



链滴

队列：队列在线程池等有限资源池中的应用

作者：[someone26671](#)

原文链接：<https://ld246.com/article/1547496243257>

来源网站：[链滴](#)

许可协议：[署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

<p>本文章是从极客时间抄写的，仅仅是看不懂，抄了一遍，分享给大家。

© 版权归极客邦科技所有</p>

<hr>

<p>我们知道，CPU 资源是有限的，任务的处理速度与线程个数并不是线性正相关。相反，过多的程反而会导致 CPU 频繁切换，处理性能下降。所以，线程池的大小一般都是综合考虑要处理任务的点和硬件环境，来事先设置的。</p>

<p>当我们向固定大小的线程池中请求一个线程时，如果线程池中没空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢？</p>

<p>实际上，这些问题并不复杂，其底层的数据结构就是我们今天要学的内容，队列（queue）。</p>

<h2 id="如何理解-队列--">如何理解“队列”？</h2>

<p>队列这个概念非常好理解。你可以把它想象成排队买票，先来的先买，后来的人只能站末尾，不许插队。**先进者先出，这就是典型的“队列”**。</p>

<p>我们知道，栈只支持两种基本操作：入栈 push()和出栈 pop()。队列跟栈常相似，支持的操作也很有限，最基本的操作也是两个：入队 enqueue()，放个数据到队列尾部；出队 dequeue()，从队列头部取出一个元素。</p>

<p></p>

<p>所以，队列跟栈一样，也是一种操作受限的线性表数据结构更。</p>

<p>队列的概念很好理解，基本操作也很容易掌握。作为一种非常基础的数据结构，队列的应用也非常广泛，特别是一些具有某些额外特性的队列，比如循环队列、阻塞队列、并发队列。它们在很多偏低系统、框架、中间件的开发中，起着关键性的作用。比如高性能队列 Disruptor、Linux 环形缓存，用到了循环并发队列；Java concurrent 并发包利用 ArrayBlockingQueue 来实现公平锁等。</p>

<h2 id="顺序队列和链式队列">顺序队列和链式队列</h2>

<p>我们知道了，队列跟栈一样，也是一种抽象的数据结构。它具有先进先出的特性，支持在队尾插元素，在队头删除元素，那究竟该如何实现一个队列呢？</p>

<p>跟栈一样，队列可以用数组来实现，也可以用链表来实现。用数组实现的栈叫做顺序栈，用链表现的栈叫做链式栈。同样，用数组实现的队列叫做顺序队列，用链表实现的队叫做链式队列。</p>

<h2 id="顺序队列">顺序队列</h2>

<p>我们先来看下基于数组的实现方法。我用 Java 语言实现了一下，不过并不包含 Java 语言的高级法，而且我做了比较详细的注释，你应该可以看懂。</p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl"> // 用数组实现的队列
</span></span><span class="highlight-line"><span class="highlight-cl"> public class Arra
Queue {
</span></span><span class="highlight-line"><span class="highlight-cl"> // 数组：items
数组大小：n
</span></span><span class="highlight-line"><span class="highlight-cl"> private String[] it
ms;
</span></span><span class="highlight-line"><span class="highlight-cl"> private int n = 0;
</span></span><span class="highlight-line"><span class="highlight-cl"> // head 表示队
下标，tail 表示队尾下标
</span></span><span class="highlight-line"><span class="highlight-cl"> private int head
0;
</span></span><span class="highlight-line"><span class="highlight-cl"> private int tail =
;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl"> // 申请一个大小为
capacity 的数组
</span></span><span class="highlight-line"><span class="highlight-cl"> public ArrayQue
e(int capacity) {
</span></span></pre>
```

