

## 原 深入理解Semaphore

2017年04月17日 16:02:28 阅读数: 24656

标签:

semaphore

线程

并发

java

更多

版权声明: 本文为博主原创文章, 未经博主允许不得转载。 [https://blog.csdn.net/qq\\_19431333/article/details/70212663](https://blog.csdn.net/qq_19431333/article/details/70212663)

## 使用

Semaphore是计数信号量。Semaphore管理一系列许可证。每个acquire方法阻塞, 直到有一个许可证可以获得然后拿走一个许可证; 每个release方法增加一个许可证, 这可能会释放一个阻塞的acquire方法。然而, 其实并没有实际的许可证这个对象, Semaphore只是维持了一个可获得许可证的数量。

Semaphore经常用于限制获取某种资源的线程数量。下面举个例子, 比如说操场上有5个跑道, 一个跑道一次只能有一个学生上面跑步, 一旦所有跑道在使用, 那后面的学生就需要等待, 直到有一个学生不跑了, 下面是这个例子:

```
1  /**
2   * 操场, 有5个跑道
3   * Created by Xingfeng on 2016-12-09.
4   */
5  public class Playground {
6
7      /**
8       * 跑道类
9       */
10     static class Track {
11         private int num;
12
13         public Track(int num) {
14             this.num = num;
15         }
16
17         @Override
18         public String toString() {
19             return "Track{" +
20                 "num=" + num +
21                 '}';
22     }
23 }
```

```
22     }
23 }
24
25 private Track[] tracks = {
26     new Track(1), new Track(2), new Track(3), new Track(4), new Track(5)};
27 private volatile boolean[] used = new boolean[5];
28
29 private Semaphore semaphore = new Semaphore(5, true);
30
31 /**
32  * 获取一个跑道
33  */
34 public Track getTrack() throws InterruptedException {
35
36     semaphore.acquire(1);
37     return getNextAvailableTrack();
38 }
39
40
41 /**
42  * 返回一个跑道
43  *
44  * @param track
45  */
46 public void releaseTrack(Track track) {
47     if (makeAsUsed(track))
48         semaphore.release(1);
49 }
50
51 /**
52  * 遍历，找到一个没人用的跑道
53  *
54  * @return
55  */
56 private Track getNextAvailableTrack() {
57
58     for (int i = 0; i < used.length; i++) {
59         if (!used[i]) {
```

4

6

```
60         used[i] = true;
61         return tracks[i];
62     }
63 }
64
65     return null;
66
67 }
68
69 /**
70  * 返回一个跑道
71  *
72  * @param track
73  */
74 private boolean makeAsUsed(Track track) {
75
76     for (int i = 0; i < used.length; i++) {
77         if (tracks[i] == track) {
78             if (used[i]) {
79                 used[i] = false;
80                 return true;
81             } else {
82                 return false;
83             }
84         }
85     }
86 }
87
88     return false;
89 }
90
91 }
```

4

6

上面可以看到，创建了5个跑道对象，并使用一个boolean类型的数组记录每个跑道是否被使用了，初始化了5个许可证的Semaphore，在获取跑道时首先调用acquire、获取一个许可证，在归还一个跑道是调用release(1)释放一个许可证。接下来再看启动程序，如下：

```
1 public class SemaphoreDemo {
2
3     static class Student implements Runnable {
4
5         private int num;
6         private Playground playground;
7
8         public Student(int num, Playground playground) {
9             this.num = num;
10            this.playground = playground;
11        }
12
13        @Override
14        public void run() {
15
16            try {
17                //获取跑道
18                Playground.Track track = playground.getTrack();
19                if (track != null) {
20                    System.out.println("学生" + num + "在" + track.toString() + "上跑步");
21                    TimeUnit.SECONDS.sleep(2);
22                    System.out.println("学生" + num + "释放" + track.toString());
23                    //释放跑道
24                    playground.releaseTrack(track);
25                }
26            } catch (InterruptedException e) {
27                e.printStackTrace();
28            }
29
30        }
31    }
32
33    public static void main(String[] args) {
34
35        Executor executor = Executors.newCachedThreadPool();
36        Playground playground = new Playground();
37        for (int i = 0; i < 100; i++) {
```

4

6

```
38         executor.execute(new Student(i+1,playground));
39     }
40
41 }
42
43 }
```

