1、并发编程的一些概念及常用的类，做下简要的说明

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

线程安全的：**当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替运行，并且不需要额外的同步及在调用方代码不必做其他的协调，这个类的行为仍然是正确的，那么这个类就是线程安全的（**在Java Concurrency in Practice中的定义）

并行与并发：

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力

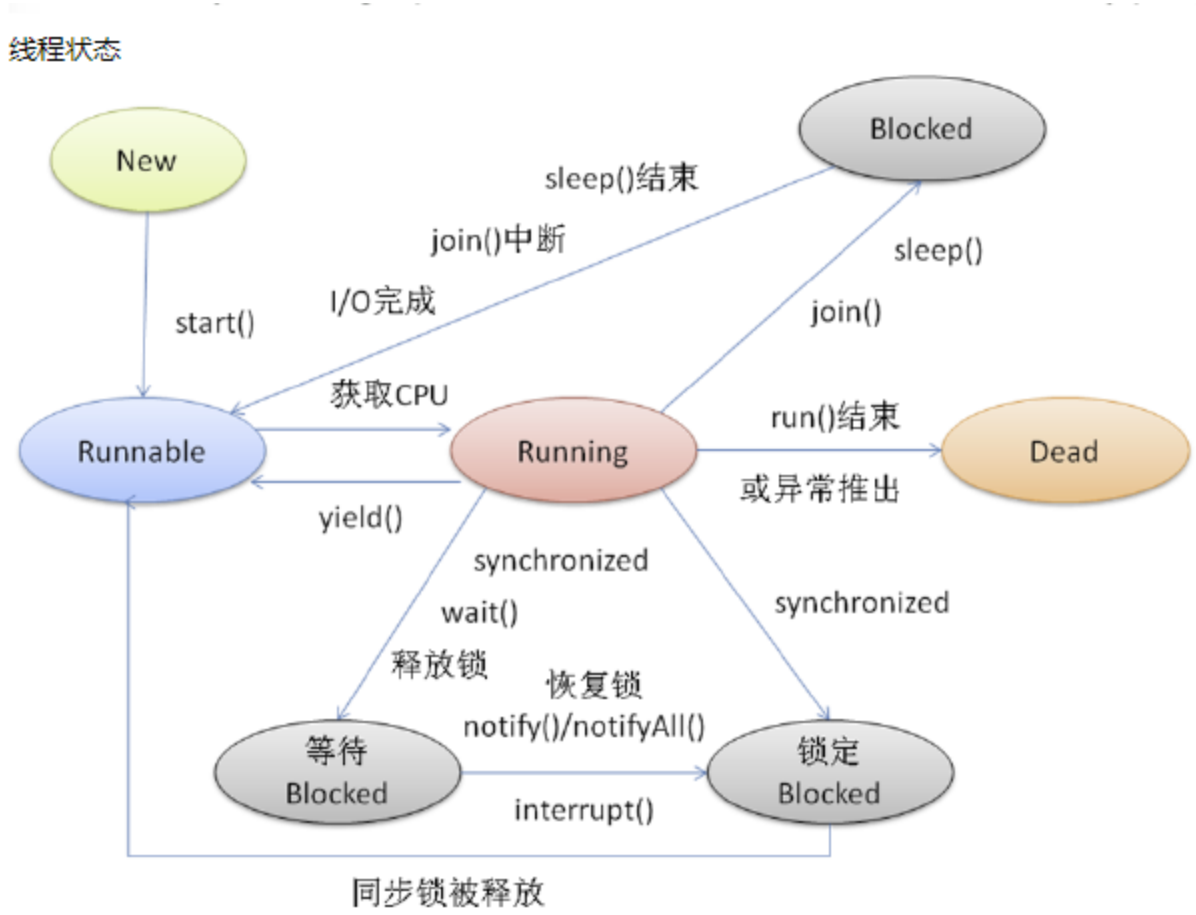
线程的状态

public static enum Thread.State
extends Enum<Thread.State>

A thread state. A thread can be in one of the following states:

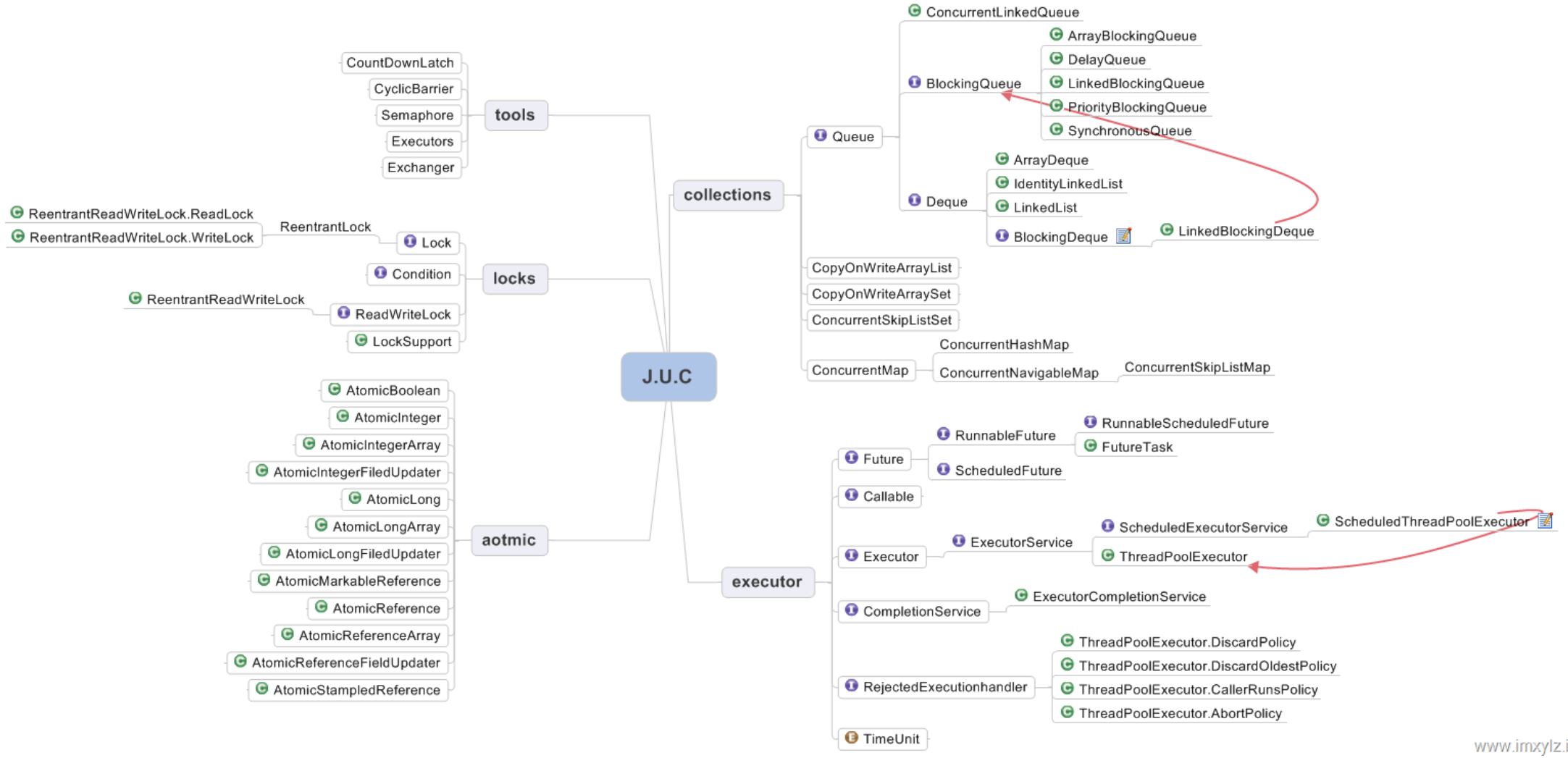
- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.



