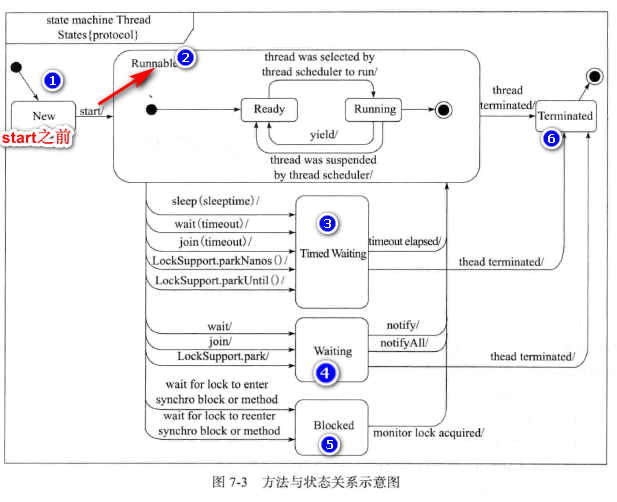
# 线程状态

线程有如下6种状态，由java.lang.State枚举给出



# [Unsafe与CAS](https://www.cnblogs.com/xrq730/p/4976007.html)

Unsafe简单讲一下这个类。Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的原子操作。

这个类尽管里面的方法都是public的，但是并没有办法使用它们，JDK API文档也没有提供任何关于这个类的方法的解释。总而言之，对于Unsafe类的使用都是受限制的，只有授信的代码才能获得该类的实例，当然JDK库里面的类是可以随意使用的。

从第一行的描述可以了解到Unsafe提供了硬件级别的操作，比如说获取某个属性在内存中的位置，比如说修改对象的字段值，即使它是私有的。不过Java本身就是为了屏蔽底层的差异，对于一般的开发而言也很少会有这样的需求。

举两个例子，比方说：

public native long staticFieldOffset(Field paramField);

这个方法可以用来获取给定的paramField的内存地址偏移量，这个值对于给定的field是唯一的且是固定不变的。再比如说：

public native int arrayBaseOffset(Class paramClass);

public native int arrayIndexScale(Class paramClass);

前一个方法是用来获取数组第一个元素的偏移地址，后一个方法是用来获取数组的转换因子即数组中元素的增量地址的。最后看三个方法：

public native long allocateMemory(long paramLong);

public native long reallocateMemory(long paramLong1, long paramLong2);

public native void freeMemory(long paramLong);

分别用来分配内存，扩充内存和释放内存的。

**CAS**

CAS，Compare and Swap即比较并交换，设计并发算法时常用到的一种技术，java.util.concurrent包全完建立在CAS之上，没有CAS也就没有此包，可见CAS的重要性。

当前的处理器基本都支持CAS，只不过不同的厂家的实现不一样罢了。CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。这是从硬件上保证的。

**CAS软件层面的 Unsafe实现的如下**

public final native boolean compareAndSwapObject(Object paramObject1, long paramLong, Object paramObject2, Object paramObject3);

public final native boolean compareAndSwapInt(Object paramObject, long paramLong, int paramInt1, int paramInt2);

public final native boolean compareAndSwapLong(Object paramObject, long paramLong1, long paramLong2, long paramLong3);

java.util.concurrent.atomic包下的原子操作类都是基于CAS实现的，下面拿AtomicInteger分析一下，首先是AtomicInteger类变量的定义：

[复制代码](javascript:void(0);)

1 private static final Unsafe unsafe = Unsafe.getUnsafe();

2 private static final long valueOffset;

3

4 static {

5 try {

6 valueOffset = unsafe.objectFieldOffset

7 (AtomicInteger.class.getDeclaredField("value"));

8 } catch (Exception ex) { throw new Error(ex); }

9 }

10

11 private volatile int value;

[复制代码](javascript:void(0);)

关于这段代码中出现的几个成员属性：

1、Unsafe是CAS的核心类，前面已经讲过了

2、valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的原值的

3、value是用volatile修饰的，这是非常关键的

下面找一个方法getAndIncrement来研究一下AtomicInteger是如何实现的，比如我们常用的addAndGet方法：

[复制代码](javascript:void(0);)

1 public final int addAndGet(int delta) {

2 for (;;) {

3 int current = get();

4 int next = current + delta;

5 if (compareAndSet(current, next))

6 return next;

7 }

8 }

[复制代码](javascript:void(0);)

1 public final int get() {

2 return value;

3 }

这段代码如何在不加锁的情况下通过CAS实现线程安全，我们不妨考虑一下方法的执行：

1、AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程1和线程2各自持有一份value的副本，值为3

2、线程1运行到第三行获取到当前的value为3，线程切换

3、线程2开始运行，获取到value为3，利用CAS对比内存中的值也为3，比较成功，修改内存，此时内存中的value改变比方说是4，线程切换

4、线程1恢复运行，利用CAS比较发现自己的value为3，内存中的value为4，得到一个重要的结论-->此时value正在被另外一个线程修改，所以我不能去修改它

5、线程1的compareAndSet失败，循环判断，因为value是volatile修饰的，所以它具备可见性的特性，线程2对于value的改变能被线程1看到，只要线程1发现当前获取的value是4，内存中的value也是4，说明线程2对于value的修改已经完毕并且线程1可以尝试去修改它

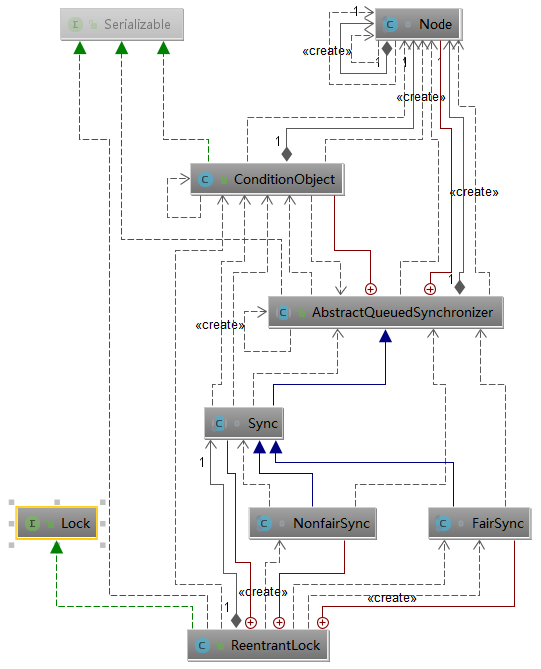
6、最后说一点，比如说此时线程3也准备修改value了，没关系，因为比较-交换是一个原子操作不可被打断，线程3修改了value，线程1进行compareAndSet的时候必然返回的false，这样线程1会继续循环去获取最新的value并进行compareAndSet，直至获取的value和内存中的value一致为止

