

DEC 2013

18

75,262 人阅读

方 腾飞

JAVA

本站原创

(16 votes, average: 4.31 out of 5)

2 条评论

## 聊聊并发（七）——Java中的阻塞队列

原文首发于[InfoQ](#)

### 1. 什么是阻塞队列？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。

当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法:

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException(“Queue full”)异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常。

返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null

一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列为空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。

超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

### 2. Java里的阻塞队列

JDK7提供了7个阻塞队列。分别是

- ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。
- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

#### ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列：

```
1 | ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true);
```

访问者的公平性是使用可重入锁实现的，代码如下：

```
1 | public ArrayBlockingQueue(int capacity, boolean fair) {
2 |     if (capacity <= 0)
3 |         throw new IllegalArgumentException();
4 |     this.items = new Object[capacity];
5 |     lock = new ReentrantLock(fair);
6 |     notEmpty = lock.newCondition();
7 |     notFull = lock.newCondition();
8 | }
```

#### LinkedBlockingQueue

LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序。

#### PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比较器comparator来指定元素的排序规则。元素按照升序排列。

#### DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将DelayQueue运用在以下应用场景：

缓存系统的设计：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。

近期文章

- [《Hyperledger Fabric官方文档》什么是Hyperledger Fabric?](#)
- [实际项目中运用责任链模式](#)
- [代码走查如何保证软件质量](#)
- [《Hyperledger Fabric官方文档》之关键概念翻译邀请](#)
- [微服务集成测试自动化探索](#)
- [《Hyperledger Fabric官方文档》介绍](#)
- [TCP 滑动窗口 与窗口缩放因子\(Window Scaling\)](#)
- [不可错过的CMS学习笔记](#)
- [白话网络通讯](#)
- [基于redis的分布式锁](#)
- [猫狗队列的再解](#)
- [如何成为一位「不那么差」的程序员](#)
- [消息队列二三事](#)
- [设计一个百万级的消息推送系统](#)
- [《Python3.6官方文档》– 4.更多流程控制语句](#)
- [Java并发编程之美](#)
- [java多线程系列：通过对战游戏学习CyclicBarrier](#)
- [《Python3.6官方文档》14 章](#)
- [源码分析Kafka之Producer](#)
- [容器\(docker\)中运行java需关注的几个小问题](#)
- [如何优雅的使用和理解线程池](#)
- [《Python3.6官方文档》– 第12章 虚拟环境和包](#)
- [《Python3.6官方文档》– 第13章 接下来](#)
- [HashMap? ConcurrentHashMap? 相信看完这篇没人能难住你!](#)
- [Raft论文读书笔记](#)
- [《Python3.6官方文档》– 第二章 使用python解释器](#)
- [你真的了解延时队列吗\(一\)](#)
- [MySQL中流式查询使用](#)
- [《Python3.6官方文档》– 第十章](#)
- [《以太坊官方文档》Ethereum Clients—Choosing a client](#)

热门文章

[Google Guava官方教程（中文版）](#) 627,830 人阅读

定时任务调度。使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行，从比如TimerQueue就是使用DelayQueue实现的。

队列中的Delayed必须实现compareTo来指定元素的顺序。比如让延时时间最长的放在队列的末尾。实现代码如下：

```
01 public int compareTo(Delayed other) {
02     if (other == this) // compare zero ONLY if same object
03         return 0;
04     if (other instanceof ScheduledFutureTask) {
05         ScheduledFutureTask x = (ScheduledFutureTask)other;
06         long diff = time - x.time;
07         if (diff < 0)
08             return -1;
09         else if (diff > 0)
10             return 1;
11     } else if (sequenceNumber < x.sequenceNumber)
12         return -1;
13     else
14         return 1;
15 }
16 long d = (getDelay(TimeUnit.NANOSECONDS) -
17         other.getDelay(TimeUnit.NANOSECONDS));
18 return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
19 }
```

