功（思考尾分叉）,此处并没有采用重试，失败之后采用别的方式进入队列
        if (compareAndSetTail(pred, node)) {
            // 把上一个尾节点的next值，指向新的尾节点
            pred.next = node;
            return node;
        }
    }
    //如果快速入队失败，尝试正常入队
    enq(node);
    return node;
}
```

```
//快速入队失败之后尝试正常入队
private Node enq(final Node node) {
    //采用死循环重试的方式
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
```

```
            //如果尾节点是空，表明是第一次进入队列，则new Node()对象，此对象没有Thread属性，称为空节点。把空节点置成CLH队列的头节点
            //所以，注意CLH队列的头节点要么是NULL,要么是没有Thread属性的空节点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            //采用CAS的方式把当前节点设置成尾结点，如果失败，会一直重试,直至设置成功
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

```
// 修改Node节点状态
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        //采用死循环的方式一直重试
        for (;;) {
            //获取当前节点的前一个节点
            final Node p = node.predecessor();
            //如果当前节点的前一个节点是头节点，则调用上面的tryAcquire方法，继续尝试获取锁
            if (p == head && tryAcquire(arg)) {
                //如果获取锁成功，则把当前节点设置成头节点，并把当前节点的thread属性置null,当前节点的prev属性置null
                setHead(node);
                //把当前节点的前节点的next属性变成null，为了方便GC回收
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            //获取锁失败的处理
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

```
//设置当前节点为头节点，并把当前节点的thread属性置null,当前节点的prev属性置null
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

```
// 如果获取锁失败，进入park状态
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
```

```
    int ws = pred.waitStatus;
    //查看当前节点的前一个节点，如果waitStatus == -1，直接返回
    if (ws == Node.SIGNAL)
        return true;
    //查看当前节点的前一个节点，如果waitStatus == 1，则一直往前找，直到找到waitStatus <= 0
的节点作为它的前节点
    if (ws > 0) {
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        //如果waitStatus <= 0，独占锁只有waitStatus == 0的场景，则通过CAS把前节点的
waitStatus变成-1
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

```
//如果设置了前节点waitStatus = -1之后，当前节点的线程park
private final boolean parkAndCheckInterrupt() {
    //未获取到锁的线程会一直park在这个位置，直到被上个节点的线程唤醒或者被别的线程中断
    LockSupport.park(this);
    //如果被唤醒，则返回是否是中断被唤醒的标识，如果返回true，表明是被别的线程中断，如果返回
false，表明是被前节点唤醒
    return Thread.interrupted();//此方法会消除中断标识
}
```

```
//如果是被别的线程中断之后获取到锁，并不是从前节点唤醒的，则线程自己中断
static void selfInterrupt() {
    Thread.currentThread().interrupt();
}
```

AQS公平锁底层原理代码分析：

```
//调用端代码
Lock lock = new ReentrantLock(true);
lock.lock();
try {
    //do buss
} finally {
    lock.unlock();
}
```

```
 //构造其中指定使用公平锁
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

```
//公平锁的lock实现
//此处并没有采用CAS直接获取锁
final void lock() {
    acquire(1);
}
```

```
//此处的获取锁的主要逻辑和非公平锁的逻辑一致,只是tryAcquire方法是公平锁的实现
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

```
//公平锁的尝试获取锁的逻辑
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        //如果state==0&&队列中没有正在等待的线程，则尝试CAS获取锁
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    //如果state!=0或者CAS获取锁失败则当前线程进入队列
    return false;
}
```

```
public final boolean hasQueuedPredecessors() {
    Node t = tail;
    Node h = head;
    Node s;
    //如果h=null t=null表明队列中没有等待的线程，则返回false，整个Node进入队列中
    //如果h非空，t非空但是t==h表明队列中没有等待的线程，则返回false，整个Node进入队列中
    //如果h非空 t是空 s = null表明  整个队里刚开始是空的，第一个线程进入队列时，刚把head空节点设置好，并未设置tail节点
    //如果h非空 t非空，s != null表明 整个队列中至少有一个值，如果s的线程是当前线程，则返回false，s是第一个等待的线程，尝试获取锁
    //如果h非空 t非空 s != null 但是s的线程不是当前线程，返回true，尝试进入队列
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

公平锁和非公平锁的解锁流程都是一样的

```
//此处的release方法调用的是AQS中的方法，所以，公平锁和非公平锁都是一个流程
public void unlock() {
    sync.release(1);
}
```

```
//释放锁
public final boolean release(int arg) {
    //调用AQS实现类的中的方法，此处调用的是ReentrantLock内部类Sync中的方法,公平锁和非公平锁都一样
    if (tryRelease(arg)) {
        Node h = head;
        //释放锁成功之后，如果头节点不是空&&头节点状态!=0，表明后续由需要头节点唤醒的线程
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

