Java中的四个并发工具类

# Java并发的四个并发工具类

## 四个工具类：都存在与java.util.concurrent包中。<重点前三个>

**CountDownLatch计数锁存器**

**CyclicBarrier循环屏障类**

**Semaphore信号量**

**Exchanger<V>交换器**

## CountDownLatch与CyclicBarrier相似

# 多线程

**问题引入**：

通过for循环创造多个线程，但是由于主循环中都是**顺序执行代码**，如何等到多个线程都创建完毕之后，一起启动呢？或者如何等所有线程都执行完毕后，才继续往下执行？

解决方法：

方法1：利用**CountDownLatch类**：计数锁存器类

创建一个**CountDownLatch对象（startSignal）**，初始计数值为1，在各个线程的任务中需要传入该**CountDownLatch对象，在任务的r**un方法中通过调用CountDownLatch对象的await方法，阻塞本线程，等所有线程都启动之后，通过调用**countDown**方法，释放开始信号。

创建一个**CountDownLatch对象（endSignal）**，初始计数值为**线程数目**N，在各个线程的任务中需要传入该**CountDownLatch对象，在任务的r**un方法最后(一般是finally中)通过调用endSignal对象的countDown方法，表示该线程结束了，等所有线程都结束了，那么就会继续往下执行。（当然，开启所有线程后，需要调用endSignal的await方法阻塞主线程）。

方法2：利用**CyclicBarrier**类：循环屏障类

# CountDownLatch类(锁存器)介绍

latch 英 [lætʃ] 美 [lætʃ]

n.**门闩;弹簧锁**

vt.& vi.闩上;用碰锁锁上（门等）;抓住，占有

## CountDownLatch类：锁存器

java.lang.Object

java.util.concurrent.CountDownLatch

**public class CountDownLatch extends Object**

**CountDownLatch**类存在于**java.util.concurrent**并发包中，直接继承与Object类。

## 功能介绍：

**CountDownLatch**：**A synchronization aid** that **allows one or more threads to wait until a set of operations being performed in other threads completes.**

等待**一个或多个线程**创建完毕，然后一起启动。类似计数器的功能。

**CountDownLatch是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。**

A CountDownLatch is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the **countDown()** method, after which all waiting threads are released and any subsequent invocations of await return immediately. **This is a one-shot phenomenon -- the count cannot be reset**. If you need a version that resets the count, consider using a **CyclicBarrier**.

**A CountDownLatch is a versatile synchronization tool** and can be used for a number of purposes. A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate: all threads invoking await wait at the gate until it is opened by a thread invoking **countDown()**. A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.

A useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass.

## 构造方法：只有1个

**CountDownLatch(int count)**

Constructs a CountDownLatch initialized with the given count.

给定一个初始化的数量，创建一个CountDownLatch对象。

**Parameters**:

**count** - the number of times countDown() must be invoked **before threads can pass through await().**

Throws: IllegalArgumentException - if count is negative

## CountDownLatch的方法

**一共5个：await()、await(timeout)、countDown()、getCount()、toString()**

主要方法：**await和countDown方法。**

**await();** // 调用此方法会一直阻塞当前线程，不会向下执行，直到计数值为0的时候该线程才会继续向下执行；使当前线程在锁存器倒计数至零之前一直等待

**countDown();** // 当前线程调用此方法，就会**递减锁存器的计数**；如果计数到达零，则释放所有等待的线程，否则计数值减一。

### public void await() throws InterruptedException

**Causes the current thread to wait** until the latch has counted down to zero, unless the thread is interrupted.

If the current count is zero then this method returns immediately.

If the current count is greater than zero then the current thread becomes disabled for **thread scheduling purposes** and lies dormant until **one of two things happen**:

* 1. The count reaches zero due to invocations of the **countDown**() method; or
  2. Some other thread **interrupts** the current thread.

If the current thread:

has its interrupted status set on entry to this method; or

is interrupted while waiting,

then **InterruptedException** is thrown and the current thread’s interrupted status is cleared.

