# 概述

同步器AbstractQueuedSynchronizer，简称AQS，主要是为了java的跨平台。C++中在使用信号量时都直接使用操作系统提供的信号量以及信号量自带的同步队列，因此不同操作系统需要使用不同的系统调用。java为了跨平台，则将同步队列抽象出来作为一个抽象类AbstractQueuedSynchronizer。

AQS是一个抽象类，也可以理解为一套框架，不能直接创建对象，主要作用是封装了各类操作，同时提供接口供子类实现以便达到不同的作用。JDK1.8中实现了多个同步器，比如Semaphore、CountDownLatch等都创建了相应的内部类来实现AQS类，AQS的继承关系如下所示。



AQS子类继承关系

为了在不同场景使用同步器，JDK特此提供多种同步器，AQS类是所有这些同步器的核心，各种同步器都只是依赖AQS提供的基本功能做了一层封装。

本文档主要学习JDK1.8中提供的AbstractQueuedSynchronizer类以及其子类的知识点。

# AbstractQueuedSynchronizer

AbstractQueuedSynchronizer同步器，简称AQS，是一类Synchronizer的抽象，这类同步器的共同点就是可以通过一个int变量来表示它的当前状态，同时在获取同步器失败时可以将该线程加入等待队列（入队列时会通知前驱节点设置为SIGNAL状态，也就是该线程会去通知下一个节点所对应的线程来获取同步器）并自我中断执行，释放同步器时会根据公平性与否唤醒线程获取。

其底层实现依赖于双向链表结构，链表节点来自于其内部类Node。

几个概念：

* Share：共享模式，是指一个同步器可以被多个线程同时拥有，比如Semaphore；
* Exclusive：独占模式，一个同步器只能被一个线程同时拥有，比如ReentrantLock，独占式的锁在修改state变量时都无需加锁，因为独占式只能存在一个线程同时去修改state
* Fair：公平性，是指处于等待状态的线程按照FIFO进行同步器的获取；
* Nonfair：非公平性，是指每一个新到的线程都可以抢占锁，如果成功直接获取锁，失败则加入队列，无需一开始就判断队列是否有排队线程；

参考

<http://ganbiao2205.iteye.com/blog/2145479>

<https://www.jianshu.com/p/1cd76cff6c0f>

## CLH变种队列

### 节点类Node

AQS中的队列是通过链表结构来实现的（因为AQS涉及大量的删除和插入节点操作，如果使用数组不便于删除操作），因此需要定义构成链表的节点类型，其结构如下：

* 每个节点有两种模式，一种是共享式，一种是独占式，同一个子类（同步组件）中只能存在一种模式；
* 每个节点包含一个Thread对象，指向排队等候的线程；
* 每个节点有一个状态变量，记录当前等候线程的状态；
* 每个节点包含前驱和后继节点引用

## 主要成员

|  |  |  |
| --- | --- | --- |
| 名字 | 类型 | 描述 |
| head | Node | 队列头指针 |
| tail | Node | 队列尾指针 |
| state | int | 被同步器同步资源的个数，如果state等于0，获取资源的线程需要排队 |
| headOffset | long | 记录head成员在AQS类中的偏移量 |
| tailOffset | long | 记录tail成员在AQS类中的偏移量 |
| stateOffset | long | 记录state成员在AQS类中的偏移量 |

通过分析源码可以得知，AQS提供的双端队列功能第一个节点是哨兵节点。

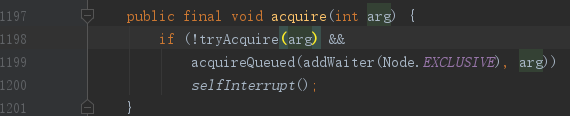
## 模板方法

此处的模板方法不是泛型，而是需要子类组件自己实现的方法，AQS中这几个模板方法都是直接抛出异常，如果需要基于AQS定义不同的同步器，需要实现这几个模板方法。模板方法分为三类：独占式获取与释放、共享式获取与释放、获取同步状态：