如何实现Delayed接口

我们可以参考ScheduledThreadPoolExecutor里ScheduledFutureTask类。这个类实现了Delayed接口。首先：在对象创建的时候，使用时间记录前对象什么时候可以使用，代码如下：

```
1 ScheduledFutureTask(Runnable r, V result, long ns, long period) {
2     super(r, result);
3     this.time = ns;
4     this.period = period;
5     this.sequenceNumber = sequencer.getAndIncrement();
6 }
```

然后使用getDelay可以查询当前元素还需要延时多久，代码如下：

```
public long getDelay(TimeUnit unit) {
    return unit.convert(time - now(), TimeUnit.NANOSECONDS);
}
```

通过构造函数可以看出延迟时间参数ns的单位是纳秒，自己设计的时候最好使用纳秒，因为getDelay时可以指定任意单位，一旦以纳秒作为单位，而延时的时间又精确不到纳秒就麻烦了。使用时请注意当time小于当前时间时，getDelay会返回负数。

如何实现延时队列

延时队列的实现很简单，当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

```
1 long delay = first.getDelay(TimeUnit.NANOSECONDS);
2 if (delay <= 0)
3     return q.poll();
4 else if (leader != null)
5     available.await();
```

## SynchronousQueue

SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue的吞吐量高于LinkedBlockingQueue 和 ArrayBlockingQueue。

## LinkedTransferQueue

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列LinkedTransferQueue多了tryTransfer和transfer方法。

transfer方法。如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。transfer方法的关键代码如下：

```
1 Node pred = tryAppend(s, haveData);
2 return awaitMatch(s, pred, e, (how == TIMED), nanos);
```

第一行代码是试图把存放当前元素的s节点作为tail节点。第二行代码是让CPU自旋等待消费者消费元素。因为自旋会消耗CPU，所以自旋一定的次数后使用Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

tryTransfer方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回。而transfer方法是必须等到消费者消费了才返回。

对于带有时间限制的tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回false，如果在超时时间内消费了元素，则返回true。

## LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque多了addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast等方法，以First单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法add等同于addLast，移除方法remove等效于removeFirst。但是take方法却等同于takeFirst，不知道是不是Jdk的bug，使用时还是用带有First和Last后缀的方法更清楚。在初始化LinkedBlockingDeque时可以初始化队列的容量，用来防止其再扩容时过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

## 3. 阻塞队列的实现原理

如果队列是空的，消费者会一直等待，当生产者添加元素时候，消费者是如何知道当前队列有元素的呢？如果让你来设计阻塞队列你会如何设计，让生产者和消费者能够高效率的进行通讯呢？让我们先来看看JDK是如何实现的。

Java NIO系列教程（一）Java NIO 概述 463,719 人阅读

Java并发性和多线程介绍目录 335,908 人阅读

Java NIO 系列教程 329,126 人阅读

Java NIO系列教程（十二）Java NIO与IO 260,566 人阅读

Java8初体验（二）Stream语法详解 236,053 人阅读

Java NIO系列教程（六）Selector 230,869 人阅读

Java NIO系列教程（三）Buffer 228,425 人阅读

Java NIO系列教程（二）Channel 223,031 人阅读

69道Spring面试题和答案 184,105 人阅读

《Storm入门》中文版 183,166 人阅读

Netty 5用户指南 176,816 人阅读

并发框架Disruptor译文 164,931 人阅读

面试题 158,939 人阅读

Java 7 并发编程指南中文版 146,012 人阅读

Java NIO系列教程（八）SocketChannel 144,744 人阅读

[Google Guava] 2.3-强大的集合工具类：ja... 141,304 人阅读

[Google Guava] 3-缓存 140,993 人阅读

如何创建并运行java线程 134,812 人阅读

聊聊并发（三）Java线程池的分析和使用 134,148 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读