```


</span></span><span class="highlight-line"><span class="highlight-cl">    items = new S
ring[capacity];
</span></span><span class="highlight-line"><span class="highlight-cl">    n = capacity;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl"> // 入队
</span></span><span class="highlight-line"><span class="highlight-cl"> public boolean e
queue(String item) {
</span></span><span class="highlight-line"><span class="highlight-cl"> // 如果 tail == n
表示队列已经满了
</span></span><span class="highlight-line"><span class="highlight-cl"> if (tail == n) ret
rn false;
</span></span><span class="highlight-line"><span class="highlight-cl">    items[tail] = i
em;
</span></span><span class="highlight-line"><span class="highlight-cl">    ++tail;
</span></span><span class="highlight-line"><span class="highlight-cl">    return true;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl"> // 出队
</span></span><span class="highlight-line"><span class="highlight-cl"> public String de
ueue() {
</span></span><span class="highlight-line"><span class="highlight-cl"> // 如果 head
= tail 表示队列为空
</span></span><span class="highlight-line"><span class="highlight-cl"> if (head == tail
return null;
</span></span><span class="highlight-line"><span class="highlight-cl"> // 为了让其他
言的同学看的更加明确，把 -- 操作放到单独一行来写了
</span></span><span class="highlight-line"><span class="highlight-cl"> String ret = it
ms[head];
</span></span><span class="highlight-line"><span class="highlight-cl"> ++head;
</span></span><span class="highlight-line"><span class="highlight-cl"> return ret;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl">>}
</span></span></code></pre>

```


比起栈的数组实现，队列的数组实现稍微有点儿复杂，但是没关系。我稍微解释一下实现思路，很容易就能明白了。

对于栈来说，我们只需要一个**栈顶指针**就可以了。但是队列需要两个指针一个是 head 指针，指向对头；一个是 tail 指针，指向队尾。

你可以结合下面这幅图来理解。当 a、b、c、d 依次入队之后，队列中的 head 指针指向下标为 0 的位置，tail 指针指向下标为 4 的位置。



当我们调用两次出队操作之后，队列中 head 指针指向下标为 2 的位置，tail 指针仍然指向下标为 4 的位置。



你肯定已经发现了，随着不停地进行入队、出队操作，head 和 tail 都会持续往后移动。当 tail 动到最右边，即使数组中还有空闲空间，也无法继续往队列中添加数据了。这个问题该如何解决呢？

你是否还记得，在数组那一节，我们也遇到过类似的问题，就是数组的删除操作会导致数组中的数据不连续。你还记得我们当时是怎么解决的吗？对，用**数据搬移**！但是，每次行出队操作都相当于删除数组下标为 0 的数据，要搬移整个队列中的数据，这样出队操作的时间复杂

就会从原来的 $O(1)$ 变为 $O(n)$ 。能不能优化一下? </p>

<p>实际上,我们在出队时可以用不用搬移数据。如果没有空闲空间了,我们只需要在入队时,再集中发一次数据的搬移操作。借助这个思想,出队函数 `dequeue()` 保持不变,我们稍加改造一下入队函数 `nqueue()` 的实现,就可以轻松解决刚才的问题了。下面是具体的代码: </p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl"> //入队操作,将item放入队尾
</span></span><span class="highlight-line"><span class="highlight-cl"> public boolean enqueue(String item){
</span></span><span class="highlight-line"><span class="highlight-cl"> // tail == n
示队列末尾没有空间了
</span></span><span class="highlight-line"><span class="highlight-cl"> if(tail == n){
</span></span><span class="highlight-line"><span class="highlight-cl"> // tail == n
&amp;&amp; head == 0,表示整个队列都占满了
</span></span><span class="highlight-line"><span class="highlight-cl"> if(head == 0
return false;
</span></span><span class="highlight-line"><span class="highlight-cl"> // 数据搬移
</span></span><span class="highlight-line"><span class="highlight-cl"> for(int i = h
ad;i &lt; tail; ++i){
</span></span><span class="highlight-line"><span class="highlight-cl"> items[i-h
ad] = item[i];
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl"> //搬移完之
重新更新head 和 tail
</span></span><span class="highlight-line"><span class="highlight-cl"> tail -= head;
</span></span><span class="highlight-line"><span class="highlight-cl"> head = 0;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl"> items[tail] = it
m;
</span></span><span class="highlight-line"><span class="highlight-cl"> ++tail;
</span></span><span class="highlight-line"><span class="highlight-cl"> return true;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>
```

<p>从代码中我们看到,当队列的 `tail` 指针移动到数组的最右边后,如果有新的数据入队,我们可以 `head` 到 `tail` 之间的数据,整体搬移到数组中 `0` 到 `tail-head` 的位置。 </p>

<p> </p>

<p>这种实现思路中,出队操作的时间复杂度仍然是 $O(1)$,但入队操作的时间复杂的还是 $O(1)$ 吗?可以用我们第 3 节、第 4 节讲的算法复杂度分析方法,自己试着分析一下。 </p>

<h2 id="链式队列">链式队列</h2>

<p>接下来,我们再来看下基于链表的队列实现方法 </p>

<p>基于链表的实现,我们同样需要两个指针: `head` 指针和 `tail` 指针。它们分别指向链表的第一个点和最后一个结点。如图所示,入队时, `tail->next=new node,tail=tail->next`;出队时, `head = head->next`。我将具体的代码放在 GitHub 上,你可以自己试着实现一下,然后再去 GitHub 跟我实现的代码对比下,看写的对不对。 </p>

<p> </p>

<h2 id="循环队列">循环队列</h2>

<p>我们刚才用数组来实现队列的时候,在 `tail==n` 时,会有数据搬移操作,这样入队操作性能就会受到影响。那有没有办法能够避免数据搬移呢?我们来看看循环队列的解决思路。 </p>

