个人资料

MONKEY_D_MENG

+ 加关注 发私信

访问： 962971次

积分： 8012

等级： BLOG 6

排名： 第2536名

原创： 98篇

转载： 23篇

译文： 0篇

评论： 278条

原 生产者/消费者问题的多种Java实现方式

标签： java 产品 任务 list signal 存储

2011-03-15 21:11 88929人阅读 评论(41) 收藏 举报

分类： Java（3）

生产者/消费者问题的多种Java实现方式

实质上，很多后台服务程序并发控制的基本原理都可以归纳为生产者/消费者模式，而这是恰恰是在本科操作系统课堂上老师反复讲解，而我们却视而不见不以为然的。在博文《一种面向作业流(工作流)的轻量级可复用的异步流水开发框架的设计与实现》中将介绍一种生产者/消费者模式的具体应用。

生产者消费者问题是研究多线程程序时绕不开的经典问题之一，它描述是有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者则可以从仓库中取走产品。解决生产者/消费者问题的方法可分为两类：（1）采用某种机制保护生产者和消费者之间的同步；（2）在生产者和消费者之间建立一个管道。第一种方式有较高的效率，并且易于实现，代码的可控制性较好，属于常用的模式。第二种管道缓冲区不易控制，被传输数据对象不易于封装等，实用性不强。因此本文只介绍同步机制实现的生产者/消费者问题。

同步问题核心在于：如何保证同一资源被多个线程并发访问时的完整性。常用的同步方法是采用信号或加锁机制，保证资源在任意时刻至多被一个线程访问。Java语言在多线程编程上实现了完全对象化，提供了对同步机制的良好支持。在Java中一共有四种方法支持同步，其中前三个是同步方法，一个是管道方法。

（1）wait() / notify()方法

（2）await() / signal()方法

（3）BlockingQueue阻塞队列方法

（4）PipedInputStream / PipedOutputStream

本文只介绍最常用的前三种，第四种暂不做讨论，有兴趣的读者可以自己去网上找答案。

一、wait() / notify()方法

wait() / nofity()方法是基类Object的两个方法，也就意味着所有Java类都会拥有这两个方法，这样，我们就可以为任何对象实现同步机制。

wait()方法：当缓冲区已满/空时，生产者/消费者线程停止自己的执行，放弃锁，使自己处于等等状态，让其他线程执行。

notify()方法：当生产者/消费者向缓冲区放入/取出一个产品时，向其他等待的线程发出可执行的通知，同时放弃锁，使自己处于等待状态。

光看文字可能不太好理解，咱来段代码就明白了：

```java
import java.util.LinkedList;

/**
 * 仓库类Storage实现缓冲区
 *
 * Email:530025983@qq.com
 *
 * @author MONKEY.D.MENG 2011-03-15
 *
 */
public class Storage
{
    // 仓库最大存储量
```

测试类Test.class加上2个线程类，保证了一个进程（Test.class）+2...

