

09 | 队列：队列在线程池等有限资源池中的应用

2018-10-10 王争



09 | 队列：队列在线程池等有限资源池中的应用

朗读人：修阳 12'34" | 5.05M

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，过多的线程反而会导致 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特点和硬件环境，来事先设置的。

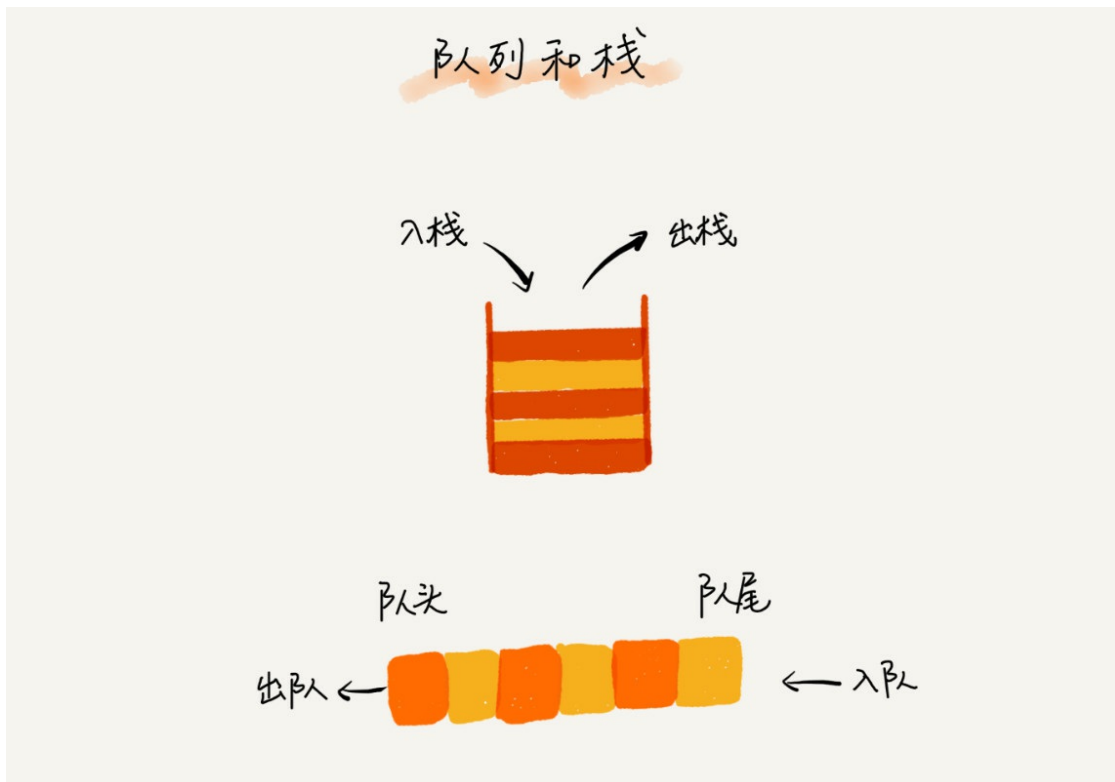
当我们向固定大小的线程池中请求一个线程时，如果线程池中没有空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢？

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）。

如何理解“队列”？

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，不允许插队。先进者先出，这就是典型的“队列”。

我们知道，栈只支持两个基本操作：入栈 `push()` 和出栈 `pop()`。队列跟栈非常相似，支持的操作也很有限，最基本的操作也是两个：入队 `enqueue()`，放一个数据到队列尾部；出队 `dequeue()`，从队列头部取一个元素。



所以，队列跟栈一样，也是一种操作受限的线性表数据结构。

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。它们在很多偏底层系统、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，都用到了循环并发队列；Java concurrent 并发包利用 **ArrayBlockingQueue** 来实现公平锁等。

顺序队列和链式队列

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，支持在队尾插入元素，在队头删除元素，那究竟该如何实现一个队列呢？