整个过程中，利用CAS机制保证了对于value的修改的线程安全性。

**CAS的缺点**

CAS看起来很美，但这种操作显然无法涵盖并发下的所有场景，并且CAS从语义上来说也不是完美的，存在这样一个逻辑漏洞：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个漏洞称为CAS操作的"ABA"问题。java.util.concurrent包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。不过目前来说这个类比较"鸡肋"，大部分情况下ABA问题并不会影响程序并发的正确性，如果需要解决ABA问题，使用传统的互斥同步可能回避原子类更加高效。

# ReentrentLock实现细节



NonfairSync和FairSync只重载了lock()和tryAcquire()获取锁方法，其余方法由父类提供等待队列中每个线程被包装成一个 node，数据结构是链表，一起看看源码吧：

static final class Node {

// 标识节点当前在共享模式下

static final Node SHARED = new Node();

// 标识节点当前在独占模式下

static final Node EXCLUSIVE = null;

// ======== 下面的几个int常量是给waitStatus用的 ===========

// 代码此线程取消了争抢这个锁

static final int CANCELLED = 1;

//后继节点需要唤醒才能工作，否则不能工作

static final int SIGNAL = -1;

//条件队列

static final int CONDITION = -2;

//非条件增长

static final int PROPAGATE = -3;

volatile int waitStatus;

// 这个就是线程本尊

volatile Thread thread;

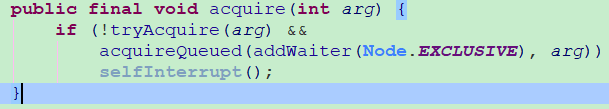
}

Node 的数据结构其实也挺简单的，就是 thread + waitStatus + pre + next 四个属性而已，大家先要有这个概念在心里。

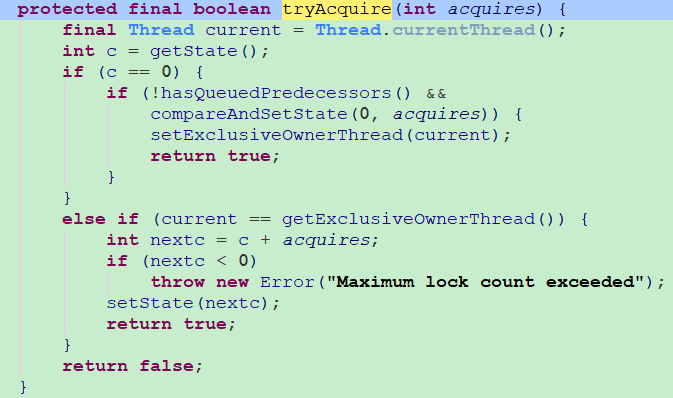
# ReentrantLock 分析：

**公平锁：**

Lock.lock()调用acquire(1)实现



If的第一个条件是尝试获取锁，具体实现



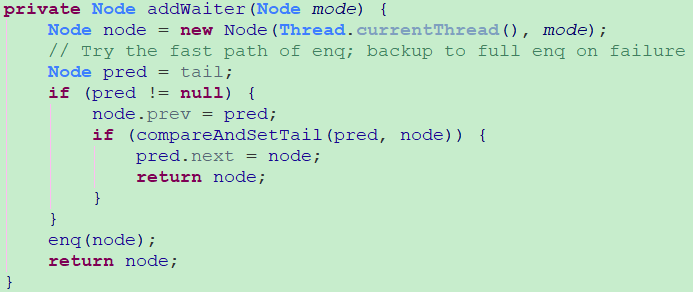
先判状态，state=0表示没锁，此时if的第一个条件判断阻塞队列为空，第二个条件将state=1，表示获取锁，若两个条件成功，将自己置为锁的拥有者。

如果有锁，就判断是不是自己已经获得了锁，都不成功表示获取锁失败

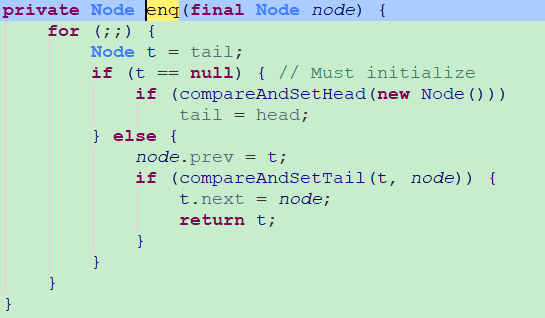
从上不难看出，tryAcquire(1)若成功，后续代码不用执行，线程已经成功获得了锁，若不成功，看if的第二个条件acquireQueued(addWaiter(Node.EXCLUSIVE), 1))

先来分析addWaiter(Node.EXCLUSIVE)

注意到static final Node EXCLUSIVE = null;



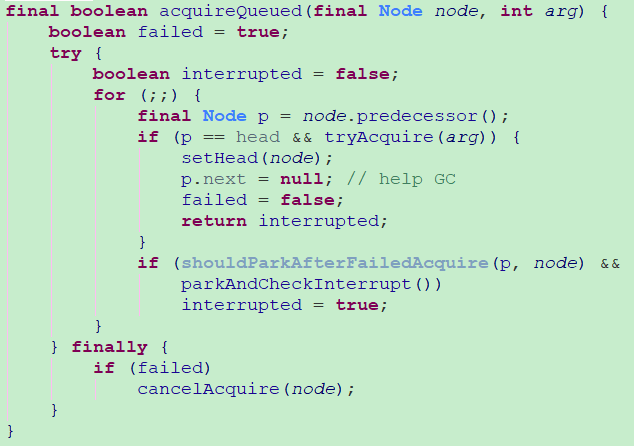
如果tail不空，将node加到waiter队列末尾。



注意到只有当tail为空时，才会进enq()。这是一个自旋，若tail和head同时为空，用新node替换head，再将tail指向head。然后通过CAS操作用参数node替换这个新加入的node，若发生竞争，通过for循环进行自旋，只到获取锁退出，返回参数node.这样addWaiter()函数将当前线程包装成node成功加入waiter队列末尾。接着分析

acquireQueued(addWaiter(Node.EXCLUSIVE), 1))

相当于acquireQueued(tail, 1))

