

09|队列：队列在线程池等有限资源池中的应用

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？

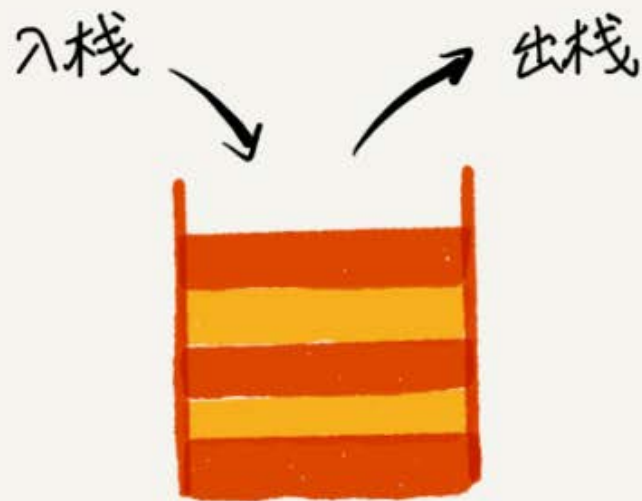
实际上，这些问题并不复杂，其底层的数据结构就是我们今天要学的内容，队列（queue）。

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。

队列和栈



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语言的高级语法，而且我做了比较详细的注释，你应该可以看懂。

```
// 用数组实现的队列
public class ArrayQueue {
    // 数组：items，数组大小：n
    private String[] items;
    private int n = 0;
    // head表示队头下标，tail表示队尾下标
    private int head = 0;
    private int tail = 0;

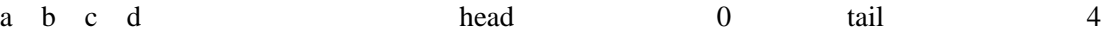
    // 申请一个大小为capacity的数组
    public ArrayQueue(int capacity) {
        items = new String[capacity];
        n = capacity;
    }

    // 入队
    public boolean enqueue(String item) {
        // 如果tail == n 表示队列已经满了
        if (tail == n) return false;
        items[tail] = item;
        ++tail;
        return true;
    }

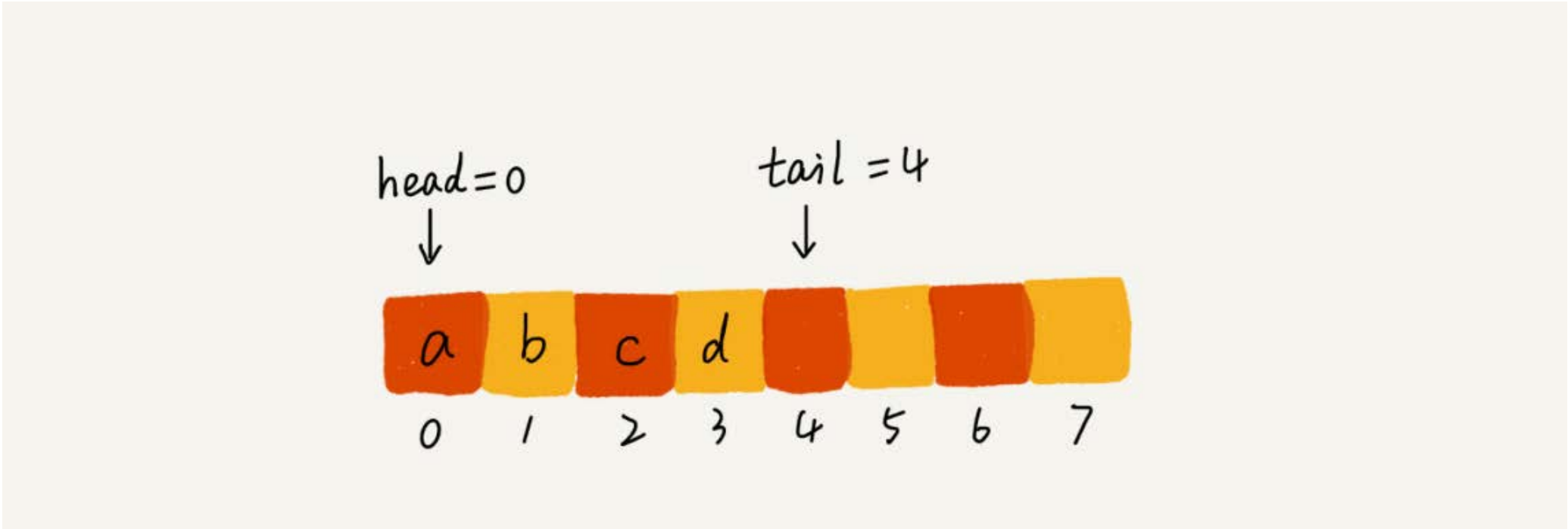
    // 出队
    public String dequeue() {
        // 如果head == tail 表示队列为空
        if (head == tail) return null;
        // 为了让其他语言的同学看的更加明确，把--操作放到单独一行来写了
        String ret = items[head];
        ++head;
        return ret;
    }
}
```