生产者/消费者问题的多种Java实
叶大帝: 写的很不错 受益匪浅

生产者/消费者问题的多种Java实
qq_35542476: lz写得通俗易懂，谢lz分享

生产者/消费者问题的多种Java实
qq_35542476: 哈哈，不是什么设计模式。就是不同的3个类而已。lz想要说的应该是策略模式。把produce,con...

```
    private final int MAX_SIZE = 100;

    // 仓库存储的载体
    private LinkedList<Object> list = new LinkedList<Object>();

    // 生产num个产品
    public void produce(int num)
    {
        // 同步代码段
        synchronized (list)
        {
            // 如果仓库剩余容量不足
            while (list.size() + num > MAX_SIZE)
            {
                System.out.println("【要生产的产品数量】:" + num + "/t【库存量】:"
                        + list.size() + "/t暂时不能执行生产任务!");
                try
                {
                    // 由于条件不满足，生产阻塞
                    list.wait();
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }

            // 生产条件满足情况下，生产num个产品
            for (int i = 1; i <= num; ++i)
            {
                list.add(new Object());
            }

            System.out.println("【已经生产产品数】:" + num + "/t【现仓储量为】:" + list.size());

            list.notifyAll();
        }
    }

    // 消费num个产品
    public void consume(int num)
    {
        // 同步代码段
        synchronized (list)
        {
            // 如果仓库存储量不足
            while (list.size() < num)
            {
                System.out.println("【要消费的产品数量】:" + num + "/t【库存量】:"
                        + list.size() + "/t暂时不能执行生产任务!");
                try
                {
                    // 由于条件不满足，消费阻塞
                    list.wait();
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }

            // 消费条件满足情况下，消费num个产品
            for (int i = 1; i <= num; ++i)
            {
                list.remove();
            }

            System.out.println("【已经消费产品数】:" + num + "/t【现仓储量为】:" + list.size());

            list.notifyAll();
        }
    }

    // get/set方法
    public LinkedList<Object> getList()
    {
        return list;
    }
```

```
92.
93.     public void setList(LinkedList<Object> list)
94.     {
95.         this.list = list;
96.     }
97.
98.     public int getMAX_SIZE()
99.     {
100.        return MAX_SIZE;
101.    }
102. }
103. /**
104.  * 生产者类Producer继承线程类Thread
105.  *
106.  * Email:530025983@qq.com
107.  *
108.  * @author MONKEY.D.MENG 2011-03-15
109.  *
110.  */
111. public class Producer extends Thread
112. {
113.     // 每次生产的产品数量
114.     private int num;
115.
116.     // 所在放置的仓库
117.     private Storage storage;
118.
119.     // 构造函数，设置仓库
120.     public Producer(Storage storage)
121.     {
122.         this.storage = storage;
123.     }
124.
125.     // 线程run函数
126.     public void run()
127.     {
128.         produce(num);
129.     }
130.
131.     // 调用仓库Storage的生产函数
132.     public void produce(int num)
133.     {
134.         storage.produce(num);
135.     }
136.
137.     // get/set方法
138.     public int getNum()
139.     {
140.         return num;
141.     }
142.
143.     public void setNum(int num)
144.     {
145.         this.num = num;
146.     }
147.
148.     public Storage getStorage()
149.     {
150.         return storage;
151.     }
152.
153.     public void setStorage(Storage storage)
154.     {
155.         this.storage = storage;
156.     }
157. }
158. /**
159.  * 消费者类Consumer继承线程类Thread
160.  *
161.  * Email:530025983@qq.com
162.  *
163.  * @author MONKEY.D.MENG 2011-03-15
164.  *
165.  */
