**阻塞队列实现--生产消费模型**

2016年10月06日 11:30:59

阅读数：452

生产消费模型是多线程安全的典型例子，当初实现的时候使用的是一个数组或者队列，加上synchronized关键字，代码量比较多，后来学习了阻塞队列，发现使用阻塞队列来实现代码简洁不少，这样在面试的时候，面试官需要你实现一个生产消费模型也会好写一些，下面我给出原始和阻塞队列实现的代码供参考：

1.原始实现

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)

1. **package** Concurrent;
3. **import** java.util.PriorityQueue;
4. **import** java.util.Queue;
6. **public** **class** TestObjProCon {
7. //定义队列的长度
8. **private** **int** queueSize=10;
10. //定义一个队列,这里使用的是优先队列PriorityQueue
11. **private** Queue<Integer> queue=**new** PriorityQueue<Integer>(queueSize);
13. **public** **static** **void** main(String[] args){
14. TestObjProCon test=**new** TestObjProCon();
15. Consumer c=test.**new** Consumer();
16. Productor p=test.**new** Productor();
18. c.start();
19. p.start();
20. }
22. //消费者类
23. **class** Consumer **extends** Thread{
24. @Override
25. **public** **void** run(){
26. comsume();
27. }
29. **private** **void** comsume() {
30. //使用一个while循环进行监听
31. **while**(**true**){
32. //同步队列
33. **synchronized** (queue) {
34. //判断队列内部元素个数,是否为0，为0等待，否则唤醒其他队列
35. **while**(queue.size()==0){
36. **try**{
37. System.out.println("队列为空，不能取，请等待");
38. //调用wait方法，使取的队列进入阻塞
39. queue.wait();
41. //调用wait方法后，线程中断抛出InterruptedException执行catch语句
42. }**catch**(InterruptedException e){
43. e.printStackTrace();
44. //并唤醒生产队列，即切换队列
45. queue.notify();
46. }
47. }
48. //若不为空，则可以进行consume
49. //poll()每次都是移走队首元素
50. queue.poll();
51. //并唤醒生成线程，因为poll一次代表取了一次，即可以继续生产
52. queue.notify();
53. System.out.println("队列剩余的元素个数为："+queue.size());
54. }
55. }
56. }
57. }
59. //生产者类
60. **class** Productor **extends** Thread{
61. **public** **void** run(){
62. produce();
63. }
64. }
66. **public** **void** produce() {
67. **while**(**true**){
68. //同步队列
69. **synchronized** (queue) {
70. **while**(queue.size()==queueSize){
71. **try**{
72. System.out.println("队列已经满了，不能继续生产了");
73. queue.wait();//使生产队列进入阻塞
74. }**catch**(InterruptedException e){
75. e.printStackTrace();
76. queue.notify();//唤醒消费队列
77. }
79. }
80. //否则可以继续生产
81. queue.offer(1);//调用offer方法，每次插入一个元素
82. queue.notify();//插入一个元素后即可以唤醒消费线程，执行消费
83. System.out.println("此时队列里面的元素为："+queue.size());
84. }
85. }
86. }
87. }

2.阻塞队列实现：

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)

1. **package** Concurrent;

4. **import** java.util.concurrent.PriorityBlockingQueue;
6. **public** **class** TestPriorityBlockingQueue {
7. **private** **int** queueSize=10;
8. **private** PriorityBlockingQueue<Integer> queue=**new** PriorityBlockingQueue<Integer>(queueSize);
10. **public** **static** **void** main(String[] agrs){
11. TestPriorityBlockingQueue test=**new** TestPriorityBlockingQueue();
12. Consumer c=test.**new** Consumer();
13. Productor p=test.**new** Productor();
15. c.start();
16. p.start();
17. }
19. **class** Consumer **extends** Thread{
20. @Override
21. **public** **void** run(){
22. //重写run方法，在run方法中写自己的线程代码和相应方法
23. comsume();
24. }
26. **private** **void** comsume() {
27. //仍需要监听，而简化的是不需要使用synchronized同步锁了
28. **while**(**true**){
29. **try**{
30. //内部take已经实现了lock机制
31. queue.take();
32. System.out.println("队列元素剩余");
33. }**catch**(InterruptedException e){
34. e.printStackTrace();
35. }
36. }
37. }
38. }
40. **class** Productor **extends** Thread{
41. @Override
42. **public** **void** run(){
43. produce();
44. }
46. **private** **void** produce() {
47. **while**(**true**){
48. **try** {
49. //内部已经实现Lock机制，不要自己使用Synchronized和wait，notify
50. queue.put(1);
51. } **catch** (NullPointerException e) {
52. e.printStackTrace();
53. }
54. }
55. }
56. }
57. }

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)



这里我简单说一下：PriorityBlockingQueue优先阻塞队列，take方法和put方法内部已经实现锁机制，采用的是ReentrantLock，内部已经实现Lock机制，不要自己使用Synchronized和wait，notify进行线程间通信，下面是相应的源码：

take：

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)

1. **public** E take() **throws** InterruptedException {
2. **final** ReentrantLock lock = **this**.lock;
3. lock.lockInterruptibly();
4. E result;
5. **try** {
6. **while** ( (result = dequeue()) == **null**)
7. notEmpty.await();
8. } **finally** {
9. lock.unlock();
10. }
11. **return** result;
12. }

put：

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)

1. **public** **void** put(E e) {
2. offer(e); // never need to block
3. }

**[java]** [view plain](https://blog.csdn.net/jy_he/article/details/52742825) [copy](https://blog.csdn.net/jy_he/article/details/52742825)

1. **public** **boolean** offer(E e) {
2. **if** (e == **null**)
3. **throw** **new** NullPointerException();
4. **final** ReentrantLock lock = **this**.lock;
5. lock.lock();
6. **int** n, cap;
7. Object[] array;
8. **while** ((n = size) >= (cap = (array = queue).length))
9. tryGrow(array, cap);
10. **try** {
11. Comparator<? **super** E> cmp = comparator;
12. **if** (cmp == **null**)
13. siftUpComparable(n, e, array);
14. **else**
15. siftUpUsingComparator(n, e, array, cmp);
16. size = n + 1;
17. notEmpty.signal();
18. } **finally** {
19. lock.unlock();
20. }
21. **return** **true**;
22. }