目录

[1、 Thread 3](#_Toc5169)

[1.1、 Thread()、Thread(Runnable target) 3](#_Toc10533)

[1.2、 synchronized void start() 3](#_Toc23022)

[1.3、 final void join() 4](#_Toc17620)

[1.4、 void interrupt() 4](#_Toc26578)

[1.5、 boolean isInterrupted() 4](#_Toc377)

[1.6、 static boolean interrupted() 4](#_Toc25488)

[1.7、 static native void yield() 5](#_Toc19284)

[1、 AbstractQueuedSynchronizer 5](#_Toc22621)

[1.1、 compareAndSetXXX() 5](#_Toc22771)

[1.2、 void acquire(int arg) 5](#_Toc16912)

[1.3、 boolean tryAcquire(int arg) 5](#_Toc24648)

[1.4、 boolean release(int arg) 6](#_Toc27238)

[1.5、 void acquireInterruptibly(int arg) 6](#_Toc2406)

[1.6、 acquireShared(int arg) 6](#_Toc28999)

[2、 ReentrantLock 6](#_Toc13362)

[2.1、ReentrantLock() 6](#_Toc24631)

[2.2、ReentrantLock(boolean fair) 7](#_Toc4234)

[2.3、void lock() 7](#_Toc27230)

[2.4、void unlock() 8](#_Toc17581)

[2.5、内部类ConditionObject 8](#_Toc17869)

[2.6、void lockInterruptibly() 10](#_Toc31729)

[2.7、boolean tryLock() 10](#_Toc12267)

[2.8、boolean tryLock(long timeout, TimeUnit unit) 10](#_Toc18432)

[3、 ReentrantReadWriteLock 10](#_Toc32292)

[3.1、ReentrantReadWriteLock() 11](#_Toc20796)

[3.2、ReentrantReadWriteLock(boolean fair) 11](#_Toc25948)

[3.3、ReentrantReadWriteLock.ReadLock readLock() 11](#_Toc30451)

[3.4、ReentrantReadWriteLock.WriteLock writeLock() 11](#_Toc13749)

[3.5、内部类ReadLock 11](#_Toc18062)

[3.6、内部类WriteLock 13](#_Toc30105)

[4、 ArrayBlockingQueue 14](#_Toc10466)

[4.1、ArrayBlockingQueue(int capacity) 14](#_Toc24124)

[4.2、ArrayBlockingQueue(int capacity, boolean fair) 14](#_Toc24385)

[4.3、boolean add(E e) 14](#_Toc3301)

[4.4、boolean contains(Object o) 15](#_Toc30223)

[4.5、void put(E e) 15](#_Toc6078)

[4.6、E poll() 15](#_Toc16600)

[4.7、E peek() 15](#_Toc27141)

[4.8、boolean remove(Object o) 16](#_Toc21564)

[4.9、E take() 16](#_Toc10955)

[5、 LinkedBlockingQueue 16](#_Toc21557)

[5.1、LinkedBlockingQueue() 16](#_Toc19109)

[5.2、LinkedBlockingQueue(int capacity) 16](#_Toc24954)

[5.3、boolean offer(E e) 16](#_Toc30721)

[5.4、E peek() 17](#_Toc1805)

[5.5、E poll() 17](#_Toc30455)

[5.6、void put(E e) 18](#_Toc209)

[5.7、boolean remove(Object o) 18](#_Toc16792)

[5.8、E take() 18](#_Toc12445)

[6、 PriorityBlockingQueue 18](#_Toc12207)

[6.1、PriorityBlockingQueue(int initialCapacity,Comparator<? super E> comparator) 19](#_Toc28038)

[6.2、boolean add(E e) 19](#_Toc25741)

[6.3、boolean offer(E e) 19](#_Toc2249)

[6.4、void clear() 20](#_Toc28494)

[6.5、Iterator<E> iterator() 20](#_Toc13442)

[6.6、E peek() 20](#_Toc25421)

[6.7、E poll() 20](#_Toc24995)

[6.8、void put(E e) 21](#_Toc29873)

[6.9、boolean remove(Object o) 21](#_Toc18123)

[7.0、E take() 21](#_Toc20062)

## Thread

①创建线程时，没有指定线程名称的话，会生成默认线程名称"Thread-" + nextThreadNum()；nextThreadNum()是同步方法，不会出现线程安全问题，nextThreadNum()被调用一次，维护的静态的int型成员变量threadInitNumber就会自增长1；

②sleep()让线程睡眠，交出CPU，让CPU去执行其他的任务。sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。sleep方法相当于让线程进入阻塞状态。

③调用yield()方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。

注意：当线程的中断标识为true时，该线程是不能被LockSupport.park()挂起的，并且，如果线程被LockSupport.park()挂起，如果此时该线程的中断标识被置为true，该线程会从挂起状态转为可运行状态；

### Thread()、Thread(Runnable target)

构造方法

①按"Thread-" + nextThreadNum()；生成此线程的名称，nextThreadNum()是同步静态方法，获取静态的int型成员变量threadInitNumber的值；