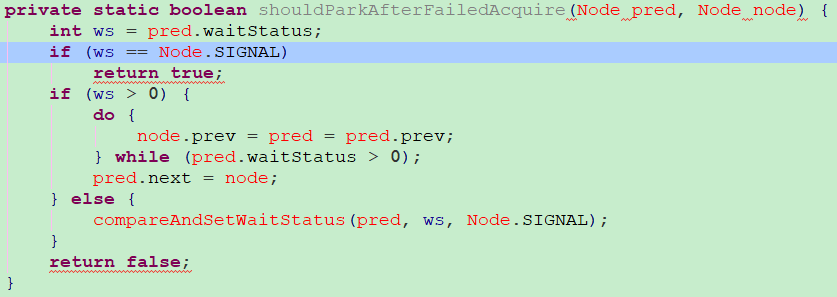


还是一个自旋，head节点也有可能是刚生成的node，并且还没进行CAS操作的，所以排在waiter队列的第一个node是有资格尝试获取锁的。如果不是第一个waiter，或者抢锁失败。进入下一个if

if (shouldParkAfterFailedAcquire(p, node) &&  
 parkAndCheckInterrupt())

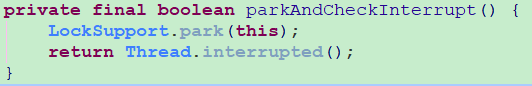
先看第一个条件：

shouldParkAfterFailedAcquire(p, node)



如果ws == Node.SIGNAL说明需要被唤醒才能工作，应该挂起。pred.waitStatus > 0退出锁竞争的线程，全部移除。当前驱节点waitStatus既不是 -1 也不是1时，必定为0，-2，-3，如果没有其它线程修改过waitStatus，它必定要被设置为 -1，返回ture，线程应当被挂起。否则还保持原来的值。此时会再次自旋，尝试获取锁。若判定为应该挂起再看第二个条件

parkAndCheckInterrupt

第一个条件断定为可挂起，第二个条件尝试挂起。挂起成功。

 acquireQueued的主要作用是把已经追加到队列的线程节点（addWaiter方法返回值）进行阻塞，但阻塞前又通过tryAccquire重试是否能获得锁，如果重试成功能则无需阻塞，直接返回。

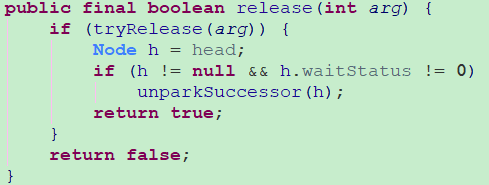
     仔细看看这个方法是个无限循环，感觉如果p == head && tryAcquire(arg)条件不满足循环将永远无法结束，当然不会出现死循环，奥秘在于第12行的parkAndCheckInterrupt会把当前线程挂起，从而阻塞住线程的调用栈。

如前面所述，LockSupport.park最终把线程交给系统（Linux）内核进行阻塞。

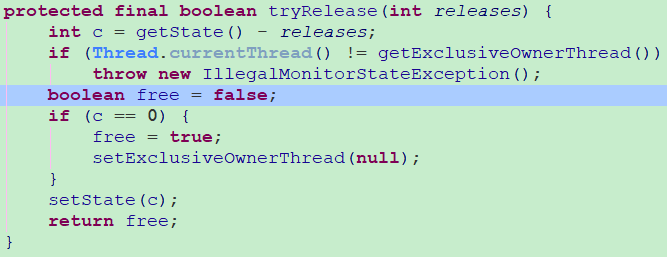
# 解锁操作

最后，就是还需要介绍下唤醒的动作了。我们知道，正常情况下，如果线程没获取到锁，线程会被 LockSupport.park(this); 挂起停止，等待被唤醒。

// 唤醒的代码还是比较简单的，你如果上面加锁的都看懂了，下面都不需要看就知道怎么回事了



// 回到ReentrantLock看tryRelease方法

 // 是否完全释放boolean free = false;

// 其实就是重入的问题，如果c==0，也就是说没有嵌套锁了，可以释放了，否则还不能释放掉

if (c == 0) {

free = true;

setExclusiveOwnerThread(null);

}

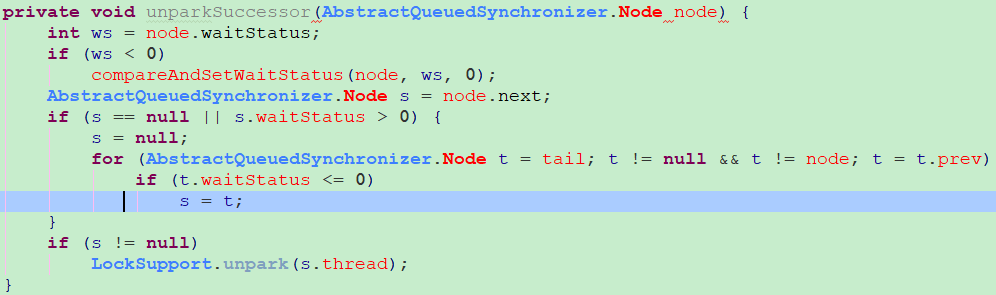
setState(c);

return free;

}

// 唤醒后继节点

// 从上面调用处知道，参数node是head头结点



// 下面的代码就是唤醒后继节点，但是有可能后继节点取消了等待（waitStatus==1）

// 从队尾往前找，找到waitStatus<=0的所有节点中排在最前面的

Node s = node.next;

if (s == null || s.waitStatus > 0) {s = null;

// 从后往前找，仔细看代码，不必担心中间有节点取消(waitStatus==1)的情况

for (Node t = tail; t != null && t != node; t = t.prev)

if (t.waitStatus <= 0)

s = t;}

if (s != null)

// 唤醒线程

LockSupport.unpark(s.thread);

}

线程唤醒后，接着挂起的地方运行。当然park()函数可以毫无理由地返回，随时都可以

总结：公平锁和非公平锁只有两处不同：

非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

[Java的LockSupport.park()实现分析](https://www.cnblogs.com/bendantuohai/p/4653543.html)

LockSupport类是Java6(JSR166-JUC)引入的一个类，提供了基本的线程同步原语。LockSupport实际上是调用了Unsafe类里的函数，归结到Unsafe里，只有两个函数：

 park：阻塞当前线程(Block current thread),字面理解park，就算占住，停车的时候不就把这个车位给占住了么？起这个名字还是很形象的。

unpark: 使给定的线程停止阻塞(Unblock the given thread blocked )。

public native void unpark(Thread jthread);

public native void park(boolean isAbsolute, long time);

isAbsolute参数是指明时间是绝对的，还是相对的。

仅仅两个简单的接口，就为上层提供了强大的同步原语。

先来解析下两个函数是做什么的。

unpark函数为线程提供“许可(permit)”，线程调用park函数则等待“许可”。这个有点像信号量，但是这个“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。

注意，unpark函数可以先于park调用。比如线程B调用unpark函数，给线程A发了一个“许可”，那么当线程A调用park时，它发现已经有“许可”了，那么它会马上再继续运行。

