链滴

Java 并发编程之（二）管程 （转）

作者：lanjian
原文链接：https://ld246.com/article/1478594315736
来源网站：链滴

<h1>定义</h1>
维基百科中定义
程为：在并发编程中，管程（monitor）为一个同步结构，具有线程互斥特性，以及能够根据某些条
来阻塞线程。根据定义，管程有三个要素：同步、互斥、条件。恰好在Java的Concurrent包中Reentr
ntLock具有上述所有特性，可以用来实现管程。管程是一个非常实用且常见的技术，可以用来实现很
常用的并发数据结构，例如阻塞队列。
<h1>阻塞队列</h1>
 队列常用于生产者消费者模型，生产者发送消息并存储到队列，消费者从队列中取出消
。一般情况下会存在多个生产者和多个消费者，普通的队列不能保证并发的安全，因此需要用到线程
全的技术。线程安全的队列又可以分为两种：

阻塞队列（BlockingQueue）
非阻塞队列（NoBlockingQueue）

阻塞队列的特征为，当队列为空时会阻塞消费者，当队列满时阻塞生产者。这种机制能够平衡生
者和消费者的负载。
分析上述定义，阻塞队列具有线程安全，阻塞，条件的特性；线程安全和阻塞就意味着要实现同
以及互斥，因此，管程能够很好地满足阻塞队列的要求。
<h1>Java源码中的管程</h1>
 如下代码所示是Concurrent包中ArrayBlockingQueue的实现，队列有两种实现方式，
组和链表。Array是数组的实现，而数组通常是有界的。
ArrayBlockingQueue结构将ReentrantLock（可重入锁）作为全局锁来实现线程安全，所谓全
锁就是整个数据结构中就这么一个锁，当一个线程在访问该对象并获得锁之后，其他线程要访问该对
都得阻塞。在后续的学习中，我们会发现全局锁有一定的弊端，因为他锁住了整个对象，使得整体的
发性不高。源代码中的newCondition（）操作会生成一个条件对象，条件对象具有唤醒线程和挂起
程的能力，具体的操作如第二段代码所示。
<pre class="prettyprint prettyprinted">public <span class="typ
>ArrayBlockingQueue(int
span class="pln"> capacity,boolea
 fair)
<span class="p
n">{ if(<span class="pln"
capacity <=0<span class=
pun">){
  throw new </
pan>IllegalArgumentException();
  }<
r /> this.<span class="p
n">items =new<span clas
="typ">Object[capacity<s
an class="pun">];
  lock=</
pan>newReentrantLock<span class
"pun">(fair);<span class="p
n">
  notEmpty =lock<span c
ass="pun">.newCondition();</span

  notFull =lock<span clas
="pun">.newCondition();
</s
an>}</pre>
 第二段代码是ArrayBlockingQueue的put操作，put操作的第一步就是获得全局锁，然
判断队列当前元素是否已满，如果队列慢就会使用notFull条件变量将线程挂起，否则就会调用enque
e函数进行入队操作。入队操作同时会使用notEmpty条件变量来唤醒一个被notEmpty阻塞的的线程（
ake操作会调用notEmpty来阻塞线程）。
<pre class="prettyprint prettyprinted">>public <span class="kw
">void put(<span class="p
n">E e)throws<span class

"typ">InterruptedException{
checkNotNull(e<span
lass="pun">);
finalReent
antLocklock=<span class=
kwd">this.lock<span clas
="pun">;
lock.
span class="pln">lockInterruptibly();
<span class
"kwd">try{
whi
e(count <span class="pun
>== items.<span class="pl
">length)
notFull.await<spa
class="pun">();//挂起线程，当其他线程调用notFull.signal()或者no
Full.signlAll()时，该线程唤醒。
enqueue(e<span c
ass="pun">);
}finally</s
an>{
lock<span cla
s="pun">.unlock();
<
pan class="pun">}}
priv
te voidenqueue<span clas
="pun">(E x)<span class="
un">{
//assert lock.getHoldCount() == 1;
</spa
>//assert items[putIndex] == null;
finalObject[]<span clas
="pln">items =this<span
class="pun">.items;<span c
ass="pln">
items[putIndex<span c
ass="pun">]=x<span clas
="pun">;
if(++</span
putIndex ==it
ms.length<span class="pu
">)
putIndex =0<span clas
="pun">;
count++;
notEmpty.signal<span
class="pun">();//唤醒一个被notEmpty阻塞的线程，因为此时队列
经非空了
}</pre>
<h1>实例</h1>
 下面我为了熟悉管程的使用，会亲自造轮子体会一下。
<pre class="prettyprint prettyprinted">package <span class="t
p">Monitors;
import</span
java.util</span
.concurrent.</s
an>locks.Condit
on;
import<span cla
s="pln">java.util<span cla
s="pun">.concurrent.<spa
class="pln">locks.ReentrantLock</
pan>;
public <span class=
kwd">class BlockingQueue<</sp
n>E>{<br /
private ReentrantLock<s