1.调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2.调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool)，释放同步锁使线程回到可运行状态（Runnable）
3.对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool)，同步锁被释放进入可运行状态(Runnable)。
此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。
Thread类中的yield方法可以让一个running状态的线程转入runnable。

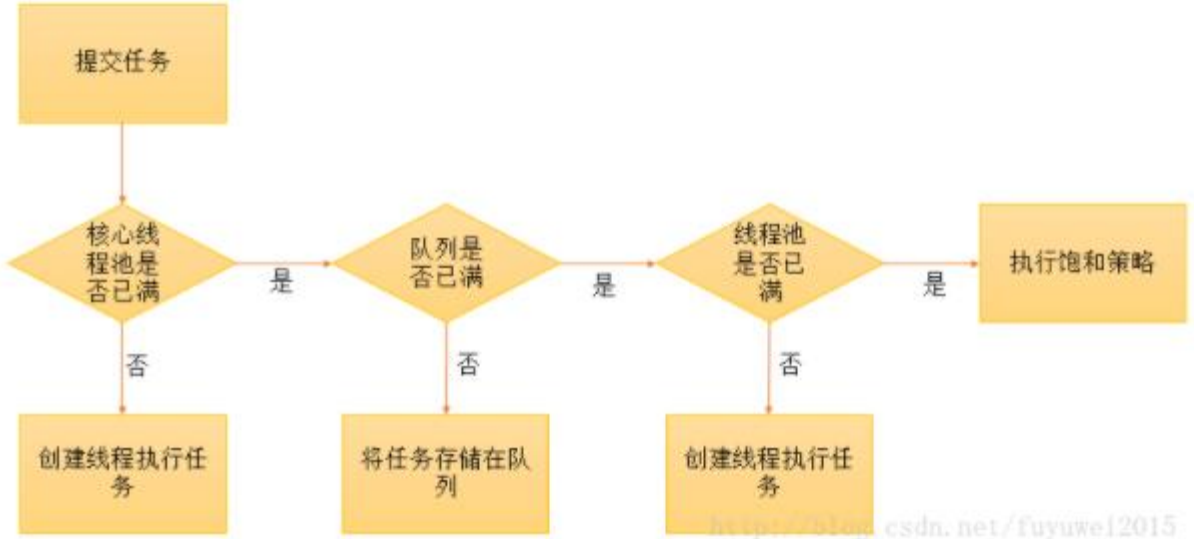
Juc的主要类

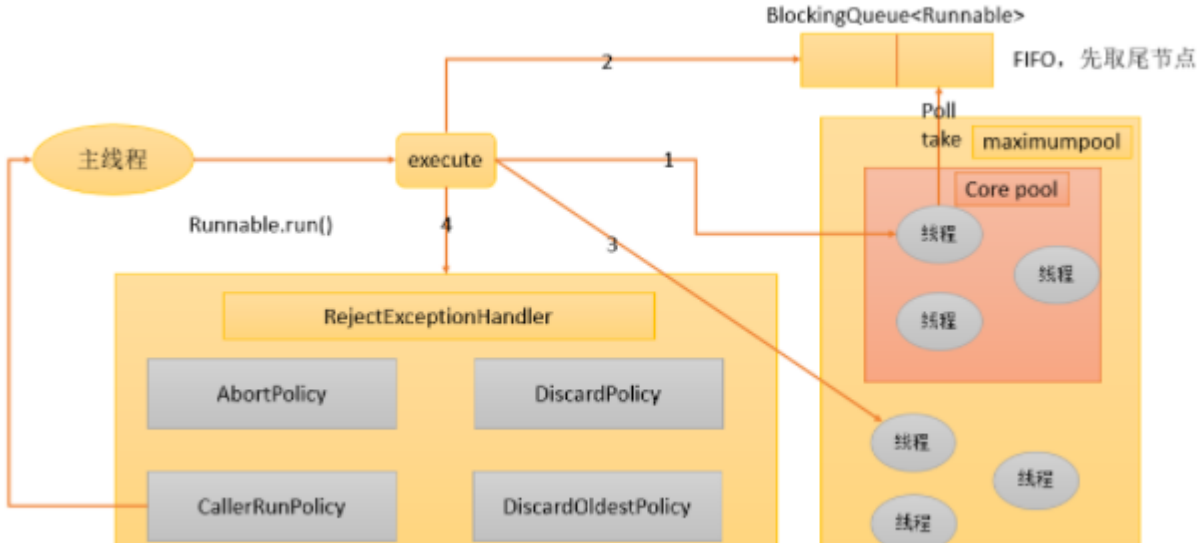


- 概念介绍的参考文献
- 1、http://www.blogjava.net/xylz/archive/2010/07/03/325168.html
- 2、https://www.cnblogs.com/wxd0108/p/5479442.html
- 3、http://www.jianshu.com/p/f4454164c017
- 4、http://blog.csdn.net/chen77716/article/details/6618779
- 5、https://www.cnblogs.com/dennyzhangdd/p/6734638.html
  （这篇对synchronized分析的很详细，因为时间，没有仔细看完）

2、线程池的实现

- 先看下实现的流程图






```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

1) corePoolSize 核心线程数

2）maximumPoolSize 最大线程数
3）keepAliveTime 线程保持激活状态的时间，如果为0，永远处于激活状态
4）unit ，keepAliveTime的单位
5）workQueue，线程池使用的队列
6）threadFactory 创建线程的工厂
7）handler 当队列已满，无更大线程处理任务时的拒绝任务的策略。
除了这些核心参数外，我觉得有必要再关注如下
8）HashSet<Worker> workers
9）completedTaskCount 完成的任务数
10）allowCoreThreadTimeOut，该值默认为false，也就是默认keepAliveTime不会生效。

线程池的状态

- RUNNING　　运行态
- SHUTDOWN　关闭，此时不接受新的任务，但继续处理队列中的任务。
- STOP　　停止，此时不接受新的任务，不处理队列中的任务，并中断正在执行的任务
- TIDYING　　所有的工作线程全部停止，并工作线程数量为0，将调用terminated方法，进入到TERMINATED
- TERMINATED　终止状态
- 线程池默认状态 RUNNING
- 如果调用shutdwon() 方法，状态从 RUNNING ---> SHUTDOWN
- 如果调用shutdwonNow()方法，状态从RUUNING|SHUTDOWN--->STOP
- SHUTDOWN ---> TIDYING
- 队列为空并且线程池空
- STOP --> TIDYING

源代码分析