166. public class Consumer extends Thread
167. {
168.     // 每次消费的产品数量
169.     private int num;
```

```
170.
171.     // 所在放置的仓库
172.     private Storage storage;
173.
174.     // 构造函数，设置仓库
175.     public Consumer(Storage storage)
176.     {
177.         this.storage = storage;
178.     }
179.
180.     // 线程run函数
181.     public void run()
182.     {
183.         consume(num);
184.     }
185.
186.     // 调用仓库Storage的生产函数
187.     public void consume(int num)
188.     {
189.         storage.consume(num);
190.     }
191.
192.     // get/set方法
193.     public int getNum()
194.     {
195.         return num;
196.     }
197.
198.     public void setNum(int num)
199.     {
200.         this.num = num;
201.     }
202.
203.     public Storage getStorage()
204.     {
205.         return storage;
206.     }
207.
208.     public void setStorage(Storage storage)
209.     {
210.         this.storage = storage;
211.     }
212. }
213. /**
214.  * <a href="http://lib.csdn.net/base/softwaretest" class='replace_word' title="软件测试知识
    库" target='_blank' style='color:#df3434; font-weight:bold;'>测试</a>类Test
215.  *
216.  * Email:530025983@qq.com
217.  *
218.  * @author MONKEY.D.MENG 2011-03-15
219.  *
220.  */
221. public class Test
222. {
223.     public static void main(String[] args)
224.     {
225.         // 仓库对象
226.         Storage storage = new Storage();
227.
228.         // 生产者对象
229.         Producer p1 = new Producer(storage);
230.         Producer p2 = new Producer(storage);
231.         Producer p3 = new Producer(storage);
232.         Producer p4 = new Producer(storage);
233.         Producer p5 = new Producer(storage);
234.         Producer p6 = new Producer(storage);
235.         Producer p7 = new Producer(storage);
236.
237.         // 消费者对象
238.         Consumer c1 = new Consumer(storage);
239.         Consumer c2 = new Consumer(storage);
240.         Consumer c3 = new Consumer(storage);
241.
242.         // 设置生产者产品生产数量
243.         p1.setNum(10);
244.         p2.setNum(10);
245.         p3.setNum(10);
246.         p4.setNum(10);
```

```
247.         p5.setNum(10);
248.         p6.setNum(10);
249.         p7.setNum(80);
250.
251.         // 设置消费者产品消费数量
252.         c1.setNum(50);
253.         c2.setNum(20);
254.         c3.setNum(30);
255.
256.         // 线程开始执行
257.         c1.start();
258.         c2.start();
259.         c3.start();
260.         p1.start();
261.         p2.start();
262.         p3.start();
263.         p4.start();
264.         p5.start();
265.         p6.start();
266.         p7.start();
267.     }
268. }
269. 【要消费的产品数量】:50    【库存量】:0 暂时不能执行生产任务!
270. 【要消费的产品数量】:30    【库存量】:0 暂时不能执行生产任务!
271. 【要消费的产品数量】:20    【库存量】:0 暂时不能执行生产任务!
272. 【已经生产产品数】:10    【现仓储量为】:10
273. 【要消费的产品数量】:20    【库存量】:10    暂时不能执行生产任务!
274. 【要消费的产品数量】:30    【库存量】:10    暂时不能执行生产任务!
275. 【要消费的产品数量】:50    【库存量】:10    暂时不能执行生产任务!
276. 【已经生产产品数】:10    【现仓储量为】:20
277. 【要消费的产品数量】:50    【库存量】:20    暂时不能执行生产任务!
278. 【要消费的产品数量】:30    【库存量】:20    暂时不能执行生产任务!
279. 【已经消费产品数】:20    【现仓储量为】:0
280. 【已经生产产品数】:10    【现仓储量为】:10
281. 【已经生产产品数】:10    【现仓储量为】:20
282. 【已经生产产品数】:80    【现仓储量为】:100
283. 【要生产的产品数量】:10    【库存量】:100    暂时不能执行生产任务!
284. 【已经消费产品数】:30    【现仓储量为】:70
285. 【已经消费产品数】:50    【现仓储量为】:20
286. 【已经生产产品数】:10    【现仓储量为】:30
287. 【已经生产产品数】:10    【现仓储量为】:40
```