**Throws**:**InterruptedException** – if the current thread is interrupted while waiting

### public boolean **await(long timeout, TimeUnit unit)** throws InterruptedException

Causes the current thread to wait **until the latch has counted down to zero**, unless the thread is interrupted, or **the specified waiting time elapses**.

If the current count is zero then this method returns immediately with the value **true**.

If the current count is greater than zero then the current thread becomes disabled for thread scheduling purposes and lies dormant until **one of three things happen**:

The count reaches zero due to invocations of the countDown() method; or

Some other thread interrupts the current thread; or

The specified waiting time **elapses**.

If the count reaches zero then the method returns with the value true.

If the current thread:

has its interrupted status set on entry to this method; or

is interrupted while waiting,

then **InterruptedException** is thrown and the current thread’s interrupted status is cleared.

If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.

Parameters: **timeout** – the maximum time to wait

**unit** – the time unit of the timeout argument

Returns:true if the count reached zero and false if the waiting time elapsed before the count reached zero

Throws:**InterruptedException** – if the current thread is interrupted while waiting

### void countDown():使当前门闩的数值减1。

**Decrements the count of the latch**, releasing all waiting threads if the count reaches zero.

If the current count is greater than zero **then it is decremented**. If the new count is zero then all waiting threads **are re-enabled for thread scheduling purposes**.If the current count equals zero then nothing happens.

如果当前数值大于0，那么就减1;若新的数值等于1，那么所有等待的线程就是同时执行；若当前数值等于0，什么也不发生。

### long **getCount()方法**：返回初始化的数目

**long getCount()**：Returns the current count.

This method is typically used for debugging and testing purposes.

### String toString()：

Returns a string identifying this latch, as well as its state.

The state, in brackets, includes the String “**Count =”** followed by the current count.

**Overrides**: toString in class Object

**Returns**: a string identifying this latch, as well as its state

### 继承的方法



## API示例1介绍：

**Sample usage: Here is a pair of classes in which a group of worker threads use two countdown latches: The first is a start signal that prevents any worker from proceeding until the driver is ready for them to proceed;The second is a completion signal that allows the driver to wait until all workers have completed.**

**class Driver { // ...**

**void main() throws InterruptedException {**

**CountDownLatch startSignal = new CountDownLatch(1);//用于开启所有线程**

**CountDownLatch doneSignal = new CountDownLatch(N);//等待所有线程结束之后，继续主线程**

**for (int I = 0; I < N; ++i){ // create and start threads**

**new Thread(new Worker(startSignal, doneSignal)).start();**

**}**

**doSomethingElse(); // don’t let run yet**

**startSignal.countDown(); // let all threads proceed**

**doSomethingElse();**

**doneSignal.await(); // wait for all to finish**

**// …等待所有线程执行完毕后，再继续执行**

**}**

**}**

**class Worker implements Runnable {**

**private final CountDownLatch startSignal;**

**private final CountDownLatch doneSignal;**

**Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {**

**this.startSignal = startSignal;**

**this.doneSignal = doneSignal;**

**}**

**public void run() {**

**try {**

**startSignal.await();**

**doWork();**

**doneSignal.countDown();**

**} catch (InterruptedException ex) {} // return;**

**}**

**void doWork() { ... }**

**}**

## API示例2介绍

**Another typical usage would be to divide a problem into N parts, describe each part with a Runnable that executes that portion and counts down on the latch, and queue all the Runnables to an Executor. When all sub-parts are complete, the coordinating thread will be able to pass through await. (When threads must repeatedly count down in this way, instead use a CyclicBarrier.)**

**class Driver2 { // ...**

**void main() throws InterruptedException {**

**CountDownLatch doneSignal = new CountDownLatch(N);**

**Executor e = ...**

**for (int I = 0; I < N; ++i) // create and start threads**

**e.execute(new WorkerRunnable(doneSignal, i));**

**doneSignal.await(); // wait for all to finish**

**}**

**}**

**class WorkerRunnable implements Runnable {**

**private final CountDownLatch doneSignal;**

**private final int I;**

**WorkerRunnable(CountDownLatch doneSignal, int i) {**

**this.doneSignal = doneSignal;**

**this.i = I;**

**}**

**public void run() {**

**try {**

**doWork(i);**

**doneSignal.countDown();//当前线程完成了**

} catch (InterruptedException ex) {} // return;

}

void doWork() { ... }

}

Memory consistency effects: Until the count reaches zero, actions in a thread prior to calling countDown() happen-before actions following a successful return from a corresponding await() in another thread.