* tryAcquire(int)：子类实现各自获取锁的算法（用于排斥性的锁，如ReentrantLock）
* tryRelease(int)：子类实现各自释放锁的算法
* tryAcquireShared(int)：用于共享性的锁（如信号量Semaphore）
* tryReleaseShared(int)
* isHeldExclusively()

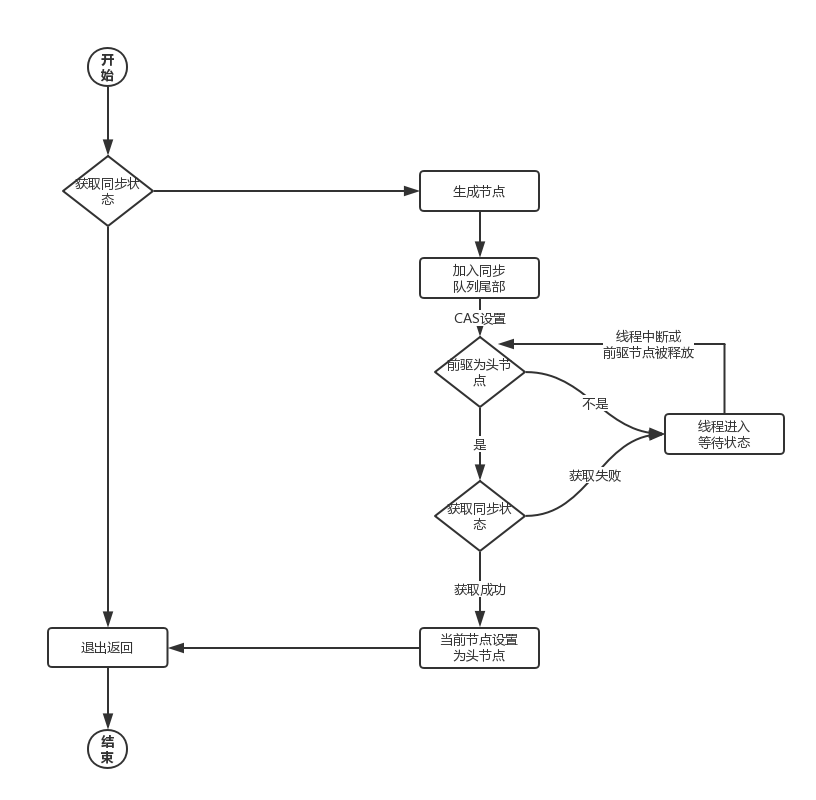
## 独占式工作原理

独占式获取锁时，调用acquire方法，该方法代码如下：



通过独占锁代码可以看出，独占锁操作state时都无需CAS操作，因为独占锁只有一个线程会操作state变量。

代码涉及其他几个方法，其中tryAcquire是各个子类需要实现的业务方法，如果返回true表示该线程获取到锁，返回false则将该节点加入到队列尾部，同时调用acquireQueued方法 ，思路如下：

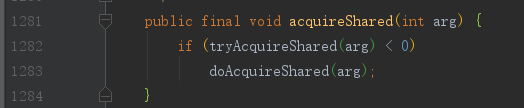


释放锁比较简单，调用release操作

采用独占式的同步器包括ReentrantLock，ReentrantReadWriteLock

## 共享式工作原理

共享式获取锁（一般是读锁）时调用acquireShared方法，该方法代码如下：



通过代码可以看出，该方法首先调用各个子类实现的tryAcquireShared方法，该方法返回可以使用的锁个数，如果小于0表示无锁可用。

采用共享式的同步器包括Semaphore、CountDownLatch，CyclicBarrier

## 辅助方法

### 方法清单

|  |  |  |  |
| --- | --- | --- | --- |
| 方法名 | 权限 | 描述 | 详细 |
| enq | private | 将传入的节点插入队列尾部 | 2.3.2 |
| addWaiter |  |  |  |