看完上述代码，对wait() / notify()方法实现的同步有了了解。你可能会对Storage类中为什么要定义public void produce(int num);和public void consume(int num);方法感到不解，为什么不直接在生产者类Producer和消费者类Consumer中实现这两个方法，却要调用Storage类中的实现呢？淡定，后文会有解释。我们先往下走。

二、**await() / signal()**方法

在JDK5.0之后，Java提供了更加健壮的线程处理机制，包括同步、锁定、线程池等，它们可以实现更细粒度的线程控制。await()和signal()就是其中用来做同步的两种方法，它们的功能基本上和wait() / nofity()相同，完全可以取代它们，但是它们和新引入的锁定机制Lock直接挂钩，具有更大的灵活性。通过在Lock对象上调用newCondition()方法，将条件变量和一个锁对象进行绑定，进而控制并发程序访问竞争资源的安全。下面来看代码：

```java
[java]
01. import java.util.LinkedList;
02. import java.util.concurrent.locks.Condition;
03. import java.util.concurrent.locks.Lock;
04. import java.util.concurrent.locks.ReentrantLock;
05.
06. /**
07.  * 仓库类Storage实现缓冲区
08.  *
09.  * Email:530025983@qq.com
10.  *
11.  * @author MONKEY.D.MENG 2011-03-15
12.  *
13.  */
14. public class Storage
15. {
16.     // 仓库最大存储量
17.     private final int MAX_SIZE = 100;
```

```
18.
19.     // 仓库存储的载体
20.     private LinkedList<Object> list = new LinkedList<Object>();
21.
22.     // 锁
23.     private final Lock lock = new ReentrantLock();
24.
25.     // 仓库满的条件变量
26.     private final Condition full = lock.newCondition();
27.
28.     // 仓库空的条件变量
29.     private final Condition empty = lock.newCondition();
30.
31.     // 生产num个产品
32.     public void produce(int num)
33.     {
34.         // 获得锁
35.         lock.lock();
36.
37.         // 如果仓库剩余容量不足
38.         while (list.size() + num > MAX_SIZE)
39.         {
40.             System.out.println("【要生产的产品数量】:" + num + "/t【库存量】:" + list.size()
41.                     + "/t暂时不能执行生产任务!");
42.             try
43.             {
44.                 // 由于条件不满足，生产阻塞
45.                 full.await();
46.             }
47.             catch (InterruptedException e)
48.             {
49.                 e.printStackTrace();
50.             }
51.         }
52.
53.         // 生产条件满足情况下，生产num个产品
54.         for (int i = 1; i <= num; ++i)
55.         {
56.             list.add(new Object());
57.         }
58.
59.         System.out.println("【已经生产产品数】:" + num + "/t【现仓储量为】:" + list.size());
60.
61.         // 唤醒其他所有线程
62.         full.signalAll();
63.         empty.signalAll();
64.
65.         // 释放锁
66.         lock.unlock();
67.     }
68.
69.     // 消费num个产品
70.     public void consume(int num)
71.     {
72.         // 获得锁
73.         lock.lock();
74.
75.         // 如果仓库存储量不足
76.         while (list.size() < num)
77.         {
78.             System.out.println("【要消费的产品数量】:" + num + "/t【库存量】:" + list.size()
79.                     + "/t暂时不能执行生产任务!");
80.             try
81.             {
82.                 // 由于条件不满足，消费阻塞
83.                 empty.await();
84.             }
85.             catch (InterruptedException e)
86.             {
87.                 e.printStackTrace();
88.             }
89.         }
90.
91.         // 消费条件满足情况下，消费num个产品
92.         for (int i = 1; i <= num; ++i)
93.         {
94.             list.remove();
95.         }
```

```
96.
97.         System.out.println("【已经消费产品数】:" + num + "/t【现仓储量为】:" + list.size());
98.
99.         // 唤醒其他所有线程
100.        full.signalAll();
101.        empty.signalAll();
102.
103.        // 释放锁
104.        lock.unlock();
105.    }
106.
107.    // set/get方法
108.    public int getMAX_SIZE()
109.    {
110.        return MAX_SIZE;
111.    }
112.
113.    public LinkedList<Object> getList()
114.    {
115.        return list;
116.    }
117.
118.    public void setList(LinkedList<Object> list)
119.    {
120.        this.list = list;
121.    }
122. }
123. 【要消费的产品数量】:50    【库存量】:0 暂时不能执行生产任务!
124. 【要消费的产品数量】:30    【库存量】:0 暂时不能执行生产任务!
125. 【已经生产产品数】:10    【现仓储量为】:10
126. 【已经生产产品数】:10    【现仓储量为】:20
127. 【要消费的产品数量】:50    【库存量】:20    暂时不能执行生产任务!
128. 【要消费的产品数量】:30    【库存量】:20    暂时不能执行生产任务!
129. 【已经生产产品数】:10    【现仓储量为】:30
130. 【要消费的产品数量】:50    【库存量】:30    暂时不能执行生产任务!
131. 【已经消费产品数】:20    【现仓储量为】:10
132. 【已经生产产品数】:10    【现仓储量为】:20
133. 【要消费的产品数量】:30    【库存量】:20    暂时不能执行生产任务!
134. 【已经生产产品数】:80    【现仓储量为】:100
135. 【要生产的产品数量】:10    【库存量】:100    暂时不能执行生产任务!
136. 【已经消费产品数】:50    【现仓储量为】:50
137. 【已经生产产品数】:10    【现仓储量为】:60
138. 【已经消费产品数】:30    【现仓储量为】:30
139. 【已经生产产品数】:10    【现仓储量为】:40
```


只需要更新仓库类Storage的代码即可，生产者Producer、消费者Consumer、测试类Test的代码均不需要进行任何更改。这样我们就知道为神马我要在Storage类中定义public void produce(int num);和public void consume(int num);方法，并在生产者类Producer和消费者类Consumer中调用Storage类中的实现了吧。将可能发生的变化集中到一个类中，不影响原有的构架设计，同时无需修改其他业务层代码。无意之中，我们好像使用了某种设计模式，具体是啥我忘记了，啊哈哈，等我想起来再告诉大家~