跟栈一样，队列可以用数组来实现，也可以用链表来实现。用数组实现的栈叫作顺序栈，用链表实现的栈叫作链式栈。同样，用数组实现的队列叫作顺序队列，用链表实现的队列叫作链式队列。

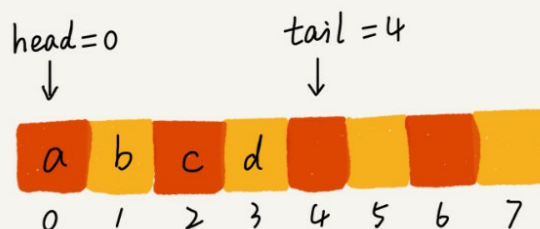
我们先看下基于数组的实现方法。我用 **Java** 语言实现了一下，不过并不包含 **Java** 语言的高级语法，而且我做了比较详细的注释，你应该可以看懂。

```
1 // 用数组实现的队列
2 public class ArrayQueue {
3     // 数组: items, 数组大小: n
4     private String[] items;
5     private int n = 0;
6     // head 表示队头下标, tail 表示队尾下标
7     private int head = 0;
8     private int tail = 0;
9
10    // 申请一个大小为 capacity 的数组
11    public ArrayQueue(int capacity) {
12        items = new String[capacity];
13        n = capacity;
14    }
15
16    // 入队
17    public boolean enqueue(String item) {
18        // 如果 tail == n 表示队列已经满了
19        if (tail == n) return false;
20        items[tail] = item;
21        ++tail;
22        return true;
23    }
24
25    // 出队
26    public String dequeue() {
27        // 如果 head == tail 表示队列为空
28        if (head == tail) return null;
29        // 为了让其他语言的同学看的更加明确, 把 -- 操作放到单独一行来写了
30        String ret = items[head];
31        ++head;
32        return ret;
33    }
34 }
```

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, 你很容易就能明白了。

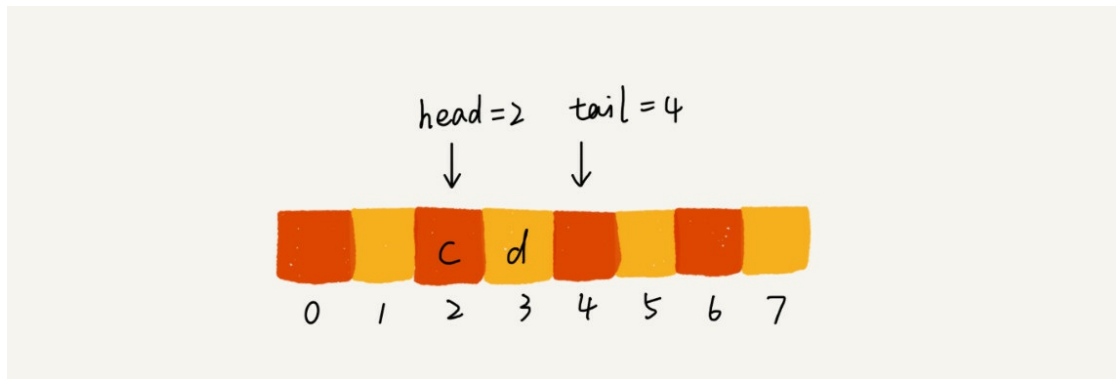
对于栈来说, 我们只需要一个栈顶指针就可以了。但是队列需要两个指针: 一个是 **head** 指针, 指向队头; 一个是 **tail** 指针, 指向队尾。

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** 移动到最右边，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？

