队列之总结篇

Java多线程总结之聊一聊Queue(转自:http://hellosure.iteye.com/blog/1126541)

博客分类:

- Java技术

BlockingQueueConcurrentLinkedQueue非阻塞算法ArrayBlockingQueueLinkedBlockingQueue

上个星期总结了一下synchronized相关的知识，这次将Queue相关的知识总结一下，和朋友们分享。

在Java多线程应用中，队列的使用率很高，多数生产消费模型的首选数据结构就是队列。Java提供的线程安全的Queue可以分为**阻塞队列和非阻塞队列**，其中阻塞队列的典型例子是BlockingQueue，非阻塞队列的典型例子是ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。

注：什么叫**线程安全**？这个首先要明确。线程安全的类 ，指的是类内共享的全局变量的访问必须保证是不受多线程形式影响的。如果由于多线程的访问（比如修改、遍历、查看）而使这些变量结构被破坏或者针对这些变量操作的原子性被破坏，则这个类就不是线程安全的。

今天就聊聊这两种Queue，本文分为以下两个部分，用分割线分开：

- BlockingQueue
- ConcurrentLinkedQueue，非阻塞算法

首先来看看**BlockingQueue**：

Queue是什么就不需要多说了吧，一句话：队列是先进先出。相对的，栈是后进先出。如果不熟悉的话先找本基础的数据结构的书看看吧。

BlockingQueue，顾名思义，“阻塞队列”：可以提供阻塞功能的队列。

首先，看看BlockingQueue提供的常用方法：

	可能报异常	返回布尔值	可能阻塞	设定等待时间
入队	add(e)	offer(e)	put(e)	offer(e, timeout, unit)
出队	remove()	poll()	take()	poll(timeout, unit)
查看	element()	peek()	无	无

从上表可以很明显看出每个方法的作用，这个不用多说。我想说的是：

- add(e) remove() element() 方法不会阻塞线程。当不满足约束条件时，会抛出IllegalStateException 异常。例如：当队列被元素填满后，再调用add(e)，则会抛出异常。
- offer(e) poll() peek() 方法即不会阻塞线程，也不会抛出异常。例如：当队列被元素填满后，再调用offer(e)，则不会插入元素，函数返回false。
- 要想要实现阻塞功能，需要调用put(e) take() 方法。当不满足约束条件时，会阻塞线程。

好，上点源码你就更明白了。以**ArrayBlockingQueue**类为例：

对于第一类方法，很明显如果操作不成功就抛异常。而且可以看到其实调用的是第二类的方法，为什么？因为第二类方法返回boolean啊。

Java代码

收藏代码

```
1. public boolean add(E e) {
2.     if (offer(e))
3.         return true;
4.     else
5.         throw new IllegalStateException("Queue full");//队列已满，抛异常
6. }
7.
8. public E remove() {
9.     E x = poll();
10.     if (x != null)
11.         return x;
12.     else
13.         throw new NoSuchElementException();//队列为空，抛异常
14. }
```

对于第二类方法，很标准的ReentrantLock使用方式（不熟悉的朋友看一下我上一篇帖子吧http://hellosure.iteye.com/blog/1121157），另外对于insert和extract的实现没啥好说的。

注：先不看阻塞与否，这ReentrantLock的使用方式就能说明这个类是线程安全类。

Java代码

收藏代码

```
1. public boolean offer(E e) {
2.         if (e == null) throw new NullPointerException();
3.         final ReentrantLock lock = this.lock;
4.         lock.lock();
5.         try {
6.             if (count == items.length)//队列已满，返回false
7.                 return false;
8.             else {
9.                 insert(e);//insert方法中发出了notEmpty.signal();
10.                 return true;
11.             }
12.         } finally {
13.             lock.unlock();
14.         }
15.     }
16.
17. public E poll() {
18.         final ReentrantLock lock = this.lock;
19.         lock.lock();
20.         try {
21.             if (count == 0)//队列为空，返回false
22.                 return null;
```

```
23.             E x = extract();//extract方法中发出了notFull.signal();
24.             return x;
25.         } finally {
26.             lock.unlock();
27.         }
28.     }
```

