队列之基础篇

转自(http://www.cnblogs.com/end/archive/2012/10/25/2738493.html)

Queue接口与List、Set同一级别，都是继承了Collection接口。LinkedList实现了Queue接 口。Queue接口窄化了对LinkedList的方法的访问权限（即在方法中的参数类型如果是Queue时，就完全只能访问Queue接口所定义的方法 了，而不能直接访问 LinkedList的非Queue的方法），以使得只有恰当的方法才可以使用。BlockingQueue 继承了Queue接口。

队列是一种数据结构．它有两个基本操作：在队列尾部加人一个元素，和从队列头部移除一个元素就是说，队列以一种先进先出的方式管理数据，如果你试图向一个 已经满了的阻塞队列中添加一个元素或者是从一个空的阻塞队列中移除一个元索，将导致线程阻塞．在多线程进行合作时，阻塞队列是很有用的工具。工作者线程可 以定期地把中间结果存到阻塞队列中而其他工作者线线程把中间结果取出并在将来修改它们。队列会自动平衡负载。如果第一个线程集运行得比第二个慢，则第二个 线程集在等待结果时就会阻塞。如果第一个线程集运行得快，那么它将等待第二个线程集赶上来。下表显示了jdk1.5中的阻塞队列的操作：

add	增加一个元索	如果队列已满，则抛出一个IIIegaISlabEepeplian异常
remove	移除并返回队列头部的元素	如果队列为空，则抛出一个NoSuchElementException异常
element	返回队列头部的元素	如果队列为空，则抛出一个NoSuchElementException异常
offer	添加一个元素并返回true	如果队列已满，则返回false
poll	移除并返问队列头部的元素	如果队列为空，则返回null
peek	返回队列头部的元素	如果队列为空，则返回null
put	添加一个元素	如果队列满，则阻塞
take	移除并返回队列头部的元素	如果队列为空，则阻塞

remove、element、offer 、poll、peek 其实是属于Queue接口。

阻塞队列的操作可以根据它们的响应方式分为以下三类：aad、removee和element操作在你试图为一个已满的队列增加元素或从空队列取得元素时 抛出异常。当然，在多线程程序中，队列在任何时间都可能变成满的或空的，所以你可能想使用offer、poll、peek方法。这些方法在无法完成任务时 只是给出一个出错示而不会抛出异常。

注意：poll和peek方法出错进返回null。因此，向队列中插入null值是不合法的。

还有带超时的offer和poll方法变种，例如，下面的调用：

```
boolean success = q.offer(x,100,TimeUnit.MILLISECONDS);
```

尝试在100毫秒内向队列尾部插入一个元素。如果成功，立即返回true；否则，当到达超时进，返回false。同样地，调用：

```
Object head = q.poll(100, TimeUnit.MILLISECONDS);
```

如果在100毫秒内成功地移除了队列头元素，则立即返回头元素；否则在到达超时时，返回null。

最后，我们有阻塞操作put和take。put方法在队列满时阻塞，take方法在队列空时阻塞。

java.ulil.concurrent包提供了阻塞队列的4个变种。默认情况下，LinkedBlockingQueue的容量是没有上限的（说的不准确，在不指定时容量为Integer.MAX_VALUE，不要然的话在put时怎么会受阻呢），但是也可以选择指定其最大容量，它是基于链表的队列，此队列按 FIFO（先进先出）排序元素。

ArrayBlockingQueue在构造时需要指定容量， 并可以选择是否需要公平性，如果公平参数被设置true，等待时间最长的线程会优先得到处理（其实就是通过将ReentrantLock设置为true来 达到这种公平性的：即等待时间最长的线程会先操作）。通常，公平性会使你在性能上付出代价，只有在的确非常需要的时候再使用它。它是基于数组的阻塞循环队列，此队列按 FIFO（先进先出）原则对元素进行排序。

PriorityBlockingQueue是一个带优先级的 队列，而不是先进先出队列。元素按优先级顺序被移除，该队列也没有上限（看了一下源码，PriorityBlockingQueue是对 PriorityQueue的再次包装，是基于堆数据结构的，而PriorityQueue是没有容量限制的，与ArrayList一样，所以在优先阻塞 队列上put时是不会受阻的。虽然此队列逻辑上是无界的，但是由于资源被耗尽，所以试图执行添加操作可能会导致 OutOfMemoryError），但是如果队列为空，那么取元素的操作take