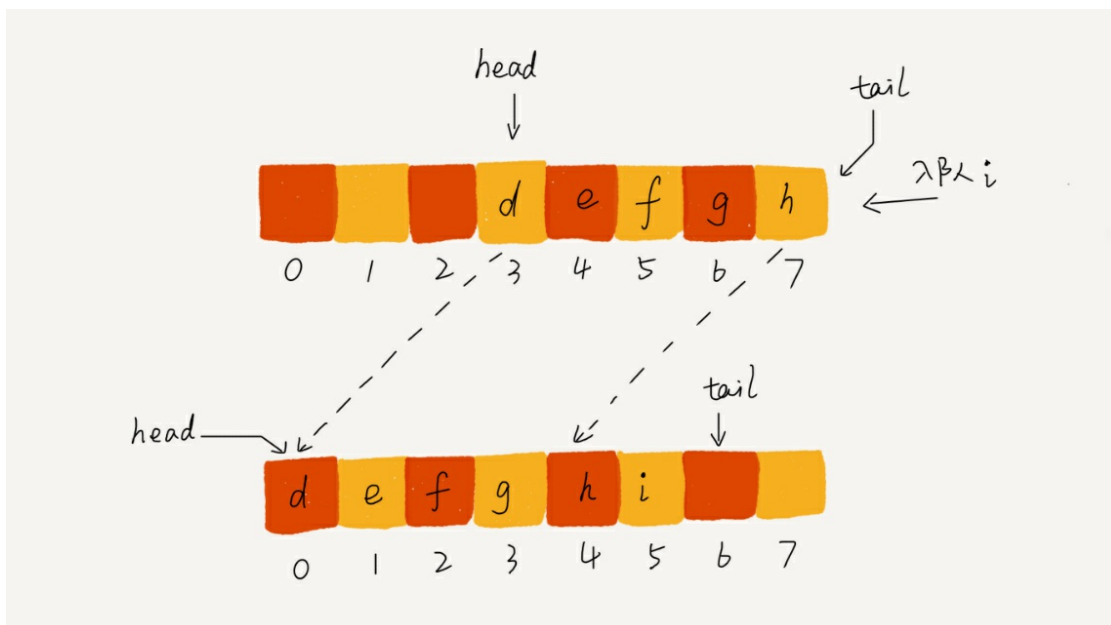
你是否还记得，在数组那一节，我们也遇到过类似的问题，就是数组的删除操作会导致数组中的数据不连续。你还记得我们当时是怎么解决的吗？对，用**数据搬移**！但是，每次进行出队操作都相当于删除数组下标为 **0** 的数据，要搬移整个队列中的数据，这样出队操作的时间复杂度就会从原来的 **$O(1)$** 变为 **$O(n)$** 。能不能优化一下呢？

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，再集中触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 **enqueue()** 的实现，就可以轻松解决刚才的问题了。下面是具体的代码：

```
1 // 入队操作，将 item 放入队尾
2 public boolean enqueue(String item) {
3     // tail == n 表示队列末尾没有空间了
4     if (tail == n) {
5         // tail == n && head == 0, 表示整个队列都占满了
6         if (head == 0) return false;
7         // 数据搬移
8         for (int i = head; i < tail; ++i) {
9             items[i-head] = items[i];
10        }
11        // 搬移完之后重新更新 head 和 tail
12        tail -= head;
13        head = 0;
14    }
15
16    items[tail] = item;
17    ++tail;
18    return true;
19 }
```