```
//添加新任务
public void execute(Runnable command) {
    //如果任务为null直接抛出异常
    if (command == null)
        throw new NullPointerException();
    //获取当前线程池的ctl值，不知道它作用的看前面说明
    int c = ctl.get();

    //如果当前线程数小于核心线程数
    if (workerCountOf(c) < corePoolSize) {
        //添加新的工作线程执行任务，addWorker方法后面分析
        if (addWorker(command, true))
            return;
        //如果添加新的工作线程失败
        c = ctl.get();
    }

    //以下两种情况继续执行后面代码
    //1.线程数小于核心线程数，且创建线程失败
    //2.线程数大于等于核心线程数，则执行以下操作

    //线程池处于RUNNING状态，且任务成功放入队列中；
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        //再次检查线程池的状态，如果线程池状态变了，非RUNNING状态下不会接收新的任务，需要将任务移除，成功从队列中删除任务，
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)//池中无线程
        // 两种情况进入以该分支
        //1.线程池处于RUNNING状态，线程池中没有线程了，因为有新任务进入队列所以要创建工作线程（这时候新任务已经在队列中，
        //2.线程池处于非RUNNING状态但是任务移除失败,导致任务队列中仍然有任务，但是线程池中的线程数为0，则创建新的工作线程；
            addWorker(null, false);
```

```
//利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态：
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0))
```

```
// 两种情况：
// 1.非RUNNING状态拒绝新的任务
// 2.队列满了启动新的线程失败（workCount > maximumPoolSize）
}else if (!addWorker(command, false))//线程池处于RUNNING状态 ，任务入队失败
    reject(command);
```

```
private boolean addWorker(Runnable firstTask, boolean core) {
    //以下for循环，增加线程数计数，ctl，只增加计数，不增加线程，只有增加计数成功，才会增加线程
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);
        这个代码块的判断，如果是STOP，TIDYING和TERMINATED这三种状态，都会返回false。(这几种状态不会接收新任务，也不再执行队
        如果是SHUTDOWN，firstTask不为空（SHUTDOWN状态下，不会接收新任务）或 者workQueue是空（队列里面都没有任务了，也就不需
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;
        //只有满足以下两种条件才会继续创建worker线程对象
        //1.RUNNING状态，
        //2.shutdown状态，且firstTask为null（因为shutdown状态下不再接收新任务），队列不是空（shutdown状态下需要继续处理队列
         通过自旋的方式增加线程池线程数
        for (;;) {
            int wc = workerCountOf(c);
            //1.如果线程数大于最大可创建的线程数CAPACITY，直接返回false；
            //2.判断当前是要根据corePoolSize，还是maximumPoolSize进行创建线程（corePoolSize是基本线程池大小，未达到corePoo
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))//将WorkerCount通过CAS操作增加1，成功的话直接跳出两层循环；
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)//否则则判断当前线程池的状态，如果现在获取到的状态与进入自旋的状态不一致的话，那么则通
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
```

```
码块是创建Worker线程对象，并启动

orkerStarted = false;
orkerAdded = false;
= null;

ew Worker(firstTask); //创建一个新的Worker对象
 Thread t = w.thread;
 != null) {
inal ReentrantLock mainLock = this.mainLock;
ainLock.lock(); //获取线程池的重入锁后，
ry {
   // Recheck while holding lock.
   // Back out on ThreadFactory failure or if
   // shut down before lock acquired.
   int rs = runStateOf(ctl.get());

   // RUNNING状态 || SHUTDOWN状态下，没有新的任务，只是处理任务队列中剩余的任务；
   if (rs < SHUTDOWN ||
       (rs == SHUTDOWN && firstTask == null)) {
       //如果线程是活动状态，直接抛出异常，因为线程刚创建，还没有执行start方法，一定不会是活动状态；
       if (t.isAlive())
           throw new IllegalThreadStateException();
       // 将新启动的线程添加到线程池中
       workers.add(w);
       // 更新largestPoolSize的值，largestPoolSize成员变量保存线程池中创建过的线程最大数量
       int s = workers.size();
       //将线程池中创建过的线程最大数量，设置给largestPoolSize，可以通过getLargestPoolSize()方法获取，注意这
       if (s > largestPoolSize)
           largestPoolSize = s;
       workerAdded = true;
   }
 finally {
   mainLock.unlock();
```

```
            }
            // 启动新添加的线程，这个线程首先执行firstTask，然后不停的从队列中取任务执行
            // 当等待keepAlieTime还没有任务执行则该线程结束。见runWoker和getTask方法的代码。
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    }finally {
        if (! workerStarted)
            addWorkerFailed(w);
        }
        return workerStarted;
    }
```

```
private final class Worker  extends AbstractQueuedSynchronizer implements Runnable
```

```
//参数为Worker线程运行后第一个要执行的任务
Worker(Runnable firstTask) {
    //设置ASQ的state为-1  设置worker处于不可加锁的状态，看后面的tryAcquire方法，只有state为0时才允许加锁，worker线程运行以后
    setState(-1);
    //设置第一个运行的任务
    this.firstTask = firstTask;
    //创建线程，将this自己传入进去；getThreadFactory()见后面详解
    this.thread = getThreadFactory().newThread(this);
}
```