```
//尝试释放锁
protected final boolean tryRelease(int releases) {
    //state值，每释放一次就减1
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    //如果state=0，表明该线程不再使用锁，需要把exclusiveOwnerThread变成null
    //如果state!=0,表明该线程还需要继续使用锁，但是只是释放了一次而已
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

```
//如果上个线程完全释放锁，即state=0&&CLH队列中头节点非空&&头结点的waitStatus!=0则唤醒CLH队列中的下一个线程
private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    //如果头节点状态<0,则采用CAS方法把statue变成0
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    Node s = node.next;
    //如果头节点的下一个节点为空或者waitStatus >0,则从尾节点开始往前遍历，找出waitStatus<=0的最前面的那一个节点。
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    //如果找到了该节点，则唤醒该节点上的线程
    if (s != null)
        LockSupport.unpark(s.thread);
```

```
}
```

共享锁的获取源码分析

```
//使用端代码逻辑
Semaphore semaphore = new Semaphore(5);
semaphore.acquire();
semaphore.release();
```

```
//锁的构造方法
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

//非公平锁的实现
NonfairSync(int permits) {
    super(permits);
}

//Semaphore内部类Sync的构造方法
Sync(int permits) {
    setState(permits);
}

//设置AQS的state属性的值
protected final void setState(int newState) {
    state = newState;
}
```

```
//调用获取共享锁的方法
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
```

```
//获取可打断的共享锁
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //如果>=0表明获取锁成功，不需要等待队列，如果方法返回值<0表示线程获取锁失败，则需要进入CLH
队列等待锁
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
```

```
//Semaphore内部NonfairSync非公平锁的实现
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}
```

```
//Semaphore内部Sync的方法
final int nonfairTryAcquireShared(int acquires) {
    //此处是一个死循环
    for (;;) {
        //获取当前的state值
        int available = getState();
        int remaining = available - acquires;
        //如果剩余的state的值<0,则直接返回负值
        //如果大于或者等于0，尝试CAS更新状态--此处会一直重试，直至成功或者直接返回负值
        //返回负值就会进入CLH队列
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

```
//线程进入队列&更新waitStatus状态，对比独占锁的acquireQueued
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    //此处的逻辑跟独占锁的一直，同一个方法
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        //此处是一个死循环，会保证进入队列一定成功
        for (;;) {
            final Node p = node.predecessor();
            //如果当前节点的前一个节点是head节点
            if (p == head) {
                //再次尝试获取锁
                int r = tryAcquireShared(arg);
                //如果获取锁成功
                if (r >= 0) {
                    //设置头节点
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

```
//设置头节点&释放共享锁
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // Record old head for check below
    //设置头节点
    setHead(node);
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next;
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

```
//设置头节点，并把thread值为空，并且prev的值为空
private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}
```

共享锁的释放源码分析

```
//释放共享锁逻辑
public void release() {
    sync.releaseShared(1);
}
```

```
//AQS中的释放共享锁逻辑
public final boolean releaseShared(int arg) {
    //尝试释放锁，释放成功返回true，否则返回false
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

```
//Semaphore中Sync中的方法，采用CAS+自旋的方式，一直尝试释放锁，直至成功，每次释放state值就+1
protected final boolean tryReleaseShared(int releases) {
    //死循环
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            return true;
    }
}
```

```
//AQS中的方法,同样是采用CAS+自旋的方式
private void doReleaseShared() {
    //死循环
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                //如果头节点状态是SIGNAL,则使用CAS把头节点状态变成0，此处只有一个线程可以完成此操作
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;            // loop to recheck cases
                //完成变更的线程，可以唤醒后续的线程，没有变更成功的线程则继续循环
                unparkSuccessor(h);
            }
            //PROPAGATE为了解决线程无法唤醒的BUG
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        //如果头结点没有发生过变化，则可以退出循环，这个是循环退出的唯一方式，否自会一直继续，形成调用风暴，增加后续节点唤醒的速度
        if (h == head)                   // loop if head changed
            break;
    }
}
```