多线程面试题及答案 133,911 人阅读



使用通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生

产者当前队列可用。通过查看JDK源码发现ArrayBlockingQueue使用了Condition来实现，代码如下：

```
01 private final Condition notFull;
02 private final Condition notEmpty;
03
04 public ArrayBlockingQueue(int capacity, boolean fair) {
05     //省略其他代码
06     notEmpty = lock.newCondition();
07     notFull = lock.newCondition();
08 }
09
10 public void put(E e) throws InterruptedException {
11     checkNotNull(e);
12     final ReentrantLock lock = this.lock;
13     lock.lockInterruptibly();
14     try {
15         while (count == items.length)
16             notFull.await();
17         insert(e);
18     } finally {
19         lock.unlock();
20     }
21 }
22
23 public E take() throws InterruptedException {
24     final ReentrantLock lock = this.lock;
25     lock.lockInterruptibly();
26     try {
27         while (count == 0)
28             notEmpty.await();
29         return extract();
30     } finally {
31         lock.unlock();
32     }
33 }
34
35 private void insert(E x) {
36     items[putIndex] = x;
37     putIndex = inc(putIndex);
38     ++count;
39     notEmpty.signal();
40 }
```

当我们往队列里插入一个元素时，如果队列不可用，阻塞生产者主要通过LockSupport.park(this);来实现

```
01 public final void await() throws InterruptedException {
02     if (Thread.interrupted())
03         throw new InterruptedException();
04     Node node = addConditionWaiter();
05     int savedState = fullyRelease(node);
06     int interruptMode = 0;
07     while (!isOnSyncQueue(node)) {
08         LockSupport.park(this);
09         if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
10             break;
11     }
12     if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
13         interruptMode = REINTERRUPT;
14     if (node.nextWaiter != null) // clean up if cancelled
15         unlinkCancelledWaiters();
16     if (interruptMode != 0)
17         reportInterruptAfterWait(interruptMode);
18 }
19 }
```

继续进入源码，发现调用setBlocker先保存下将要阻塞的线程，然后调用unsafe.park阻塞当前线程。

```
1 public static void park(Object blocker) {
2     Thread t = Thread.currentThread();
3     setBlocker(t, blocker);
4     unsafe.park(false, 0L);
5     setBlocker(t, null);
6 }
```

unsafe.park是个native方法，代码如下：

```
1 public native void park(boolean isAbsolute, long time);
```

park这个方法会阻塞当前线程，只有以下四种情况中的一种发生时，该方法才会返回。

与park对应的unpark执行或已经执行时。注意：已经执行是指unpark先执行，然后再执行的park。

线程被中断时。

如果参数中的time不是零，等待了指定的毫秒数时。

发生异常现象时。这些异常事先无法确定。

我们继续看一下JVM是如何实现park方法的，park在不同的操作系统使用不同的方式实现，在linux下使用的是系统方法pthread\_cond\_wait

实现。实现代码在JVM源码路径src/os/linux/vm/os\_linux.cpp里的 os::PlatformEvent::park方法，代码如下：

```
01 void os::PlatformEvent::park() {
02     int v ;
03     for (;;) {
04         v = _Event ;
05         if (Atomic::cmpxchg (v-1, &_amp;_Event, v) == v) break ;
06     }
07     guarantee (v >= 0, "invariant") ;
08     if (v == 0) {
09         // Do this the hard way by blocking ...
10         int status = pthread_mutex_lock(&_amp;mutex);
11         assert_status(status == 0, status, "mutex_lock");
12         guarantee (_nParked == 0, "invariant") ;
13         ++ _nParked ;
14         while (_Event < 0) {
15             status = pthread_cond_wait(&_amp;cond, &_amp;mutex);
16             // for some reason, under 2.7 lwp_cond_wait() may return ETIME ...
17             // Treat this the same as if the wait was interrupted
18             if (status == ETIME) { status = EINTR; }
19             assert_status(status == 0 || status == EINTR, status, "cond_wait");
20         }
21         -- _nParked ;
22
23         // In theory we could move the ST of 0 into _Event past the unlock(),
24         // but then we'd need a MEMBAR after the ST.
25         _Event = 0 ;
26         status = pthread_mutex_unlock(&_amp;mutex);
27         assert_status(status == 0, status, "mutex_unlock");
28     }
29     guarantee (_Event >= 0, "invariant") ;
30 }
31
32 }
```