实际上，park函数即使没有“许可”，有时也会无理由地返回，这点等下再解析。

park和unpark的灵活之处

上面已经提到，unpark函数可以先于park调用，这个正是它们的灵活之处。

一个线程它有可能在别的线程unPark之前，或者之后，或者同时调用了park，那么因为park的特性，它可以不用担心自己的park的时序问题，否则，如果park必须要在unpark之前，那么给编程带来很大的麻烦！！

考虑一下，两个线程同步，要如何处理？

在Java5里是用wait/notify/notifyAll来同步的。wait/notify机制有个很蛋疼的地方是，比如线程B要用notify通知线程A，那么线程B要确保线程A已经在wait调用上等待了，否则线程A可能永远都在等待。编程的时候就会很蛋疼。

另外，是调用notify，还是notifyAll？

notify只会唤醒一个线程，如果错误地有两个线程在同一个对象上wait等待，那么又悲剧了。为了安全起见，貌似只能调用notifyAll了。

park/unpark模型真正解耦了线程之间的同步，线程之间不再需要一个Object或者其它变量来存储状态，不再需要关心对方的状态。

HotSpot里park/unpark的实现

每个java线程都有一个Parker实例，Parker类是这样定义的：

[cpp] [view plaincopy](http://blog.csdn.net/hengyunabc/article/details/28126139)[在CODE上查看代码片](https://code.csdn.net/snippets/373846)

class Parker : public os::PlatformParker {

private:

  volatile int \_counter ;

  ...

public:

  void park(bool isAbsolute, jlong time);

  void unpark();

  ...

}

class PlatformParker : public CHeapObj<mtInternal> {

  protected:

    pthread\_mutex\_t \_mutex [1] ;

    pthread\_cond\_t  \_cond  [1] ;

    ...

}

可以看到Parker类实际上用Posix的mutex，condition来实现的。

在Parker类里的\_counter字段，就是用来记录所谓的“许可”的。

当调用park时，先尝试直接能否直接拿到“许可”，即\_counter>0时，如果成功，则把\_counter设置为0,并返回：

[cpp] [view plaincopy](http://blog.csdn.net/hengyunabc/article/details/28126139)[在CODE上查看代码片](https://code.csdn.net/snippets/373846)

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

  // Ideally we'd do something useful while spinning, such

  // as calling unpackTime().

  // Optional fast-path check:

  // Return immediately if a permit is available.

  // We depend on Atomic::xchg() having full barrier semantics

  // since we are doing a lock-free update to \_counter.

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

如果不成功，则构造一个ThreadBlockInVM，然后检查\_counter是不是>0，如果是，则把\_counter设置为0，unlock mutex并返回：

[cpp] [view plaincopy](http://blog.csdn.net/hengyunabc/article/details/28126139)[在CODE上查看代码片](https://code.csdn.net/snippets/373846)

ThreadBlockInVM tbivm(jt);

if (\_counter > 0)  { // no wait needed

  \_counter = 0;

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

否则，再判断等待的时间，然后再调用pthread\_cond\_wait函数等待，如果等待返回，则把\_counter设置为0，unlock mutex并返回：

[cpp] [view plaincopy](http://blog.csdn.net/hengyunabc/article/details/28126139)[在CODE上查看代码片](https://code.csdn.net/snippets/373846)

if (time == 0) {

  status = pthread\_cond\_wait (\_cond, \_mutex) ;

}

\_counter = 0 ;

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

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

OrderAccess::fence();

当unpark时，则简单多了，直接设置\_counter为1，再unlock mutext返回。如果\_counter之前的值是0，则还要调用pthread\_cond\_signal唤醒在park中等待的线程：

[cpp] [view plaincopy](http://blog.csdn.net/hengyunabc/article/details/28126139)[在CODE上查看代码片](https://code.csdn.net/snippets/373846)

void Parker::unpark() {

  int s, status ;

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

  assert (status == 0, "invariant") ;

  s = \_counter;

  \_counter = 1;

  if (s < 1) {

     if (WorkAroundNPTLTimedWaitHang) {

        status = pthread\_cond\_signal (\_cond) ;

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

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

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

     } else {

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

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

        status = pthread\_cond\_signal (\_cond) ;

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

     }

  } else {

    pthread\_mutex\_unlock(\_mutex);

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

  }

}

简而言之，是用mutex和condition保护了一个\_counter的变量，当park时，这个变量置为了0，当unpark时，这个变量置为1。  
值得注意的是在park函数里，调用pthread\_cond\_wait时，并没有用while来判断，所以posix condition里的"Spurious wakeup"一样会传递到上层Java的代码里。

LockSupport是不可重入的，如果一个线程连续2次调用LockSupport.park()，那么该线程一定会一直阻塞下去。

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

{

Thread thread = Thread.currentThread();

LockSupport.unpark(thread);

System.out.println("a");

LockSupport.park();

System.out.println("b");

LockSupport.park();

System.out.println("c");

}

这段代码打印出a和b，不会打印c，因为第二次调用park的时候，线程无法获取许可出现死锁。

下面我们来看下LockSupport对应中断的响应性

public static void t2() throws Exception

{

Thread t = new Thread(new Runnable()

{

private int count = 0;

@Override

public void run()

{

long start = System.currentTimeMillis();

long end = 0;

while ((end - start) <= 1000)

{

count++;

end = System.currentTimeMillis();

}

System.out.println("after 1 second.count=" + count);

//等待或许许可

LockSupport.park();

System.out.println("thread over." + Thread.currentThread().isInterrupted());

}

});

t.start();

Thread.sleep(2000);

// 中断线程

t.interrupt();

System.out.println("main over");

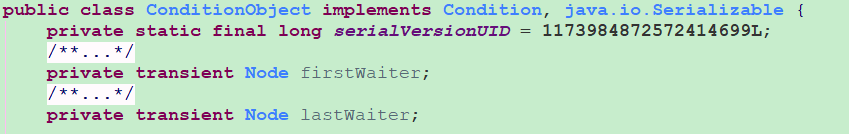
}

最终线程会打印出thread over.true。这说明线程如果因为调用park而阻塞的话，能够响应中断请求(中断状态被设置成true)，但是不会抛出InterruptedException。

Condition

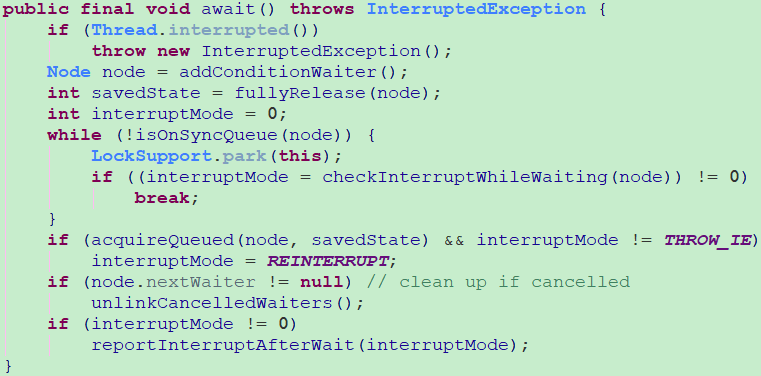
Condition 是基于 ReentrantLock 实现的，而 ReentrantLock 是依赖于AbstractQueuedSynchronizer 实现的下面看下ReentrantLock中Conditino的实现