<p>循环队列,顾名思义,它长得像一个环。原本数组是有头有尾的,是一条直线。现在我们把首尾

连，扳成了一个环。我画了一张图，你可以直观地感受一下。



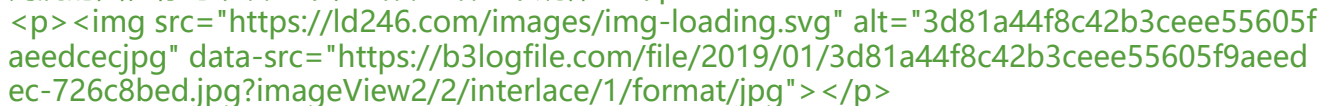
我们可以看到，图中这个队列的大小为 8，当前 head=4，tail=7。当有一个新的元素 a 入列时我们放入下标为 7 的位置。但这个时候，我们并不把 tail 更新为 8，而是将其在环中后移一位，到下为 0 的位置。当再有一个元素 b 入队时，我们将 b 放入下标为 0 的位置，然后 tail 加 1 更新为 1。以，在 a, b 依次入队之后，循环队列中的元素就变成了下面的样子：



通过这样的方法，我们成功避免了数据搬移操作。看起来不难理解，但是循环队列的代码实现难要比前面讲的非循环队列难多了。要想写出没有 Bug 的循环队列的实现代码，我个人觉得，最关键的是，**确定好队空和队满的判定条件**。

在用数组实现的非循环队列中，对满的判断条件是 $tail == n$ ，队空的判断条件是 $head == tail$ 。针对循环队列，如何判断队空和队满呢？

队列为空的判断条件仍然是 $head == tail$ 。但队列满的判断条件就稍微有点复杂了。我画了一张列满的图，你可以看一下，试着总结一下规律。



就像我图中画的队满的情况， $tail=3$ ， $head=4$ ， $n=8$ ，所以总结一下规律就是： $(3+1)\%8=4$ 。画几张队满的图，你就会发现，当队满时， $*(tail+1)\%n=head$ 。

你有没有发现，当队列满时，图中的 tail 指向的位置实际上是没有存储数据的。所以，循环队列浪费一个数组的存储空间。

Talk is cheap，如果还是没怎么理解，那就 show you code 吧。

```
public class CircularQueue{
    //数组: items
    数组大小: n
    private String[]
    items;
    private int n =
    0;
    //head表示队
    下标, tail表示队尾下标
    private int he
    d = 0;
    private int tail
    = 0;
    //申请一个大
    为capacity的数组
    public Circula
    Queue(int capacity){
        items = new
    String[capacity];
        n = capacity

    }
    //入队
    public boolean
    enqueue(String item){
```

```

</span></span><span class="highlight-line"><span class="highlight-cl"> //队列满了
</span></span><span class="highlight-line"><span class="highlight-cl"> if((tail + 1 )
% n ==head) return false;
</span></span><span class="highlight-line"><span class="highlight-cl"> items[tail] =
tem;
</span></span><span class="highlight-line"><span class="highlight-cl"> tail = (tail +
1) % n;
</span></span><span class="highlight-line"><span class="highlight-cl"> return true;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl"> //出队
</span></span><span class="highlight-line"><span class="highlight-cl"> public String
enqueue(){
</span></span><span class="highlight-line"><span class="highlight-cl"> //如果 head
== tail 表示队列为空
</span></span><span class="highlight-line"><span class="highlight-cl"> if(head ==ta
l) return null;
</span></span><span class="highlight-line"><span class="highlight-cl"> String ret =
tems[head];
</span></span><span class="highlight-line"><span class="highlight-cl"> head = (he
d +1) % n;
</span></span><span class="highlight-line"><span class="highlight-cl"> return ret ;
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span><span class="highlight-line"><span class="highlight-cl"> }
</span></span></code></pre>

```

阻塞队列和并发队列

前面讲的内容理论比较多，看起来很难跟实际的项目开发扯上关系。确实，队列这种数据结构很基础，平时的业务开发不大可能从零实现一个队列，甚至都不会直接用到。而一些具有特殊特性的队列用却比较广泛，比如阻塞队列和并发队列。

阻塞队列

阻塞队列其实就是在队列的基础上增加乐阻塞操作。简单来说，就是在队列为空的时候，从队头数据会被阻塞。因为此时还没有数据可取，直到队列中有了数据才能返回；如果队列已经满了，那么入数据的操作就会被阻塞，直到队列中有空闲位置后再插入数据，然后再返回。