三、**BlockingQueue**阻塞队列方法

BlockingQueue是JDK5.0的新增内容，它是一个已经在内部实现了同步的队列，实现方式采用的是我们第2种await() / signal()方法。它可以在生成对象时指定容量大小。它用于阻塞操作的是put()和take()方法。

put()方法：类似于我们上面的生产者线程，容量达到最大时，自动阻塞。

take()方法：类似于我们上面的消费者线程，容量为0时，自动阻塞。

关于BlockingQueue的内容网上有很多，大家可以自己搜，我在这不多介绍。下面直接看代码，跟以往一样，我们只需要更改仓库类Storage的代码即可：

```
[java]
01. import java.util.concurrent.LinkedBlockingQueue;
02.
03. /**
04.  * 仓库类Storage实现缓冲区
05.  *
06.  * Email:530025983@qq.com
07.  *
08.  * @author MONKEY.D.MENG 2011-03-15
```

```
 *
 */
public class Storage
{
    // 仓库最大存储量
    private final int MAX_SIZE = 100;

    // 仓库存储的载体
    private LinkedBlockingQueue<Object> list = new LinkedBlockingQueue<Object>(
            100);

    // 生产num个产品
    public void produce(int num)
    {
        // 如果仓库剩余容量为0
        if (list.size() == MAX_SIZE)
        {
            System.out.println("【库存量】:" + MAX_SIZE + "/t暂时不能执行生产任务!");
        }

        // 生产条件满足情况下，生产num个产品
        for (int i = 1; i <= num; ++i)
        {
            try
            {
                // 放入产品，自动阻塞
                list.put(new Object());
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }

            System.out.println("【现仓储量为】:" + list.size());
        }
    }

    // 消费num个产品
    public void consume(int num)
    {
        // 如果仓库存储量不足
        if (list.size() == 0)
        {
            System.out.println("【库存量】:0/t暂时不能执行生产任务!");
        }

        // 消费条件满足情况下，消费num个产品
        for (int i = 1; i <= num; ++i)
        {
            try
            {
                // 消费产品，自动阻塞
                list.take();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        System.out.println("【现仓储量为】:" + list.size());
    }

    // set/get方法
    public LinkedBlockingQueue<Object> getList()
    {
        return list;
    }

    public void setList(LinkedBlockingQueue<Object> list)
    {
        this.list = list;
    }

    public int getMAX_SIZE()
    {
        return MAX_SIZE;
    }
```

关闭

```
}
【库存量】:0 暂时不能执行生产任务!
【库存量】:0 暂时不能执行生产任务!
【现仓储量为】:1
【现仓储量为】:1
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:8
【现仓储量为】:9
【现仓储量为】:10
【现仓储量为】:11
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:13
【现仓储量为】:14
【现仓储量为】:17
【现仓储量为】:19
【现仓储量为】:20
【现仓储量为】:21
【现仓储量为】:22
【现仓储量为】:23
【现仓储量为】:24
【现仓储量为】:25
【现仓储量为】:26
【现仓储量为】:12
【现仓储量为】:1
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:27
【现仓储量为】:8
【现仓储量为】:6
【现仓储量为】:18
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:8
【现仓储量为】:9
【现仓储量为】:10
【现仓储量为】:16
【现仓储量为】:11
【现仓储量为】:12
【现仓储量为】:13
【现仓储量为】:14
【现仓储量为】:15
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:3
【现仓储量为】:15
【现仓储量为】:1
【现仓储量为】:0
【现仓储量为】:1
【现仓储量为】:1
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:0
【现仓储量为】:1
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:8
【现仓储量为】:9
【现仓储量为】:10
【现仓储量为】:11
【现仓储量为】:12
```

```
【现仓储量为】:13
【现仓储量为】:14
【现仓储量为】:15
【现仓储量为】:16
【现仓储量为】:17
【现仓储量为】:1
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:3
【现仓储量为】:3
【现仓储量为】:1
【现仓储量为】:2
【现仓储量为】:3
【现仓储量为】:4
【现仓储量为】:5
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:8
【现仓储量为】:9
【现仓储量为】:10
【现仓储量为】:11
【现仓储量为】:12
【现仓储量为】:13
【现仓储量为】:14
【现仓储量为】:15
【现仓储量为】:16
【现仓储量为】:17
【现仓储量为】:18
【现仓储量为】:19
【现仓储量为】:6
【现仓储量为】:7
【现仓储量为】:8
【现仓储量为】:9
【现仓储量为】:10
【现仓储量为】:11
【现仓储量为】:12
【现仓储量为】:13
【现仓储量为】:14
【现仓储量为】:15
【现仓储量为】:16
【现仓储量为】:17
【现仓储量为】:18
【现仓储量为】:19
【现仓储量为】:20
【现仓储量为】:21
【现仓储量为】:22
【现仓储量为】:23
【现仓储量为】:24
【现仓储量为】:25
【现仓储量为】:26
【现仓储量为】:27
【现仓储量为】:28
【现仓储量为】:29
【现仓储量为】:30
【现仓储量为】:31
【现仓储量为】:32
【现仓储量为】:33
【现仓储量为】:34
【现仓储量为】:35
【现仓储量为】:36
【现仓储量为】:37
【现仓储量为】:38
【现仓储量为】:39
【现仓储量为】:40
```