ConditionObject实现了一个队列，我们称这个队列为条件队列，每个Condition对象都有这样一个队列

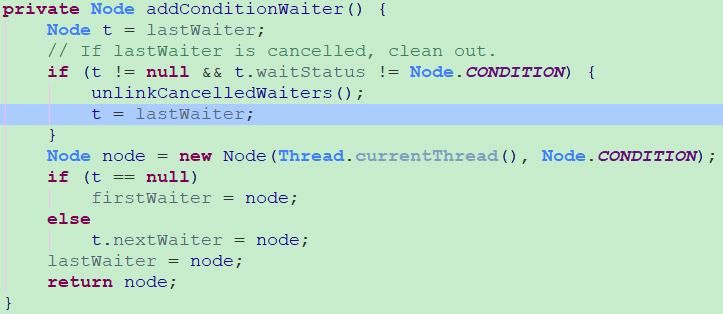


线程 1 调用 condition1.await() 方法即可将当前线程 1 包装成 Node 加入到当前Condition条件队列中，然后阻塞在这里，不继续往下执行，条件队列是一个单向链表

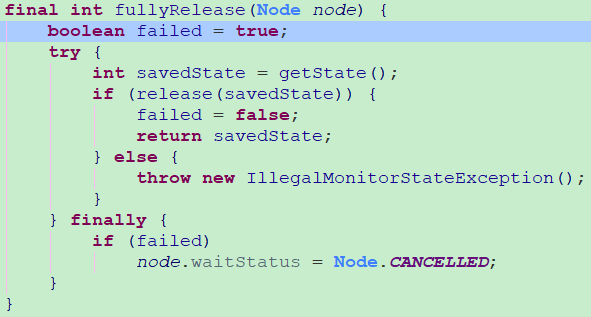
调用 condition1.signal() 会将condition1 对应的条件队列的 firstWaiter 移到阻塞队列的队尾，等待获取锁，获取锁后 await 方法返回，继续往下执行。具体实现如下



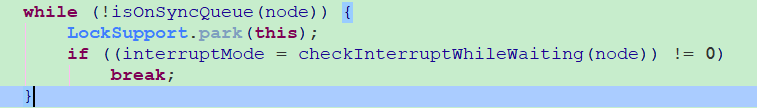
第一步就是要将调用await()的线程加入到Condition的条件队列中，并且去掉已经不处于Node.CONDITION状态的节点（线程），返回值为当前线程的node



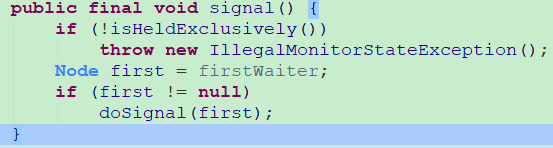
线程加入Condition条件队列后挂起，释放锁，锁的状态是记录在node上的



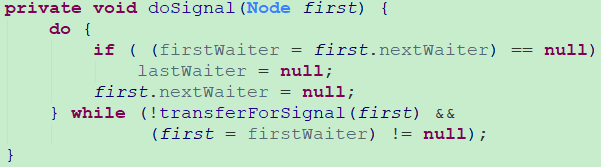
释放锁后，若不在阻塞队列，挂起并可响应中断

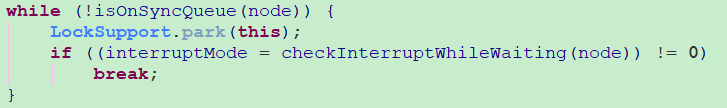


下面看唤醒阻塞，先判断是否由独占锁，没获取锁直接抛异常



将条件队列第一个转移到阻塞队列，使用unpark()唤醒线程。





唤醒后检查中断状态

先解释下 interruptMode。interruptMode 可以取值为 REINTERRUPT（1），THROW\_IE（-1），0

REINTERRUPT： 代表 await 返回的时候，需要重新设置中断状态

THROW\_IE： 代表 await 返回的时候，需要抛出 InterruptedException 异常

0 ：说明在 await 期间，没有发生中断

有以下三种情况会让 LockSupport.park(this); 这句返回继续往下执行：

常规路劲。signal -> 转移节点到阻塞队列 -> 获取了锁（unpark）

线程中断。在 park 的时候，另外一个线程对这个线程进行了中断

signal 的时候我们说过，转移以后的前驱节点取消了，或者对前驱节点的CAS操作失败了

假唤醒。这个也是存在的，和 Object.wait() 类似，都有这个问题

线程唤醒后第一步是调用 checkInterruptWhileWaiting(node) 这个方法，此方法用于判断是否在线程挂起期间发生了中断，如果发生了中断，是 signal 调用之前中断的，还是 signal 之后发生的中断。

// 1. 如果在 signal 之前已经中断，返回 THROW\_IE

// 2. 如果是 signal 之后中断，返回 REINTERRUPT

// 3. 没有发生中断，返回 0

private int checkInterruptWhileWaiting(Node node) {

return Thread.interrupted() ?

(transferAfterCancelledWait(node) ? THROW\_IE : REINTERRUPT) :

0;

}

Thread.interrupted()：如果当前线程已经处于中断状态，那么该方法返回 true，同时将中断状态重置为 false，所以，才有后续的 重新中断（REINTERRUPT） 的使用。

看看怎么判断是 signal 之前还是之后发生的中断：

// 只有线程处于中断状态，才会调用此方法

// 如果需要的话，将这个已经取消等待的节点转移到阻塞队列

// 返回 true：如果此线程在 signal 之前被取消，

final boolean transferAfterCancelledWait(Node node) {

// 用 CAS 将节点状态设置为 0

// 如果这步 CAS 成功，说明是 signal 方法之前发生的中断，因为如果 signal 先发生的话，signal 中会将 waitStatus 设置为 0

if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {

// 将节点放入阻塞队列

// 这里我们看到，即使中断了，依然会转移到阻塞队列

enq(node);

return true;

}

// 到这里是因为 CAS 失败，肯定是因为 signal 方法已经将 waitStatus 设置为了 0

// signal 方法会将节点转移到阻塞队列，但是可能还没完成，这边自旋等待其完成

// 当然，这种事情还是比较少的吧：signal 调用之后，没完成转移之前，发生了中断

while (!isOnSyncQueue(node))

Thread.yield();

return false;