Condition接口源码解析

```
//Condition是个接口
//ConditionObject是AQS的内部类,ConditionObject实现了Condition接口
//ConditionObject中有firstWaiter\lastWaiter两个属性
//condition必须和lock配合使用
//Lock接口有个方法newCondition，返回Condition对象
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();
condition.await();
condition.signal();
condition.signalAll();
```

```
//调用的是ReentrantLock内部类Sync中的newCondition方法
public Condition newCondition() {
    return sync.newCondition();
}
```

```
//在该方法中直接new ConditionObject
final ConditionObject newCondition() {
    return new ConditionObject();
}
```

await方法源码解析

```
//AQS中ConditionObject内部类中的方法
//能调用此方法的必定是持有锁的线程--此处的锁必须是实现了Lock的锁
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    //添加一个新的Node节点到等待队列中(注意，此处并不是把同步队列中的Node节点加入等待队列，而是创建了一个新的Node)
    Node node = addConditionWaiter();
    //释放锁，此处的释放锁，会释放该线程所有的锁，（即，如果一个线程加了两次锁，state=2,则此处直接state=0）并返回持有的锁的个数
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    //查看当前节点是否在同步队列中
    while (!isOnSyncQueue(node)) {
        //当前线程挂起
        LockSupport.park(this);
        //----------------------------------------------------
        //以下是线程被唤醒之后的逻辑--包括了被中断和被signal两种唤醒方式
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

```
//新建Node节点并加入到等待队列中
private Node addConditionWaiter() {
    Node t = lastWaiter;
    //如果Node的waitStatus非CONDITION，则从firstWaiter开始，往后遍历找出最后一个状态是CONDITION节点作为尾部节点
    //等待队列中的状态有三种,CONDITION(初始状态),0,和CANCELLED
    //同步队列中的状态有三种,0(初始状态),SIGNAL和取消，以及PROPAGATE
    if (t != null && t.waitStatus != Node.CONDITION) {
        //从前往后遍历找出最后一个状态是CONDITION节点作为尾部节点
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    //创建一个新的节点（注意，节点状态是CONDITION）
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    //如果头节点是空，则当前节点为头节点
    if (t == null)
        firstWaiter = node;
    //否则，把尾节点的nextWaiter指向当前节点
    else
        t.nextWaiter = node;
    //尾节点指向当前节点
    lastWaiter = node;
    return node;
}
```

```
//释放线程拥有的所有锁,即state=0,此处返回值表明的是该线程持有的锁的个数（重入的个数）
final int fullyRelease(Node node) {
    boolean failed = true;
    try {
        int savedState = getState();
        //排他锁释放线程方法一样
        if (release(savedState)) {
            failed = false;
            return savedState;
        } else {
            throw new IllegalMonitorStateException();
        }
    } finally {
        if (failed)
            node.waitStatus = Node.CANCELLED;
    }
}
```

```
//判定Node是否在同步队列中
final boolean isOnSyncQueue(Node node) {
    //在等待队列中
    if (node.waitStatus == Node.CONDITION || node.prev == null)
        return false;
    //在同步队列中
    if (node.next != null) // If has successor, it must be on queue
        return true;
    //从尾节点开始查询Node是否在同步队列中
    return findNodeFromTail(node);
}
```

```
//从尾节点开始查询Node是否在同步队列中
private boolean findNodeFromTail(Node node) {
    Node t = tail;
    for (;;) {
        if (t == node)
            return true;
        if (t == null)
            return false;
        t = t.prev;
    }
}
```

signal方法源码解析

```
//AQS中ConditionObject内部类中的方法
//能调用此方法的必定是持有锁的线程--此处的锁必须是实现了Lock的锁
public final void signal() {
    //检查当前线程是否持有锁
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        //通知第一个waiter
        doSignal(first);
}
```

```
//通知第一个waiter
private void doSignal(Node first) {
    do {
        //如果头节点的下一个节点是空，则把等待队列置空
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}
```

```
final boolean transferForSignal(Node node) {
    /*
     * If cannot change waitStatus, the node has been cancelled.
     */
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
        return false;

    /*
     * 加入到同步队列中，同获取锁中的enq方法是同一个方法,返回值是当前节点的前一个节点
     */
    Node p = enq(node);
    int ws = p.waitStatus;
    //如果前一个节点取消或者更改状态失败，则唤醒首节点
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
        LockSupport.unpark(node.thread);
    //返回true，则退出循环，表明唤醒节点
    return true;
}
```

await方法被唤醒之后的逻辑

```
private int checkInterruptWhileWaiting(Node node) {
    return Thread.interrupted() ?
        (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
        0;
}
```

```
final boolean transferAfterCancelledWait(Node node) {
    if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
        enq(node);
```

```
            return true;
        }
        /*
         * If we lost out to a signal(), then we can't proceed
         * until it finishes its enq().  Cancelling during an
         * incomplete transfer is both rare and transient, so just
         * spin.
         */
        while (!isOnSyncQueue(node))
            Thread.yield();
        return false;
    }
```

```
private void reportInterruptAfterWait(int interruptMode)
    throws InterruptedException {
    if (interruptMode == THROW_IE)
        throw new InterruptedException();
    else if (interruptMode == REINTERRUPT)
        selfInterrupt();
}
```