复制代码

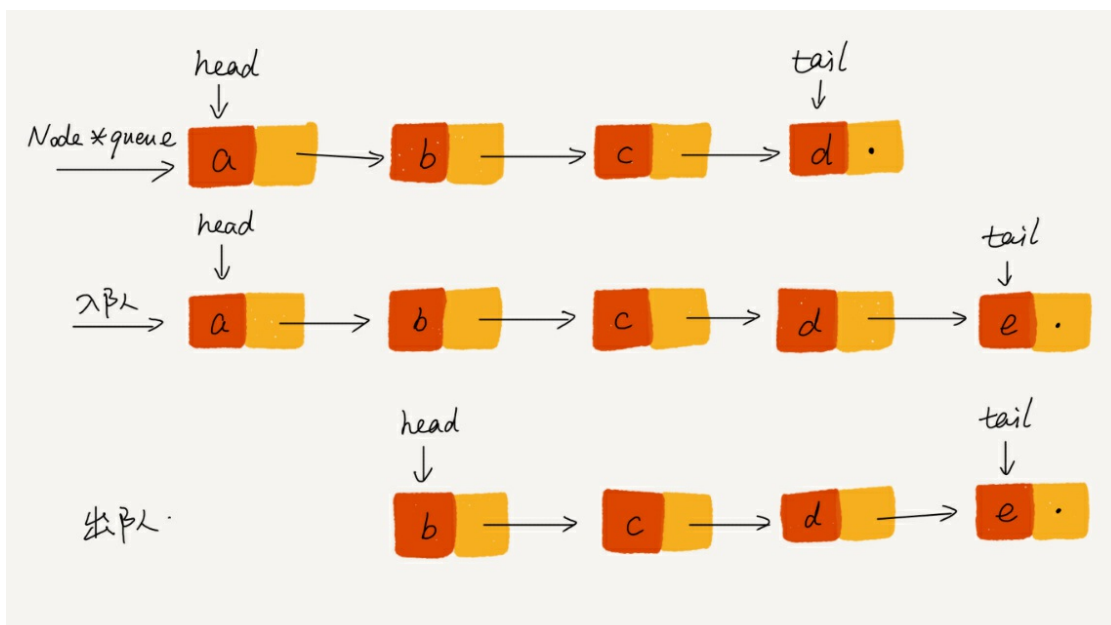
从代码中我们看到，当队列的 **tail** 指针移动到数组的最右边后，如果有新的数据入队，我们可以将 **head** 到 **tail** 之间的数据，整体搬移到数组中 **0** 到 **tail-head** 的位置。



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度还是 $O(1)$ 吗？你可以用我们第 3 节、第 4 节讲的算法复杂度分析方法，自己试着分析一下。

接下来，我们再来看下基于链表的队列实现方法。

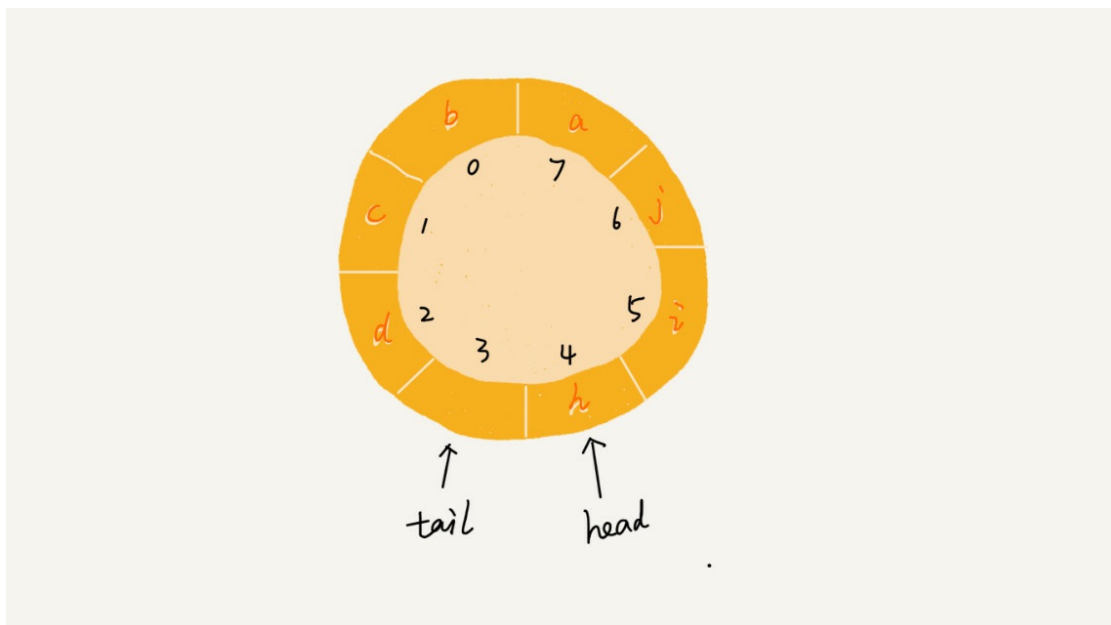
基于链表的实现，我们同样需要两个指针：**head** 指针和 **tail** 指针。它们分别指向链表的第一个结点和最后一个结点。如图所示，入队时，**tail->next = new_node**, **tail = tail->next**；出队时，**head = head->next**。我将具体的代码放到 [Github](#) 上，你可以自己试着实现一下，然后再去 [Github](#) 上跟我实现的代码对比下，看写得对不对。