}

这里再说一遍，即使发生了中断，节点依然会转移到阻塞队列。

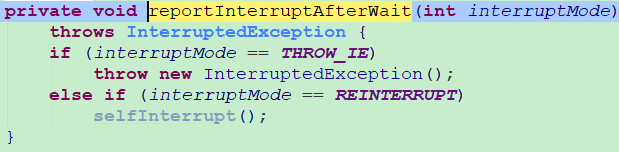
到这里，大家应该都知道这个 while 循环怎么退出了吧。要么中断，要么转移成功。

获取独占锁



不管有没有发生中断，都会进入到阻塞队列，而 acquireQueued(node, savedState) 的返回值就是代表线程是否被中断。如果返回 true，说明被中断了，而且 interruptMode != THROW\_IE，说明在 signal 之前就发生中断了，这里将 interruptMode 设置为 REINTERRUPT，用于待会重新中断。这边说说 node.nextWaiter != null 怎么满足。我前面也说了 signal 的时候会将节点转移到阻塞队列，有一步是 node.nextWaiter = null，将断开节点和条件队列的联系。

可是，在判断发生中断的情况下，是 signal 之前还是之后发生的？ 这部分的时候，我也介绍了，如果 signal 之前就中断了，也需要将节点进行转移到阻塞队列，这部分转移的时候，是没有设置 node.nextWaiter = null 的。响应中断



共享模式的使用

synchronized底层语义原理

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现， 无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法 并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的，关于这点，稍后详细分析。下面先来了解一个概念Java对象头，这对深入理解synchronized实现原理非常关键。

理解Java对象头与Monitor

在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。如下：

实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。

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，这点了解即可。

而对于顶部，则是Java头对象，它实现synchronized的锁对象的基础，这点我们重点分析它，一般而言，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，其结构说明如下表：

| 虚拟机位数 | 头对象结构 | 说明 |
| --- | --- | --- |
| 32/64bit | Mark Word | 存储对象的hashCode、锁信息或分代年龄或GC标志等信息 |
| 32/64bit | Class Metadata Address | 类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的实例。 |

其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构

| 锁状态 | 25bit | 4bit | 1bit是否是偏向锁 | 2bit 锁标志位 |
| --- | --- | --- | --- | --- |
| 无锁状态 | 对象HashCode | 对象分代年龄 | 0 | 01 |

由于对象头的信息是与对象自身定义的数据没有关系的额外存储成本，因此考虑到JVM的空间效率，Mark Word 被设计成为一个非固定的数据结构，以便存储更多有效的数据，它会根据对象本身的状态复用自己的存储空间，如32位JVM下，除了上述列出的Mark Word默认存储结构外，还有如下可能变化的结构：

其中轻量级锁和偏向锁是Java 6 对 synchronized 锁进行优化后新增加的，稍后我们会简要分析。这里我们主要分析一下重量级锁也就是通常说synchronized的对象锁，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如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，其主要数据结构如下（位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的）

ObjectMonitor() {

\_header = NULL;

\_count = 0; //记录个数

\_waiters = 0,

\_recursions = 0;

\_object = NULL;

\_owner = NULL;

\_WaitSet = NULL; //处于wait状态的线程，会被加入到\_WaitSet

\_WaitSetLock = 0 ;

\_Responsible = NULL ;

\_succ = NULL ;

\_cxq = NULL ;

FreeNext = NULL ;

\_EntryList = NULL ; //处于等待锁block状态的线程，会被加入到该列表

\_SpinFreq = 0 ;

\_SpinClock = 0 ;

OwnerIsThread = 0 ;

}

ObjectMonitor中有两个队列，\_WaitSet 和 \_EntryList，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)。如下图所示

由此看来，monitor对象存在于每个Java对象的对象头中(存储的指针的指向)，synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因，同时也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因(关于这点稍后还会进行分析)，ok~，有了上述知识基础后，下面我们将进一步分析synchronized在字节码层面的具体语义实现。

synchronized代码块底层原理

现在我们重新定义一个synchronized修饰的同步代码块，在代码块中操作共享变量i，如下

public class SyncCodeBlock {

public int i;

public void syncTask(){

//同步代码库

synchronized (this){

i++;

}

}

}

编译上述代码并使用javap反编译后得到字节码如下(这里我们省略一部分没有必要的信息)：

Classfile /Users/zejian/Downloads/Java8\_Action/src/main/java/com/zejian/concurrencys/SyncCodeBlock.class

Last modified 2017-6-2; size 426 bytes

MD5 checksum c80bc322c87b312de760942820b4fed5

Compiled from "SyncCodeBlock.java"

public class com.zejian.concurrencys.SyncCodeBlock

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

//........省略常量池中数据

//构造函数

public com.zejian.concurrencys.SyncCodeBlock();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 7: 0

//===========主要看看syncTask方法实现================

public void syncTask();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=3, locals=3, args\_size=1

0: aload\_0

1: dup

2: astore\_1

3: monitorenter //注意此处，进入同步方法

4: aload\_0

5: dup

6: getfield #2 // Field i:I

9: iconst\_1

10: iadd

11: putfield #2 // Field i:I

14: aload\_1

15: monitorexit //注意此处，退出同步方法

16: goto 24

19: astore\_2

20: aload\_1

21: monitorexit //注意此处，退出同步方法

22: aload\_2

23: athrow

24: return

Exception table:

//省略其他字节码.......

}

SourceFile: "SyncCodeBlock.java"

我们主要关注字节码中的如下代码

3: monitorenter //进入同步方法

//..........省略其他

15: monitorexit //退出同步方法

16: goto 24

//省略其他.......

21: monitorexit //退出同步方法

从字节码中可知同步语句块的实现使用的是monitorenter 和 monitorexit 指令，其中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 (关于重入性稍后会分析)，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 。值得注意的是编译器将会确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都有执行其对应 monitorexit 指令，而无论这个方法是正常结束还是异常结束。为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令。从字节码中也可以看出多了一个monitorexit指令，它就是异常结束时被执行的释放monitor 的指令。

synchronized方法底层原理

方法级的同步是隐式，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（虚拟机规范中用的是管程一词）， 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。在方法执行期间，执行线程持有了monitor，其他任何线程都无法再获得同一个monitor。如果一个同步方法执行期间抛 出了异常，并且在方法内部无法处理此异常，那这个同步方法所持有的monitor将在异常抛到同步方法之外时自动释放。下面我们看看字节码层面如何实现：

public class SyncMethod {

public int i;

public synchronized void syncTask(){

i++;

}

}

使用javap反编译后的字节码如下：

Classfile /Users/zejian/Downloads/Java8\_Action/src/main/java/com/zejian/concurrencys/SyncMethod.class