### enq

将传递进来的node节点插入队列尾部。



# 子类（同步组件）

## 信号量Semaphore

Semaphore是共享式锁，提供两种同步器，一种是公平同步器FairSync，一种是非公平同步器NonfairSync（默认采用非公平方式），这两个同步器具有很多相似的地方，因此对两种同步器抽象出了一个基类Sync。

Sync、FairSync、NonfairSync都是Semaphore的内部类。

### 构造方法

第一种方式只传递允许同步的线程数permits，默认使用非公平同步器NonfairSync。



第二种方式传递允许同步的线程数，同时设置同步器类型，如果fair是true创建公平同步器FairSync，否则为非公平同步器。



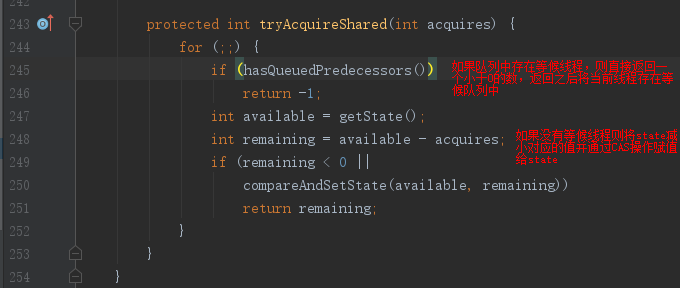
### 重要操作

#### acquire

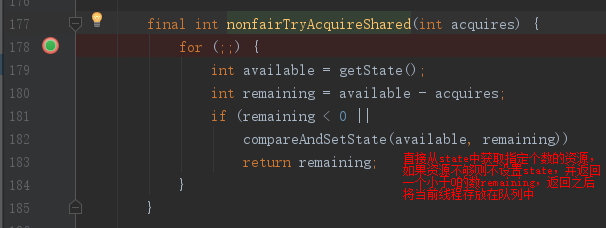
当线程调用acquire方法时，最终都会都会通过多态调用各自的tryAcquireShared（共享式）方法，因此需要分情况讨论。

注意：共享式获取锁会采用死循环的方式获取直到获取成功为止。

公平信号量：



非公平信号量



总结

公平信号量

* 线程调用acquire操作，获取指定个数（permits）的锁资源；
* 判断当前队列是否存在等候线程，如果存在则返回，返回之后将当前线程存在等候队列中
* 如果队列中不存在等候线程，从state 中减去permits得到remaining，如果remaining 小于0则返回，返回后将当前线程存放在队列中；如果remaining大于0，则通过cas操作将state值设置为 remaining并返回，如果cas操作失败则for循环继续执行，执行从第二步重新开始；

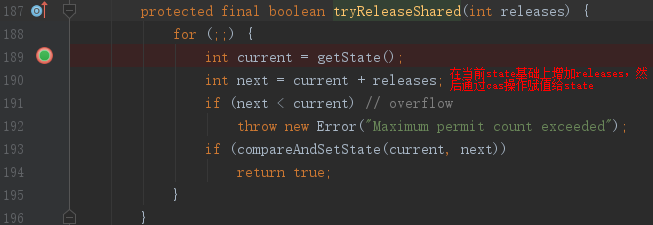
非公平信号量

* 线程调用acquire操作，获取指定个数（permits）的锁资源
* 无需判断队列中是否存在等候线程，直接从state 中减去permits得到remaining，如果remaining 小于0则返回，返回后将当前线程存放在队列中；如果remaining大于0，则通过cas操作将state值设置为 remaining并返回，如果cas操作失败则for循环继续执行，执行从第二步重新开始

#### release

release操作不区分公平性，也就是无论是公平信号量还是非公平信号量，release操作都执行一样的算法；最终执行子类实现的tryReleaseShared方法。