4

6

上面的代码中，Student类代表学生，首先获取跑道，一旦获取到就打印一句话，然后睡眠2s，然后再打印释放，最后归还跑道。

## 源码解析

Semaphore有两种模式，公平模式和非公平模式。公平模式就是调用acquire的顺序就是获取许可证的顺序，遵循FIFO；而非公平模式是抢占式的，也就是有可能新的获取线程恰好在一个许可证释放时得到了这个许可证，而前面还有等待的线程。

## 构造方法

Semaphore有两个构造方法，如下：

```
1     public Semaphore(int permits) {
2         sync = new NonfairSync(permits);
3     }
4
5     public Semaphore(int permits, boolean fair) {
6         sync = fair ? new FairSync(permits) : new NonfairSync(permits);
7     }
```

从上面可以看到两个构造方法，都必须提供许可的数量，第二个构造方法可以指定是公平模式还是非公平模式，默认非公平模式。

Semaphore内部基于AQS的共享模式，所以实现都委托给了Sync类。

这里就看一下NonfairSync的构造方法：

```
1     NonfairSync(int permits) {
2         super(permits);
3     }
```

可以看到直接调用了父类的构造方法，Sync的构造方法如下：

```
1 Sync(int permits) {  
2     setState(permits);  
3 }
```

可以看到调用了setState方法，也就是说AQS中的资源就是许可证的数量。

## 获取许可

先从获取一个许可看起，并且先看非公平模式下的实现。首先看acquire方法，acquire方法有几个重载，但主要是下面这个方法

```
1 public void acquire(int permits) throws InterruptedException {  
2     if (permits < 0) throw new IllegalArgumentException();  
3     sync.acquireSharedInterruptibly(permits);  
4 }
```

从上面可以看到，调用了Sync的acquireSharedInterruptibly方法，该方法在父类AQS中，如下：

```
1 public final void acquireSharedInterruptibly(int arg)  
2     throws InterruptedException {  
3     //如果线程被中断了，抛出异常  
4     if (Thread.interrupted())  
5         throw new InterruptedException();  
6     //获取许可失败，将线程加入到等待队列中  
7     if (tryAcquireShared(arg) < 0)  
8         doAcquireSharedInterruptibly(arg);  
9 }
```

AQS子类如果要使用共享模式的话，需要实现tryAcquireShared方法，下面看NonfairSync的该方法实现：

```
1 protected int tryAcquireShared(int acquires) {  
2     return nonfairTryAcquireShared(acquires);  
3 }
```

该方法调用了父类中的nonfairTyAcquireShared方法，如下：

```
1 final int nonfairTryAcquireShared(int acquires) {
2     for (;;) {
3         //获取剩余许可数量
4         int available = getState();
5         //计算给完这次许可数量后的个数
6         int remaining = available - acquires;
7         //如果许可不够或者可以将许可数量重置的话，返回
8         if (remaining < 0 ||
9             compareAndSetState(available, remaining))
10            return remaining;
11    }
12 }
```

4

6

从上面可以看到，只有在许可不够时返回值才会小于0，其余返回的都是剩余许可数量，这也就解释了，一旦许可不够，后面的线程将会阻塞。看完了不公平的获取，看下公平的获取，代码如下：

```
1 protected int tryAcquireShared(int acquires) {
2     for (;;) {
3         //如果前面有线程再等待，直接返回-1
4         if (hasQueuedPredecessors())
5             return -1;
6         //后面与非公平一样
7         int available = getState();
8         int remaining = available - acquires;
9         if (remaining < 0 ||
10            compareAndSetState(available, remaining))
11            return remaining;
12    }
13 }
```

上面可以看到，FairSync与非FairSync的区别就在于会首先判断当前队列中有没有线程在等待，如果有，就老老实实进入到等待队列；而不像NonfairSync一样首先...一把，说不定就恰好获得了一个许可，这样就可以插队了。

完了获取许可后，再看一下释放许可。

## 释放许可

释放许可也有几个重载方法，但都会调用下面这个带参数的方法，

```
1 public void release(int permits) {
2     if (permits < 0) throw new IllegalArgumentException();
3     sync.releaseShared(permits);
4 }
```

releaseShared方法在AQS中，如下：

```
1 public final boolean releaseShared(int arg) {
2     //如果改变许可数量成功
3     if (tryReleaseShared(arg)) {
4         doReleaseShared();
5         return true;
6     }
7     return false;
8 }
```