pthread\_cond\_wait是一个多线程的条件变量函数，cond是condition的缩写，字面意思可以理解为线程在等待一个条件发生，这个条件是一个

全局变量。这个方法接收两个参数，一个共享变量\_cond，一个互斥量\_mutex。而unpark方法在linux下是使用pthread\_cond\_signal实现的。

park 在windows下则是使用WaitForSingleObject实现的。

架构 (38)

活动 (6)

网络 (7)

面试 (1)

CN22

当队列满时，生产者往阻塞队列里插入一个元素，生产者线程会进入WAITING (parking)状态。我们可以使用jstack dump阻塞的生产者线程看

到这点：

```
1 | "main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting on condition [0x000000010164d000]
2 |   java.lang.Thread.State: WAITING (parking)
3 |     at sun.misc.Unsafe.park(Native Method)
4 |       - parking to wait for  <0x0000000140559fe8> (a
5 | java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
6 |   at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
7 |   at
8 | java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
   at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:324)
   at blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQueueTest.java:11)
```

#### 4. 参考资料

[JDK6.0阻塞队列API文档](#)

JDK1.7源码

[JVM Park的windows实现](#)

[JVM Park的linux实现代码](#)

原创文章，转载请注明： 转载自[并发编程网 – ifeve.com](#)本文链接地址：[聊聊并发（七）——Java中的阻塞队列](#)

## CacheKit

### 从零开始打造一个分布式缓存系统

AboutLatest Posts



方 腾飞

花名清英，并发网(ifeve.com)创始人，畅销书《Java并发编程的艺术》作者，蚂蚁金服技术专家。目前工作于支付宝微贷事业部，关注互联网金融，并发编程和敏捷实践。微信公众号aliqinying。

★[添加本文到我的收藏](#)

#### Related Posts:

- 聊聊并发（二）**Java SE1.6**中的**Synchronized**
- JUC LinkedBlockingQueue**
- 聊聊并发（一）深入分析**Volatile**的实现原理
- 聊聊并发（十）生产者消费者模式
- 聊聊并发（六）**ConcurrentLinkedQueue**的实现原理分析
- 聊聊并发（五）原子操作的实现原理
- 聊聊并发（四）深入分析**ConcurrentHashMap**
- 聊聊并发（三）**Java**线程池的分析和使用
- 聊聊并发（八）——**Fork/Join**框架介绍
- 聊聊并发-**Java**中的**Copy-On-Write** 容器
- 怎么理解**Condition**
- Java**锁的种类以及辨析（三）： 阻塞锁
- 聊聊并发系列文章
- 《 **Java**并发编程从入门到精通》第**5**章 多线程之间交互：线程阀
- JAVA**互斥锁(**synchronized&Lock**): 行为分析及源码

发表评论

RSS 订阅评论

Trackback 关闭

评论 (2)



Snway

2013/12/20 12:49 下午

[登录以回复](#)

[引用](#)

归纳得不错，可作为手册参考。



田红瀚

2014/06/11 12:05 下午

[登录以回复](#)

[引用](#)

LinkedBlockingDeque和ArrayBlockingQueue中都使用唯一的ReentrantLock,高并发时竞争还是很严重啊。JCP书里的218页的测试结果显示，高并发下还是两个ReentrantLock的LinkedBlockingQueue的可伸缩性要好。奇怪的是LinkedBlockingDeque为什么采用一个ReentrantLock,这样两端操作还是有竞争啊。

您必须 [登陆](#) 后才能发表评论

基于锁的原子操作

ReentrantLock(重入锁)以及公平性