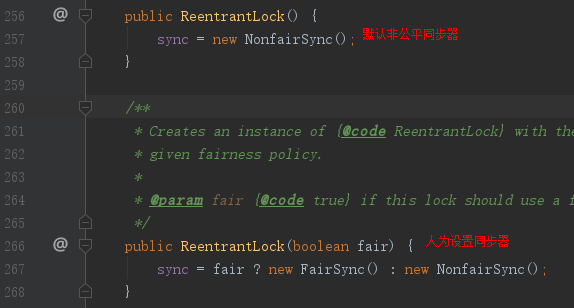
注意：共享式释放锁都是采用死循环的方式给state赋值直到CAS操作成功为止。



## 可重入锁ReentrantLock

可重入锁和Semaphore类似（），都包含公平同步器和非公平同步器，但是ReentrantLock是独占式锁。可重入锁的state变量的值如果为0表示没有线程占有锁，如果为1表示有一个线程占有锁，如果大于1表示该锁被同一个线程多次获取并占有（可以通过多次调用lock操作，调试查看锁中state变量值的变化，每调用一次lock，state自增1，调用一次unlock，state自减）。

### 构造方法



### 重要操作

ReentrantLock对外提供的常规API主要包括两个，一个是lock，一个是unlock。

#### lock

调用lock方法时，如果是非公平的可重入锁，首先判断state变量是否为0，为0则将state通过CAS设置为1，同时将当前线程作为占有锁的线程；如果不为0，则调用对应类实现的tryAcquire方法（多态）；如果是公平的可重入锁，直接调用对应类的tryAcquire方法。

公平可重入锁的tryAcquire方法



非公平可重入锁的tryAcquire方法



总结

公平可重入锁lock操作

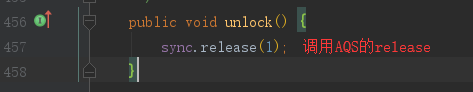
* 判断state是否为0
* 如果state=0，并且无其他线程排队，则通过CAS将state设置为1，同时将占有锁的线程设置为当前线程；
* 如果state>0，判断占有锁的线程和当前线程是否相同，相同则将state+1，否则返回false；
* 上述操作都不成功时，说明获取锁失败，则将当前线程存放在AQS队列，并将当前线程中断。

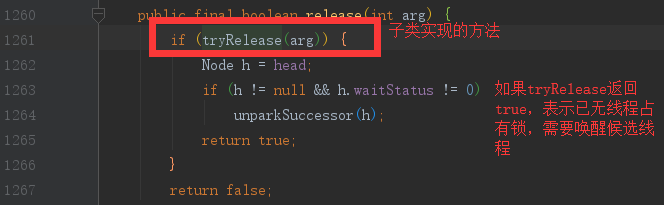
非公平可重入锁lock操作

* 判断state是否为0
* 如果state=0，则通过CAS将state设置为1，同时将占有锁的线程设置为当前线程；
* 如果state>0，判断占有锁的线程和当前线程是否相同，相同则将state+1，否则返回false；
* 上述操作都不成功时，说明获取锁失败，则将当前线程存放在AQS队列（线程放进AQS同步队列前会再判断一次state是否为0，如果为0会再一次发起tryAcquire请求，如果这一次成功获取锁则占有锁，否则放进AQS队列），并将当前线程中断。
* 非公平方式获取锁时，线程被加入队列前会尝试两次tryAcquire请求 ；

#### unlock

无论是公平锁还是非公平说，unlock操作都采用相同的方式，代码如下：







总结

* 公平和非公平可重入锁的unlock操作算法是一样的；
* state减1
* 判断调用线程和锁占有线程是否相同，如果不同抛出异常
* 如果state为0，把占有锁的线程置为null，设置state为0，同时返回true，之后唤醒候选线程；
* 如果state不为0，设置state的新值，返回false

## 可重入读写锁ReentrantReadWriteLock

ReentrantReadWriteLock对象中包含两把锁，一个是ReadLock，另一个是WriteLock，以及包含一个Sync对象，Sync对象既可以是公平锁也可以是非公平锁。