当然，你会发现这时对于public void produce(int num);和public void consume(int num);方法业务逻辑上的实现跟前面两个例子不太一样，没关系，这个例子只是为了说明BlockingQueue阻塞队列的使用。

有时使用BlockingQueue可能会出现put()和System.out.println()输出不匹配的情况，这是由于它们之间没有同步造成的。当缓冲区已满，生产者在put()操作时，put()内部调用了await()方法，放弃了线程的执行，然后消费者线程执行，调用take()方法，take()内部调用了signal()方法，通知生产者线程可以执行，致使在消费者的println()还没运行的情况下生产者的println()先被执行，所以有了输出不匹配的情况。

对于BlockingQueue大家可以放心使用，这可不是它的问题，只是在它和别的对象之间的同步有问题。

对于Java实现生产者/消费者问题的方法先总结到这里面吧，过几天实现一下C++版本的，接下来要马上着手于基于生产者/消费者模式的《异步工作流服务框架的设计与实现》，请持续关注本博客。

上一篇 SQL注入
下一篇 一种轻量级对象池的设计与实现

相关文章推荐

- Java Executor并发框架（十四）Executor框架线程...
- 【直播】计算机视觉原理及实战—屈教授
- Java设计模式—生产者消费者模式（阻塞队列实现...
- 【套餐】Spark+Scala课程包--陈超
- java多线程之生产者消费者经典问题
- 【套餐】Linux应用和网络编程实战套餐--朱有鹏
- 设计模式 工厂模式 从卖肉夹馍说起
- 【直播】机器学习&数据挖掘7周实训--韦玮
- 设计模式 观察者模式 以微信公众服务为例
- 【套餐】从0蜕变为自动化测试工程师--李晓鹏
- 设计模式 装饰者模式 带你重回传奇世界
- 【直播】广义线性模型及其应用——李科
- java多线程之消费者生产者模式
- 生产者/消费者问题的多种Java实现方式
- java 多线程并发系列之 生产者消费者模式的两种实..
- 一种面向作业流(工作流)的轻量级可复用的异步流...

查看评论

30楼 lzbhnr 2017-06-29 16:36发表

每个消费者，或者生产者，只能生产或者消费一次吧

29楼 WorldWelcome 2017-06-21 09:45发表

这个代码模式的测试类Test.class加上2个线程类，保证了一个进程（Test.class）+2个线程的模式，2个线程共享Test.class的 内存空间，开销小，效率好。

28楼 叶大帝 2017-06-15 20:01发表

写的很不错 受益匪浅

27楼 qq_35542476 2017-06-08 11:48发表

lz写得通俗易懂，谢lz分享

26楼 qq_35542476 2017-06-08 11:38发表

哈哈，不是什么设计模式。就是不同的3个类而已。

lz想要说的应该是策略模式。 把produce,consumer，换成不同的子类来实现。

25楼 梁山boy 2017-04-04 18:17发表

简化的桥接模式？

24楼 Dreamer-1 2017-01-24 17:32发表

使用synchronized和wait()/notify()的第一个示例中：
Storage中的list在生产和消费时都会上锁内置的list对象，那么同一时间就只能有【生产】或【消费】两个动作之中的一个发生？
对于多核处理器来说，完全可以实现一个cpu执行生产任务，一个cpu执行消费任务吧，那么这个模拟是不是有些问题呢？