## CountDownLatch应用

只需要掌握好下面三种方法即可：

public void await() throws InterruptedException

public boolean **await(long timeout, TimeUnit unit)** throws InterruptedException

void countDown():使当前门闩的数值减1。

### CountDownLatch中发生阻塞的方法只有await方法，对于countDown方法不会阻塞；

鉴于此，CountDownLatch的countDown不仅可以应用于多个线程中，也可以都应用于一个线程中。本质上就是计数，一个线程计数于多个线程同时计数达到的效果是一样的，都是在到达0时，调用await方法的线程可以继续往下执行。

同时，await方法也可以在多个线程中同时调用，调用await方法的线程都是根据count值是否为0了，来决定是否继续往下执行。举个例子，跑步比赛活动，5个小伙伴负责计数，5个小伙伴听到0时，就开始跑，也就是计数的5个小伙伴利用countDown减一，而跑步的5个小伙伴只关心是否到达零。

当count值已经到达零，再次调用countDown方法，什么都不做，立刻返回；count值不会递减。

### await方法:

防止等待时间过长，await方法可以传递等待的最大时长，超过等待时长，调用await方法的线程就会继续往下执行。

## 使用场景1：（不如API示例）

类介绍有两个示例，见API介绍。

场景1：当需要等待**所有线程都准备好**之后，一起启动的场景（公平原则）。

import java.util.Scanner;

import java.util.concurrent.CountDownLatch;

import org.junit.Test;

public class **CountDownLatchTest** {

@Test

public void test() throws InterruptedException{

int N = 10;

CountDownLatch countDownLatch = new CountDownLatch(N);

for(int i = 0;i < N;i++ ){

**new Thread(new Target(countDownLatch)).start();//创建线程**

countDownLatch.countDown();

System.out.print(countDownLatch.getCount()+",");//获取计数值的大小

}

Scanner scanner = new Scanner(System.in);

scanner.next();//阻塞主线程

}

}

/\*

\* 封装任务类

\*/

class Target implements Runnable{

private CountDownLatch countDownLatch;

**public Target(CountDownLatch countDownLatch){**

**this.countDownLatch = countDownLatch;**

**}**

@Override

public void run() {

try {

**countDownLatch.await();//线程等待,计数值为0**

doSomethingElse();//线程其他任务

}catch (InterruptedException e) {

e.printStackTrace();

}finally{

**countDownLatch.countDown();//目的：一旦发生异常，保证计数值减一**

}

}

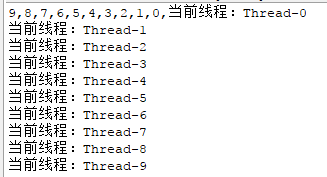
private void doSomethingElse() {

System.out.println("当前线程："+Thread.currentThread().getName());

}

}

执行结果：



实现方法：需要将创建的CountDownLatch锁存器对象传入到任务中，使多个线程同时共享一个CountDownLatch对象，这样锁存器对象才可以统一管理这多个线程。在Runnable实现类的run方法中首先执行**countDownLatch.await()方法**，阻塞线程。注意在**finally**中调用countDown()方法,避免异常发生，导致线程始终无法启动。

在线程执行的外部，调用countDownLatch.countDown()正常递减计数值，等到减为0，同时启动多个线程。

## 使用场景2：（不如API示例）

程序启动的时候，根据业务**开启几个线程**去执行检查服务是否正常，主线程一直等待，当检查的线程都结束的时候，主线程才能去判断对应的所有的线程返回结果，检查是否正常。