* ReadLock是共享锁，WriteLock是独占锁。
* ReentrantReadWriteLock中会将state分为两部分，高16位记录readlock占有次数，低16位记录writelock占有次数。
* 因此readlock最多能被2^16个线程占有；
* 如果低16位不为0，高16位一定为0，因为有writelock被占用时readlock不能被获取成功（读写互斥）

ReentrantReadWriteLock主要提供四种操作，ReadLock上锁（Lock），释放锁（unlock）；WriteLock上锁（Lock），释放锁（unlock）。

能够成功获取锁的情况只有

* 读读
* 写写，并且是同一个线程执行获取写锁。

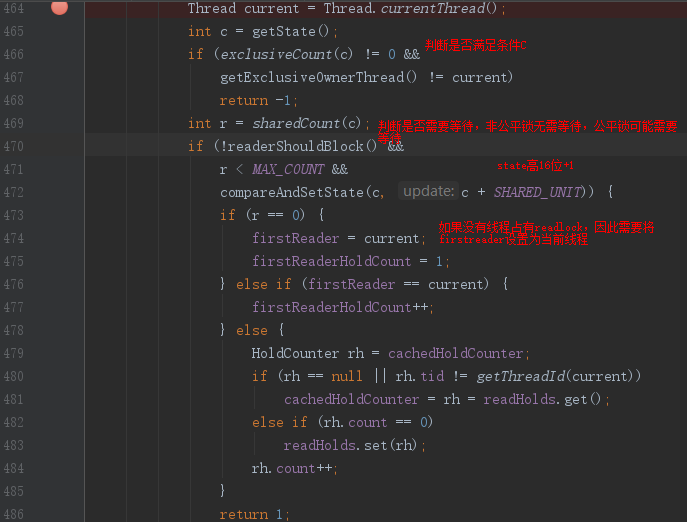
接下来主要针对这四种操作进行解释

### ReadLock上锁

因为ReadLock是共享锁，因此调用ReadLock的lock操作时，最终会执行Sync的tryAcquireShared方法，该方法是AQS提供的抽象方法，需要子类实现具体的获取锁方法。

一个线程在获取ReadLock锁时，只有满足下面的条件才会继续往下执行获取锁，其他情况将直接返回-1，这些条件是：

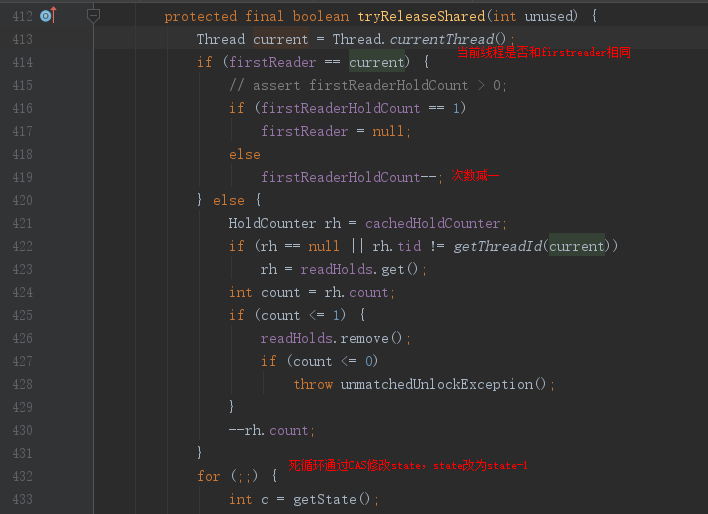
1. state=0，表示既没有readlock也没有writelock被占有；
2. 只有readlock被占用（state高16位不为0，低16位为0），writelock没有被占用
3. 只有writelock被占用（state高16位为0，低16位为0）并且申请readlock的线程和占用writelock的线程相同，也就是占用writelock的线程去获取readlock；