还有while (list.size() + num > MAX_SIZE)条件判断的问题，假设生产者p1执行时发现此时list里容量为20，p1的任务是生产81个东西，由于20+81>100，则p1被挂起；
但是按照生产逻辑来说不应该是p1先生产80个，然后在生产第81个的时候发现容器已满再被挂起么？

ps：楼主总体讲解得很好~赞一个~~~

Re: WorldWelcome 2017-05-26 10:37发表

> 引用“zlhqwz”的评论：
>
> 使用synchronized和wait()/notify()的第一个示例中：
>
> Storage中的li...

对多核处理器来讲，程序仍不会有问题，因为storage对象在内存始终只有一份拷贝，无论有多少线程，如何被多CPU分配，只要读取的是同一内存对象，就不用担心。

Re: 怎呼虹 2017-03-26 20:38发表

回复Dreamer-1：厉害了word哥

23楼 qq_27270567 2016-10-04 16:27发表

作者第一个例子生成量大于消费量和仓库最大容量总和的时候会出现阻塞，据说可以使用PV操作，但是还不清楚怎么写PV。只想说，作者第一个栗子还是正常的，不过修改每个生产者或者消费者的数据时就需要考虑停止线程的问题了（就是发现完全不能生产，，或者完全没有消费的可能时，，要自动关闭的设计吧）

22楼 qq_27270567 2016-10-04 16:26发表

作者第一个例子生成量大于消费量和仓库最大容量总和的时候会出现阻塞，据说可以使用PV操作，但是还不清楚怎么写PV。只想说，作者第一个栗子还是正常的，不过修改每个生产者或者消费者的数据时就需要考虑停止线程的问题了（就是发现完全不能生产，，或者完全没有消费的可能时，，要自动关闭的设计吧）

21楼 qq_27855813 2016-09-07 11:52发表

实际测试，多个线程执行顺序是随机的，但仓库剩余总量是一定的，并且同步只需要对list同步，如果对生产者或消费者方法同步而不对list同步就会出错。

20楼 黑客江湖 2016-09-07 11:50发表

实际测试，多个线程执行顺序是随机的，但仓库剩余总量是一定的，并且同步只需要对list同步，如果对生产者或消费者方法同步而不对list同步就会出错。

19楼 羲皇 2016-08-18 16:12发表

在list加锁的话，应该会报错吧。应该在当前对象上加锁this.wait

18楼 qq_35587085 2016-08-09 17:23发表

看了评论一脸懵逼

17楼 armon0602 2016-04-08 09:58发表

问大家一个问题，wait/notify, condition中，为什么要用while，我用if试过也可以。wait应该会堵塞的

Re: WorldWelcome 2017-05-26 14:36发表

回复armon0602：肯定要while循环，否则执行一次后代码段结束执行，在达到满足条件后，代码段也不执行，也就失去了响应的机会。例子里设计的数量满足就乱了，引起阻塞。

Re: lzbhnr 2017-06-29 16:45发表

回复WorldWelcome：如果用if，条件为true，执行代码块，到wait()方法，线程阻塞。在拿到锁的时候，则不回去再去判断是否满足条件，直接执行。后面的代码。如果while，则会再次判断。这样应该是不用if，用while的原因吧

Re: gongyi1101 2016-06-04 13:45发表

回复armon0602：多个生产者和消费者线程的时候只能用while,只有1个的时候无所谓if还是while

16楼 chaser401 2016-04-05 19:03发表

有没有用ReentrantLock这个锁的方法？

15楼 WenJie_top 2016-03-10 15:25发表

第一种方法你的Wati() 、notify()是有问题的。 比较赞同一楼的观点。 作者的方法，设置只有一个消费者，多个生产者的时候，会出现生产者生产成功之后，唤醒其他生产者的情况，造成实际元素的数量超出设定的容量的问题。