```
/** Delegates main run loop to outer runWorker  */
public void run() { runWorker( w: this); }
```

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 在构造方法里面将state设置为了-1，执行该方法就将state置为了0，这样就可以加锁了，-1状态下是无法加锁的，看Worker类的tr
    w.unlock();
    //该变量代表任务执行是否发生异常，默认值为true发生了异常，后面会用到这个变量
    boolean completedAbruptly = true;
    try {
        //如果创建worker时传入了第一个任务，则执行第一个任务，否则 从任务队列中获取任务getTask()，getTask()后面分析；
        while (task != null || (task = getTask()) != null) {
            //线程加锁
            w.lock();
            /**
             * 先判断线程池状态是否允许继续执行任务：
             * 1.如果是stop<tidying<terminated，并且线程是非中断状态
             * 2.shutingdown，runing，处于中断状态（并复位中断标志），如果这个时候其它线程执行了shutdownNow方法，shutdown
             * 这个时候则中断线程
             **/
            if ((
                    runStateAtLeast(ctl.get(), STOP) ||
                    (
                        Thread.interrupted() && runStateAtLeast(ctl.get(), STOP)
                    )

                )
                 &&
                !wt.isInterrupted())
                wt.interrupt();
```

```
/**
 *开始执行任务
 */

try {
    //任务执行前要做的处理：这个方法是空的，什么都不做，一般会通过继承ThreadPoolExecute类后重写该方法实现自己的
    beforeExecute(wt, task);
    Throwable thrown = null;
    try {
        task.run();
    } catch (RuntimeException x) {
        thrown = x; throw x;
    } catch (Error x) {
        thrown = x; throw x;
    } catch (Throwable x) {
        thrown = x; throw new Error(x);
    } finally {
        //任务执行后要做的处理：这个方法也是空的，什么都不做，一般会通过继承ThreadPoolExecute类后重写该方法实现
        afterExecute(task, thrown);
    }
} finally {
    task = null;
    //增加完成任务计数
    w.completedTasks++;
    w.unlock();
}
```

```
        /**
         *退出while循环，线程结束；
         **/

        //判断task.run()方法是否抛出了异常，如果没有则设置它为false，如果发生了异常，前面会直接抛出，中断方法继续执
        completedAbruptly = false;
    } finally {
        /**
         * 线程退出后的处理
         */
        processWorkerExit(w, completedAbruptly);
    }
```

```
private Runnable getTask() {
    // 记录for循环中，workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)操作是否超时，如果超时，在下次执行for循环时根据ti
    boolean timedOut = false;

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);
        /**
        * 判断线程池状态是否允许继续获取任务：
        * RUNNING<shutdown<stop<tidying<terminated;
        *1.如果是stop、tidying、terminated；这时不再处理队列中的任务，直接返回null
        *2.如果是rs = SHUTDOWN ，rs>=STOP不成立，这时还需要处理队列中的任务除非队列为空，没有任务要处理，则返回null
        */
        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            //自旋锁将ctl减1（也就是将线程池中的线程数减1）
            decrementWorkerCount();
            return null;
        }

        /***
         *以下做线程超时控制
         */

        //获取线程池中线程数量
        int wc = workerCountOf(c);
```

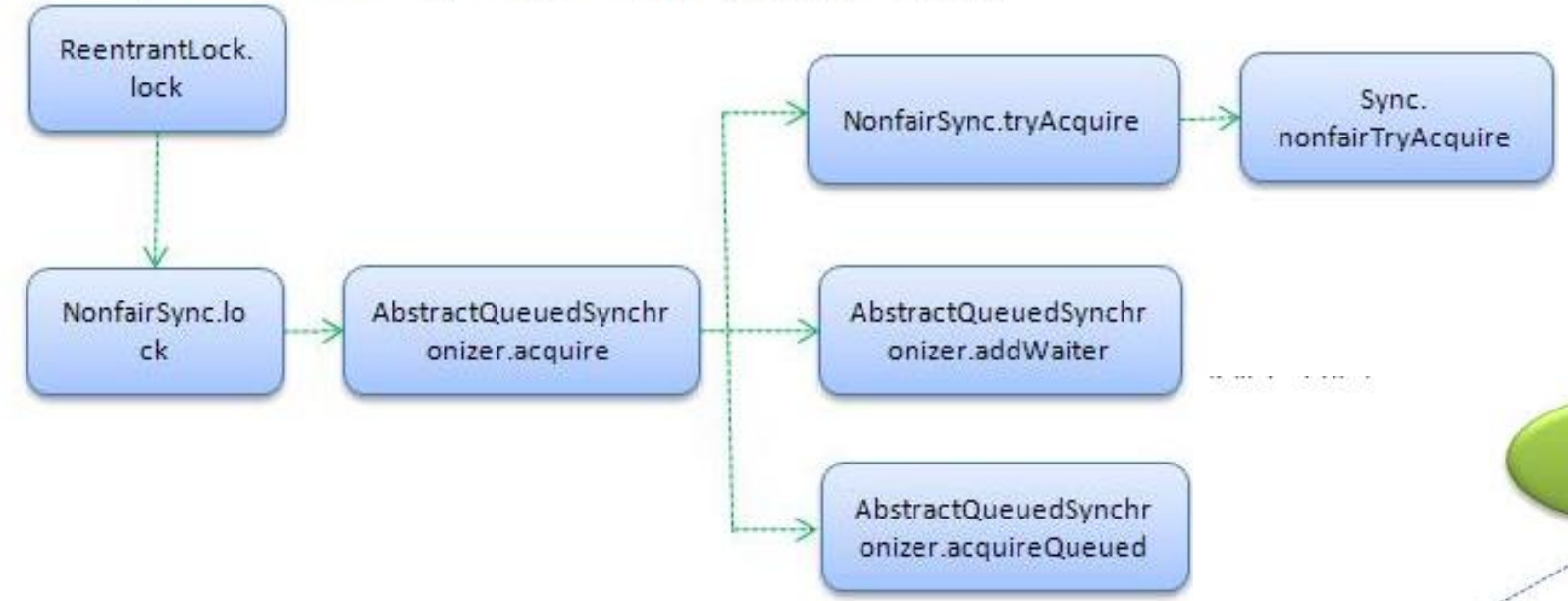
```
    /**
     *取任务操作
     */
    try {
    //根据timed来判断执行poll操作还是执行take()操作还是执行有时间限制
        Runnable r = timed ?
            workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
            workQueue.take();
        if (r != null)
            return r;
        //如果poll操作等待超时,没有取到任务；
        timedOut = true;
    } catch (InterruptedException retry) {
        //如果是因为线程中断导致没有取到任务；
        timedOut = false;
    }
}
}
```