在获取ReadLock锁成功后，会通过HoldCounter对象将当前线程占用ReadLock的次数记录下来并存放在一个ThreadLocal对象中。

### ReadLock释放锁

ReadLock释放锁时会调用tryReleaseShared方法，该方法的思路可以概括为如下：

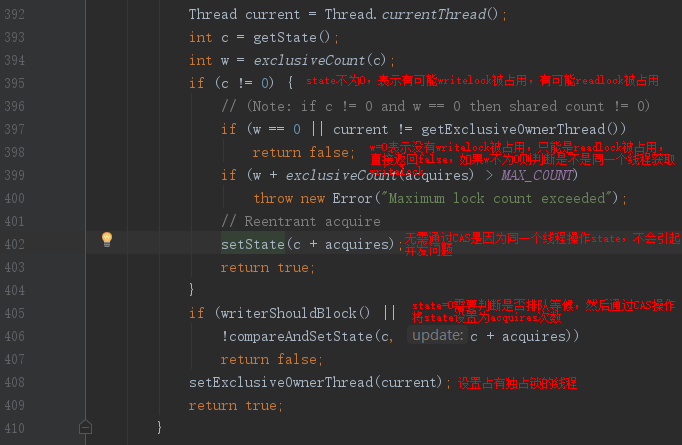


### WriteLock上锁

WriteLock上锁更简单一些，因为WriteLock是独占式的，无需额外维护线程占用次数。

因为WriteLock是独占式，所以上锁操作需要调用tryAcquire操作。能够完整执行tryAcquire方法的情况之一满足如下条件才能进行获取锁，其他情况都直接返回false：

1. state=0
2. 只有writelock被占用（state高16位为0，低16位不为0）并且当前线程和占用writelock的线程相同

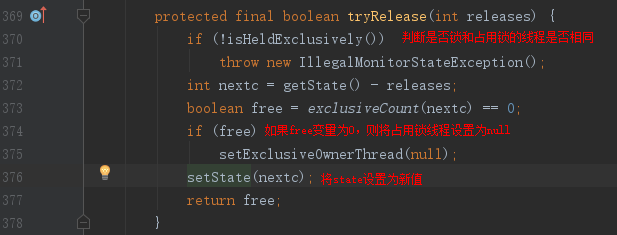


线程获取writelock失败后会通过AQS提供的方法将当前线程放到等待队列中。

### WriteLock释放锁

WriteLock释放锁思路如下：

* 判断占用锁的线程和当前线程是否相同，不同则直接抛出异常；
* state低16位减去releases得到新的state
* 如果state为0，则将占用锁的线程设置为null，并将state设置为0
* 否则将state设置为新的state（无需cas，因为是独占锁，只有一个线程可以操作state）

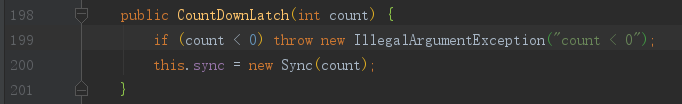


## 锁存器CountDownLatch

CountDownLatch用于一个线程或者多个线程等待多个操作（其他线程中）的完成。当计数器减为1后，所有调用wait的线程都会被唤醒。

### 构造方法

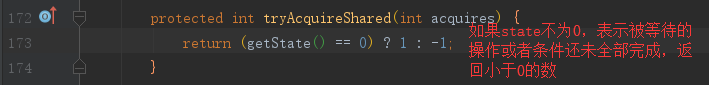
CountDownLatch只有一个构造方法，创建对象是需要设置等待条件（操作）的个数，代码如下所示：



### 重要方法

#### await

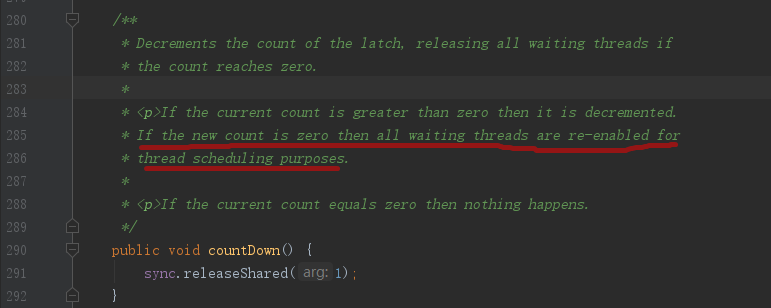
await操作会转换为调用AQS的acquireShardedInterruptibly方法，而该方法会通过多态调用AQS的资料Sync的tryAcquireShared方法：