②将第一步生成的名称赋值给成员变量name；

③获取当前正在执行线程的引用作为父线程；若是在main方法中创建的线程，则此时返回的就是主线程的引用；

④将父线程的线程组作为本线程的线程组，并将线程组的未启动线程数加一；

⑤如果父线程是守护线程则将本线程也是守护线程（将父线程的daemon布尔变量值赋值给本线程的daemon）；

⑥本线程的优先级也等于父线程的优先级；

⑦如果新建线程时传入了Runnable的实现，则将实现赋值到成员变量target中；

⑧将成员变量stackSize设置为0；

⑨调用nextThreadID()生成线程ID。

### synchronized void start()

方法做用：导致此线程开始执行; Java虚拟机调用此线程的run方法。

①检查线程状态threadStatus是否为0，不是，则抛出非法线程状态异常；

②通知组此线程即将start，以便可以将其添加到组的线程列表中并且可以减少组的unstarted线程的计数；

③调用本地方法start0()启动线程；

④如果线程启动报错，则将此线程从线程组中移出。

### final void join()

方法做用：假设在线程1中调用线程2的join()，线程1会被挂起，直到线程2死亡（执行完毕），线程1才会继续执行下去

①方法内部的原理就是一个while循环，执行条件是isAlive()方法（即线程2还活着），执行语句是wait()（及线程2的wait()方法）；其实就是在线程2死亡之前，线程1一直在执行这个while循环，直到线程2死亡，线程1 就跳出循环继续向下执行。

②如果在调用join()时传入了参数，单位为毫秒的时间值，则线程1只会被挂起这么长时间，无论此时线程2是否死亡，线程1都能跳出我还while循环，继续往下执行

### void interrupt()

方法作用：将线程的中断标识置为true，并不是直接中断线程，线程会继续执行

注意：

①在某线程调用此方法时，如果该线程正在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法，或者自身的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法并在阻塞中，则这些阻塞方法将会抛出InterruptedException异常，阻塞将会中断，当异常被捕获，此线程的中断状态将被清除（置为false）。

②在某线程调用此方法时（在其他线程中调用），如果该线程已经调用LockSupport.park()被挂起，也会立即被唤醒；

### boolean isInterrupted()

方法作用：返回调用此方法线程的中断标识状态

### static boolean interrupted()

方法作用：返回当前线程的中断标识状态，并清除中断状态（置为false）

### static native void yield()

方法作用：使当前线程从运行状态转为就绪状态，并和线程池中相同优先级的线程共同抢夺cpu资源，有可能还是当前线程抢到

## AbstractQueuedSynchronizer

AQS是Java中锁的基础，主要由两个队列组成。一个队列是同步队列，另一个是等待队列（等待队列可以有多个）；AQS支持实现独占锁和共享锁。

### compareAndSetXXX()

方法作用：这是一组方法，通过CAS修改变量的值，在多线程下被修改的变量是线程安全的；

### void acquire(int arg)

方法作用：以独占模式获取锁资源

①当前线程尝试抢夺锁资源（及调用tryAcquire方法），如果成功，方法直接结束；

②如果抢夺锁资源失败，则将当前线程新建为节点再从同步队列队尾入队；

③判断当前线程的前接点是否为首节点，如果是就进行锁资源的抢夺，如果不是就被挂起等待被唤醒；如果抢夺锁资源成功，则将自己设置为首节点，如果失败，则挂起等待被唤醒再次抢夺锁资源；

④直达抢夺到资源方法才能退出；

### boolean tryAcquire(int arg)

方法作用：以独占模式尝试获取锁资源，成功返回true，失败返回false；

注意：此方法直接抛出异常，需要子类重写此方法，实现抢夺锁资源的逻辑；

### boolean release(int arg)

方法作用：以独占模式释放锁资源

①尝试释放锁资源，及调用tryRelease()方法，此方法需要子类重写来实现释放逻辑；如果释放成功并且此时锁资源全部释放完（及state=0），则tryRelease()方法返回true，否则返回false；

②如果第①步返回的是true，说明锁资源空闲，则唤醒等待队列中的下一个节点（线程），整个方法结束并返回true；如果第①步返回的是true，则整个方法结束并返回false；

### void acquireInterruptibly(int arg)

方法作用：以独占模式获取，如果线程中断则中止。

注意：此方法与acquire(int arg)唯一的不同点在于：当被挂起的线程醒来后发现自己的中断标识位为true时，会直接抛出中断异常，则当前线程会退出锁资源的抢夺；而在acquire(int arg)方法中，如果被挂起的线程醒来后发现自己的中断标识位为true时，会继续抢夺锁资源，不会退出。

### acquireShared(int arg)

方法作用：在共享模式下获取锁资源

①

## ReentrantLock

跟synchronized相同，是可重入的重量级锁，并且是悲观锁、独占锁（排他锁）。但是其用法则相当不同，首先ReentrantLock要显式的调用lock方法表示接下来的这段代码已经被当前线程锁住，其他线程需要执行时需要拿到这个锁才能执行，而当前线程在执行完之后要显式的释放锁。

### 2.1、ReentrantLock()

方法作用：无参构造方法，默认新建非公平同步器

①调用内部类NonfairSync无参构造赋值给成员变量sync；

②依次调用父类的默认构造函数，各级父类无参构造函数没有做任何事情；

### 2.2、ReentrantLock(boolean fair)

方法作用：带布尔参数构造方法，fair为true，创建公平同步器，否则创建非公平同步器

### 2.3、void lock()

方法作用：获得锁

假如创建的是非公平同步器：

根据假设，本方法实际就是调用了内部类NonfairSync（非公平同步器

）的lock()方法；

调用本方法的的当前线程就是要抢占本锁的锁资源；

首先要将NonfairSync的成员变量state设置为1，条件是此时state的值必须是0（没有其他线程占用锁资源），这个过程在本地方法中完成，并且是同步的；

如果成功将state设置为1，则说明当前线程抢占到锁资源，则将NonfairSync的成员变量exclusiveOwnerThread赋值为当前线程，exclusiveOwnerThread的作用是保存占用锁资源的线程，此时整个方法执行完毕；