对于第三类方法，这里面涉及到Condition类，简要提一下，

await方法指：造成当前线程在接到信号或被中断之前一直处于等待状态。

signal方法指：唤醒一个等待线程。

Java代码

收藏代码

```
1. public void put(E e) throws InterruptedException {
2.         if (e == null) throw new NullPointerException();
3.         final E[] items = this.items;
4.         final ReentrantLock lock = this.lock;
5.         lock.lockInterruptibly();
6.         try {
7.             try {
8.                 while (count == items.length)//如果队列已满，等待notFull这个条件，这时当前线程被阻塞
9.                     notFull.await();
10.             } catch (InterruptedException ie) {
11.                 notFull.signal(); //唤醒受notFull阻塞的当前线程
12.                 throw ie;
13.             }
14.             insert(e);
15.         } finally {
16.             lock.unlock();
17.         }
18.     }
19.
20. public E take() throws InterruptedException {
21.         final ReentrantLock lock = this.lock;
22.         lock.lockInterruptibly();
23.         try {
24.             try {
25.                 while (count == 0)//如果队列为空，等待notEmpty这个条件，这时当前线程被阻塞
26.                     notEmpty.await();
27.             } catch (InterruptedException ie) {
28.                 notEmpty.signal();//唤醒受notEmpty阻塞的当前线程
29.                 throw ie;
30.             }
31.             E x = extract();
32.             return x;
33.         } finally {
34.             lock.unlock();
35.         }
```

```
36.    }
```

第四类方法就是指在有必要时等待指定时间，就不详细说了。

再来看看BlockingQueue接口的具体实现类吧：

- ArrayBlockingQueue，其构造函数必须带一个int参数来指明其大小
- LinkedBlockingQueue，若其构造函数带一个规定大小的参数，生成的BlockingQueue有大小限制，若不带大小参数，所生成的BlockingQueue的大小由Integer.MAX_VALUE来决定
- PriorityBlockingQueue，其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序

上面是用ArrayBlockingQueue举得例子，下面看看**LinkedBlockingQueue**：
首先，既然是链表，就应该有Node节点，它是一个内部静态类：
Java代码

收藏代码

```
1. static class Node<E> {
2.         /** The item, volatile to ensure barrier separating write and read */
3.         volatile E item;
4.         Node<E> next;
5.         Node(E x) { item = x; }
6.     }
```

然后，对于链表来说，肯定需要两个变量来标示头和尾：
Java代码

收藏代码

```
1. /** 头指针 */
2. private transient Node<E> head;  //head.next是队列的头元素
3. /** 尾指针 */
4. private transient Node<E> last;  //last.next是null
```

那么，对于入队和出队就很自然能理解了：
Java代码

收藏代码

```
1. private void enqueue(E x) {
2.     last = last.next = new Node<E>(x);  //入队是为last再找个下家
3. }
4.
5. private E dequeue() {
6.     Node<E> first = head.next;  //出队是把head.next取出来，然后将head向后移一位
7.     head = first;
8.     E x = first.item;
9.     first.item = null;
```

```
10.     return x;
11. }
```

另外，LinkedBlockingQueue相对于ArrayBlockingQueue还有不同是，有两个ReentrantLock，且队列现有元素的大小由一个AtomicInteger对象标示。

注：AtomicInteger类是以原子的方式操作整型变量。

Java代码

收藏代码

```
1. private final AtomicInteger count = new AtomicInteger(0);
2. /** 用于读取的独占锁*/
3. private final ReentrantLock takeLock = new ReentrantLock();
4. /** 队列是否为空的条件 */
5. private final Condition notEmpty = takeLock.newCondition();
6. /** 用于写入的独占锁 */
7. private final ReentrantLock putLock = new ReentrantLock();
8. /** 队列是否已满的条件 */
9. private final Condition notFull = putLock.newCondition();
```

有两个Condition很好理解，在ArrayBlockingQueue也是这样做的。但是为什么需要两个ReentrantLock呢？下面会慢慢道来。

让我们来看看offer和poll方法的代码：

Java代码

收藏代码

```
1.  public boolean offer(E e) {
2.      if (e == null) throw new NullPointerException();
3.      final AtomicInteger count = this.count;
4.      if (count.get() == capacity)
5.          return false;
6.      int c = -1;
7.      final ReentrantLock putLock = this.putLock;//入队当然用putLock
8.      putLock.lock();
9.      try {
10.         if (count.get() < capacity) {
11.             enqueue(e); //入队
12.             c = count.getAndIncrement(); //队长度+1
13.             if (c + 1 < capacity)
14.                 notFull.signal(); //队列没满，当然可以解锁了
15.         }
16.     } finally {
17.         putLock.unlock();
18.     }
19.     if (c == 0)
20.         signalNotEmpty();//这个方法里发出了notEmpty.signal();
21.     return c >= 0;
22. }
```

```
23.
24. public E poll() {
25.     final AtomicInteger count = this.count;
26.     if (count.get() == 0)
27.         return null;
28.     E x = null;
29.     int c = -1;
30.     final ReentrantLock takeLock = this.takeLock;出队当然用takeLock
31.     takeLock.lock();
32.     try {
33.         if (count.get() > 0) {
34.             x = dequeue();//出队
35.             c = count.getAndDecrement();//队长度-1
36.             if (c > 1)
37.                 notEmpty.signal();//队列没空，解锁
38.         }
39.     } finally {
40.         takeLock.unlock();
41.     }
42.     if (c == capacity)
43.         signalNotFull();//这个方法里发出了notFull.signal();
44.     return x;
45. }
```