比如有一个任务A，它要等待**其他3个检查任务执行完毕之后**才能执行，此时就可以利用**CountDownLatch**来实现这种功能了。

代码示例：

import java.util.concurrent.CountDownLatch;

import org.junit.Test;

public class **CountDownLatchTest** {

@Test

public void test() throws InterruptedException{

int N = 3;

CountDownLatch countDownLatch = new CountDownLatch(N);

**new Thread(new Check1(countDownLatch)).start();//执行检查1任务**

**new Thread(new Check2(countDownLatch)).start();//执行检查2任务**

**new Thread(new Check3(countDownLatch)).start();//执行检查3任务**

countDownLatch.await();//等待以上三次检查完毕，才能继续执行

System.out.print("三次检查完毕,继续执行.....");//获取计数值的大小

System.out.print("------end------");//获取计数值的大小

}

}

/\*

\* 封装任务类

\*/

class Check1 implements Runnable{

private CountDownLatch countDownLatch;

public Check1(CountDownLatch countDownLatch){

this.countDownLatch = countDownLatch;

}

public void run() {

try {

Thread.currentThread().sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("第一步检查结束!");

**countDownLatch.countDown();/**/目的：一旦发生异常，保证计数值减一

}

}

class Check2 implements Runnable{

private CountDownLatch countDownLatch;

public Check2(CountDownLatch countDownLatch){

this.countDownLatch = countDownLatch;

}

public void run() {

try {

Thread.currentThread().sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("第二步检查结束!");

**countDownLatch.countDown();**//目的：一旦发生异常，保证计数值减一

}

}

class Check3 implements Runnable{

private CountDownLatch countDownLatch;

public Check3(CountDownLatch countDownLatch){

this.countDownLatch = countDownLatch;

}

public void run() {

try {

Thread.currentThread().sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("第三步检查结束!");

countDownLatch.countDown();//目的：一旦发生异常，保证计数值减一

}

}

## 使用场景3

在一个线程任务中，多次调用countDown方法；

也可以开启多个线程，在这多个线程中都调用await方法。

Runnable task1 = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("任务1完成...");  
 countDownLatch.countDown();  
 System.*out*.println("任务2完成...");  
 countDownLatch.countDown();  
 System.*out*.println("任务3完成...");  
 countDownLatch.countDown();  
 }  
};

Runnable task2 = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("等待所有任务执行完毕......");  
 try {  
 countDownLatch.await();  
 System.*out*.println("count = " + countDownLatch.getCount());  
 }catch(Exception e){  
 System.*out*.println("异常e = " + e);  
 }  
 System.*out*.println("所有任务已经执行完毕了");  
  
 }  
};

# CyclicBarrier类-循环屏障类

## CyclicBarrier类

public class **CyclicBarrier** extends Object

**CyclicBarrier**类存在于java.util.concurrent包中，直接继承于Object，属于并发包中的一个类。

CyclicBarrier可翻译为**可循环使用的(Cyclic)屏障(Barrier)**;

**cyclic** 英[ˈsaɪklɪk] 美[ˈsaɪklɪk, ˈsɪklɪk] adj. 周期的，循环的; 轮转的;

## 功能：

A **synchronization** aid that **allows a set of threads to all wait for each other to reach a common barrier point**. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. **The barrier is called cyclic because it can be re-used after the waiting threads are released**.(重复使用，通过**reset方法**)

A CyclicBarrier supports an optional Runnable command that is run **once per barrier point**, after the last thread in the party arrives, but before any threads are released. (线程执行之前)This barrier action is useful **for updating shared-state** before any of the parties continue.