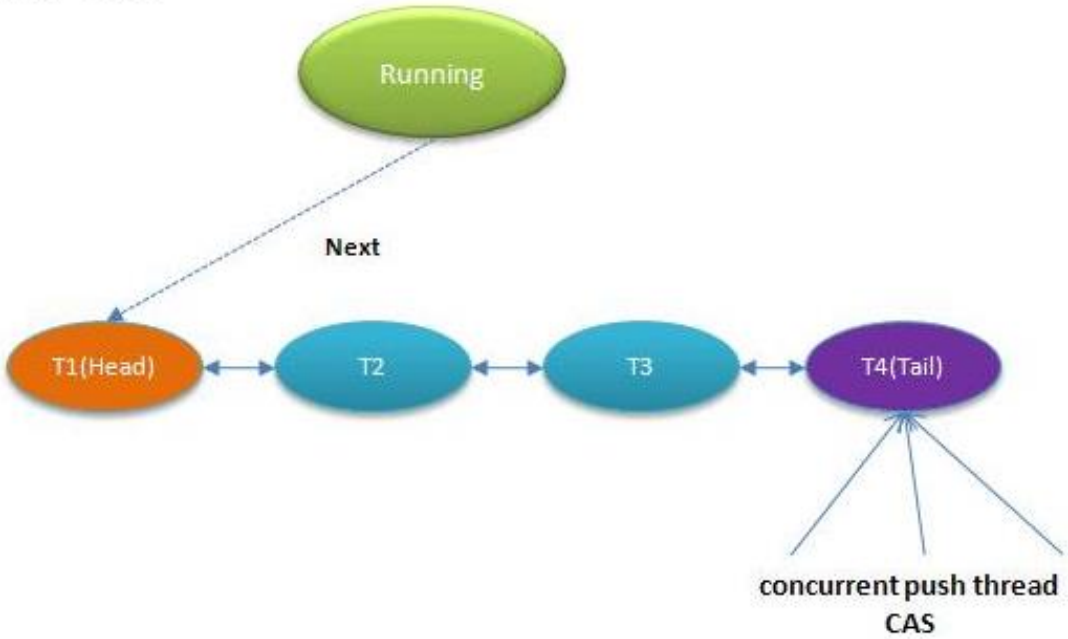
```
// timed变量用于判断是否需要进行超时控制。
// allowCoreThreadTimeOut默认是false，也就是核心线程不允许进行超时；
// wc > corePoolSize，表示当前线程池中的线程数量大于核心线程数量；
// 对于超过核心线程数量的这些线程，需要进行超时控制；
boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

/*
* wc > maximumPoolSize的情况是因为可能在此方法执行阶段同时执行了setMaximumPoolSize方法；
* timed && timedOut 如果为true，表示当前操作需要进行超时控制，并且上次循环从任务队列中获取任务发生了超时
* 接下来判断，如果线程数量大于1，或者线程队列是空的，那么尝试将workerCount减1；
* 如果减1失败，则返回重试。
* 如果wc == 1时，也就说明当前线程是线程池中唯一的一个线程了。
*/
if ((wc > maximumPoolSize || (timed && timedOut))
    && (wc > 1 || workQueue.isEmpty())) {
    //尝试将线程池线程数量减一
    if (compareAndDecrementWorkerCount(c))
        return null;
    //如果将线程池数量减一不成功则循环重试
    continue;
}
```

- 参考文献
- http://blog.csdn.net/zqz_zqz/article/details/69488570?locationNum=12&fps=1
- http://www.blogjava.net/xylz/archive/2010/07/08/325587.html
- http://blog.csdn.net/baidu_37107022/article/details/77415936

ReentrantLock的实现

先理一下Reentrant.lock()方法的调用过程（默认非公平锁）：





```
    */
    public ReentrantLock() {
        sync = new NonfairSync();
    }


    */
  public void lock() { sync.lock(); }

static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    /**
     * Performs lock.  Try immediate barge, backing up to normal
     * acquire on failure.
     */
    final void lock() {
        if (compareAndSetState( expect: 0,  update: 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire( arg: 1);
    }

    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}

  */
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

该方法会首先判断当前状态，如果c==0说明没有线程正在竞争该锁，如果不c !=0 说明有线程正拥有了该锁。

如果发现c==0，则通过CAS设置该状态值为acquires,acquires的初始调用值为1，每次线程重入该锁都会+1，每次unlock都会-1，但为0时释放锁。如果CAS设置成功，则可以预计其他任何线程调用CAS都不会再成功，也就认为当前线程得到了该锁，也作为Running线程，很显然这个Running线程并未进入等待队列。

如果c !=0 但发现自己已经拥有锁，只是简单地++acquires，并修改status值，但因为没有竞争，所以通过setStatus修改，而非CAS，也就是说这段代码实现了偏向锁的功能

```java
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

其中参数mode是独占锁还是共享锁，默认为null，独占锁。追加到队尾的动作分两步：
1.如果当前队尾已经存在(tail!=null)，则使用CAS把当前线程更新为Tail
2.如果当前Tail为null或则线程调用CAS设置队尾失败，则通过enq方法继续设置Tail

```java
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

该方法就是循环调用CAS，即使有高并发的场景，无限循环将会最终成功把当前线程追加到队尾（或设置队头）。总而言之，addWaiter的目的就是通过CAS把当前现在追加到队尾，并返回包装后的Node实例

- 把线程要包装为Node对象的主要原因，除了用Node构造供虚拟队列外，还用Node包装了各种线程状态，这些状态被精心设计为一些数字值：
- SIGNAL(-1)：线程的后继线程正/已被阻塞，当该线程release或cancel时要重新这个后继线程(unpark)
- CANCELLED(1)：因为超时或中断，该线程已经被取消
- CONDITION(-2)：表明该线程被处于条件队列，就是因为调用了Condition.await而被阻塞
- PROPAGATE(-3)：传播共享锁
- 0：0代表无状态

```java
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

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

```java
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        /*
         * This node has already set status asking a release
         * to signal it, so it can safely park.
         */
        return true;
    if (ws > 0) {
        /*
         * Predecessor was cancelled. Skip over predecessors and
         * indicate retry.
         */
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        /*
         * waitStatus must be 0 or PROPAGATE.  Indicate that we
         * need a signal, but don't park yet.  Caller will need to
         * retry to make sure it cannot acquire before parking.
         */
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

检查原则在于：

•规则1：如果前继的节点状态为SIGNAL，表明当前节点需要unpark，则返回成功，此时acquireQueued方法的第12行（parkAndCheckInterrupt）将导致线程阻塞

•规则2：如果前继节点状态为CANCELLED(ws>0,即CANCELLED(1)：因为超时或中断，该线程已经被取消)，说明前置节点已经被放弃，则回溯到一个非取消的前继节点，返回false，acquireQueued方法的无限循环将递归调用该方法，直至规则1返回true，导致线程阻塞

•规则3：如果前继节点状态为非SIGNAL、非CANCELLED，则设置前继的状态为SIGNAL（初始化是0），返回false后进入acquireQueued的无限循环，与规则2同

总体看来，shouldParkAfterFailedAcquire就是靠前继节点判断当前线程是否应该被阻塞，如果前继节点处于CANCELLED状态，则顺便删除这些节点重新构造队列

```
public void unlock() {
    sync.release( arg: 1);
}

*/
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

tryRelease语义很明确：如果线程多次锁定，则进行多次释放，直至status==0则真正释放锁，所谓释放锁即设置status为0，因为无竞争所以没有使用CAS。

