

Vector vs ArrayList  
前者是线程安全的

Stack & Queue 关键点

Stack: 先入后出; 添加、删除皆为  $O(1)$

Queue: 先入先出; 添加、删除皆为  $O(1)$

Stack & Queue & Deque 查询皆为  $O(n)$ , 因为无序 所以查找需要遍历

Deque: Double-End Queue  
队列前后都可以插入删除



内部保持了数据的有序性

**Priority Queue**

- 1. 插入操作:  $O(1)$
- 2. 取出操作:  $O(\log N)$  - 按照元素的优先级取出
- 3. 底层具体实现的数据结构较为多样和复杂: heap, bst, treap

Java 的 PriorityQueue  
<https://docs.oracle.com/javase/10/docs/api/java/util/PriorityQueue.html>

Stack & Queue 工程实现

使用数组模拟, 在数组上加一些 API

示例代码 - Stack

```
Stack<Integer> stack = new Stack<>();
stack.push(1);
stack.push(2);
stack.push(3);
stack.push(4);
System.out.println(stack);
System.out.println(stack.search(4));

stack.pop();
stack.pop();
Integer topElement = stack.peek();
System.out.println(topElement);
System.out.println("3的位置" + stack.search(3));
```

```
/**
 * Returns the position of an Object on the stack, with the top
 * most Object being at position 1, and each Object deeper in the
 * stack at depth + 1.
 *
 * @param o The Object to search for
 * @return The 1 based depth of the Object, or -1 if the Object
 *         is not on the stack
 */
public synchronized int search(Object o)
{
    int i = elementCount;
    while (--i >= 0)
        if (equals(o, elementData[i]))
            return elementCount - i;
    return -1;
}
```

示例代码 - Deque

```
Deque<String> deque = new LinkedList<String>();

deque.push("a");
deque.push("b");
deque.push("c");
System.out.println(deque);

String str = deque.peek();
System.out.println(str);
System.out.println(deque);

while (deque.size() > 0) {
    System.out.println(deque.pop());
}
System.out.println(deque);
```

示例代码 - Queue

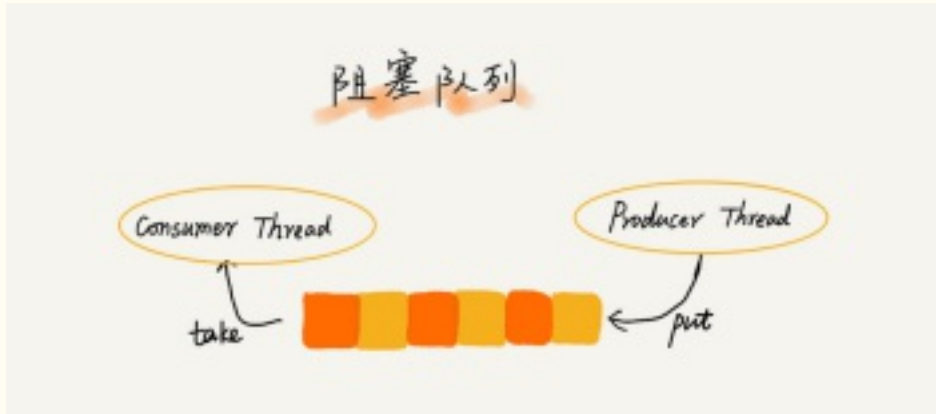
```
Queue<String> queue = new LinkedList<String>();
queue.offer("one");
queue.offer("two");
queue.offer("three");
queue.offer("four");
System.out.println(queue);

String polledElement = queue.poll();
System.out.println(polledElement);
System.out.println(queue);

String peekedElement = queue.peek();
System.out.println(peekedElement);
System.out.println(queue);

while (queue.size() > 0) {
    System.out.println(queue.poll());
}
```

Common Data Structure Operations									
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$



