1. CyclicBarrier
2. 概述

同步屏障，在JDK1.5引入的一个同步辅助类，它好比一扇门，堵住了线程执行的道路，直到所有线程就位，门才会打开，让所有线程一起通过。

1. 实现

CyclicBarrier的内部是使用重入锁ReentrantLock和条件Condition实现的。首先，CyclicBarrier中的线程需要等待，因此使用Condition来设置线程的等待，同时还需要指定唤醒哪些线程，因此使用Condition更加地合理。使用ReentrantLock是因为多线程执行操作需要锁来实现原子性操作。

1. 构造方法

有两个构造方法：

1. CyclicBarrier(int parties)

parties代表着拦截指定数量的线程，当这些数量的线程处于等待状态时，屏障启动，但它不会在启动屏障时执行预定义的操作。

1. CyclicBarrier(int parties, Runnable barrierAction)

在拦截到指定数量的线程后启动屏障，并且执行给定的屏障操作，该操作由最后一个进入屏障的线程执行。

1. await()

每个线程通过调用await()方法告诉CyclicBarrier已经达到了屏障位置，线程被阻塞，如果该线程不是到达的最后一个线程，则它会一直处于等待状态。除非发生以下情况：

1. 最后一个线程到达；
2. 超出了指时间；
3. 其他的某个线程中断当前线程；
4. 其他的某个线程在等待屏障时超时；
5. 其他的某个线程在此屏障调用reset()方法，将屏障重置为初始状态；

当执行完await()方法后返回后，就会执行await()方法后的代码。

1. 实例

多个运动员需要都就位以后，才能够起跑。

package thread7;  
  
import java.nio.channels.GatheringByteChannel;  
import java.util.ArrayList;  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CyclicBarrier;  
  
public class CyclicBarrierDemo {  
 public static void main(String[] args) {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(5);  
 ArrayList<Thread> threadArrayList = new ArrayList<Thread>();  
 for(int i = 0; i < 5; ++i){  
 Thread t = new Thread(new Athlete(cyclicBarrier, "运动员" + i));  
 threadArrayList.add(t);  
 }  
 for(Thread t :threadArrayList){  
 t.start();  
 }  
 }  
  
 static class Athlete implements Runnable{  
 private CyclicBarrier cyclicBarrier;  
 private String name;  
  
 public Athlete(CyclicBarrier cyclicBarrier, String name){  
 this.cyclicBarrier = cyclicBarrier;  
 this.name = name;  
 }  
  
 @Override  
 public void run() {  
 System.*out*.println(name +"就位");  
 try {  
 cyclicBarrier.await();  
 System.*out*.println(name + "跑到终点");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

1. CountDownLatch
2. 概述

CountDownLatch是一个计数的闭锁，作用与CyclicBarrier相似。该类用给定的计数初始化CountDownLatch，然后线程调用await()方法进入阻塞状态，在指定的其他多个线程执行完成前，计数器不会达到0，等待的线程也会一直阻塞。当计数器达到0时，等待的线程会被释放，await()的后续调用都会立即返回。

1. 与CyclicBarrier的区别

CountDownLatch的重点是一个线程或者多个线程等待，其他的多个线程在完成某件事情后，唤醒等待的线程。通常用于异步操作的处理，不同的线程之间需要等待不同的线程完成任务之后才能继续执行，因此通过异步等待的方式实现。

CyclicBarrier的重点是多个线程在任意线程没有完成的情况下，都必须等待下去。

CountDownLatch的计数是不能重置的，如果需要重置计数，考虑使用CyclicBarrier。

1. 实现

CountDownLatch内部依赖Sync实现的，Sync又是依赖于AQS同步队列实现的，内部通过AQS的共享锁来实现。CountDownLatch维护一个计数器，初始值为指定的线程数量，每当一个线程完成了任务，就会调用countDown()方法，将计数器减一，当计数器值为0时，可以唤醒等待的线程继续执行了。

1. await()和countDown()

await()方法使当前线程在计数器到0之前一直等待，除非线程被中断。内部使用AQS的getState()方法获取计数器，如果计数器不为0，则会以自旋的方式持续尝试获取同步状态。

countdown()方法递减计数器的计数，如果计数为0，则释放所有等待的线程，内部调用AQS的releaseShared(int arg)来释放共享锁同步状态。

1. 实例

在CyclicBarrier的实例上修改，添加接力运动员，接力运动员只需要关心和自己同队的起点远动员是否到达接力点，之后既可以起跑了。

package thread8;  
  
import thread7.CyclicBarrierDemo;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.CyclicBarrier;  
  
public class CountDownLatchDemo {  
  
 public static void main(String[] args) {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(5);  
  
 List<Thread> threadList = new ArrayList<Thread>();  
 for(int i = 0; i < 5; ++i){  
 CountDownLatch countDownLatch = new CountDownLatch(1);  
 Thread t1 = new Thread(new Athlete(cyclicBarrier, countDownLatch, "起点远动员" + i));  
 Thread t2 = new Thread(new Athlete(countDownLatch, "接力运动员"+ i));  
 threadList.add(t1);  
 threadList.add(t2);  
 }  
  
 for(Thread t : threadList){  
 t.start();  
 }  
 }  
  
 static class Athlete implements Runnable{  
  
 private CyclicBarrier cyclicBarrier;  
 private String name;  
  
 CountDownLatch countDownLatch;  
  
