目录

[1、 Thread 2](#_Toc7068)

[1.1、 Thread()、Thread(Runnable target) 2](#_Toc9158)

[1.2、 synchronized void start() 2](#_Toc4854)

[1.3、 final void join() 3](#_Toc19530)

[1.4、 void interrupt() 3](#_Toc29139)

[1.5、 boolean isInterrupted() 3](#_Toc6683)

[1.6、 static boolean interrupted() 3](#_Toc28236)

[1、 AbstractQueuedSynchronizer 3](#_Toc910)

[2、 ReentrantLock 4](#_Toc25468)

[2.1、ReentrantLock() 4](#_Toc9473)

[2.2、ReentrantLock(boolean fair) 4](#_Toc15155)

[2.3、void lock() 4](#_Toc11341)

[2.4、void unlock() 5](#_Toc12832)

## 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。

### 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：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）。

### 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）

## AbstractQueuedSynchronizer

## ReentrantLock

跟synchronized相同，是可重入的重量级锁，并且是悲观锁、独占锁。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。

如果有多个线程同时操作此内部类，应该是线程不安全的，编程时是否应该避免此情况？

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，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？没理解），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，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，然后方法结束。

2.5.2、void signalAll()

方法作用：唤醒此等待队列中所有节点，

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；

### 3、