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。

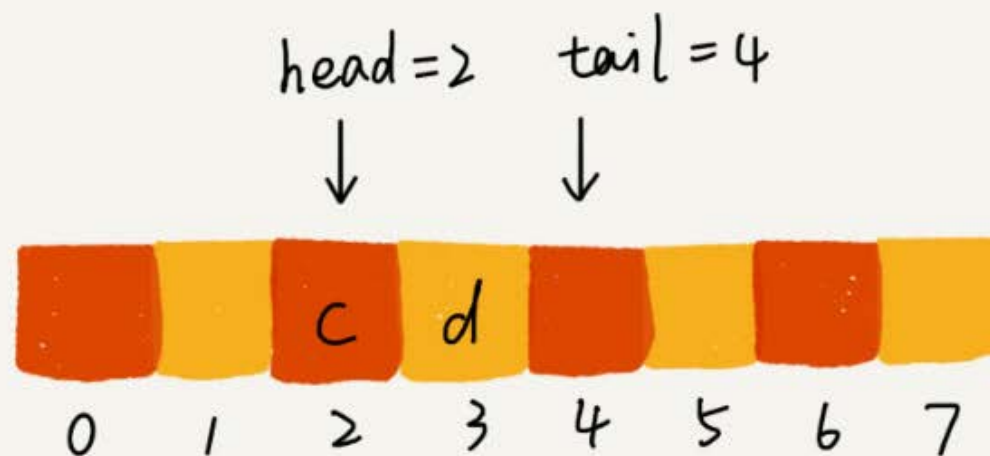
对于栈来说，我们只需要一个栈顶指针就可以了。但是队列需要两个指针：一个是head指针，指向队头；一个是tail指针，指向队尾。



你可以结合下面这幅图来理解。当 `a`、`b`、`c` 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？

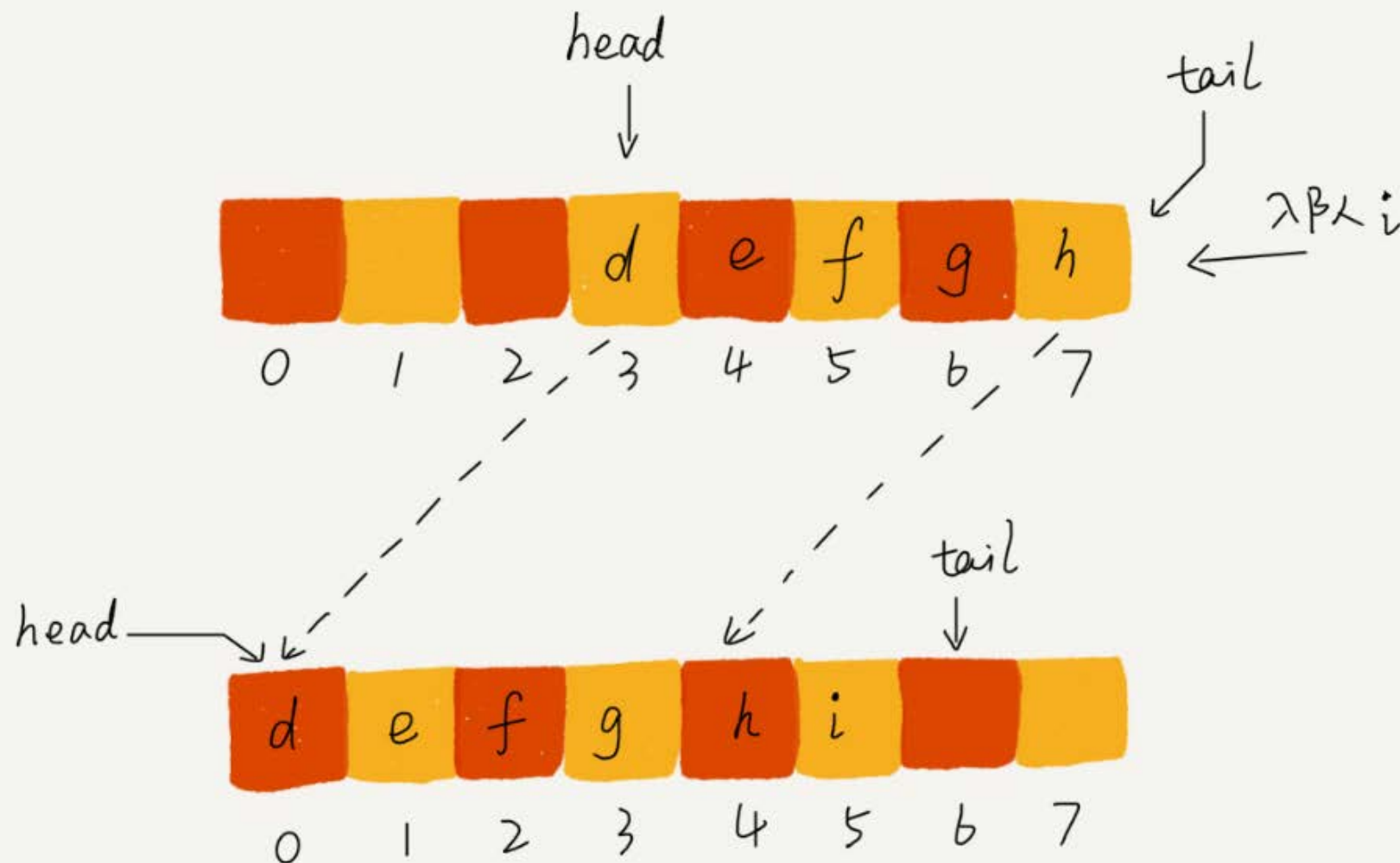
你是否还记得，在数组那一节，我们也遇到过类似的问题，就是数组的删除操作会导致数组中的数据不连续。你还记得我们当时是怎么解决的吗？对，用数据搬移！但是，每次进行出队操作都相当于删除数组下标为0的数据，要搬移整个队列中的数据，这样出队操作的时间复杂度就会从原来的 $O(1)$ 变为 $O(n)$ 。能不能优化一下呢？