这个**barrierAction**执行时间是**所有线程都达到了屏障点，所有线程开始执行之前，先执行barrierAction**。

CyclicBarrier要做的事情是，**让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活**。CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier还提供一个**更高级的构造函数CyclicBarrier(int parties, Runnable barrierAction)**，用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。

**trip**  [英][trɪp][美][trɪp]

n. 旅游，出行;**摔倒，绊倒**;绊（使某人跌倒的动作）;错误，过失

vi. **绊倒，绊**;旅行;轻快地走，跳

vt. 使犯错误;松开（离合器等）以开动

## 构造方法2个

**CyclicBarrier(int parties)**

Creates a new CyclicBarrier that will **trip** when the given number of parties (threads) are waiting upon it, and **does not perform a predefined action** when the barrier is tripped.

**CyclicBarrier(int parties, Runnable barrierAction)**

Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and **which will execute the given barrier action** when the barrier is tripped, **performed by the last thread entering the barrier**.

## 方法介绍

### await() 、await(long timeout, TimeUnit unit)

和CountDownLatch的相似，但是返回类型不一样。

int await()

Waits until all parties have invoked await on this barrier.

int await(long timeout, TimeUnit unit)

Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.

Returns: **the arrival index of the current thread**, where index **getParties() - 1** indicates the first to arrive and zero indicates the last to arrive.

### getNumberWaiting

int getNumberWaiting()

Returns the number of parties **currently waiting at the barrier**.

### getParties

int getParties()

Returns the number of parties **required to trip this barrier**.

### isBroken

boolean **isBroken**()

Queries if this barrier is in a broken state.

### reset

void reset()

Resets the barrier to **its initial state.**

## 应用场景-与API场景类似

CyclicBarrier可以用于**多线程计算数据**，最后合并计算结果的应用场景。比如我们用一个Excel保存了用户所有银行流水，每个Sheet保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用**barrierAction**用这些线程的计算结果，计算出整个Excel的日均银行流水。

**Sample usage**: Here is an example of using a barrier in a parallel decomposition design:

class Solver {

final int N;

final float[][] data;

**final CyclicBarrier barrier;**

class Worker implements Runnable {

int myRow;

Worker(int row) { myRow = row; }

public void run() {

while (!done()) {

**processRow(myRow);//先处理指定行，然后等待**

try {

**barrier.await();**

} catch (InterruptedException ex) {

return;

} catch (BrokenBarrierException ex) {

return;

}

}

}

}

public Solver(float[][] matrix) {

data = matrix;

N = matrix.length;

**Runnable barrierAction =**

**new Runnable() { public void run() { mergeRows(...); }};//线程退出之前，合并所有行**

**barrier = new CyclicBarrier(N, barrierAction);**

List<Thread> threads = new ArrayList<Thread>(N);

for (int i = 0; i < N; i++) {

Thread thread = new Thread(new Worker(i));

threads.add(thread);

thread.start();

}

// wait until done

for (Thread thread : threads)

thread.join();

}

}

Here, **each worker thread processes a row of the matrix** then waits at the barrier until all rows have been processed. When all rows are processed the supplied Runnable barrier action is executed and merges the rows. If the merger determines that a solution has been found then done() will return true and each worker will terminate.

注意：barrierAction是在所有线程调用**CyclicBarrier.await()方法**之后执行，所以可以在await方法之前，执行一定的工作。

If the barrier action does not rely on the parties being suspended when it is executed, then any of the threads in the party could execute that action when it is released. **To facilitate this, each invocation of await() returns the arrival index of that thread at the barrier**. You can then choose which thread should execute the barrier action, for example:

if (barrier.await() == 0) {

// log the completion of this iteration

}

The CyclicBarrier uses an **all-or-none breakage model** for failed synchronization attempts: If a thread leaves a barrier point prematurely because of interruption, failure, or timeout, **all other threads waiting at that barrier point will also leave abnormally via BrokenBarrierException (or InterruptedException if they too were interrupted at about the same time).（同生共死）**

**Memory consistency effects**: Actions in a thread prior to calling await() **happen-before actions** that are part of the barrier action, which in turn happen-before actions following a successful return from the corresponding await() in other threads.