看看源代码发现和上面ArrayBlockingQueue的很类似，关键的问题在于：**为什么要用两个ReentrantLock**putLock和takeLock？

我们仔细想一下，入队操作其实操作的只有队尾引用last，并且没有牵涉到head。而出队操作其实只针对head，和last没有关系。那么就是说入队和出队的操作完全不需要公用一把锁，所以就设计了两个锁，这样就实现了多个不同任务的线程入队的同时可以进行出队的操作，另一方面由于两个操作所共同使用的count是AtomicInteger类型的，所以完全不用考虑计数器递增递减的问题。

另外，还有一点需要说明一下：await()和singal()这两个方法执行时都会检查当前线程是否是独占锁的当前线程，如果不是则抛出java.lang.IllegalMonitorStateException异常。所以可以看到在源码中这两个方法都出现在Lock的保护块中。

----------我是分割线----------

下面再来说说**ConcurrentLinkedQueue**，它是一个无锁的并发线程安全的队列。

以下部分的内容参照了这个帖子http://yanxuxin.iteye.com/blog/586943

对比锁机制的实现，使用无锁机制的难点在于要充分考虑线程间的协调。简单的说就是多个线程对内部数据结构进行访问时，如果其中一个线程执行的中途因为一些原因出现故障，其他的线程能够检测并帮助完成剩下的操作。这就需要把对数据结构的操作过程精细的划分成多个状态或阶段，考虑每个阶段或状态多线程访问会出现的情况。

ConcurrentLinkedQueue有两个volatile的线程共享变量：head，tail。要保证这个队列的线程安全就是保证对这两个Node的引用的访问（更新，查看）的原子性和可见性，由于volatile本身能够保证可见性，所以就是对其修改的原子性要被保证。

下面通过offer方法的实现来看看在无锁情况下如何保证原子性：

Java代码

```
1. public boolean offer(E e) {
2.     if (e == null) throw new NullPointerException();
3.     Node<E> n = new Node<E>(e, null);
4.     for (;;) {
5.         Node<E> t = tail;
6.         Node<E> s = t.getNext();
7.         if (t == tail) { //--------------------------------a
8.             if (s == null) { //---------------------------b
9.                 if (t.casNext(s, n)) { //--------------------c
10.                    casTail(t, n); //-------------------------d
11.                    return true;
12.                }
13.            } else {
14.                casTail(t, s); //----------------------------e
15.            }
16.        }
17.    }
18. }
```

此方法的循环内首先获得尾指针和其next指向的对象，由于tail和Node的next均是volatile的，所以保证了获得的分别都是最新的值。

代码a：t==tail是最上层的协调，如果其他线程改变了tail的引用，则说明现在获得不是最新的尾指针需要重新循环获得最新的值。

代码b：s==null的判断。静止状态下tail的next一定是指向null的，但是多线程下的另一个状态就是中间态：tail的指向没有改变，但是其next已经指向新的结点，即完成tail引用改变前的状态，这时候s!=null。这里就是协调的典型应用，直接进入*代码e*去协调参与中间态的线程去完成最后的更新，然后重新循环获得新的tail开始自己的新一次的入队尝试。另外值得注意的是a,b之间，其他的线程可能会改变tail的指向，使得协调的操作失败。从这个步骤可以看到无锁实现的复杂性。

代码c：t.casNext(s, n)是入队的第一步，因为入队需要两步：更新Node的next，改变tail的指向。代码c之前可能发生tail引用指向的改变或者进入更新的中间态，这两种情况均会使得t指向的元素的next属性被原子的改变，不再指向null。这时代码c操作失败，重新进入循环。