AQS子类实现共享模式的类需要实现tryReleaseShared类来判断是否释放成功，实现如下：

```
1 protected final boolean tryReleaseShared(int releases) {
2     for (;;) {
3         //获取当前许可数量
4         int current = getState();
5         //计算回收后的数量
6         int next = current + releases;
7         if (next < current) // overflow
8             throw new Error("Maximum permit count exceeded");
9         //CAS改变许可数量成功，返回true
10        if (compareAndSetState(current, next))
11            return true;
12    }
13 }
```

从上面可以看到，一旦CAS改变许可数量成功，那么就会调用doReleaseShared()方法释放阻塞的线程。



## 减小许可数量

Semaphore还有减小许可数量的方法，该方法可以用于当资源用完不能再用时，这时就可以减小许可证。代码如下：

```
1  protected void reducePermits(int reduction) {
2      if (reduction < 0) throw new IllegalArgumentException();
3      sync.reducePermits(reduction);
4  }
```

可以看到，委托给了Sync，Sync的reducePermits方法如下：

```
1  final void reducePermits(int reductions) {
2      for (;;) {
3          //得到当前剩余许可数量
4          int current = getState();
5          //得到减完之后的许可数量
6          int next = current - reductions;
7          if (next > current) // underflow
8              throw new Error("Permit count underflow");
9          //如果CAS改变成功
10         if (compareAndSetState(current, next))
11             return;
12     }
13 }
```

从上面可以看到，就是CAS改变AQS中的state变量，因为该变量代表许可证的数量。

## 获取剩余许可数量

maphore还可以一次将剩余的许可数量全部取走，该方法是drain方法，如下：

```
1  public int drainPermits() {
2      return sync.drainPermits();
3  }
```

Sync的实现如下：

```
1 final int drainPermits() {  
2     for (;;) {  
3         int current = getState();  
4         if (current == 0 || compareAndSetState(current, 0))  
5             return current;  
6     }  
7 }
```

4

6

可以看到，就是CAS将许可数量置为0。

## 总结

Semaphore是信号量，用于管理一组资源。其内部是基于AQS的共享模式，AQS的状态表示许可证的数量，在许可证数量不够时，线程将会被挂起；而一旦有一个线程释放一个资源，那么就有可能重新唤醒等待队列中的线程继续执行。



想对作者说点什么



星空dream: 博主，请求转载一下 (07-02 20:08 #3楼)



zdy845986584: [reply]java\_augur[/reply] 你好，能说的详细点吗？ (03-27 17:29 #2楼)



hai319: 楼主，你好。我有个疑问想请教一下，在demo里面，Playground类用于记录跑道使用情况的布尔数组是否需要加上volatile修饰呢？没有这个修饰的话，会不会有可能出现，两个线程同时获取了布尔数组里的同一个值（比如used[0]），而该值刚好为false，然后导致同时获取了同一个跑道这种情况？ (03-08 20:11 #1楼) [查看回复\(3\)](#)

## Java并发编程中Semaphore的用法

1.2万

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。pub...

## Semaphore的介绍和使用

1850

<https://wosyingjun.iteye.com/blog/2299860> 一个计数信号量。从概念上讲，信号量维护了一个许可集。如有必要...

## Java并发之Semaphore详解

 2.2万

一、入题 Semaphore是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信...

## Semaphore详解1

 1579

Semaphore类其实就是synchronized关键字的升级版，这个类主要作用就是控制线程并发的数量，而在这方面sy...

## Semaphore

 94

Semaphore信号量，可以控同时访问的线程个数，acquire() 失败就等待，而 release() 释放一个许可。来看看...

## Semaphore实现原理分析

 2138

synchronized的语义是互斥锁，就是在同一时刻，只有一个线程能获得执行代码的锁。但是现实生活中，有好多...



## 征婚-找个程序猿当男朋友

想找个合适的男朋友，谈以结婚为目的的恋爱

## semaphore的简介

 901

(一)关于semaphore的介绍Semaphore 通常是限制访问某些资源的现成数目的工具类,自从java5.0开始 在java的i...

## 聊一聊Semaphore

 623

原文链接: <http://www.cnblogs.com/liuling/p/2013-8-20-03.html>

<http://ifeve.com/conc...>

## Semaphore的工作原理及实例

 289

Semaphore是一种在多线程环境下使用的设施，该设施负责协调各个线程，以保证它们能够正确、合理的...