# CyclicBarrier和CountDownLatch的区别

## 使用次数不同

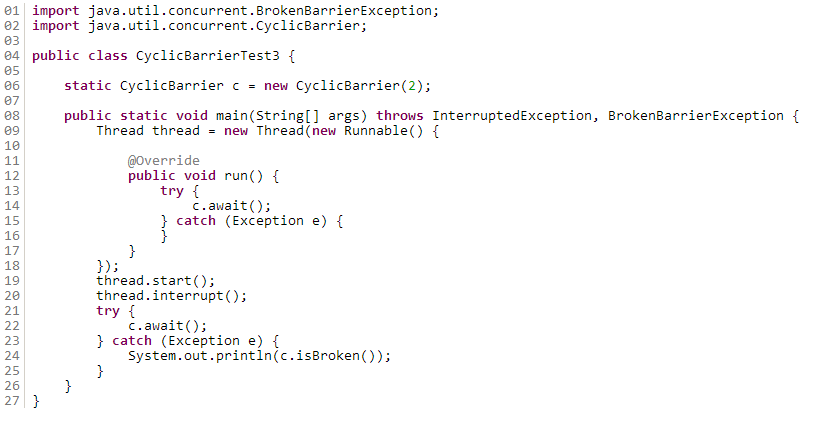
**CountDownLatch的计数器**只能使用一次；而CyclicBarrier的计数器可以使用**reset() 方法**重置，循环使用，这也是为啥叫做**CyclicBarrier**的原因。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

## 使用上的区别

CyclicBarrier只需要调用**await方法**，自动记录到达线程的个数；而CountDownLatch需要await和countDown方法结合，await只能阻塞当前线程，而真正启动这些线程需要调用countDown方法。

## CyclicBarrier提供方法较多

CyclicBarrier还提供其他有用的方法，比如**getNumberWaiting**方法可以获得CyclicBarrier阻塞的线程数量；**isBroken**方法用来知道阻塞的线程是否被中断。比如以下代码执行完之后会返回true。



输出：true 屏障已经破坏，无法循环使用。

## 总结：

CountDownLatch和CyclicBarrier**都能够实现线程之间的等待**，只不过它们侧重点不同：CountDownLatch一般用于**某个线程A等待若干个其他线程执行完任务之后，它才执行**；而CyclicBarrier一般用于**一组线程互相等待至某个状态，然后这一组线程再同时执行**；另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

# Semaphore-信号量--介绍

## Semophore信号量的包及继承关系

单词介绍： **semaphore** 英[ˈseməfɔ:(r)] 美[ˈsɛməˌfɔr, -ˌfor]

n. 臂板信号系统，（铁道）臂板信号装置; vt. **发出信号，打旗语**

public class **Semaphore** extends **Object** implements **Serializable**

**Semaphore**类存在于java.util.concurrent包中，直接继承于Object类，实现了Serializable接口。

## 功能介绍

**Semaphore**又称信号量，是操作系统中的一个概念，在**Java并发编程**中，信号量控制的是**线程并发的数量**。Semaphore类用于**控制并发线程数**。

public **Semaphore**(int permits)；其中参数permits就是允许**同时运行的线程数目**。

**Semaphore（信号量）**是**用来控制同时访问特定资源的线程数量**，它通过协调各个线程，以保证合理的使用公共资源。很多年以来，我都觉得从字面上很难理解Semaphore所表达的含义，只能把它比作是控制流量的红绿灯，比如XX马路要限制流量，只允许同时有一百辆车在这条路上行使，其他的都必须在路口等待，所以前一百辆车会看到绿灯，可以开进这条马路，后面的车会看到红灯，不能驶入XX马路，但是如果前一百辆中有五辆车已经离开了XX马路，那么后面就允许有5辆车驶入马路，这个例子里说的车就是线程，驶入马路就表示线程在执行，离开马路就表示线程执行完成，看见红灯就表示线程被阻塞，不能执行。

**A counting semaphore**. Conceptually, a semaphore maintains **a set of permits**. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.

Semaphores are often used **to restrict the number of threads** than can access some (physical or logical) resource.