如果返回值小于0，则表示还需要等待其他操作的完成，需要将当前线程中断并挂起，存放在AQS队列中。

#### countDown

该操作会将CountDownLatch对象的count计数器减一，如果当前count为0，则什么操作都不做直接返回，如果大于0则减一，新的count如果等于0，则唤醒所有调用过await方法的线程。



## CyclicBarrier

参考<https://blog.csdn.net/carson0408/article/details/79471490>

CyclicBarrier又名同步屏障，其主要作用是使一组线程同时达到一个“栅栏”（执行点）时才继续往下执行，如果某个线程到达“栅栏”处计数器不为0，则将该线程存放在ReentrantLock的条件队列中，直到最后一个到达“栅栏”的线程会唤醒条件队列中的所有线程。

翻译JDK源码的注释：

CyclicBarrier是一个同步器，其目的是为了使一组线程之间的相互等待直到所有线程达到一个公共的点（“栅栏”）。CyclicBarrier之所以被称作为循环同步器，原因是一个CyclicBarrier对象在所有线程释放该对象之后可以重复的被使用。

CyclicBarrier可以提供一个可选的命令任务，该任务由最后一个到达“栅栏”的线程触发并且在其他线程被释放之前执行。

如果一个线程在离开Barrier之前被字段、执行异常或者超时，其他所有的线程也都需要被被释放。

### 属性

* lock：可重入锁，主要作用是锁住线程对count变量的修改
* parties：
* trip：ReentrantLock中定义的Condition
* barrierCommand：Runnable对象，当唤醒trip中线程中时调用其run方法
* generation：轮数，类型是内部类Generation
* count：本轮计数器，记录还需要等待的线程数

### 内部类

Generation类，只包含一个boolean类型的broken属性。Generation用来控制屏障的循环使用，如果generation.broken为true的话，说明这个屏障已经损坏，当某个线程await的时候，直接抛出异常。

### 构造方法

所有的构造方法核心构造方法

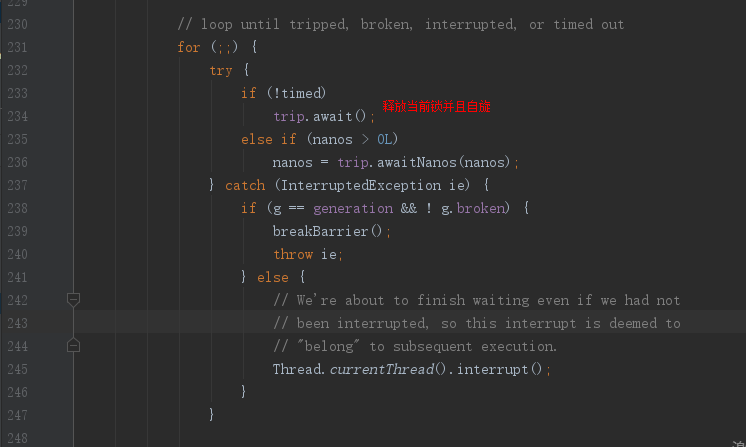
### 主要方法

CyclicBarrier所有方法中await、reset两个方法最重要

#### await

在CyclicBarrier的内部定义了一个Lock对象，每当一个线程调用CyclicBarrier的await方法时，将剩余拦截的线程数减1，然后判断剩余拦截数是否为0，如果不是，进入Lock对象的条件队列等待。如果是，执行barrierAction对象的Runnable方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从await方法返回，再从CyclicBarrier的await方法中返回。

这个方法最重要的是ConditionObject对象trip调用await方法时会释放当前锁。



#### reset

将屏障设置为初始状态。