Last modified 2017-6-2; size 308 bytes

MD5 checksum f34075a8c059ea65e4cc2fa610e0cd94

Compiled from "SyncMethod.java"

public class com.zejian.concurrencys.SyncMethod

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool;

//省略没必要的字节码

//==================syncTask方法======================

public synchronized void syncTask();

descriptor: ()V

//方法标识ACC\_PUBLIC代表public修饰，ACC\_SYNCHRONIZED指明该方法为同步方法

flags: ACC\_PUBLIC, ACC\_SYNCHRONIZED

Code:

stack=3, locals=1, args\_size=1

0: aload\_0

1: dup

2: getfield #2 // Field i:I

5: iconst\_1

6: iadd

7: putfield #2 // Field i:I

10: return

LineNumberTable:

line 12: 0

line 13: 10

}

SourceFile: "SyncMethod.java"

从字节码中可以看出，synchronized修饰的方法并没有monitorenter指令和monitorexit指令，取得代之的确实是ACC\_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。这便是synchronized锁在同步代码块和同步方法上实现的基本原理。同时我们还必须注意到的是在Java早期版本中，synchronized属于重量级锁，效率低下，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，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。庆幸的是在Java 6之后Java官方对从JVM层面对synchronized较大优化，所以现在的synchronized锁效率也优化得很不错了，Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了轻量级锁和偏向锁，接下来我们将简单了解一下Java官方在JVM层面对synchronized锁的优化。

Java虚拟机对synchronized的优化

锁的状态总共有四种，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，关于重量级锁，前面我们已详细分析过，下面我们将介绍偏向锁和轻量级锁以及JVM的其他优化手段，这里并不打算深入到每个锁的实现和转换过程更多地是阐述Java虚拟机所提供的每个锁的核心优化思想，毕竟涉及到具体过程比较繁琐，如需了解详细过程可以查阅《深入理解Java虚拟机原理》。

偏向锁

偏向锁是Java 6之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。下面我们接着了解轻量级锁。

轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟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，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

/\*\*

\* Created by zejian on 2017/6/4.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\* 消除StringBuffer同步锁

\*/

public class StringBufferRemoveSync {

public void add(String str1, String str2) {

//StringBuffer是线程安全,由于sb只会在append方法中使用,不可能被其他线程引用

//因此sb属于不可能共享的资源,JVM会自动消除内部的锁

StringBuffer sb = new StringBuffer();

sb.append(str1).append(str2);

}

public static void main(String[] args) {

StringBufferRemoveSync rmsync = new StringBufferRemoveSync();

for (int i = 0; i < 10000000; i++) {

rmsync.add("abc", "123");

}

}

}

关于synchronized 可能需要了解的关键点

synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功，在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。如下：

public class AccountingSync implements Runnable{

static AccountingSync instance=new AccountingSync();

static int i=0;

static int j=0;

@Override

public void run() {

for(int j=0;j<1000000;j++){

//this,当前实例对象锁

synchronized(this){

i++;

increase();//synchronized的可重入性

}

}

}

public synchronized void increase(){

j++;

}

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

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

正如代码所演示的，在获取当前实例对象锁后进入synchronized代码块执行同步代码，并在代码块中调用了当前实例对象的另外一个synchronized方法，再次请求当前实例锁时，将被允许，进而执行方法体代码，这就是重入锁最直接的体现，需要特别注意另外一种情况，当子类继承父类时，子类也是可以通过可重入锁调用父类的同步方法。注意由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。

线程中断与synchronized

线程中断

正如中断二字所表达的意义，在线程运行(run方法)中间打断它，在Java中，提供了以下3个有关线程中断的方法

//中断线程（实例方法）

public void Thread.interrupt();

//判断线程是否被中断（实例方法）

public boolean Thread.isInterrupted();

//判断是否被中断并清除当前中断状态（静态方法）

public static boolean Thread.interrupted();

当一个线程处于被阻塞状态或者试图执行一个阻塞操作时，使用Thread.interrupt()方式中断该线程，注意此时将会抛出一个InterruptedException的异常，同时中断状态将会被复位(由中断状态改为非中断状态)，如下代码将演示该过程：

public class InterruputSleepThread3 {

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

Thread t1 = new Thread() {

@Override

public void run() {

//while在try中，通过异常中断就可以退出run循环

try {

while (true) {

//当前线程处于阻塞状态，异常必须捕捉处理，无法往外抛出

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

System.out.println("Interruted When Sleep");

boolean interrupt = this.isInterrupted();

//中断状态被复位

System.out.println("interrupt:"+interrupt);

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

//中断处于阻塞状态的线程

t1.interrupt();

/\*\*

\* 输出结果:

Interruted When Sleep

interrupt:false

\*/

}

}

如上述代码所示，我们创建一个线程，并在线程中调用了sleep方法从而使用线程进入阻塞状态，启动线程后，调用线程实例对象的interrupt方法中断阻塞异常，并抛出InterruptedException异常，此时中断状态也将被复位。这里有些人可能会诧异，为什么不用Thread.sleep(2000);而是用TimeUnit.SECONDS.sleep(2);其实原因很简单，前者使用时并没有明确的单位说明，而后者非常明确表达秒的单位，事实上后者的内部实现最终还是调用了Thread.sleep(2000);，但为了编写的代码语义更清晰，建议使用TimeUnit.SECONDS.sleep(2);的方式，注意TimeUnit是个枚举类型。ok~，除了阻塞中断的情景，我们还可能会遇到处于运行期且非阻塞的状态的线程，这种情况下，直接调用Thread.interrupt()中断线程是不会得到任响应的，如下代码，将无法中断非阻塞状态下的线程：

public class InterruputThread {

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

Thread t1=new Thread(){

@Override

public void run(){

while(true){

System.out.println("未被中断");

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果(无限执行):

未被中断

未被中断

未被中断

......

\*/

}

}

虽然我们调用了interrupt方法，但线程t1并未被中断，因为处于非阻塞状态的线程需要我们手动进行中断检测并结束程序，改进后代码如下：

public class InterruputThread {

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

Thread t1=new Thread(){

@Override

public void run(){

while(true){

//判断当前线程是否被中断

if (this.isInterrupted()){

System.out.println("线程中断");

break;

}

}

System.out.println("已跳出循环,线程中断!");

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果:

线程中断

已跳出循环,线程中断!