an class="kwd">lock;//全局锁
  private Condition notFull;//条件变量，来监控队列是否已满
  private ondition notEmpty;//条件变量，用来监控队列是否为空
  private </pan>final int cpacity;//容量，这是一个定容的队列br /> private int<span cl ss="pln"> head;//队列头部哨兵
  private int<span class "pln"> tail;//队列尾部哨兵
</ pan> private int<span class="pln > count;//当然元素个数
</sp n> private E[] /span> data;/ 保存元素的数组
/*初始化阻 队列*/
  public <spn class="typ">BlockingQueue(int< span> c){
<span class="kwd" this.lock<span cl ss="pun">=newReentrantLock</sp n>();
  this.notEmpty=lock.newCondition</s an>();
  this<span cla s="pun">.notFull=<span cl ss="kwd">lock.newCondition</spa >();
  this<span class "pun">.capacity=<span cla s="pln">c;
  this</sp n>.count=</sp n>0;
 this.head=0;
<span class= kwd"> this.tail<sp n class="pun">=0;<span cla s="pln">
  data=(E<span cla s="pun">[]) newObject<sp n class="pun">[c+<span cl ss="lit">1];
  }<< r />/*向队列中添加一个元素 */
  public void /span> put(E e< span>){
  lock.lock();
 span class="kwd"> try{
  while(this span class="pun">.count==</span this.capacity< pan>)//判断队列是否已满<span cla s="pln">
  notFull.await<spa

class="pun">();//挂起线程
ail++;
count++;
i
(tail=
this.ca
acity+1)
tail=0</
pan>;
data[tail<span clas
="pun">]=e;<span class="
om">//插入元素
notEmpty.signal<sp
n class="pun">();//通知其他线程，队列不为空
<sp
n class="pun">}catch(</
pan>InterruptedExceptione1<span
lass="pun">){
/ TODO Auto-generated catch block
e1.printStackTrace<
pan class="pun">();
}<span class="kwd
>finally{
lock</s
an>.unlock();<b
/>}}
<span c
ass="com">/* 获取队列容量 */
public int<span class
"pln">getCap(){return<sp
n class="kwd">this.capacity</span
;}
/* </s
an>判读队列是否为空 */
<sp
n class="kwd">public booleanisE
mpty(){lock<span class="
un">.lock();<span class="
wd">try{return<span clas
="pln">count==0<span cla
s="pun">?true:<span clas
="kwd">false;}<span clas
="kwd">finally{lock<span
class="pun">.unlock();<sp
n class="pun">}}
/*
span class="com">判断队列是否已满*/
<span clas
="kwd">public booleanisFull</s
an>(){lock.</s
an>lock();try<
span>{returnc
unt==capacity<span class
"pun">?true:<span class=
kwd">false;}<span class="
wd">finally{lock<span cla
s="pun">.unlock();<span cl
ss="pun">}}
/* <sp
n class="com">从队列中取出一个元素 */
<span cl
ss="kwd">publicE take() </span
throws InterruptedException<span
lass="pun">{
lock<span class="p
n">.lock();
<span cla

s="kwd"> try{while
/span>(this.</
pan>count==0</
pan>)//判断队列是否为空<span cla
s="pln">
notEmpty.await<
pan class="pun">();//阻塞队列
head++;
count--;
if(head<span class="
un">==this.<span class=
pln">capacity+1<span class
"pun">)head=<span class
"lit">0;
E x=data<span
class="pun">[head];<span
lass="com">//获得头部元素
notFull.signal<
pan class="pun">();//通知其他线程，队列未满<span clas
="kwd">return x;
<s
an class="pun"> }finally<span class="pu
">{
  lock.</
pan>unlock();
<span class="
un"> }}
}</pre>
 在上锁所有操作中，几乎每个操作都在如下的结构体中进行，为了防止代码异常退出而
致锁没有被释放，必须使用finally关键字确保锁的释放。该代码块的作用和synchronized同步块作用
似，在进入同步块之前都要先获得锁，然后进行同步操作；在退出同步块的时候都要释放锁并再次同
。
<pre class="prettyprint prettyprinted">lock
lock();
<span class=
kwd">try{
  return
span class="pln"> count==capacit
?true:
/span>false;
<span class="p
n">}finally{
<span cl
ss="kwd"> lock.unlock<s
an class="pun">();
}</pre>