 5ef3326181907dea0964f61290185eb.jpg" data-bbox="76 561 916 609"/>

你应该已经发现了，上述的定义就是一个“生产者 - 消费者模型”！是的，我们可以使用阻塞队列，轻松实现一个“生产者 - 消费者模型”！

这种基于阻塞队列实现的“生产者 - 消费者模型”，可以有效地协调生产和消费的速度。当“生产者”生产数据的速度过快，“消费者”来不及消费时，存储数据的队列很快就会满了。这个时候，生产者就阻塞等待，直到“消费者”消费了数据，“生产者”才会被唤醒继续“生产”。

而且不仅如此，基于阻塞队列，我们还可以通过协调“消费者”消费了数据，“生产者”才会唤醒继续“生产”。

而且不仅如此，基于阻塞队列，我们还可以通过协调“生产者”和“消费者”的个数，来提高数据的处理效率。比如前面的例子，我们可以多配置几个“消费者”，来应付一个“生产者”。

 9f539cc0f1edc20e7fa655913898067.jpg" data-bbox="76 747 917 795"/>

并发队列

前面我们讲了阻塞队列，在多线程情况下，会有多个线程同时操作队列，这个时候就会存在线程安全问题，那如何实现一个线程安全的队列呢？

线程安全的队列我们叫作**并发队列**。最简单直接的实现方式是直接在 enqueue()、dequeue()方法上加锁，但是锁粒度大并发度会比较低，同一时刻仅允许一个存或者取操作。实际上，基于数组的循环队列，利用 CAS 原子操作，可以实现非常高效的并发队列。这也是循环队列

链式队列应用更加广泛的原因。在实战篇讲 Disruptor 的时候，我会再详细讲并发队列的应用。

解答开篇

队列的知识就讲完了，我们现在回过来看下开篇的问题。线程池没有空闲线程是，新的任务请求资源时，线程池该如何处理？各种处理策略又是如何实现的呢？

我们一般有两种处理策略。第一种是非阻塞的处理方式，直接拒绝任务请求；另一种是阻塞的方式，将请求排队，等到有空闲线程是，取出排队的请求继续处理。那如何存储排队的请求呢？

我们希望公平地处理每个排队的请求，先进者先服务，所以队列这种数据结构很适合来存储排队请求。我们前面说过，队列有基于链表和基于数组这两种实现方式。这两种实现方式对于排队请求又有什么区别呢？

基于链表的实现方式，可以实现一个支持无限排队的无界队列（unbounded queue），但是会导致过多的请求排队等待，请求处理的响应时间过长。所以，针对响应时间比较敏感的系统，基于表实现的无限排队的线程池是不合适的。

而基于数组实现的有界队列（bounded queue），队列的大小有限，所以线程池中排队的请求过队列大小时，接下来的请求就会被拒绝，这种方式对响应时间敏感的系统来说，就相对更加合理。过，设置一个合理的队列大小，也是非常研究的。队列太大导致等待的请求太多，队列太小会导致无法充分利用系统资源、发挥最大性能。

除了前面讲到队列应用在线程池请求排队的场景之外，队列可以应用有任何有限资源池中，用于队请求，比如数据库连接池等。实际上，对于大部分资源有限的场景，当没有空闲资源时基本上都可以通过“队列”这种数据结构来实现请求排队。

内容小结

今天我们讲了一种跟栈很相似的数据结构，队列。关于队列，你能掌握下面的内容，这节就没问题了。

队列最大的特点就是先进先出，主要的两个操作是入队和出队。跟栈一样，它既可以用数组来实现，也可以用链表来实现。用数组实现的叫顺序队列，用链表实现的叫链式队列。特别是长额像一个环循环队列。在数组实现队列的时候，会有数据搬移操作，要想解决数据搬移的问题，我们就需要像环样的循环队列。

循环队列是我们这节的重点。要想写出没有 bug 的循环队列实现代码，关键要确定好队空和队的判定条件，具体的代码你要能写出来。

除此之外，我们还讲了几种高级的队列结构，阻塞队列、并发队列，底层都还是队列这种数据结构，只不过在之上附加了很多其他功能。阻塞队列就是入队、出队操作可能阻塞，并发队列就是队列的作多线程安全。

课后思考

-


除了线程池这种池结构会用到队列排队请求，你还知道有哪些类似的池结构或者场景中会用到队的排队请求呢？

-

今天讲到并发队列，关于如何实现无锁并发队列，网上有非常多的讨论。对这个问题，你怎么看？

欢迎留言和我分享，我会第一时间给你反馈。

我已将本节内容相关的详细代码更新到 GitHub，[戳此](https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Fwangzheng0822%2Falgo)即可查看。

 <https://b3logfile.com/file/2019/01/8e603e3d795fc0ab2698f6f5eabf143-108230fc.jpg?imageView2/2/interlace/1/format/jpg>