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：

```
// 入队操作，将item放入队尾
public boolean enqueue(String item) {
    // tail == n表示队列末尾没有空间了
    if (tail == n) {
        // tail == n && head == 0, 表示整个队列都占满了
        if (head == 0) return false;
        // 数据搬移
        for (int i = head; i < tail; ++i) {
            items[i-head] = items[i];
        }
        // 搬移完之后重新更新head和tail
        tail -= head;
        head = 0;
    }

    items[tail] = item;
    ++tail;
    return true;
}
```

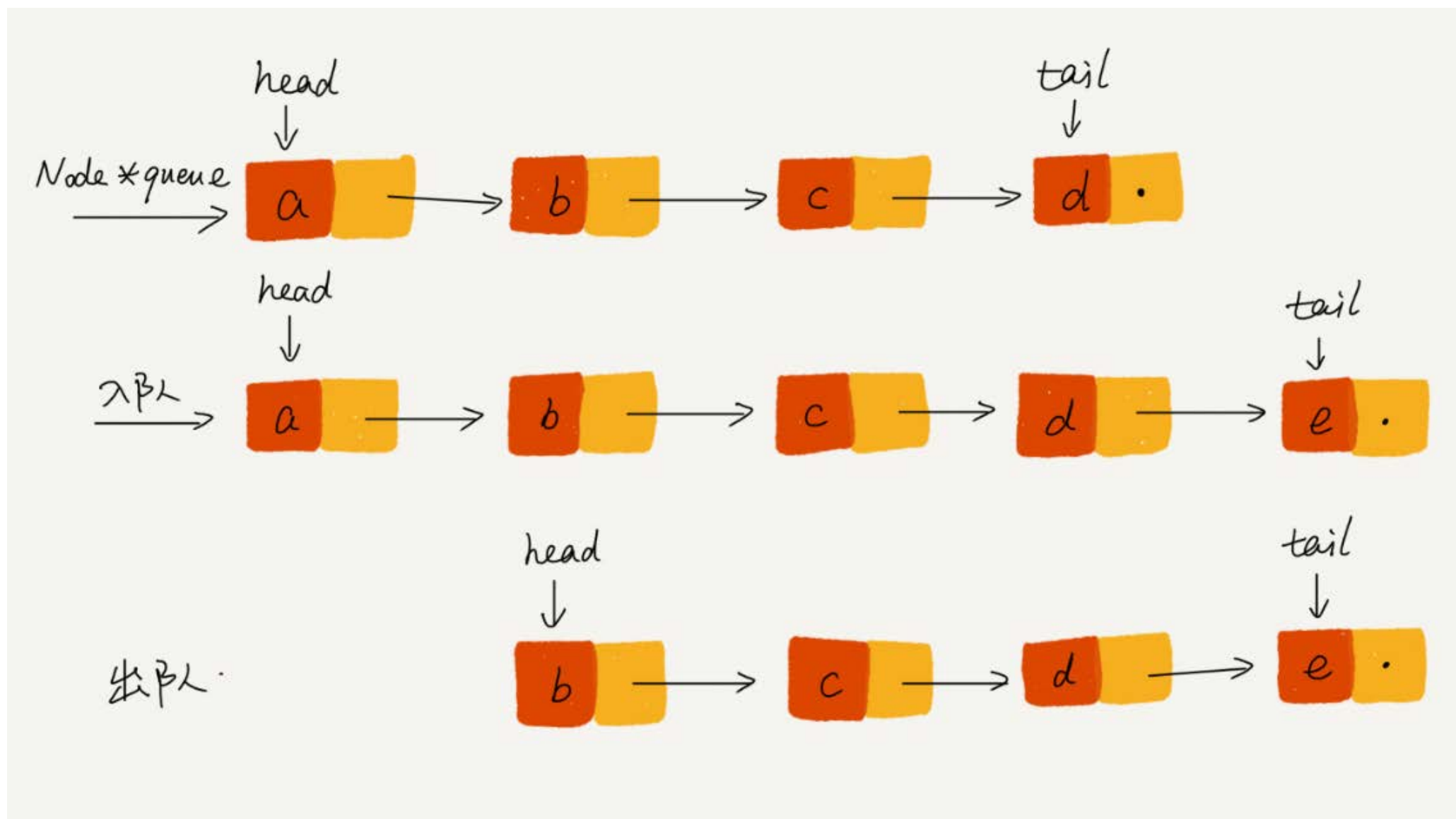
从代码中我们看到，当队列的tail指针移动到数组的最右边后，如果有新的数据入队，我们可以将head到tail之间的数据，整体搬移到数组中0到tail-head的位置。



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。

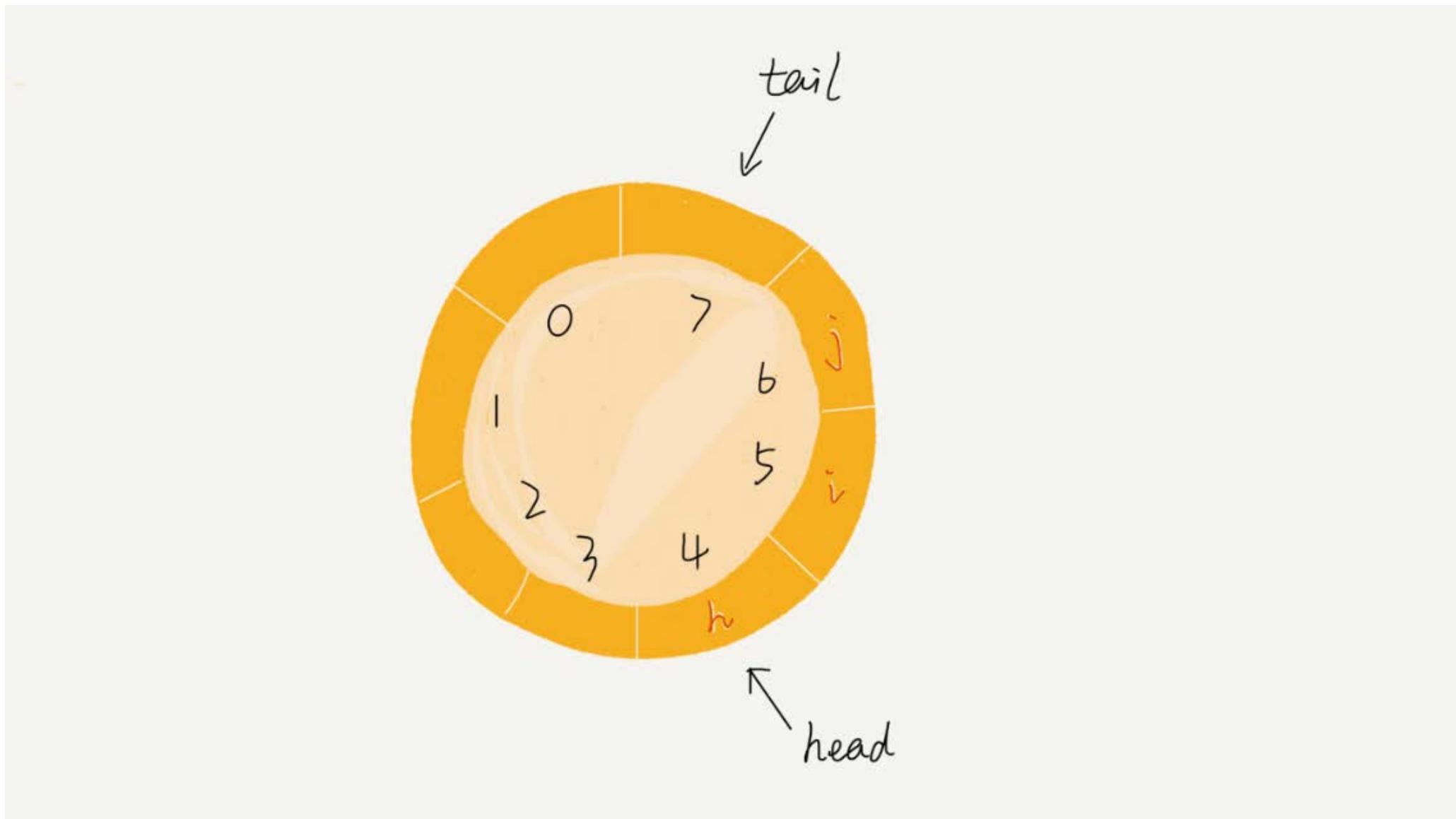
基于链表的实现，我们同样需要两个指针：head指针和tail指针。它们分别指向链表的第一个结点和最后一个结点。如图所示，入队时，tail->next= new_node, tail = tail->next；出队时，head = head->next。我将具体的代码放到GitHub上，你可以自己试着实现一下，然后再去GitHub上跟我实现的代码对比下，看写得对不对。