release的语义在于：如果可以释放锁，则唤醒队列第一个线程（Head）

```java
private void unparkSuccessor(Node node) {
    /*
     * If status is negative (i.e., possibly needing signal) try
     * to clear in anticipation of signalling.  It is OK if this
     * fails or if status is changed by waiting thread.
     */
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, update: 0);

    /*
     * Thread to unpark is held in successor, which is normally
     * just the next node.  But if cancelled or apparently null,
     * traverse backwards from tail to find the actual
     * non-cancelled successor.
     */
    Node s = node.next;
    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;
    }
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

这段代码的意思在于找出第一个可以unpark的线程，一般说来head.next == head，Head就是第一个线程，但Head.next可能被取消或被置为null，因此比较稳妥的办法是从后往前找第一个可用线程。貌似回溯会导致性能降低，其实这个发生的几率很小，所以不会有性能影响。之后便是通知系统内核继续该线程，在Linux下是通过pthread_mutex_unlock完成。之后，被解锁的线程进入上面所说的重新竞争状态（重新执行acquireQueued方法）

- 参考文献
- 1、http://blog.csdn.net/Luxia_24/article/details/52403033

在JUC锁机制(Lock)学习笔中，我们了解到AQS有一个队列，同样Condition也有一个等待队列，两者是相对独立的队列，因此一个Lock可以有多个Condition，Lock(AQS)的队列主要是阻塞线程的，而Condition的队列也是阻塞线程，但是它是有阻塞和通知解除阻塞的功能

Condition阻塞时会释放Lock的锁

await()就是在当前线程持有锁的基础上释放锁资源，并新建Condition节点加入到Condition的队列尾部，阻塞当前线程

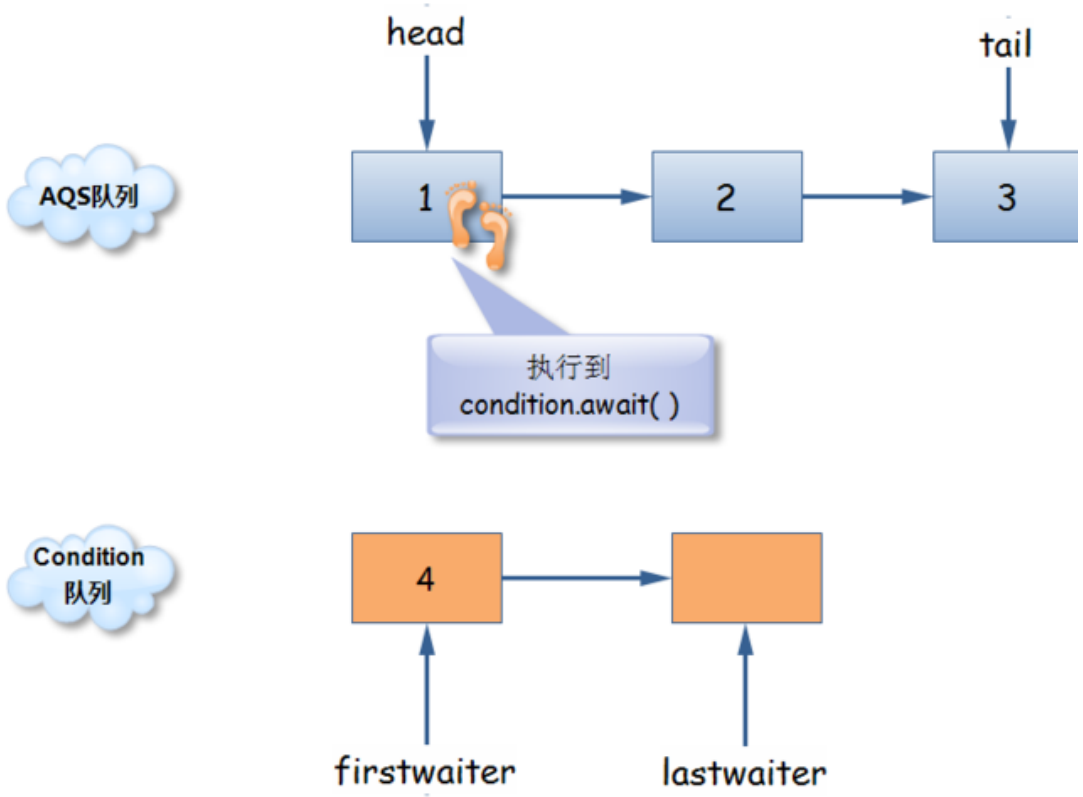
signal()就是将Condition的头节点移动到AQS等待节点尾部，让其等待再次获取锁

```
 */
public Condition newCondition() { return sync.newCondition(); }
```

```
final ConditionObject newCondition() {
    return new ConditionObject();
}
```