如果没有抢占到锁资源（设置state失败），则获取当前state的值，如果state=0（此时没有线程占用锁资源），则再次尝试将state的值设置为1，如果成功，则将NonfairSync的成员变量exclusiveOwnerThread赋值为当前线程，此时整个方法执行完毕；如果state=1（此时有线程占用锁资源），则判断是否是当前线程自己占用了锁资源，如果是，则将state的值加1（重入锁，多次获取锁资源），此时整个方法执行完毕；

引用：如果再次设置state的值失败，则将当前线程保存进新建的节点中（并且设置nextWaiter（写一个等待节点）为null），并放到节点双向链表的队尾（若此时链表中没有节点，则需要初始化链表，新建一个无参数（无参构造）的节点作为首节点，此过程也是在本地方法中进行的，是同步的，可能失败，失败原因是其他线程率先对链表进行了初始化），此过程也是写在本地方法中的，是同步的，如果失败则一直尝试，直到成功为止；

接下来，判断新建的当前线程节点的前节点是否为首部节点，如果是，则当前线程再次抢夺锁资源，且成功了，则将新建节点设置为首节点，并将新建节点的线程和前节点置为null，将老首节点的下一个节点置为null，并返回false，此时整个方法执行完毕；如果当前线程节点的前节点不是首部节点，或者当前线程节点的前节点是首部节点但当前线程再次抢夺锁资源失败；此时需要判断是否需要挂起当前线程，判断新建节点的前节点的成员变量waitStatus是否等于SIGNAL（此状态的节点会在释放锁资源后唤醒后节点），如果是，则直接返回true（当前线程将被挂起）；如果waitStatus大于0（表示该节点已经失效），则进入循化，依次判端当前线程节点之前的节点waitStatus是否大于0，如果是则将此节点踢出双向链表（队列），直到遇到waitStatus不大于0的节点，将此节点作为当前线程节点的前节点，返回false，当前线程不挂起，继续循环抢夺资源，直到抢夺成功或者被挂起；如果waitStatus为其他值，则将前节点的waitStatus设置为SIGNAL（此过程是在本地方法中进行，是同步的），当前线程不挂起，继续循环抢夺资源，直到抢夺成功或者被挂起；

只有抢夺资源成功，此方法才会正常退出。

但是如果在抢夺到资源之前程序抛出了异常（暂且将抛出异常的线程成为当前线程），则将取消当前线程抢夺资源的资格：将当前线程节点的成员变量thread置为null，并唤醒下一个节点。

当挂起的当前线程被唤醒后，获取当前线程的中断状态并返回，再清除线程的中断状态（置为false），如果获取到的线程的中断状态为true（说明在当前线程被中断过，及interrupt()方法被调用过），则在抢夺到资源成功后会将自己的中断状态置为true；

假如创建的是公平同步器：

根据假设，本方法实际就是调用了内部类FairSync（公平同步器

）的lock()方法；

获取当前state的值，如果state=0（此时没有线程占用锁资源），则再判断队列中（双向链表）是否有线程排在当前线程之前并且还在等待锁资源抢夺，如果有，则当前线程不进行资源的抢夺，后续流程与非公平同步器流程中蓝色引用字样后的内容相同；

### 2.4、void unlock()

方法作用：释放锁

无论是非公平同步器还是公平同步器，本方法实际就是调用了AQS的release(int arg)方法，参数arg默认为1；

先检查当前线程是否是占用资源的线程，如果不是，直接抛出异常；（我认为是防止非锁资源占用线程调用此方法）

用state减去arg，将结果赋值给state，判断结果是否为0，如果是0（锁资源被当前线程释放），则将成员变量exclusiveOwnerThread（是独占锁资源的线程的引用）置为空，然后唤醒同步队列中下一个waitStatus <= 0的节点。

### 2.5、内部类ConditionObject

作用：将等待获取资源的线程独立出来分队，新建一个ConditionObject，就是一个等待队列，可以有多个等待队列。

此类只能在同步块中使用，也是就是说，只在lock()和unlock()之间使用，因为此类中的public方法在执行时都会检查当前线程是否是占用锁资源的线程，不是的话会直接抛出异常，所以在同步块之外使用此类是没有意义的。

2.5.1、void await()

方法作用：

①如果当前线程的中断标志位为true，直接抛出中断异常；（还未清楚此处的作用）；

②将当前线程新建为节点（并将等待状态设置为CONDITION），插入到等待队列的尾部，在插入之前先判断此时等待队列的尾部节点的等待状态是否为是CONDITION（进入等待队列时设置的），如果不是，则将从等待队列首部开始，逐一将等待状态不为CONDITION的等待节点剔除出队列，再将新节点加入队尾；

③将当前线程占用的锁资源全部释放（注意：在释放锁的过程中会校验当前线程是否是暂用锁资源的线程，不是的话会抛出非法监视器状态异常，在此也印证了ConditionObject只能在同步块中使用），过程见2.4中release(int arg)方法的讲解，并记录释放的资源次数savedState；

④判断新建的节点是否在同步队列中（此节点是在第②步新建的节点，肯定不在同步队列中啊？解答：在做此判断时，当前线程已经将锁资源全部释放了，其他线程是有可能在抢夺到锁资源后调用ConditionObject的signal()或signalAll()将当前线程所在的等待节点移动到同步节点中的），如果没有在同步队列，则将当前线程挂起；一直等到被唤醒，被唤醒后（注意：此处被唤醒有两种可能：1、被挂起的线程中断标识由false变为true；2、当处于等待队列的线程被移动到同步队列，会被其他线程唤醒），检查当前线程在等待的过程中是否被中断（中断位被置为true），如果是，则将当前线程等待节点的waitStatus从CONDITION（处于等待状态）改为0，如果成功，则当前线程节点进入同步队列，如果修改waitStatus失败（当前线程未进入同步队列），则当前线程被阻塞（一直调用Thread.yield()，直到当前线程被其他线程放进同步队列）；如果没有被被中断过，再判断当前线程是否进入了同步队列，如果没有，则再次被挂起；