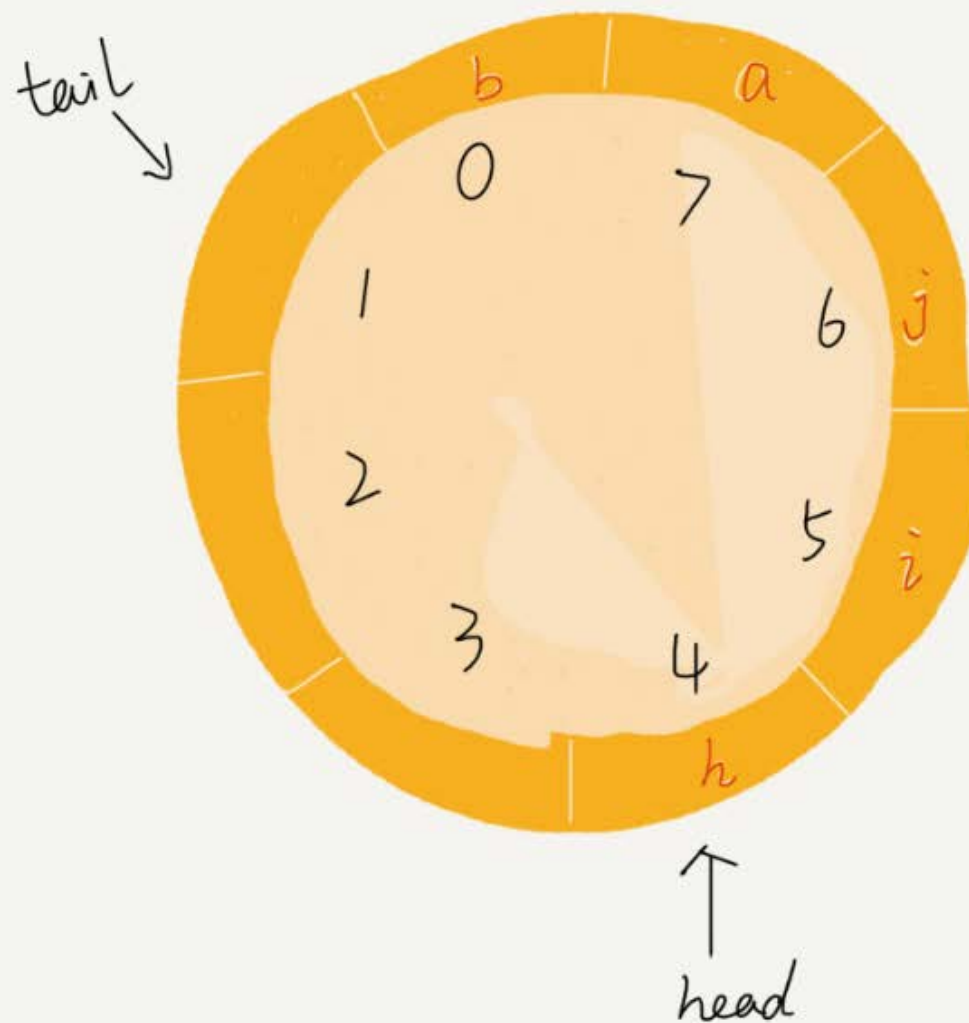
循环队列

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。



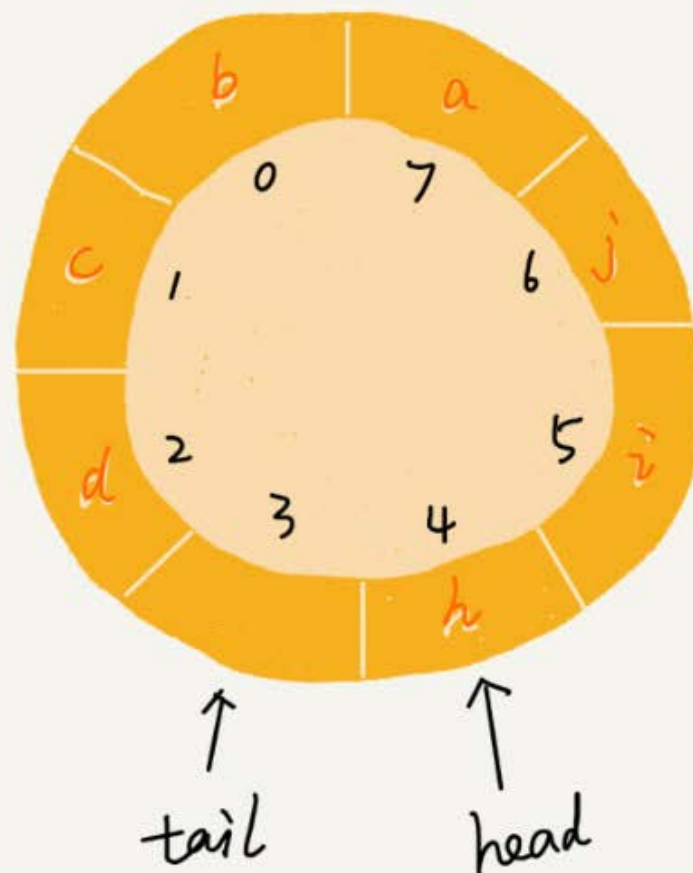
我们可以看到，图中这个队列的大小为8，当前`head=4`，`tail=7`。当有一个新的元素`a`入队时，我们放入下标为7的位置。但这个时候，我们并不把`tail`更新为8，而是将其在环中后移一位，到下标为0的位置。当再有一个元素`b`入队时，我们将`b`放入下标为0的位置，然后`tail`加1更新为1。所以，在`a`，`b`依次入队之后，循环队列中的元素就变成了下面的样子：



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度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，队满的判断条件是 $\text{tail} == n$ ，队空的判断条件是 $\text{head} == \text{tail}$ 。那针对循环队列，如何判断队空和队满呢？

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



循环队列，队尾下标可能在队头前面，所以需要取余

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。

```
public class CircularQueue {
    // 数组：items，数组大小：n
    private String[] items;
    private int n = 0;
```

```
// head表示队头下标, tail表示队尾下标
private int head = 0;
private int tail = 0;

// 申请一个大小为capacity的数组
public CircularQueue(int capacity) {
    items = new String[capacity];
    n = capacity;
}

// 入队
public boolean enqueue(String item) {
    // 队列满了
    if ((tail + 1) % n == head) return false;
    items[tail] = item;
    tail = (tail + 1) % n;
    return true;
}

// 出队
public String dequeue() {
    // 如果head == tail 表示队列为空
    if (head == tail) return null;
    String ret = items[head];
    head = (head + 1) % n;
    return ret;
}
}
```