代码d：这是完成更新的最后一步了，就是更新tail的指向，最有意思的协调在这儿又有了体现。从代码看casTail(t, n)不管是否成功都会接着返回true标志着更新的成功。首先如果成功则表明本线程完成了两步的更新，返回true是理所当然的；如果 casTail(t, n)不成功呢？要清楚的是完成代码c则代表着更新进入了中间态，代码d不成功则是tail的指向被其他线程改变。意味着对于其他的线程而言：它们得到的是中间态的更新，s!=null，进入*代码e*帮助本线程执行最后一步并且先于本线程成功。这样本线程虽然代码d失败了，但是是由于别的线程的协助先完成了，所以返回true也就理所当然了。

通过分析这个入队的操作，可以清晰的看到无锁实现的每个步骤和状态下多线程之间的协调和工作。

注：上面这大段文字看起来很累，先能看懂多少看懂多少，现在看不懂先不急，下面还会提到这个算法，并且用示意图说明，就易懂很多了。

在使用ConcurrentLinkedQueue时要注意，如果直接使用它提供的函数，比如add或者poll方法，这样我们自己不需要做任何同步。

但如果是非原子操作，比如：

Java代码

```
1. if(!queue.isEmpty()) {
2.    queue.poll(obj);
3. }
```

我们很难保证，在调用了isEmpty()之后，poll()之前，这个queue没有被其他线程修改。所以对于这种情况，我们还是需要自己同步：

Java代码

收藏代码

```
1. synchronized(queue) {
2.    if(!queue.isEmpty()) {
3.       queue.poll(obj);
4.    }
5. }
```

注：这种需要进行自己同步的情况要视情况而定，不是任何情况下都需要这样做。

另外还说一下，ConcurrentLinkedQueue的size()是要遍历一遍集合的，所以尽量要避免用size而改用isEmpty()，以免性能过慢。

好，最后想说点什么呢，阻塞算法其实很好理解，简单点理解就是加锁，比如在BlockingQueue中看到的那样，再往前推点，那就是synchronized。相比而言，非阻塞算法的设计和实现都很困难，要通过低级的原子性来支持并发。下面就简要的介绍一下**非阻塞算法**，以下部分的内容参照了一篇很经典的文章http://www.ibm.com/developerworks/cn/java/j-jtp04186/

注：我觉得可以这样理解，**阻塞对应同步，非阻塞对应并发。也可以说：同步是阻塞模式，异步是非阻塞模式**

举个例子来说明什么是非阻塞算法：非阻塞的计数器

首先，使用同步的线程安全的计数器代码如下

Java代码

收藏代码

```
1. public final class Counter {
2.    private long value = 0;
3.    public synchronized long getValue() {
4.       return value;
5.    }
6.    public synchronized long increment() {
7.       return ++value;
8.    }
9. }
```

下面的代码显示了一种最简单的非阻塞算法：使用 AtomicInteger的compareAndSet()（**CAS方法**）的计数器。compareAndSet()方法规定“将这个变量更新为新值，但是如果从我上次看到这个变量之后其他线程修改了它的值，那么更新就失败”

Java代码

收藏代码

```
1. public class NonblockingCounter {
2.    private AtomicInteger value;//前面提到过，AtomicInteger类是以原子的方式操作整型变量。
3.    public int getValue() {
4.       return value.get();
5.    }
6.    public int increment() {
7.       int v;
8.       do {
9.          v = value.get();
10.       while (!value.compareAndSet(v, v + 1));
11.       return v + 1;
12.    }
13. }
```

非阻塞版本相对于基于锁的版本有几个性能优势。首先，它用硬件的原生形态代替 JVM 的锁定代码路径，从而在更细的粒度层次上（独立的内存位置）进行同步，失败的线程也可以立即重试，而不会被挂起后重新调度。更细的粒度降低了争用的机会，不用重新调度就能重试的能力也降低了争用的成本。即使有少量失败的 CAS 操作，这种方法仍然会比由于锁争用造成的重新调度快得多。

NonblockingCounter 这个示例可能简单了些，但是它演示了所有非阻塞算法的一个基本特征——有些算法步骤的执行是要冒险的，因为知道如果 CAS 不成功可能不得不重做。非阻塞算法通常叫作乐观算法，因为它们继续操作的假设是不会有干扰。如果发现干扰，就会回退并重试。在计数器的示例中，冒险的步骤是递增——它检索旧值并在旧值上加一，希望在计算更新期间值不会变化。如果它的希望落空，就会再次检索值，并重做递增计算。

再来一个例子，Michael-Scott **非阻塞队列算法**的插入操作，ConcurrentLinkedQueue 就是用这个算法实现的，现在来结合示意图分析一下，很明朗：