```
private transient Node firstWaiter;
/** Last node of condition queue. */
private transient Node lastWaiter;
```

I.初始化状态：AQS等待队列有3个Node，Condition队列有1个Node(也有可能1个都没有)



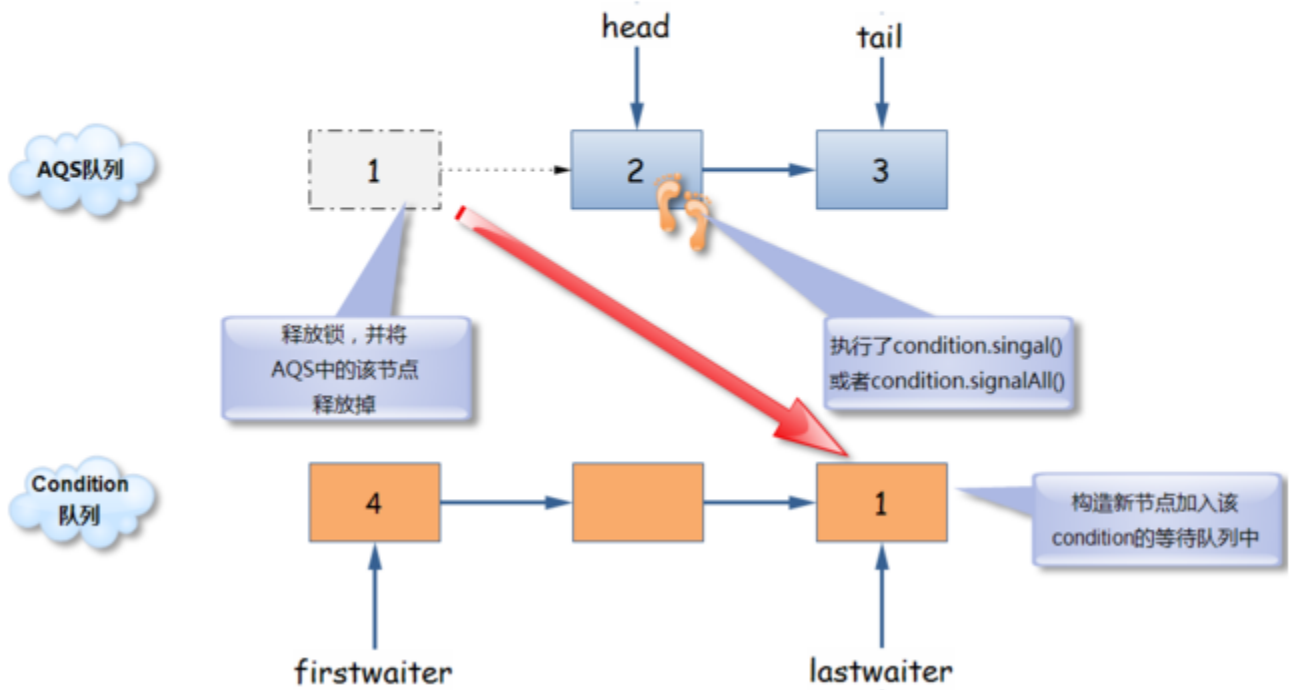
II.节点1执行Condition.await()

1.将head后移

2.释放节点1的锁并从AQS等待队列中移除

3.将节点1加入到Condition的等待队列中

4.更新lastWaiter为节点1



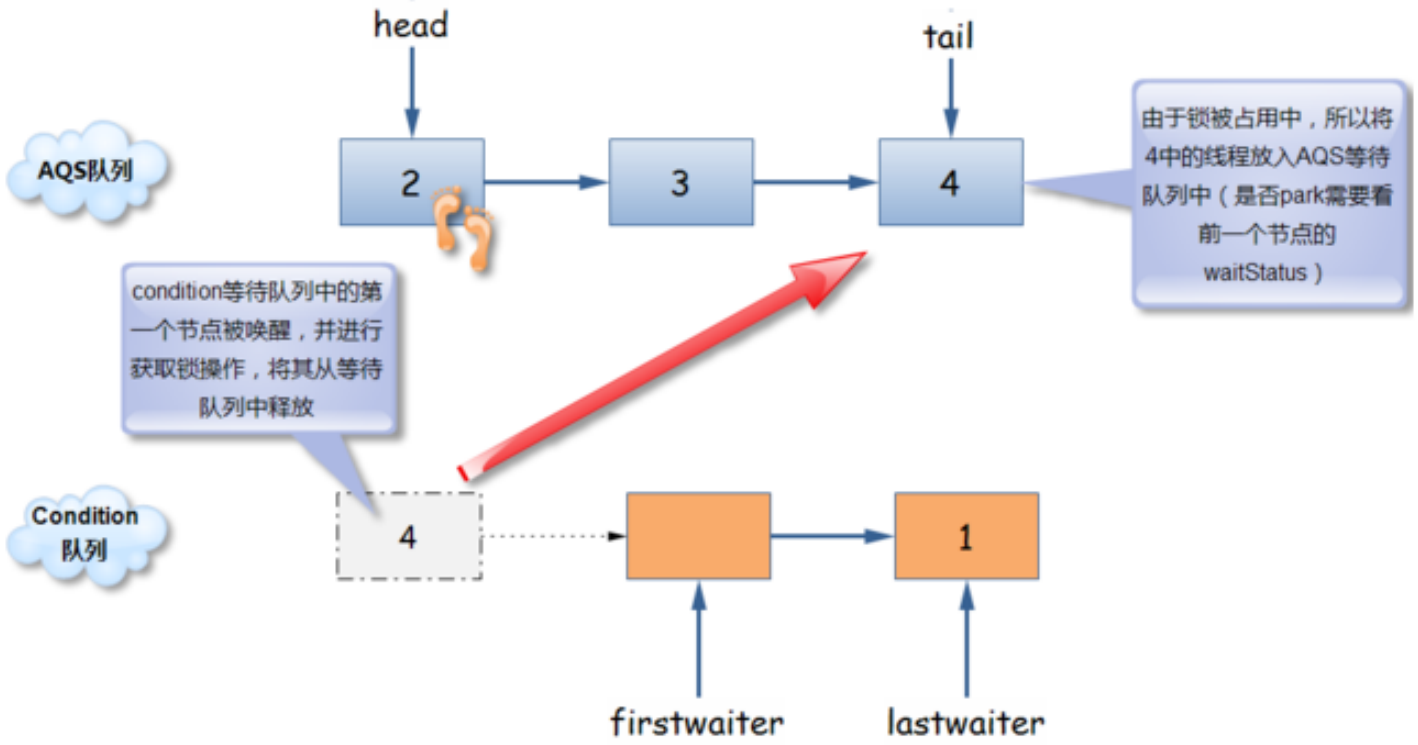
III.节点2执行signal()操作

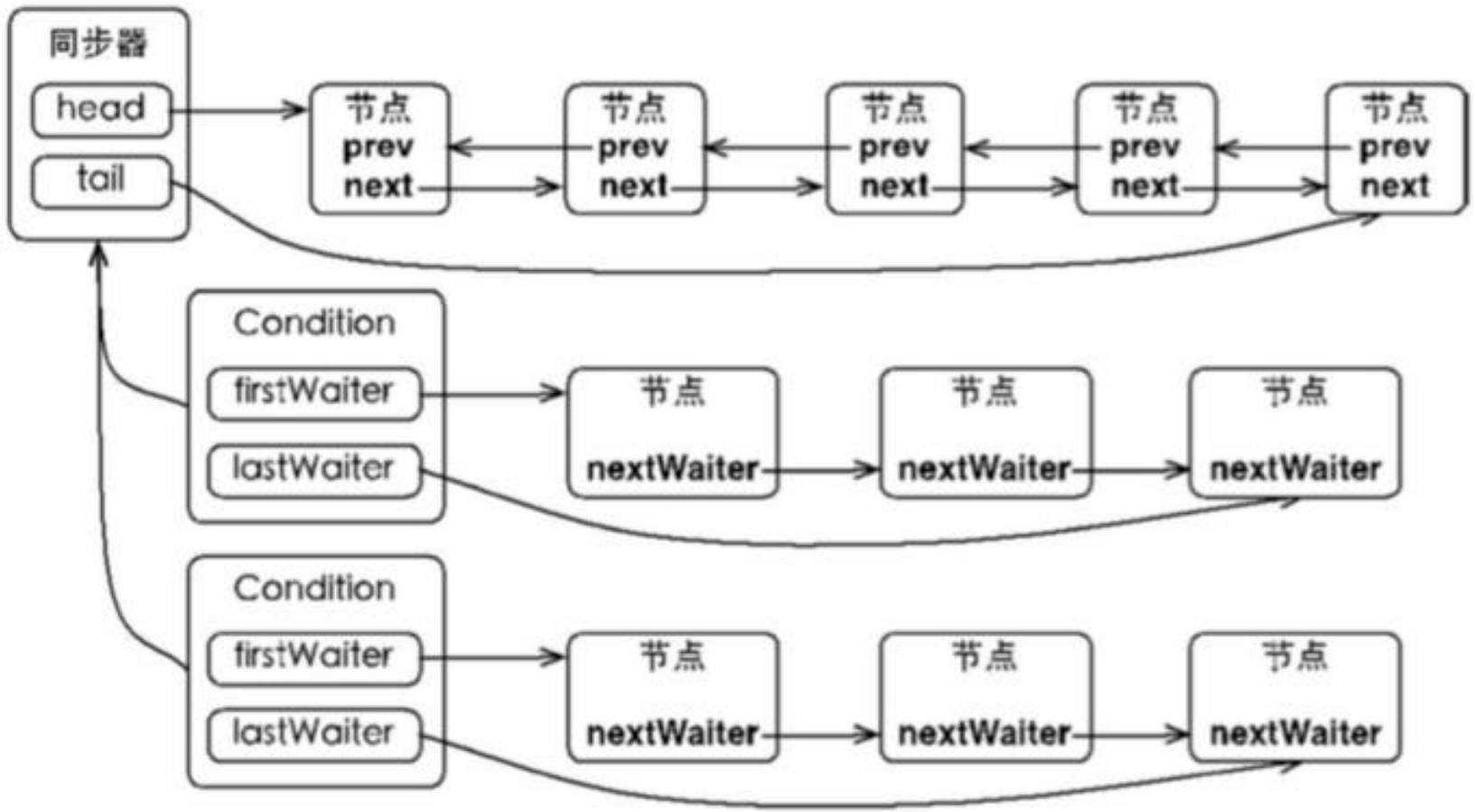
5.将firstWaiter后移

6.将节点4移出Condition队列

7.将节点4加入到AQS的等待队列中去

8.更新AQS的等待队列的tail




同步器
head
tail
节点
prev
next
节点
prev
next
节点
prev
next
节点
prev
next
节点
prev
next
Condition
firstWaiter
lastWaiter
节点
nextWaiter
节点
nextWaiter
节点
nextWaiter
Condition
firstWaiter
lastWaiter
节点
nextWaiter
节点
nextWaiter
节点
nextWaiter