⑤如果等待队列的线程进入了同步队列，则按照同步队列的顺序，轮到自己获取锁资源时，再一次性获取savedState次锁资源，获取成功后，再对等待队列进行一次清理（将待状态不为CONDITION的等待节点剔除出队列）；

⑥最后根据线程在等待和同步时的中断情况判断要不要抛出中断异常；（此出未理解）

2.5.2、void signalAll()

方法作用：将此等待队列中所有节点移动到同步队列，

注意：只有获得锁资源的线程才能调用此方法，不然会抛出非法监视器状态异常；

①检查调用此方法的当前线程是否是占用锁资源的线程，不是会抛出非法监视器状态异常；

②按等待队列的顺序从头到尾将各个节点插入到同步节点的末尾；

2.5.3、void signal()

方法作用：唤醒此等待队列中第一个未被取消的节点

①检查调用此方法的当前线程是否是占用锁资源的线程，不是会抛出非法监视器状态异常；

②按等待队列的顺序从头到尾遍历，如果遍历到的节点未被取消，则将此节点入队同步队列，并结束方法；

### 2.6、void lockInterruptibly()

方法作用：以独占模式获取，如果线程中断则中止。

注意：此方法与lock()唯一的不同点在于：当被挂起的线程醒来后发现自己的中断标识位为true时，会直接抛出中断异常，则当前线程会退出锁资源的抢夺；而在lock()方法中，如果被挂起的线程醒来后发现自己的中断标识位为true时，会继续抢夺锁资源，不会退出。

①此方法就是调用了AQS中acquireInterruptibly(int arg)方法；

### 2.7、boolean tryLock()

方法作用：以非公平的方式进行一次锁资源的抢夺，抢夺成功返回true，失败返回false，无论抢夺是否成功，方法都会退出；

### 2.8、boolean tryLock(long timeout, TimeUnit unit)

方法作用：在指定时间内获取锁资源，时间到就退出此方法；

## ReentrantReadWriteLock

可重入读写锁是将被锁保护的临界资源的读操作和写操作分开，允许同时有多个线程同时对临界资源进行读操作，任意时刻只允许一个线程对资源进行写操作。简单的说，对与读操作采用的是共享锁，对于写操作采用的是排他锁。

可重入读写锁内部维护了一个读锁一个写锁，并且这两个锁的同步器都引用了可重入读写锁的同步器。

注意：

①写锁只能被一个线程占用，读锁可以被多个线程共同占用，读写锁都是可重入锁；

②当某个线程在占用了写锁的情况下，再去抢夺读锁，那这个线程永远不会被挂起，因为一旦被挂起，那这个线程持有的写锁就释放不了，其他线程没法获取写锁，也没法获取读锁，这样就造成死锁了；

③当某个线程在占用了写锁的情况下，只有它自己能获取到读锁，其他线程是获取不了读锁的，只能到同步队列中去排队；

④当有线程占用了读锁，且写锁没有被占用，此时任何线程都无法抢夺写锁，包括占用读锁的线程；当某一线程同时占用了读锁和写锁，则它继续重入读锁和写锁；

⑤由③和④可以得出结论：当某一线程在持有读锁的情况下，不能再持有写锁；但是，当某一线程在持有写锁的情况下，却能再持有读锁；

### 3.1、ReentrantReadWriteLock()

方法做用：无参构造函数，内部调用有参构造函数，传参false，及创建非公平锁；

### 3.2、ReentrantReadWriteLock(boolean fair)

方法做用：有参构造函数，根据传入参数创建非公平锁或者公平锁

①如果传入的参数是true，则创建公平同步器（AQS子类），否则创建非公平同步器，无论是公平还是非公平同步器，都会默认调用父类Sync的无参构造，此构造中初始化了成员变量readHolds；再将新创建的同步器赋值给成员变量sync；

②然后分别新建读锁和写锁，将上一步创建的同步器分别赋值给读锁和写锁的成员变量sync，再将读锁和写锁赋值给成员变量readerLock和writerLock；

### 3.3、ReentrantReadWriteLock.ReadLock readLock()

方法做用：返回维护的读锁

### 3.4、ReentrantReadWriteLock.WriteLock writeLock()

方法做用：返回维护的写锁

### 3.5、内部类ReadLock

注意：读锁被获取一次，state值就增加216，也就是说，当读锁被第一次获取，state变为216，第一次获取未释放的前提下，第二次获取后，state的值变为217（216+216），前两次未释放，打三次获取后，state的值变为217+216（216+216+216），依此类推。

3.5.1、ReadLock(ReentrantReadWriteLock lock)

方法做用：构造函数，使成员变量sync指向外部类创建的同步器

3.5.2、void lock()

方法做用：获取读锁

假设读写锁中维护的是非公平同步器：

①本方法就是调用AQS中的acquireShared(int arg)方法；

②acquireShared(int arg)方法先调用tryAcquireShared(arg)方法，此方法由AQS子类也就是读写锁的内部类Sync重写实现；

③在tryAcquireShared(arg)方法中，先判断此时是否有线程在占用写锁，是的话，并且占用写锁的正是当前线程，则当前线程可以去抢夺读锁，否则当前线程进入同步队列；此处的意思就是一旦有线程占用了写锁，其他线程就不能再获取读锁了；