循环队列

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。

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。现在我们把首尾相连，扳成了一个环。我画了一张图，你可以直观地感受一下。



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

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，循环队列会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。

```
1 public class CircularQueue {
2     // 数组: items, 数组大小: n
3     private String[] items;
4     private int n = 0;
5     // head 表示队头下标, tail 表示队尾下标
6     private int head = 0;
7     private int tail = 0;
8
9     // 申请一个大小为 capacity 的数组
10    public CircularQueue(int capacity) {
11        items = new String[capacity];
12        n = capacity;
13    }
14
15    // 入队
16    public boolean enqueue(String item) {
17        // 队列满了
18        if ((tail + 1) % n == head) return false;
19        items[tail] = item;
20        tail = (tail + 1) % n;
21        return true;
22    }
23
24    // 出队
25    public String dequeue() {
26        // 如果 head == tail 表示队列为空
27        if (head == tail) return null;
28        String ret = items[head];
29        head = (head + 1) % n;
30        return ret;
31    }
32 }
```

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应用却比较广泛, 比如阻塞队列和并发队列。

阻塞队列其实就是在队列基础上增加了阻塞操作。简单来说, 就是在队列为空的时候, 从队头取数据会被阻塞。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, 那么插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。

阻塞队列

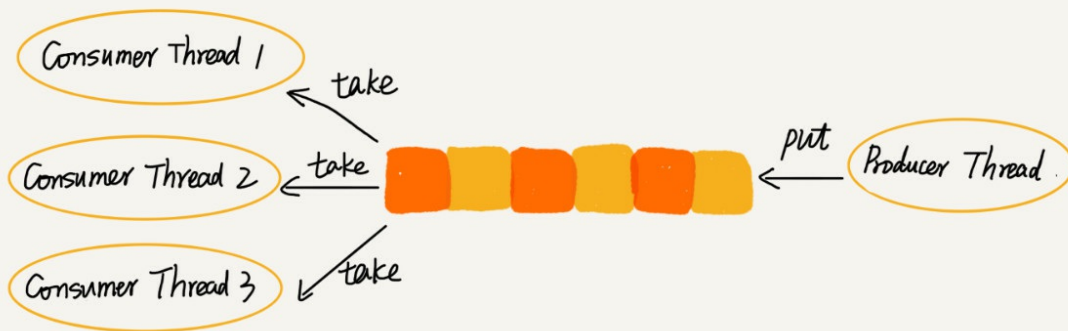


你应该已经发现了，上述的定义就是一个“生产者 - 消费者模型”！是的，我们可以使用阻塞队列，轻松实现一个“生产者 - 消费者模型”！

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“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”，来应对一个“生产者”。

阻塞队列



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。这也是循环队列比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时，新的任务请求线

程资源时，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；另一种是阻塞的处理方式，将请求排队，等到有空闲线程时，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**），但是可能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，基于链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超过队列大小时，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。不过，设置一个合理的队列大小，也是非常有讲究的。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，队列可以应用在任何有限资源池中，用于排队请求，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。特别是长得像一个环的循环队列。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，我们就需要像环一样的循环队列。

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满的判定条件，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、出队操作可以阻塞，并发队列就是队列的操作多线程安全。

课后思考

1. 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列的排队请求呢？
2. 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，你怎么看呢？

欢迎留言和我分享，我会第一时间给你反馈。

[戳此查看本节内容相关的详细代码](#)



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



lpccwin

👍 25

我觉得出队的时间复杂度还是 $O(n)$ ，每次下面都有好多答案，老师不能事后给个标准答案吗？

2018-10-10



蝴蝶

👍 9

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度还是 $O(1)$ 吗想了下，考虑到head可能等于1,2,n-1,经过计算，觉得均摊和平均时间复杂度还是 $O(1)$,对么？