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

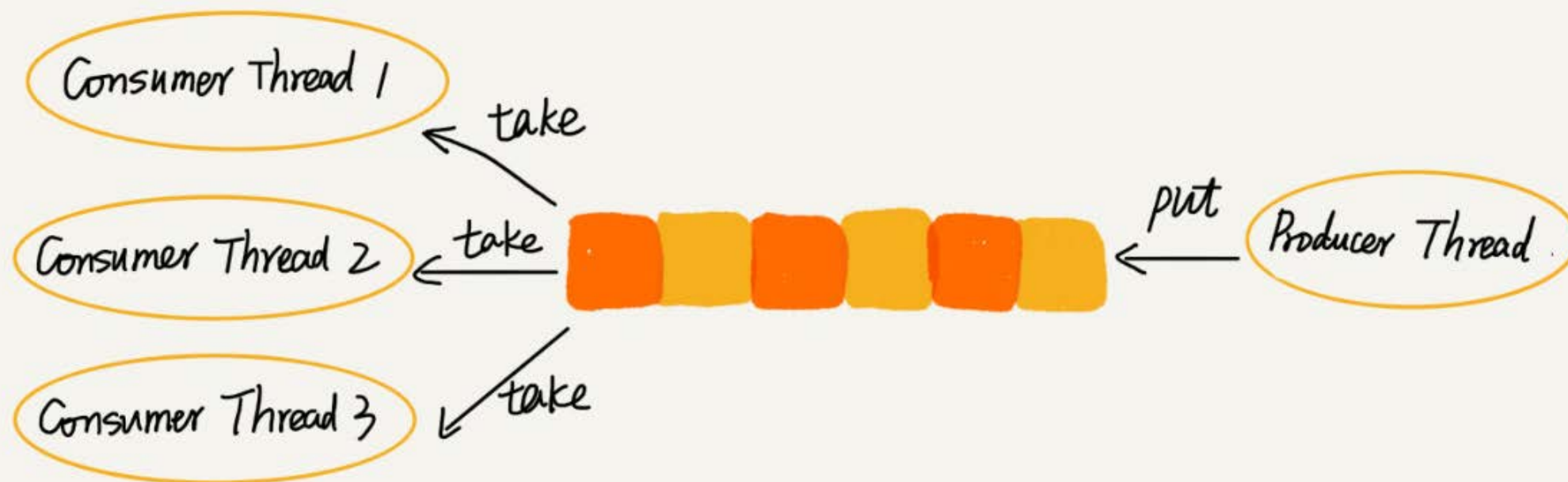


你应该已经发现了，上述的定义就是一个“生产者-消费者模型”！是的，我们可以使用阻塞队列，轻松实现一个“生产者-消费者模型”！

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。

我已将本节内容相关的详细代码更新到GitHub，[戳此](#)即可查看。



数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言：

- 城 2018-10-10 00:46:32
 - 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 [63赞]

作者回复2018-10-10 01:34:47

- wean 2018-10-10 11:58:30

队列也是一种“操作受限”的线性表，只支持两种基本操作：入队和出队。

队列的应用非常广泛，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。它们在很多偏底层的系统、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。

关于如何实现无锁并发队列

可以使用 **cas + 数组** 的方式实现。

队列的其他应用

分布式消息队列，如 **kafka** 也是一种队列。 [12赞]

作者回复2018-10-11 15:02:39

- 花见笑 2018-10-10 01:05:54

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。 [8赞]

作者回复2018-10-10 02:04:31

- 只会安卓De小鹿 2018-10-11 13:35:38

王争老师，为了更好的区分队列和栈，小鹿给大家一个更好的口诀。

“吃多了拉就是队列，吃多了吐就是栈”。哈哈！ [158赞]

作者回复2018-10-12 02:18:38

- 计科一班 2018-10-10 06:19:53

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++** [65赞]

09|队列：队列在线程池等有限资源池中的应用

作者回复2018-10-11 01:52:16

你这个思路挺巧妙的 我暂时还没有想到破绽

- Peter、桥 2018-10-10 04:36:48

老师要是有时间对课后问题集中式做下解答就好了 [21赞]

作者回复2018-10-11 01:52:33

行的

- 姜威 2018-10-12 00:51:09

队列实现

一、数组实现

```
public class ArrayQueue {
```

```
//存储数据的数组
```

```
private String[] items;
```

```
//记录数组容量
```

```
private int n;
```

```
private int size;
```

```
//head记录队头索引，tail记录队尾索引
```

```
private int head = 0;
```

```
private int tail = 0;
```

```
//申请一个指定容量的队列
```

```
public ArrayQueue(int capacity){
```

```
items = new String[capacity];
```

```
n = capacity;
```

```
}
```

```
/*
```

```
* 入队：
```

```
* 1.堆满的时，入队失败
```

```
* 1.1频繁出入队，造成数组使用不连续
```

```
* 1.2在入队的时候，集中触发进行数据搬移
```

```
* 2.在末尾插入数据，注意tail指向队尾元素的索引+1
```

```
*/
```

```
public boolean enqueue(String item){
```

```
//
```