Java代码

收藏代码

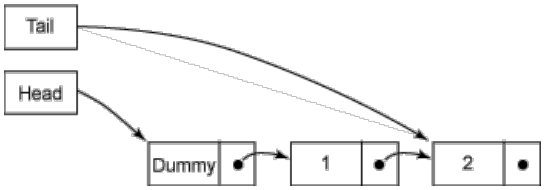
```
1. public class LinkedQueue <E> {
2.    private static class Node <E> {
3.       final E item;
4.       final AtomicReference<Node<E>> next;
5.       Node(E item, Node<E> next) {
6.          this.item = item;
7.          this.next = new AtomicReference<Node<E>>(next);
8.       }
9.    }
10.    private AtomicReference<Node<E>> head
11.       = new AtomicReference<Node<E>>(new Node<E>(null, null));
12.    private AtomicReference<Node<E>> tail = head;
13.    public boolean put(E item) {
14.       Node<E> newNode = new Node<E>(item, null);
15.       while (true) {
16.          Node<E> curTail = tail.get();
17.          Node<E> residue = curTail.next.get();
18.          if (curTail == tail.get()) {
19.             if (residue == null) /* A */ {
20.                if (curTail.next.compareAndSet(null, newNode)) /* C */ {
```

```
21.                     tail.compareAndSet(curTail, newNode) /* D */ ;
22.                     return true;
23.                 }
24.             } else {
25.                 tail.compareAndSet(curTail, residue) /* B */;
26.             }
27.         }
28.     }
29. }
30. }
```

看看这代码完全就是ConcurrentLinkedQueue 源码啊。

插入一个元素涉及头指针和尾指针两个指针更新，这两个更新都是通过 CAS 进行的：从队列当前的最后节点（C）链接到新节点，并把尾指针移动到新的最后一个节点（D）。如果第一步失败，那么队列的状态不变，插入线程会继续重试，直到成功。一旦操作成功，插入被当成生效，其他线程就可以看到修改。还需要把尾指针移动到新节点的位置上，但是这项工作可以看成是 “清理工作”，因为任何处在这种情况下的线程都可以判断出是否需要这种清理，也知道如何进行清理。

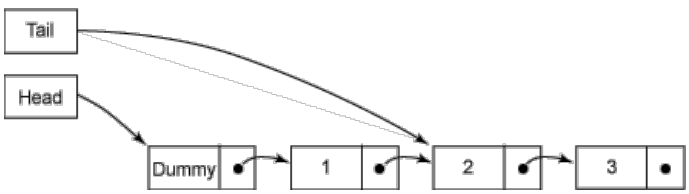
队列总是处于两种状态之一：正常状态（或称静止状态，图 1 和 图 3）或中间状态（图 2）。在插入操作之前和第二个 CAS（D）成功之后，队列处在静止状态；在第一个 CAS（C）成功之后，队列处在中间状态。在静止状态时，尾指针指向的链接节点的 next 字段总为 null，而在中间状态时，这个字段为非 null。任何线程通过比较 tail.next 是否为 null，就可以判断出队列的状态，这是让线程可以帮助其他线程 “完成” 操作的关键。



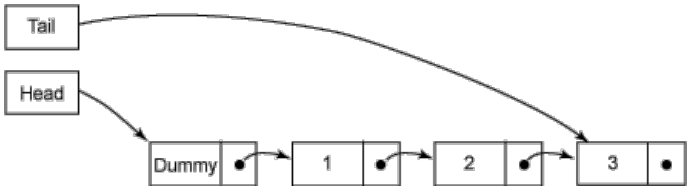
上图显示的是：有两个元素，处在静止状态的队列

插入操作在插入新元素（A）之前，先检查队列是否处在中间状态。如果是在中间状态，那么肯定有其他线程已经处在元素插入的中途，在步骤（C）和（D）之间。不必等候其他线程完成，当前线程就可以 “帮助” 它完成操作，把尾指针向前移动（B）。如果有必要，它还会继续检查尾指针并向前移动指针，直到队列处于静止状态，这时它就可以开始自己的插入了。

第一个 CAS（C）可能因为两个线程竞争访问队列当前的最后一个元素而失败；在这种情况下，没有发生修改，失去 CAS 的线程会重新装入尾指针并再次尝试。如果第二个 CAS（D）失败，插入线程不需要重试 —— 因为其他线程已经在步骤（B）中替它完成了这个操作！



上图显示的是：处在插入中间状态的队列，在新元素插入之后，尾指针更新之前


Tail
Head
Dummy
1
2
3