2018-10-10



Peter、桥

👍 8

老师要是有时间对课后问题集中式做下解答就好了

2018-10-10

作者回复

行的

2018-10-11



城

👍 7

1.分布式应用中的消息队列，也是一种队列结构

2.考虑使用CAS实现无锁队列，则在入队前，获取tail位置，入队时比较tail是否发生变化，如果否，则允许入队，反之，本次入队失败。出队则是获取head位置，进行cas。

个人浅见，请批评指正

2018-10-10

作者回复

👍

2018-10-10



bro.

👍 5

老师，课后习题有空讲解一下理解呀！每次看评论，有的还是不太明白的地方

2018-10-10

作者回复

行的呢 我抽空集中答疑一下

2018-10-11



liyghting

👍 3

队列可用于资源有限的场景，比如卖火车票，抢购等

2018-10-10



老司机

👍 3

循环队列真的是比较牛逼的思路，尤其是linux内核源码的kfifo的实现，无论是取模运算转换成取与运算，还是考虑head，tail的溢出，牛逼

2018-10-10



王威人

👍 2

老师，循环队列的数组实现，在您的代码中，入队时会空留出一个位置，而且我感觉不太好理解。我定义一个记录队列大小的值size，当这个值与数组大小相等时，表示队列已满，当tail达到最底时，size不等于数组大小时，tail就指向数组第一个位置。当出队时，size-，入队时size++

2018-10-10

作者回复

你这个思路挺巧妙的 我暂时还没有想到破绽

2018-10-11



最初的印象

👍 2

能不能写下阻塞队列和并发队列的代码

2018-10-10

作者回复

等我有空了吧 最近有点忙

2018-10-11



熊先生

👍 2

常规操作，看一遍，再看一遍，做思考总结，再实现一遍

2018-10-10



苏志辉

👍 2

感觉入队时head为1，2...n-1的概率都是1/n-1,而每种情况对应的复杂度为n-1...,1，所以复杂度为O(n),不知道对不对

2018-10-10



花见笑

👍 2

循环队列的长度设定需要对并发数据有一定的预测，否则会丢失太多请求。

2018-10-10

作者回复

👍

2018-10-10



Lost In The Echo。

👍 2

为什么数组可以用CAS而链表不能？

2018-10-10



objcoding

👍 1

一个基于链表的链式队列:

```
public class LinkedQueue {
    private Node head = null;
    private Node tail = null;
    private static class Node {
        int value;
        Node next;
        public Node(int value) {
            this.value = value;
            this.next = null;
        }
    }

    public boolean enqueue(int value) {
        Node newNode = new Node(value);
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        return true;
    }

    public int dequeue() {
        if (head == null) {
            return -1;
        }
        Node node = head.next;
        int value = node.value;
        head = node;
        return value;
    }
}
```

2018-10-11



路漫漫其修远兮

👍 1

自己尝试了一下不让循环队列有空的存储空间，结果发现很难实现。难道循环队列注定要有一个空的存储空间吗？有没有办法不让循环队列有一个空的存储空间。

2018-10-10



lpccwin

👍 1

我觉得是每种都是 $1/n$ ，然后每种的操作分别是1、 $2...n$ ，然后算法是 $1/n * (1+2+...+n)$ 加权平均算法得到结果 $O(n)$

2018-10-10



lovetechlovelife

👍 1

Spark在处理job的时候，可以配置使用FIFO或者Fair模式

2018-10-10

作者回复



2018-10-11



钢

👍 1

redis的消息队列以及redis的管道技术都用到队列

2018-10-10

作者回复



2018-10-11



传说中的成大大

👍 1

内存池

2018-10-10



飞羽

👍 1

课后思考

1. 作为一个前端开发，首先能想到的比较典型的场景就是事件队列、WebScket连接池、动画队列这三个。

2. 暂时没啥思路，等大神的回答

2018-10-10