表示队满

```
if(head == 0 && tail == n)
```

```
return false;
```

//表示需要数据搬移

```
else if(head != 0 && tail == n){
```

```
for (int i = head; i < tail; i++) {
```

```
items[i-head] = items[i];
```

```
}
```

```
head = 0;
```

```
tail = tail - head;
```

```
}
```

//将数据加入队列

```
items[tail++] = item;
```

```
size++;
```

```
return true;
```

```
}
```

//出队：1.队空时，出队失败;2.出队，head索引+1

```
public String dequeue(){
```

```
String res = null;
```

```
if(head == tail) return res;
```

```
res = items[head++];
```

```
size--;
```

```
return res;
```

```
}
```

```
}
```

二、循环队列

```
public class LoopArrayQueue {
```

//存储数据的数组

```
private String[] items;
```

//记录数组容量

```
private int n;
```

```
private int size = 0;
```

//head记录队头索引，tail记录队尾索引

```
private int head = 0;
private int tail = 0;
//申请一个指定容量的队列
public LoopArrayQueue(int capacity){
    items = new String[capacity];
    n = capacity;
}
//入队：关键在于队满的条件
public boolean enqueue(String item){
    if ((tail + 1) % n == head) return false;
    items[tail] = item;
    tail = (tail + 1) % n;
    size++;
    return true;
}
//出队：关键在于队空的条件
public String dequeue(){
    String res = null;
    if(head == tail) return res;
    res = items[head];
    head = (head + 1) % n;
    size--;
    return res;
}
}
```

三、链表实现

```
public class LinkedQueue {
    //定义一个节点类
    private class Node{
        String value;
        Node next;
    }
    //记录队列元素个数
```

```
private int size = 0;
//head指向队头结点， tail指向队尾节点
private Node head;
private Node tail;
//申请一个队列
public LinkedList(){ }
//入队
public boolean enqueue(String item){
Node newNode = new Node();
newNode.value = item;
if (size == 0) head = newNode;
else tail.next = newNode;
tail = newNode;
size++;
return true;
}
//出队
public String dequeue(){
String res = null;
if(size == 0) return res;
if(size == 1) tail = null;
res = head.value;
head = head.next;
size--;
return res;
}
} [18赞]
```

作者回复2018-10-12 02:09:43

- 姜威 2018-10-12 00:49:35

总结

一、什么是队列？

- 1.先进者先出，这就是典型的“队列”结构。
- 2.支持两个操作：入队`enqueue()`，放一个数据到队尾；出队`dequeue()`，从队头取一个元素。
- 3.所以，和栈一样，队列也是一种操作受限的线性表。

二、如何实现队列？

1.队列API

```
public interface Queue<T> {  
    public void enqueue(T item); //入队  
    public T dequeue(); //出队  
    public int size(); //统计元素数量  
    public boolean isEmpty(); //是否为空  
}
```

- 2.数组实现（顺序队列）：见下一条留言
- 3.链表实现（链式队列）：见下一条留言
- 4.循环队列（基于数组）：见下一条留言

三、队列有哪些常见的应用？

1.阻塞队列

- 1) 在队列的基础上增加阻塞操作，就成了阻塞队列。
- 2) 阻塞队列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，然后在返回。
- 3) 从上面的定义可以看出这就是一个“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，这时生产者就阻塞等待，直到“消费者”消费了数据，“生产者”才会被唤醒继续生产。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，来提高数据处理效率，比如配置几个消费者，来应对一个生产者。

2.并发队列

- 1) 在多线程的情况下，会有多个线程同时操作队列，这时就会存在线程安全问题。能够有效解决线程安全问题的队列就称为并发队列。
- 2) 并发队列简单的实现就是在`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取操作。
- 3) 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。

3.线程池资源枯竭是的处理

在资源有限的场景，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。

四、思考

- 1.除了线程池这种池结构会用到队列排队请求，还有哪些类似线程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，关于如何实现无锁的并发队列，网上有很多讨论。对这个问题，你怎么看？[16赞]

作者回复2018-10-12 02:09:23

- 蝴蝶 2018-10-09 17:40:48

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 $O(1)$ 吗想了一下，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？ [15赞]

- 老司机 2018-10-10 01:03:58

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 [11赞]