阻塞队列  
e.g 生产者 - 消费者模型

并发队列  
线程安全的队列

Queue 实现

对于大部分资源有限的场景, 当没有空闲资源时, 可以通过使用队列来实现请求排队

拒绝请求  
线程池已满  
阻塞并将请求排队

有界队列, 响应灵敏, 但是要合理利用资源设置好队列大小

无界队列, 但是可能会导致过多的请求排队等待, 响应时间长

队列需要两个指针  
head 和 tail

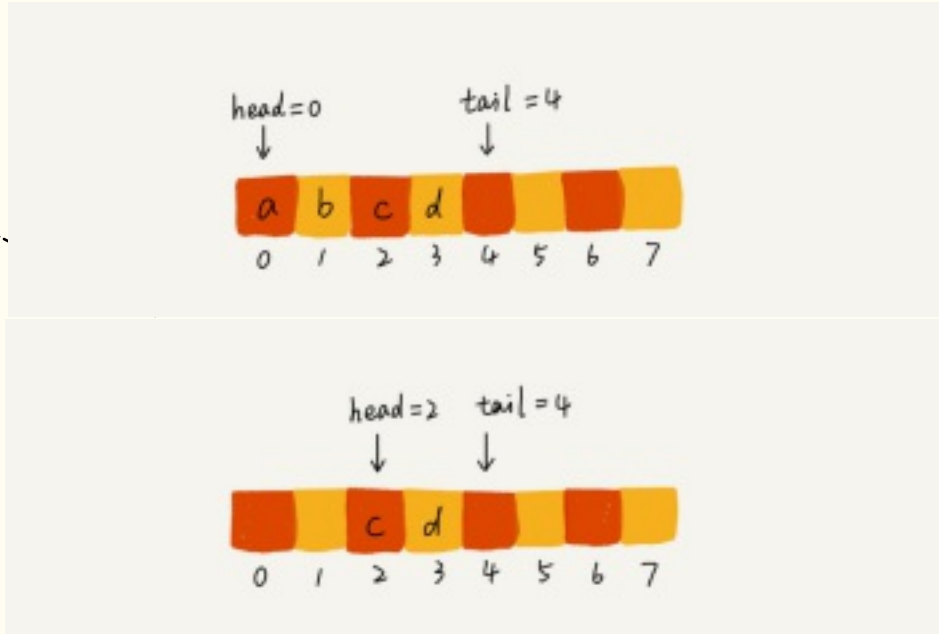
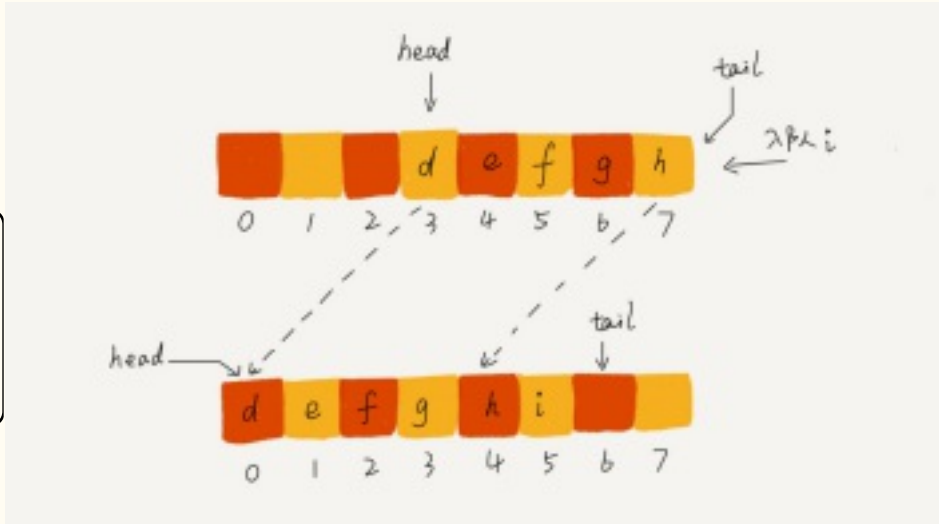
入队时检查是否有空闲空间, 有则做数据搬移, 再入队  $O(n+1)$

解决数据搬移 -> 循环队列

确定好队空和队满的判定条件

队满判断  $tail ==$   
队空判断  $head == tail$

队空  $tail == head$   
队满  $(tail + 1) \% n = head$



Stack

最近相关性 -> 用栈来解决

现实情况用栈这种逻辑来解决, 就能用栈来解决

工程里面有很多 从外向内、从内向外的扩散

Queue

先来到, 公平性

只用队列来做栈 或 只用栈来做队列  
那么你需要两个队列 或者 两个栈

栈实现队列, 需要 A B 两个栈, 出栈时 将 A 的元素全部转移至 B 栈, 然后弹出 B 栈即可