④如果当前线程可以进行读锁资源的抢夺，则先判断以下2个条件是否同时满足：1、等待队列中队首线程要抢夺的是读锁资源；2、此时读锁被占用次数小于216-1（因为读锁占用了int的高16位，如果大于了216-1，就超过了int的范围了）；

⑤如果2个条件都满足，则进行读锁资源抢夺，如果抢夺成功，并且当前线程是第一个成功抢夺到读锁资源的线程，则将当前线程赋值给成员变量firstReader，并将成员变量firstReaderHoldCount进行加1；如果当前线程不是第一个成功抢夺到读锁资源的线程，则新建内部类HoldCounter来记录当前线程占用读锁资源的次数，HoldCounter还记录了当前线程的线程id，所以每个线程都独自拥有一个HoldCounter，并将HoldCounter保存在线程成员变量threadLocals中；

⑥如果2个条件中有不满足的或者抢夺读锁资源失败，则进入fullTryAcquireShared(current)方法（完整尝试获取读锁阶段）；

⑦进入fullTryAcquireShared(current)方法后就进入了一个无线循环中，首先判断是否有线程在占用写锁，是的话，并且占用写锁的不是当前线程，则当前线程进入同步队列（跳出了无线循环）；若占用写锁的是当前线程，则继续进行读锁资源的抢夺；

⑧如果没有线程在占用写锁，则判断同步队列中队首线程要抢夺的是否是写锁资源，如果不是，则直接进行读锁资源的抢夺；如果是，则再判断当前线程是否是第一个占用读锁的线程，如果是，则直接进行读锁资源的抢夺，否则，获取此时cachedHoldCounter（缓存最后一个成功获取读锁的线程的重入次数）的引用，如果cachedHoldCounter不为空并且cachedHoldCounter正是属于当前线程，则再判断cachedHoldCounter中维护的读锁资源占用数是否为0，若是，则使当前线程进入同步队列（跳出了无线循环），否则，继续读锁资源的抢夺；如果cachedHoldCounter为空或者cachedHoldCounter不属于当前线程，则获取当前线程的HoldCounter，如果HoldCounter.count=0（说明当前线程是首次抢夺读锁资源，或者之前获取的读锁资源已经释放完，再次抢夺），则将当前线程中保存的HoldCounter进行删除，再使当前线程进入同步队列；如果HoldCounter.count不等于0，则当前线程可以继续抢夺读锁资源；

⑨进入同步队列的线程，当它的前节点是头节点时，就可以进行读锁的抢夺，如果失败，则修改头节点的waitStatus为SIGNAL（释放锁后唤醒下一个节点）后被挂起，当被唤醒后，可以再次进行锁资源的抢夺；如果抢夺读锁资源成功，则将当前线程节点设置为头节点，并将成员变量thread和prev置为null；

注意（非公平同步器）：

①如果写锁没有被占用，并且同步队列队首线程（队首指的是head.next）要抢夺的是写锁资源，并且当前抢夺读锁的线程是firstReader（第一个抢夺到读锁的线程），则当前线程可以继续抢夺读锁；

②如果写锁没有被占用，并且同步队列队首线程（队首指的是head.next）要抢夺的是写锁资源，并且当前抢夺读锁的线程已经持有读锁，则当前线程可以继续抢夺读锁，否者当前线程进入同步队列；

假设读写锁中维护的是公平同步器：

与非公平同步器的区别：

①某个线程进入lock()方法后，在抢夺读锁资源之前，会先检查同步队列中是否有线程在排队，有的话就不抢夺锁资源，而是进入同步队列排队；而非公平同步器下，线程进入lock()方法后，不管同步队列是否有线程在等待，先抢夺一次锁资源再说；

3.5.3、void unlock()

方法作用：释放锁资源

①如果当前线程是firstReader引用的线程，并且firstReaderHoldCount值为1，则直接将firstReader置位null，再将state减216；

②如果当前线程不是firstReader引用的线程，则获取当前线程中维护的HoldCounter，将HoldCounter的成员变量count减1，如果count=1，直接将HoldCounter从当前线程中移除，最后将state减216；

③如果state减216后等于0了（读锁和写锁都没被占用），尽可能唤醒同步队列中的所有线程；

### 3.6、内部类WriteLock

3.6.1、WriteLock(ReentrantReadWriteLock lock)

方法做用：构造函数，使成员变量sync指向外部类创建的同步器

3.6.2、void lock()

方法做用：获取写锁

假设读写锁中维护的是非公平同步器：

①本方法就是调用AQS中的acquire(1)方法；

②获取state的值和当前写锁被占用的次数w；

③如果state不等于0并且w=0（读锁被占用，写锁未被占用），则直接让当前线程进入同步队列；

④如果state不等于0并且w不等于0（读锁被占用，写锁也被占用），如果当前线程正是占用写锁的线程，那当前线程直接重入写锁；

⑤如果state=0（读锁写锁都未被占用），则当前线程直接进行写锁的抢夺，若果抢夺成功，则将当前线程赋值给AQS成员变量exclusiveOwnerThread，否则当前线程进入同步队列；

假设读写锁中维护的是公平同步器：

与非公平同步器的唯一区别：当state=0（读锁写锁都未被占用），先判断同步队列中是否有线程在排队，如果有的话，当前线程需要排队（入同步队列并排在队尾）；

3.5.3、void unlock()