Re: sinat_33994921 2016-05-21 10:24发表

回复WenJie_top：锁加while循环，确保不会发生这种情况。

Re: peppengliu 2016-03-30 12:39发表

回复WenJie_top：第一个是在list上加了同步，我理解，多生产者的时候，也是多个生产者抢夺这个锁，不会出现超出

容量的情况。

14楼 WenJie_top 2016-03-10 15:24发表

第一种方法你的Wati() 、notify()是有问题的。 比较赞同一楼的观点。 作者的方法，设置只有一个消费者，多个生产者的时候，会出现生产者生产成功之后，唤醒其他生产者的情况，造成实际元素的数量超出设定的容量的问题。

13楼 WenJie_top 2016-03-10 15:23发表

第一种方法你的Wati() 、notify()是有问题的。 比较赞同一楼的观点。 作者的方法，设置只有一个消费者，多个生产者的时候，会出现生产者生产成功之后，唤醒其他生产者的情况，造成实际元素的数量超出设定的容量的问题。

Re: leoge0113 2017-04-12 14:29发表

回复WenJie_top：不会有问题的

12楼 cool010 2015-09-16 00:10发表

第一种实现，Storage 类的42行和76行都有问题：
在wait方法前进行的容量检查，等当前线程再次运行时，容量可能已经变了，有可能不满足容量上下限了，需要重新进行检查。

11楼 gxfcome 2015-08-07 16:44发表

最后一个linkedblockingqueue，判断满和空应该用while循环吧if (list.size() == MAX_SIZE) 和 if (list.size() == 0)

10楼 AC是男孩 2015-06-26 12:01发表

await() / signal()方法中，不明白为什么要先signal，然后再unlock，为什么不先unlock，然后再signal呢？但是我先unlock，然后再signal运行一会后就报错了，为什么？

9楼 干净的句号 2015-03-05 14:53发表

第一部分代码，看了很多博客都会出现阻塞的情况，而唯独这个例子没有，仔细分析发现是因为生产者生产的数量肯定大于消费者要消耗的数量，所以就不会出现阻塞。
如果生产者的数量减少到3个，就会出现阻塞的情况。比如，我p4,p5,p6,p7注释掉，就会出现阻塞。

Re: WorldWelcome 2017-05-26 14:41发表

是的，这个程序，总体消费100，生产140，所以不会阻塞。当生产<消费，就有阻塞，阻塞的可能是20或30或50。

8楼 wuzetiandaren 2015-01-09 09:34发表

策略模式

Re: WorldWelcome 2017-05-26 16:25发表

这个不是什么模式，别被作者误导往高难度方向去想，代码必须写成一个类，2个方法，保证内存中只有一个拷贝。

7楼 lxydo 2014-11-22 22:39发表

第一部分的代码有错误吧

6楼 Xujian0000abcd 2014-10-27 11:06发表

大神讲的很通俗易懂呦~赞一个~
只是await()/signal中，produce(int num)函数中的full.signalAll()可以去掉，因为生产之后能保证仓库不为空，但不能保证仓库是满的。同理consume（int num）中的empty.signalAll()也可以去掉。经过验证去掉之后完全不影响结果。

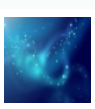
5楼 wcs504066130 2014-09-06 20:03发表

有个问题不明白，为什么用lock 的时候，唤醒必须放在unlock之前，这样的话，唤醒了其他线程，但是Unlock未执行，资源并没有释放锁，不是又等待了么？

Re: haocnc 2014-10-20 16:15发表

回复wcs504066130：unlock未执行，其他线程拿不到锁，就进不了代码块，因此不会被await()，而是处于临时阻塞状态，直到锁被释放，开始争抢执行权。

4楼 放牧的太阳 2014-07-18 17:41发表



很不错！

3楼 kongtiao5 2014-07-07 12:25发表

Good

2楼 bingaabing 2013-11-28 19:41发表

讲解的通俗易懂，对于我这样的菜鸟，刚刚合胃口，赞一个

1楼 xidianyejijie 2013-04-06 20:39发表

感觉对于多生产者和多消费者问题的话好像还少一对PV操作。

发表评论

用 户 名： qq_36596145

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场