 //起跑运动员  
 public Athlete(CyclicBarrier cyclicBarrier, CountDownLatch countDownLatch, String name){  
 this.countDownLatch = countDownLatch;  
 this.cyclicBarrier = cyclicBarrier;  
 this.name = name;  
 }  
 //接力远动员  
 public Athlete(CountDownLatch countDownLatch, String name){  
 this.countDownLatch = countDownLatch;  
 this.name = name;  
 }  
  
 @Override  
 public void run() {  
 //判断是否是棋牌  
 if(cyclicBarrier != null){  
 System.*out*.println(name + "就位");  
 try {  
 cyclicBarrier.await();  
 System.*out*.println(name + "到达交接点");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
  
 //判断是否是接力  
 if(cyclicBarrier == null){  
 System.*out*.println(name + "就位");  
 try {  
 countDownLatch.await();  
 System.*out*.println(name + "到达终点");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

1. Semaphore
2. 概述

Semaphore是一个控制访问多个共享资源的计数器，本质上是一个共享锁。他维护了一个信号量许可集，线程可以获取信号量的许可并执行线程。当线程释放它所持有的信号量许可时，被释放的许可归还到许可集中，可以被其他线程再次获取。线程在竞争信号量许可的时候是根据公平锁或者非公平锁的机制排序的。

1. 用途

Semaphore常用于约束访问一些资源的线程数量。且当信号量初始化为1时，可以当做互斥锁使用，因为该信号量集只有两个状态，在这种状态下，互斥锁可以被其他线程控制和释放，而不是主线程控制释放。

1. 实现

Semaphore通过Sync了实现，Sync类通过AQS实现，因此Semaphore包含着公平锁和非公平锁，在构造信号量时设置给定的许可数，也可以选择使用公平锁或者非公平锁，内部调用Sync类的对应类构建锁，默认是非公平锁。

public Semaphore(int premits){

sync = new NonfairSync(permits);

}

public Semaphore(int permits, Boolean fair){

sync = fair ? new FairSync(premits) : new NonfairSync(permits);

}

1. 信号量的获取

Semaphore提供acquire()方法来获取一个许可：

public void acquire() throws InterruptedException{

sync.acquireSharedInterruptibly(1);

}

内部使用AQS以共享模式获取同步状态，对于公平锁，首先判断当前线程是否位于CLH队列的头部，然后获取当前可用的信号量许可数量，减去索取的信号量数量，通过CAS的compareAndSetState()方法来更改内存，更新信号量。

//公平 判断该线程是否位于CLH队列的列头

protected int tryAcquireShared(int acquires) {

   for (;;) {

       //判断该线程是否位于CLH队列的列头

       if (hasQueuedPredecessors())

           return -1;

       //获取当前的信号量许可

       int available = getState();

​

       //设置“获得acquires个信号量许可之后，剩余的信号量许可数”

       int remaining = available - acquires;

​

       //CAS设置信号量

       if (remaining < 0 ||

           compareAndSetState(available, remaining))

           return remaining;

  }

}

对于非公平锁，不需要判断当前线程是否位于CLH队列的队头，直接获取即可：  
//非公平 对于非公平而言，因为它不需要判断当前线程是否位于CLH同步队列列头。

protected int tryAcquireShared(int acquires) {

   return nonfairTryAcquireShared(acquires);

}

​

final int nonfairTryAcquireShared(int acquires) {

   for (;;) {

       int available = getState();

       int remaining = available - acquires;

       if (remaining < 0 ||

           compareAndSetState(available, remaining))

           return remaining;

  }

}

1. 信号量释放

当线程执行完后，需要释放信号量，通过release()方法释放许可。

public void release(){

sync.releaseShared(1);

}

内部通过调用AQS来释放许可，释放的时候就不需要区分公平锁和非公平锁了，直接释放到信号量许可集中即可，通过CAS操作更新内存中的信号量许可数量。

protected final boolean tryReleaseShared(int releases) {

   for (;;) {

       int current = getState();

       //信号量的许可数 = 当前信号许可数 + 待释放的信号许可数

       int next = current + releases;

       if (next < current) // overflow

           throw new Error("Maximum permit count exceeded");

       //设置可获取的信号许可数为next

       if (compareAndSetState(current, next))

           return true;

  }

}

1. 实例

将汽车要进入停车场作为实例，停车场的空位就是信号量许可，只有存在空位，其他的车辆才可以进入停车场停车。

package thread9;  
  
import java.util.concurrent.Semaphore;  
  
public class SemaphoreDemo {  
  
 public static void main(String[] args) {  
 Parking parking = new Parking(3);  
 for(int i = 0; i < 5; ++i){  
 new Car(parking).start();  
 }  
 }  
  
 static class Parking{  
 private Semaphore semaphore;  
 Parking(int count){  
 semaphore = new Semaphore(count);  
 }  
  
 public void park(){  
 try {  
 //获取信号量  
 semaphore.acquire();  
 long time = (long)(Math.*random*() \* 10);  
 System.*out*.println(Thread.*currentThread*().getName() + "进入停车场，停车" + time + "秒");  
 Thread.*sleep*(time);  
 System.*out*.println(Thread.*currentThread*().getName() + "开出停车场");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 semaphore.release();  
 }  
 }  
 }  
  
 static class Car extends Thread{  
 Parking parking;  
  
 Car(Parking parking){  
 this.parking = parking;  
 }  
  
 @Override  
 public void run(){  
 parking.park();  
 }  
 }  
}