方法作用：释放锁资源

①如果当前线程是占用写锁资源的线程，则将state进行减1操作；

②如果此次是当前线程最后一次释放写锁，则将exclusiveOwnerThread置位null，并唤醒同步队列中下一条线程；

## ArrayBlockingQueue

数组阻塞队列，内部维护了一个final的object数组items，被初始化后就不能再进行修改；内部还维护了两个int型的变量putIndex（表示下一个元素进入items数组的下标）和takeIndex（表示下一个被取出元素在items数组中的下标）；

### 4.1、ArrayBlockingQueue(int capacity)

方法做用：构造函数之一，直接调用ArrayBlockingQueue(int capacity, boolean fair)构造函数，第二个参数默认传false；

### 4.2、ArrayBlockingQueue(int capacity, boolean fair)

方法做用：构造函数之一，

①校验传入的参数capacity不能小于等于0，如果不满足要求直接抛出非法参数异常；

②根据传入的capacity初始化final成员变量object数组items；

③根据传入的fair参数新建ReentrantLock赋值给final成员变量lock；

④使用上一步新建的ReentrantLock分别新建ConditionObject赋值给成员变量notEmpty（等待被取出的等待队列）和notFull（等待被放入的等待队列）；

### 4.3、boolean add(E e)

方法作用：新增元素入队列（插入到items数组），方法是同步的

①通过父类及多态性调用到了自己的offer(E e)方法；

②校验传入的参数e不能为空，否则直接抛出空指针异常；

③调用成员变量可重入锁lock的lock()方法进行锁资源的获取，获取过程见ReentrantLock；

④如果获取锁成功：如果成员变量count和items.length相等，则释放锁资源，再抛出异常提示队列已满；如果count和items.length不相等，则将传入的元素e入队，将e放入items数组的putIndex（当前元素应该插入的数组索引值）索引下，并将putIndex加1，加1后的putIndex如果等于items.length，则再将putIndex置为0，然后count++，最后将等待队列notEmpty中第一个等待的元素移动到同步队列中，然后释放锁资源返回true，整个方法结束；

### 4.4、boolean contains(Object o)

方法作用：判断阻塞队列中（及items数组中）是否包含传入的对象，传入对象为空直接返回false；方法是同步的

注意：遍历items数组是从takeIndex开始到putIndex结束。

### 4.5、void put(E e)

方法作用：新增元素入队列（插入到items数组），如果队列已经满了，则进入等待队列notFull中，等待获取锁资源进入items数组；

①校验传入的参数e不能为空，否则直接抛出空指针异常；

②获取可中断锁；

③判断队列是否已经满了(count == items.length)如果是的话，当前线程进去等待队列notFull中，直到被移入同步队列排队抢夺锁资源，如果抢夺锁资源再判断队列是否已经满了，如果满了再次进入notFull中，一直循环直到插入items数组成功，或者是当前线程被中断；

### 4.6、E poll()

方法作用：取出队列中下一个元素，并将此元素从队列中删除；

①判断队列是否为空，如果是则返回null；

②如果队列不为空，则返回takeIndex索引下的元素，并将takeIndex索引处置位null，takeIndex加1，如果加1后的takeIndex等于items数组的长度，则将takeIndex置位0；count（队列中元素个数）减1；将notFull等待队列中等待的第一个线程移动到同步队列中去；最后

### 4.7、E peek()

方法作用：返回下一个要出队列的元素，是null就返回null；

### 4.8、boolean remove(Object o)

方法作用： 如果队列中存在传入的对象，则将其从队列中移除；

①如果传入的对象为null，则直接放回false；

②从下标takeIndex到putIndex遍历数组items，如果数组中存在该对象则将其从该队列中移除；

### 4.9、E take()

方法作用：获取队列中下一个元素并将其从队列中移除，如果队列中没有元素，则等到有元素进去队列再取出；

## LinkedBlockingQueue

### 5.1、LinkedBlockingQueue()

方法作用：无参构造，调用LinkedBlockingQueue(int capacity)