```java
public final void await() throws InterruptedException {
    // 1.如果当前线程被中断，则抛出中断异常
    if (Thread.interrupted())
        throw new InterruptedException();
    // 2.将节点加入到Condition队列中去，这里如果lastWaiter是cancel状态，那么会把它踢出Condition队列。
    Node node = addConditionWaiter();
    // 3.调用tryRelease，释放当前线程的锁
    long savedState = fullyRelease(node);
    int interruptMode = 0;
    // 4.为什么会有在AQS的等待队列的判断？
    // 解答：signal操作会将Node从Condition队列中拿出并且放入到等待队列中去，在不在AQS等待队列就看signal是否执行
    // 如果不在AQS等待队列中，就park当前线程，如果在，就退出循环，这个时候如果被中断，那么就退出循环
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
    }
    // 5.这个时候线程已经被signal()或者signalAll()操作给唤醒了，退出了4中的while循环
    // 自旋等待尝试再次获取锁，调用acquireQueued方法
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null)
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

ReentrantLock是独占锁，一个线程拿到锁后如果不释放，那么另外一个线程肯定是拿不到锁，所以在lock.lock()和lock.unlock()之间可能有一次释放锁的操作（同样也必然还有一次获取锁的操作）。在进入lock.lock()后唯一可能释放锁的操作就是await()了。也就是说await()操作实际上就是释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁！

```java
private Node addConditionWaiter() {
    Node t = lastWaiter;
    // If lastWaiter is cancelled, clean out.
    if (t != null && t.waitStatus != Node.CONDITION) {
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    Node node = new Node(Thread.currentThread(), Node.CONDITION);
    if (t == null)
        firstWaiter = node;
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```

添加条件等待节点，根据链表的特征，直接在尾部节点的nextWaiter指向新建的节点，并将新建的节点设置为整个链表的尾部，首先要知道如下数据结构：

```
object {
        Node firstWaiter;
        Node lastWaiter;
        node {
            node nextWaiter;
           该节点承载的业务数据，比如这里的
Thread t;等
        }
    }
```

知道上述结构，其实整个链的数据维护，基本一目了然，自己都可以实现下面的逻辑。

代码@1,如果最后一个等待节点的状态不是Node.CONDITION,则，则先删除等待链中节点状态不为Node.CONDITION的节点。具体代码分析请参照下文unlinkCancelledWaiters的解读。

代码@2开始，就是普通链表的节点添加的基本方法

```
 ,
private void unlinkCancelledWaiters() {
    Node t = firstWaiter;  //
    Node trail = null;       //@1
    while (t != null) {
        Node next = t.nextWaiter;
        if (t.waitStatus != Node.CONDITION) {  // @3
            t.nextWaiter = null;
            if (trail == null)                          // @4
                firstWaiter = next;
            else
                trail.nextWaiter = next;       //@5
            if (next == null)    // @6
                lastWaiter = trail;
        }
        else   // @4
            trail = t;
        t = next;
    }
}
```

该方法的思路为，从第一节点开始，将不等于Node.CONDITION的节点。
代码@1,设置尾部节点临时变量，用来记录最终的尾部节点。
代码@1,第一次循环，是循环第一个节点，如果它的状态为Node.CONDITION,则该链的头节点保持不变，设置临时尾节点为t,然后进行一个节点的判断，如果节点不为Node.CONDITION, 重置头节点的下一个节点，或尾部节点的下一个节点(@4,@5)。代码@6代表整个循环结束，设置 ConditionObject对象的lastWaiter为trail的值

- 参考文献
- 1、http://www.jianshu.com/p/be2dc7c878dc
- 2、http://blog.csdn.net/prestigeding/article/details/53158246
- 3、http://www.cnblogs.com/cm4j/p/juc_condition.html

```java
public ArrayBlockingQueue(int capacity) {
    this(capacity, fair: false);
}

*/
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}

*/
public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        insert(e);
    } finally {
        lock.unlock();
    }
}
*/
private void insert(E x) {
    items[putIndex] = x;
    putIndex = inc(putIndex);
    ++count;
    notEmpty.signal();
}
```

```java
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return extract();
    } finally {
        lock.unlock();
    }
}

    call only when holding lock.
*/
private E extract() {
    final Object[] items = this.items;
    E x = this.<E>cast(items[takeIndex]);
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
}
```

```java
private Object handleDirectValue(final long startTime, final MethodInvoker methodInvoker, final Cacheable cacheable,
        final String cacheKey, Object cacheValue, Cache<String, Object> cache) throws Throwable {
    final Method method = methodInvoker.getMethod();

    try {
        readWriteLock.readLock().lock(); // 获取读锁

        if (cacheValue == null) {
            readWriteLock.readLock().unlock(); // 获取写锁之前先释放读锁
            readWriteLock.writeLock().lock(); // 获取写锁
            try {
                cacheValue = getCacheValue(method, cacheKey, cache);
                if (cacheValue == null) {
                    cacheValue = invokeMethodAndCache(startTime, methodInvoker, method, cacheable, cacheKey, cache);
                } else {
                    logGetValueFromCache(startTime, method, cacheKey, cache);
                }
            } finally {
                readWriteLock.readLock().lock(); // 释放写锁之前先获取读锁
                readWriteLock.writeLock().unlock(); // 释放写锁
            }

        } else {
            logGetValueFromCache(startTime, method, cacheKey, cache);
        }
    } finally {
        readWriteLock.readLock().unlock(); // 释放读锁
    }

    if (cacheValue instanceof CacheObject) {// 这行代码必须加，否则高并发下，刚开始部分可能会出现类型强转异常，因为getCacheValue可能是返回的CacheObject类型的数据
        return ((CacheObject) cacheValue).getValue();
    }

    return cacheValue;
}
```