A semaphore **initialized to one**, and which is used such that it only has at most one permit available, can serve as a mutual exclusion lock. This is more commonly known as a binary semaphore, because it only has two states: one permit available, or zero permits available. When used in this way, the binary semaphore has the property (unlike many Lock implementations), that the "lock" can be released by a thread other than the owner (as semaphores have no notion of ownership). This can be useful in some specialized contexts, such as deadlock recovery.

This class also provides convenience methods to acquire and release multiple permits at a time. Beware of the increased risk of indefinite postponement when these methods are used without fairness set true.

Memory consistency effects: Actions in a thread prior to calling a "**release**" method such as **release**() happen-before actions following a successful "**acquire**" method such as **acquire**() in another thread.

## 构造方法—Constructors

### Semaphore(int permits)

Creates a Semaphore with **the given number of permits** and nonfair fairness setting.

Parameters:**permits** - the initial number of permits available. This value may be negative, in which case **releases** must occur **before** any **acquires** will be granted.

### Semaphore(int permits, boolean fair)

Creates a Semaphore with the given number of permits and **the given fairness setting**.

**Parameters**: **permits** - the initial number of permits available. This value may be negative, in which case releases must occur before any acquires will be granted.

**fair** - true if this semaphore will guarantee **first-in first-out**(FIFO) granting of permits under contention, else false

对于fair的详细解释：

The constructor for this class optionally accepts **a fairness parameter**. ①When set false, this class makes no guarantees about the order in which threads acquire permits. In particular, barging is permitted, that is, a thread invoking acquire() can be allocated a permit ahead of a thread that has been waiting - logically the new thread places itself at the head of the queue of waiting threads. ②When fairness is set true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed (**first-in-first-out; FIFO**). Note that FIFO ordering necessarily applies to specific internal points of execution within these methods. So, it is possible for one thread to invoke acquire before another, but reach the ordering point after the other, and similarly upon return from the method. Also note that the untimed **tryAcquire** methods do not honor the fairness setting, but will take any permits that are available.

Generally, semaphores used to control resource access should be initialized **as fair**, to ensure that no thread is starved out from accessing a resource. When using semaphores for other kinds of synchronization control, the throughput advantages of non-fair ordering often outweigh fairness considerations.当用于控制资源连接的时候，建议使用公平，对其他用途，不公平更好一些。

## 方法介绍

最重要的方法就是acquire和release

### acquire():获取一个permit许可证。

void **acquire()**

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

具体见acquire(int permits)，就是个数不同。

### acquire(int permits)：获取指定个数个permit许可证。

void acquire(int permits)

**Acquires the given number of permits** from this semaphore, blocking until all are available, or the thread is interrupted.

Acquires the given number of permits, **if they are available,** and returns immediately, reducing the number of available permits by the given amount.

**If insufficient permits are available** then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

1. Some other thread invokes one of the release methods for this semaphore, the current thread is next to be assigned permits and the number of available permits satisfies this request; or
2. Some other thread interrupts the current thread.

If the current thread:

has its interrupted status set on entry to this method; or

is interrupted while waiting for a permit,

then InterruptedException is thrown and the current thread's interrupted status is cleared. Any permits that were to be assigned to this thread are instead assigned to other threads trying to acquire permits, as if permits had been made available by a call to **release()**.

**Parameters**:permits - the number of permits to acquire

Throws:

InterruptedException - if the current thread is interrupted

IllegalArgumentException - if permits is negative

### release()：归还permit许可证。

void release()