### 5.2、LinkedBlockingQueue(int capacity)

方法作用：带参构造，初始化阻塞队列

①将传入的参数赋值给成员变量capacity；

②新建节点，并将节点值设置为null，并将新建的节点赋值给头节点和尾节点；

### 5.3、boolean offer(E e)

方法作用：想队列中插入元素

①如果传入的元素为null，直接抛出异常，方法结束；

②如果成员变量原子类count的值等于容量capacity（队列已经装满），直接返回false；

③用传入的参数新建节点；

④获取入队列锁；

⑤获取锁成功后，再次判断count的值是否小于容量capacity，如果不是，则直接返回false，方法结束；

⑥如果count小于capacity，则将步骤③中新建的节点进行入队，即尾节点的next指向新建的节点；

⑦count使用CAS进行加1操作，并获取到加1之前的值；

⑧判断count加1后的值是否小于capacity，如果是则将入队列等待队列中的第一个等待线程放进入队锁同步队列；

⑨释放入队列锁；

⑩如果count加1前的值为0，则将出队列等待队列中第一个等待的线程放进出队锁同步队列；

### 5.4、E peek()

方法作用：获取队列中第一个元素的值

①判断当前count的值是否等于0，如果是则返回null；

②如果count不为null，则获取出队列锁；

③获取到锁之后，取出队列中第二个节点（因为头节点指向的元素为null）；

④如果第二节点为null，则返回null，否则返回节点指向的元素；

⑤最后释放锁资源；

### 5.5、E poll()

方法作用：取出队列中的第一个元素

①获取count的值，如果count的值为0，则直接返回null；

②获取出队锁；

③获取锁成功后，判断count是否大于0，如果不是，则释放锁并返回null；

④如果count大于0，则第一个元素进行出队操作；

⑤对count进行减1操作；

⑥如果count减1前的值大于1（说明本次出队后队列中还存在元素），则将出队列等待队列中的第一个线程移入同步队列；

⑦释放锁资源；

⑧如果count减1前的值等于队列容量，则将入队列等待队列中的第一个线程移入同步队列；

⑨最后返回本次获取的元素；

### 5.6、void put(E e)

方法作用：将传入的元素插入到队列中，如果队列已满，则阻塞

①判断要入队的元素是否为null，若为null，则直接抛出空指针异常；

②将传入的元素新建为节点；

③获取可中断入队列锁；

④获取锁成功，往下执行；

⑤判断count（队列中现有元素）是否等于capacity（队列最大容量），如果相等，则当前线程进入入队列等待队列；

⑥直到判断出队列中有空位时，就将第二步新建的节点进行入队；

对conut进行加1操作；

⑧如果count加1后的值小于capacity（说明队列中还有空位），则唤醒入队等待队列中的节点进去入入队同步队列；

⑨释放锁资源；

⑩如果count加1前的值等于0（说明之前队列中没有元素，则出队等待队列中可能会有线程），唤醒出队等待队列中第一个节点进入出队同步队列中；

### 5.7、boolean remove(Object o)

方法作用：移除队列节点中包含传入元素的节点

①如果传入的元素为null，则直接返回false；

②获取全锁，也就是获取入队锁和出队锁；

③从队列头部开始遍历队列，判断节点中所含元素是否等于传入的元素，如果有等于传入元素的节点，就将此节点移出队列，并返回true，否则返回false；

④注意，在将节点移出队列时后，如果count在减1 前的值等于capacity则唤醒入队等待队列中的节点到入队同步队列中；

### 5.8、E take()

方法作用：取出队列中的第一个元素，如果队列中没有元素，则将阻塞（与poll的差异）；

## PriorityBlockingQueue

优先级阻塞队列，队列中的元素是一个完全二叉树结构，并且满足任意节点的值小于其子节点（即最小堆结构），所以每次取出的值都是整个队列最小的。

### 6.1、PriorityBlockingQueue(int initialCapacity,Comparator<? super E> comparator)

方法作用：构造方法，其他构造方法都是调用此方法；

①判断传入的初始容量值initialCapacity是否小于1，如果是，直接抛出不合法参数异常；

②初始化可重入锁；

③初始化出队列等待队列；

④将传入的比较器comparator赋值给成员变量comparator；

⑤根据传入的初始容量值initialCapacity初始化队列数组queue；

### 6.2、boolean add(E e)

方法作用：向队列中插入元素

①调用offer方法；

### 6.3、boolean offer(E e)

方法作用：向队列中插入元素

①如果出入的元素是null，则抛出空指针异常；

②获取锁资源；

③如果队列中存在的元素个数size大于等于数组queue的长度，则对数组进行扩容；

④扩容流程：释放锁资源；如果allocationSpinLock的值不为0，或者通过CAS的方式将allocationSpinLock改为1失败，说明有其他线程在对数组进行扩容，则调用yield方法让出cpu资源；如果获取allocationSpinLock锁成功，则根据数组容量得到扩容后的容量，使用扩容后的容量新建数组，最后释放扩容锁；

⑤获取锁资源；

⑥用新数组替换之前的旧数组；

⑦如果有比较器，就使用比较器，如果没有，则使用元素的compareTo方法比较传入元素与其父元素的大小，如果传入的元素大于等于其父元素，则传入元素直接进入数组；如果传入元素小于其父元素，则与其父元素交换位置，一直比较到比自己小的父元素或者比较到0索引位置；

⑧size++；

⑨唤醒出队列等待队列中的线程进入同步队列；

⑩释放锁资源；

### 6.4、void clear()

方法作用：清空队列

①获取锁资源；

②将size置为0；

③将队列数组中的每个索引下的对象都置为null；

④释放锁资源；

### 6.5、Iterator<E> iterator()

方法作用：获取此队列数组的迭代器

①先新建数组，将队列数组中的元素复制到新数组中；

②新建迭代器，并将上一步新建的数组作为参数，赋值给迭代器的成员变量；

### 6.6、E peek()

方法作用：获取队列数组中索引下的元素

①获取锁资源；

②判断队列中元素的数量size的值是否等于0，如果是，则返回null，如果不是0，则返回数组中0索引处的元素；

③释放锁资源；

### 6.7、E poll()

方法作用：

①获取锁资源；

②判断size是否等于0，如果是直接返回null；

③将队列数组中0索引下的元素取出（将作为最终的返回值），并将size-1索引下的元素取出放到0索引位置（实际源码中并未如此操作，此处为了便于理解），size-1索引处置位null，从0索引处开始，比较它与两个子节点中较小那个的大小，如果比子元素大，则交换他们，直到符合最小堆数据结构；

④释放做资源；

⑤返回结果；

### 6.8、void put(E e)

方法作用：向队列中插入元素，与add的区别是没有返回值

①调用offer方法；

### 6.9、boolean remove(Object o)

方法作用：从队列中移除传入的元素

①获取锁资源；

②遍历队列数组，判断传入的对象是否在队列中，如果不在，直接返回false；

③如果传入的元素存在于队列中，则找到它所在的索引；

④将size-1处的元素赋值给传入元素所在的索引处，然后向下（与子节点）比较大小以满足最小堆结构；

⑤如果第④步中的元素未挪动位置（它比它的子节点都小），则向上（与父节点）比较大小以满足最小堆结构；

⑥释放锁资源；

⑦返回true；

### 6.10、E take()

方法作用：从队列中取出0索引处的元素，如果队列中没有元素，则被阻塞

①获取可中断锁；

②从队列中取出0索引下的元素，如果队列中没有元素，则进入出队等待队列，直到被入队线程唤醒；

③获取到元素后释放锁资源；

④将元素返回；

## SynchronousQueue

同步队列，队列中要么维护的是传输队列，要么维护的是传输堆栈；如果维护的是传输队列，有线程1往队列中插入元素时，如果没有其它线程来取元素则线程1将阻塞，取元素的线程也是一样的逻辑；