## Semaphore使用

 39

Semaphore 实现了信号量，概念上讲，一个信号量相当于持有一些许可（permits），线程可以调用Semaphore...

4

6

## 文章热词

java word打开 java 模板+参数 java收集控制台一行 java代码抽奖 java 对象动态堆

## 相关热词

和的深入 it深入技能 docker深入 nioio深入 apache深入

## Semaphore用法

 101

semaphore用法 #- encoding:utf-8 -\*- import multiprocessing import time import sys reload(sys) sys.s...

## Semaphore的介绍和使用

 185

一个计数信号量。从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire(), ...

## Java多线程系列--【JUC锁11】 - Semaphore信号量的原理和示例

 296

参考：<http://www.cnblogs.com/skywang12345/p/3534050.html> 概要 本章，我们对JUC包中的信号量Semaphor...

## 并发工具类（三）控制并发线程的数量 Semaphore

 184

前言 JDK中为了处理线程之间的同步问题，除了提供锁机制之外，还提供了几个非常实用的并发工具类：C...

## JDK 源码解析 —— Semaphore

 1673

这是一个用来对并发计数的信号量，并发量超过一定数值则只能等待。从概念上来说，semaphore 维持着一组...

## 当一切面对面实时,尽是那样残忍\*\_\*~~~

 551

原来可以回家的,但因为天公不做美,铁路中断,整个线路全部中断,我被遗弃在了广州..... 很不情愿, ...

## Java并发编程--深入理解Semaphore

 674

Semaphore简介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...

## 聊高并发（二十一）解析java.util.concurrent各个组件（三）深入...

 5079

S是AbstractQueuedSynchronizer的缩写，AQS是Java并发包里大部分同步器的基础构件，利用AQS可以很方...

深入理解OkHttp源码（一）——提交请求

 2117

本篇文章主要介绍OkHttp执行同步和异步请求的大体流程。主要流程如下图： 主要分析到getResponseWidthInt...

iOS学习之GCD 信号量详解，dispatch\_semaphore、NSOperationQ...

 3322

当我们在处理一系列线程的时候，当数量达到一定量，在以前我们可能会选择使用NSOperationQueue来处理并...

个人资料




xingfeng\_co...

关注

原创	粉丝	喜欢	评论
70	124	97	45

等级：**博客 5**      访问：19万+

积分：2293      排名：2万+

勋章：

最新文章

- 开发Gradle插件并上传至本地maven库流程
- EventBus源码分析之线程分发
- EventBus源码分析之订阅-发布模型
- EventBus配置、粘性事件、优先级和取消事件分发
- EventBus的线程分发

个人分类

Android基础	14篇
OkHttp框架解析	5篇
Android官方文档译文	1篇
深入浅出Android	3篇
.java集合库源码解析	9篇

展开

归档

2018年8月	2篇
2018年7月	4篇
2017年10月	2篇
2017年9月	5篇
2017年8月	3篇

展开

热门文章

- 深入理解Semaphore  
阅读量：24515
- 美化你的APP——从Toolbar开始  
阅读量：18023

4
6

- 深入理解读写锁—ReadWriteLock源码分析

阅读量：16518
- 使用OkHttp进行网络同步异步操作

阅读量：16090
- android WebView拦截请求详解

阅读量：11253

最新评论

- 深入理解OkHttp源码（二）——...

XG1057415595：感谢作者啊！
- ThreadPoolExecuto...

weixin\_42917883：赞
- 深入理解阻塞队列（二）——Arra...

qq\_19431333：[reply]hhq420684[/reply] 是的。final之后不能修改了，ArrayLis...
- 深入理解阻塞队列（二）——Arra...

hhq420684：大哥能不能解释一下，为什么这个Arrayblockingqueue的底层存储数组是final修饰的...
- 深入理解阻塞队列（四）——Link...

qq\_19431333：[reply]xiaoqiang\_329[/reply] 那块是在解释两把锁+四个条件+Atomi...

联系我们



请扫描二维码联系客服

webmaster@csdn.net

400-660-0108

QQ客服 客服论坛


关于

招聘

广告服务

网站地图

©18 CSDN版权所有 京ICP证09002463号

 百度提供搜索支持

经营性网站备案信息

4
6

网络110报警服务  
中国互联网举报中心  
北京互联网违法和不良信息举报中心

4
6