就会阻塞，所以它的检索操作take是受阻的。另外，往入该队列中的元 素要具有比较能力。

最后，DelayQueue（基于PriorityQueue来实现的）是一个存放Delayed 元素的无界阻塞队列，只有在延迟期满时才能从中提取元素。该队列的头部是延迟期满后保存时间最长的 Delayed 元素。如果延迟都还没有期满，则队列没有头部，并且poll将返回null。当一个元素的 getDelay(TimeUnit.NANOSECONDS) 方法返回一个小于或等于零的值时，则出现期满，poll就以移除这个元素了。此队列不允许使用 null 元素。 下面是延迟接口：
Java代码

```
1. public interface Delayed extends Comparable<Delayed> {
2.     long getDelay(TimeUnit unit);
3. }
```

放入DelayQueue的元素还将要实现compareTo方法，DelayQueue使用这个来为元素排序。

下面的实例展示了如何使用阻塞队列来控制线程集。程序在一个目录及它的所有子目录下搜索所有文件，打印出包含指定关键字的文件列表。从下面实例可以看出，使用阻塞队列两个显著的好处就是：多线程操作共同的队列时不需要额外的同步，另外就是队列会自动平衡负载，即那边（生产与消费两边）处理快了就会被阻塞掉，从而减少两边的处理速度差距。下面是具体实现：
Java代码

```
1. public class BlockingQueueTest {
2.     public static void main(String[] args) {
3.         Scanner in = new Scanner(System.in);
4.         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
5.         String directory = in.nextLine();
6.         System.out.print("Enter keyword (e.g. volatile): ");
7.         String keyword = in.nextLine();
8.
9.         final int FILE_QUEUE_SIZE = 10;// 阻塞队列大小
10.         final int SEARCH_THREADS = 100;// 关键字搜索线程个数
11.
12.         // 基于ArrayBlockingQueue的阻塞队列
13.         BlockingQueue<File> queue = new ArrayBlockingQueue<File>(
14.                 FILE_QUEUE_SIZE);
15.
16.         //只启动一个线程来搜索目录
17.         FileEnumerationTask enumerator = new FileEnumerationTask(queue,
18.                 new File(directory));
19.         new Thread(enumerator).start();
20.
21.         //启动100个线程用来在文件中搜索指定的关键字
22.         for (int i = 1; i <= SEARCH_THREADS; i++)
23.             new Thread(new SearchTask(queue, keyword)).start();
24.     }
25. }
26. class FileEnumerationTask implements Runnable {
27.     //哑元文件对象，放在阻塞队列最后，用来标示文件已被遍历完
28.     public static File DUMMY = new File("");
29.
30.     private BlockingQueue<File> queue;
31.     private File startingDirectory;
32.
33.     public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory) {
34.         this.queue = queue;
35.         this.startingDirectory = startingDirectory;
```

```
36.     }
37.
38.     public void run() {
39.         try {
40.             enumerate(startingDirectory);
41.             queue.put(DUMMY);//执行到这里说明指定的目录下文件已被遍历完
42.         } catch (InterruptedException e) {
43.         }
44.     }
45.
46.     // 将指定目录下的所有文件以File对象的形式放入阻塞队列中
47.     public void enumerate(File directory) throws InterruptedException {
48.         File[] files = directory.listFiles();
49.         for (File file : files) {
50.             if (file.isDirectory())
51.                 enumerate(file);
52.             else
53.                 //将元素放入队尾，如果队列满，则阻塞
54.                 queue.put(file);
55.         }
56.     }
57. }
58. class SearchTask implements Runnable {
59.     private BlockingQueue<File> queue;
60.     private String keyword;
61.
62.     public SearchTask(BlockingQueue<File> queue, String keyword) {
63.         this.queue = queue;
64.         this.keyword = keyword;
65.     }
66.
67.     public void run() {
68.         try {
69.             boolean done = false;
70.             while (!done) {
71.                 //取出队首元素，如果队列为空，则阻塞
72.                 File file = queue.take();
73.                 if (file == FileEnumerationTask.DUMMY) {
74.                     //取出来后重新放入，好让其他线程读到它时也很快的结束
75.                     queue.put(file);
76.                     done = true;
77.                 } else
78.                     search(file);
79.             }
80.         } catch (IOException e) {
81.             e.printStackTrace();
82.         } catch (InterruptedException e) {
83.         }
84.     }
85.     public void search(File file) throws IOException {
86.         Scanner in = new Scanner(new FileInputStream(file));
87.         int lineNumber = 0;
88.         while (in.hasNextLine()) {
89.             lineNumber++;
```

```
90.             String line = in.nextLine();
91.             if (line.contains(keyword))
92.                 System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber,
93.                         line);
94.         }
95.         in.close();
96.     }
97. }
```