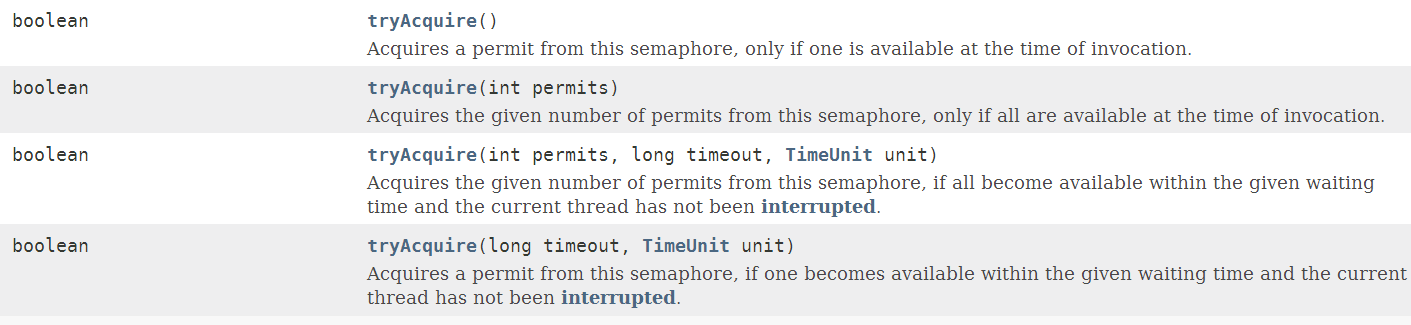
Releases a **permit**, returning it to the semaphore.

### release(int permits)：归还指定个数个permit许可证。

void release(int permits)

**Releases the given number of permits**, returning them to the semaphore.

### tryAcquire



public boolean **tryAcquire**(int permits)

Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.

Acquires the given number of permits, if they are available, and returns immediately, with the value true, reducing the number of available permits by the given amount.

If insufficient permits are available then this method will return immediately with the value **false** and the number of available permits is unchanged.

Even when this semaphore has been set to use a fair ordering policy, a call to tryAcquire will immediately acquire a permit if one is available, whether or not other threads are currently waiting. This "**barging**" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting, then use **tryAcquire(permits, 0, TimeUnit.SECONDS)** which is almost equivalent (it also detects interruption).

Parameters: **permits** - the number of permits to acquire

Returns:**true** if the permits were acquired and false otherwise

Throws:IllegalArgumentException - if permits is negative

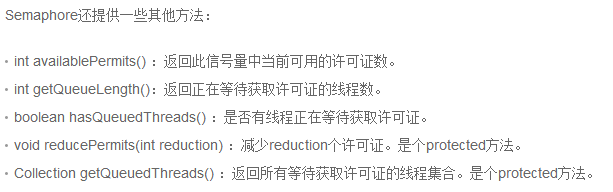
### drainPermits():获取并返回可用许可证数目，并将许可证置为0.

public int **drainPermits()**

**Acquires** and **returns** all permits that are immediately available.

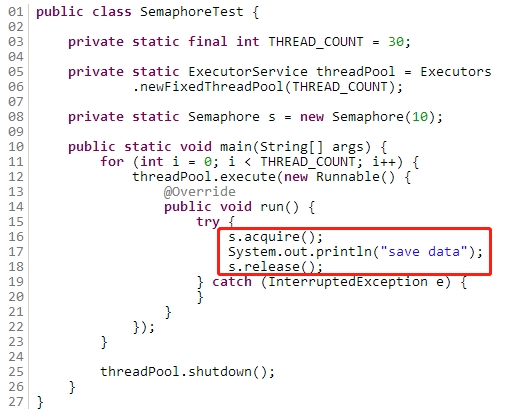
**Returns**:the number of permits acquired

### 其他方法



## 应用场景

**Semaphore**可以用于做**流量控制**，特别公用资源有限的应用场景，比如**数据库连接**。假如有一个需求，要读取几万个文件的数据，因为都是**IO密集型任务**，我们可以启动几十个线程并发的读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有十个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，我们就可以使用**Semaphore**来做流控。代码如下



在代码中，虽然有30个线程在执行，但是只允许10个并发的执行。Semaphore的构造方法Semaphore(int permits) 接受一个整型的数字，表示可用的**许可证数量**。Semaphore(10)表示允许10个线程获取许可证，**也就是最大并发数是10**。Semaphore的用法也很简单，首先线程**使用Semaphore的acquire()获取一个许可证**，使用完之后**调用release()归还许可证**。**还可以用tryAcquire()方法尝试获取许可证**。