### 7.1、SynchronousQueue(boolean fair)

方法做用：构造函数，另一个无参构造也是调用此方法

①根据传入的布尔值，true就创建传输队列，false就创建传输堆栈，并赋值给成员变量transferer；

### 7.2、void put(E e)

方法做用：将传入的元素添加到此队列，等待另一个线程接收它

假设transferer指向的是传输队列：

①校验传入的元素是否为null，如果是，则抛出空指针异常；

②进入TransferQueue的transfer方法；

### 7.3、内部类TransferQueue

### 7.3.0、TransferQueue()

方法作用：无参构造函数，初始化头尾节点

①新建节点使其指向null，成员变量isData赋值为false；

②头尾节点都指向上一步新建的节点；

### 7.3.1、E transfer(E e, boolean timed, long nanos)

方法作用：向传输队列插入一个元素或从传输队列取出一个元素；

②进入死循环；

③获取此时传输队列中的头尾节点h和t，如果其中之一为null，则终止本次循化，转到下一轮循环；如果头尾节点都不为null，则向下执行；

④如果头节点等于尾节点（队列中没有等待的线程），并且尾节点中的线程与当前线程不是同一种线程（要么是入队线程，要么是出队线程）；

⑤判断此时的尾节点是否还等于t，如果不等，则终止本次循化，转到下一轮循环；

⑥判断t的下一个节点tn是否为null，若不为null，并且此时的尾节点等于t，则使用cas尝试将尾节点的下一节点设置为尾节点；终止本次循化，转到下一轮循环；

⑥．①如果设置了超时，则判断超时时间是否小于0，是的话，直接返回null；

⑦以传入的元素新建入队节点s（出队线程传入的元素为null）；

⑧如果tn为null，并且成功使用cas将s设置为t的下一个节点，就往下执行，否者终止本次循化，转到下一轮循环；

⑨尝试使用cas将s替换t成为尾节点；

⑩进入awaitFulfill方法，

⑱判断新建节点是否等于awaitFulfill方法返回的值，如果等于，则进入clean方法；

③②回到transfer方法，返回null；

### 7.3.2、Object awaitFulfill(QNode s, E e, boolean timed, long nanos)

方法作用：自旋直到传入的节点s被满足

⑩进入自旋阻塞状态；

⑪判断当前线程中断标识位是否是true，如果是，尝试将新建节点指向的对象从传入对象更改为新建节点自己（取消当前线程的任务，无论是入队还是出队）；

⑫判断此时s指向的对象还是否是传入的元素，如果不是，则直接返回此时s指向的对象（awaitFulfill方法唯一返回点）；

⑬如果设置了超时，则判断是否超时，如果超时，尝试将s指向的对象从传入对象更改为新建节点自己；并结束本次自旋（循环），进入下一次自旋（循环）；

⑭如果spins（目前还不清楚是什么）大于0，则spins减减操作，并调用Thread.onSpinWait()（方法体为空，只是一个象征意义吗）；本次循环结束；

⑮如果spins不大于0，并且s里的等待线程为null，则将当前线程赋值给s的等待线程；本次循环结束；

⑯如果spins不大于0，并且s里的等待线程不为null，并且没有时间限制，则当前线程被挂起；被唤醒后，本次循环结束；

⑰如果spins不大于0，并且s里的等待线程不为null，并且有时间限制，并且限制时间大于1000毫秒，则按限制时间挂起当前线程；被唤醒后，本次循环结束；

### 7.3.3、void clean(QNode pred, QNode s)

方法作用：

⑲将新建节点的等待线程置为null；

⑳进入while循化，条件是新建节点的前接点还指向新建节点；

②①获取头节点h，和头节点的下一节点hn；

②②如果hn不为null，并且hn已经取消（节点指向自己），则在h还是头节点的情况下使用cas将hn设置为头节点，如果设置成功，则将旧的头节点h指向h自己；本次循环结束，进入下次循环；

②③获取尾节点t，如果h等于t，直接返回到transfer方法；

②④获取t的下一节点tn；

②⑤如果t已经不等于此时的尾节点，结束此次循环，进入下次循化；

②⑥若果tn不为null，则在t还是尾节点的情况下尝试将tn通过cas设置为尾节点，结束此次循环，进入下次循化；

②⑦如果s不等于t（新建节点不是尾节点），获取s的下一节点sn，如果s等于sn或者（||）将s的前接点的下一节点设置为sn成功，则直接返回到transfer方法；

②⑧获取成员变量cleanMe赋值给dp；

②⑨如果dp不等于null，获取dp的下一个节点d；如果d等于null，或者d等于dp，或者d没有被取消，或者（d不等于t并且d的下一节点dn不等于null并且dn不等于d并且将dn设置为dp的下一节点成功），则尝试通过cas将成员变量cleanMe置为null；

③〇如果dp等于pred，直接返回到transfer方法；

③①如果dp等于null并且将新建节点的前节点设置为cleanMe成功，则直接返回到transfer方法；