\*/

}

}

是的，我们在代码中使用了实例方法isInterrupted判断线程是否已被中断，如果被中断将跳出循环以此结束线程,注意非阻塞状态调用interrupt()并不会导致中断状态重置。综合所述，可以简单总结一下中断两种情况，一种是当线程处于阻塞状态或者试图执行一个阻塞操作时，我们可以使用实例方法interrupt()进行线程中断，执行中断操作后将会抛出interruptException异常(该异常必须捕捉无法向外抛出)并将中断状态复位，另外一种是当线程处于运行状态时，我们也可调用实例方法interrupt()进行线程中断，但同时必须手动判断中断状态，并编写中断线程的代码(其实就是结束run方法体的代码)。有时我们在编码时可能需要兼顾以上两种情况，那么就可以如下编写：

public void run(){

try {

//判断当前线程是否已中断,注意interrupted方法是静态的,执行后会对中断状态进行复位

while (!Thread.interrupted()) {

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

}

}

中断与synchronized

事实上线程的中断操作对于正在等待获取的锁对象的synchronized方法或者代码块并不起作用，也就是对于synchronized来说，如果一个线程在等待锁，那么结果只有两种，要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效。演示代码如下

/\*\*

\* Created by zejian on 2017/6/2.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class SynchronizedBlocked implements Runnable{

public synchronized void f() {

System.out.println("Trying to call f()");

while(true) // Never releases lock

Thread.yield();

}

/\*\*

\* 在构造器中创建新线程并启动获取对象锁

\*/

public SynchronizedBlocked() {

//该线程已持有当前实例锁

new Thread() {

public void run() {

f(); // Lock acquired by this thread

}

}.start();

}

public void run() {

//中断判断

while (true) {

if (Thread.interrupted()) {

System.out.println("中断线程!!");

break;

} else {

f();

}

}

}

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

SynchronizedBlocked sync = new SynchronizedBlocked();

Thread t = new Thread(sync);

//启动后调用f()方法,无法获取当前实例锁处于等待状态

t.start();

TimeUnit.SECONDS.sleep(1);

//中断线程,无法生效

t.interrupt();

}

}

我们在SynchronizedBlocked构造函数中创建一个新线程并启动获取调用f()获取到当前实例锁，由于SynchronizedBlocked自身也是线程，启动后在其run方法中也调用了f()，但由于对象锁被其他线程占用，导致t线程只能等到锁，此时我们调用了t.interrupt();但并不能中断线程。

等待唤醒机制与synchronized

所谓等待唤醒机制本篇主要指的是notify/notifyAll和wait方法，在使用这3个方法时，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，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

synchronized (obj) {

obj.wait();

obj.notify();

obj.notifyAll();

}

需要特别理解的一点是，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。

1. 线程状态及状态转换

    当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

Contention List：所有请求锁的线程将被首先放置到该竞争队列

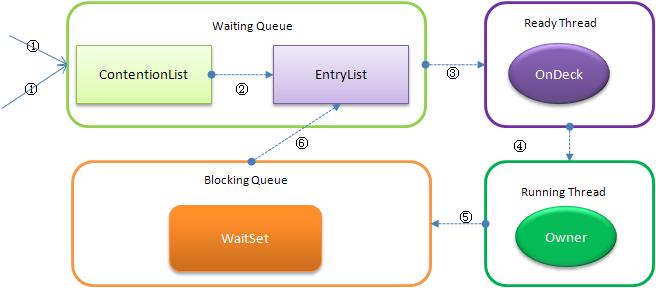
Entry List：Contention List中那些有资格成为候选人的线程被移到Entry List

Wait Set：那些调用wait方法被阻塞的线程被放置到Wait Set

OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck

Owner：获得锁的线程称为Owner

!Owner：释放锁的线程

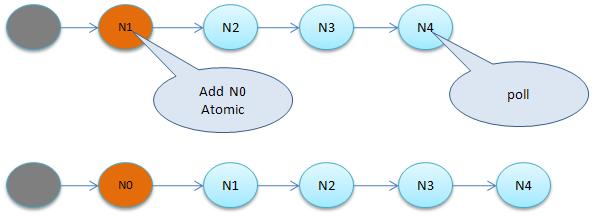
下图反映了个状态转换关系：  


　　新请求锁的线程将首先被加入到ConetentionList中，当某个拥有锁的线程（Owner状态）调用unlock之后，如果发现EntryList为空则从ContentionList中移动线程到EntryList，下面说明下ContentionList和EntryList的实现方式：

1.1 Contention List虚拟队列

　　Contention List并不是一个真正的Queue，而只是一个虚拟队列，原因在于ContentionList是由Node及其next指针逻辑构成，并不存在一个Queue的数据结构。ContentionList是一个后进先出（LIFO）的队列，每次新加入Node时都会在队头进行，通过CAS改变第一个节点的的指针为新增节点，同时设置新增节点的next指向后续节点，而取得操作则发生在队尾。显然，该结构其实是个Lock-Free的队列。

　　因为只有Owner线程才能从队尾取元素，也即线程出列操作无争用，当然也就避免了CAS的ABA问题。



ps.这里怎么看都像FIFO，查了很久，没找到关于Contention List 的详细说明，这里先不纠结了。

1.2 EntryList

EntryList与ContentionList逻辑上同属等待队列，ContentionList会被线程并发访问，为了降低对ContentionList队尾的争用，而建立EntryList。Owner线程在unlock时会从ContentionList中迁移线程到EntryList，并会指定EntryList中的某个线程（一般为Head）为Ready（OnDeck）线程。Owner线程并不是把锁传递给OnDeck线程，只是把竞争锁的权利交给OnDeck，OnDeck线程需要重新竞争锁。这样做虽然牺牲了一定的公平性，但极大的提高了整体吞吐量，在Hotspot中把OnDeck的选择行为称之为“竞争切换”。

OnDeck线程获得锁后即变为owner线程，无法获得锁则会依然留在EntryList中，考虑到公平性，在EntryList中的位置不发生变化（依然在队头）。如果Owner线程被wait方法阻塞，则转移到WaitSet队列；如果在某个时刻被notify/notifyAll唤醒，则再次转移到EntryList。

# Executor框架

**corePoolSize**：核心线程数，如果运行的线程少于corePoolSize，则创建新线程来执行新任务，即使线程池中的其他线程是空闲的

**maximumPoolSize**:最大线程数，可允许创建的线程数，

**keepAliveTime**:如果线程数多于corePoolSize,则这些多余的线程的空闲时间超过**keepAliveTime**时将被终止

unit:keepAliveTime参数的时间单位

**workQueue**:保存任务的阻塞队列，与线程池的大小有关：

  当运行的线程数少于corePoolSize时，在有新任务时直接创建新线程来执行任务而无需再进队列

  当运行的线程数等于或多于corePoolSize，在有新任务添加时则选加入队列，不直接创建线程

  当队列满时，在有新任务时就创建新线程

**threadFactory**:使用ThreadFactory创建新线程，默认使用defaultThreadFactory创建线程

**handle**:定义处理被拒绝任务的策略，默认使用ThreadPoolExecutor.AbortPolicy,任务被拒绝时将抛出RejectExecutorException