1. Deque基础概念
2. 定义

Deque的含义是“double ended queue”，即双端队列，**既可以当作栈使用，也可以当作队列使用。**一般而言，由于需要移动元素，数组的插入和删除效率比较低，但ArrayDeque的效率比ArrayList高。

1. Queue接口和Deque接口方法对比

|  |  |  |
| --- | --- | --- |
| Queue Method | Equivalent Deque Method | 说明 |
| add(e) | addLast(e) | 向队尾插入元素，失败则抛出异常 |
| offer(e) | offerLast(e) | 向队尾插入元素，失败则返回false |
| remove() | removeFirst() | 获取并删除队首元素，失败则抛出异常 |
| poll() | pollFirst() | 获取并删除队首元素，失败则返回null |
| element() | getFirst() | 获取但不删除队首元素，失败则抛出异常 |
| peek() | peekFirst() | 获取但不删除队首元素，失败则返回null |

1. Stack和Deque接口方法对比

|  |  |  |
| --- | --- | --- |
| Stack Method | Equivalent Deque Method | 说明 |
| push(e) | addFirst(e) | 向栈顶插入元素，失败则抛出异常 |
| 无 | offerFirst(e) | 向栈顶插入元素，失败则返回false |
| pop() | removeFirst() | 获取并删除栈顶元素，失败则抛出异常 |
| 无 | pollFirst() | 获取并删除栈顶元素，失败则返回null |
| peek() | peekFirst() | 获取但不删除栈顶元素，失败则抛出异常 |
| 无 | peekFirst() | 获取但不删除栈顶元素，失败则返回null |

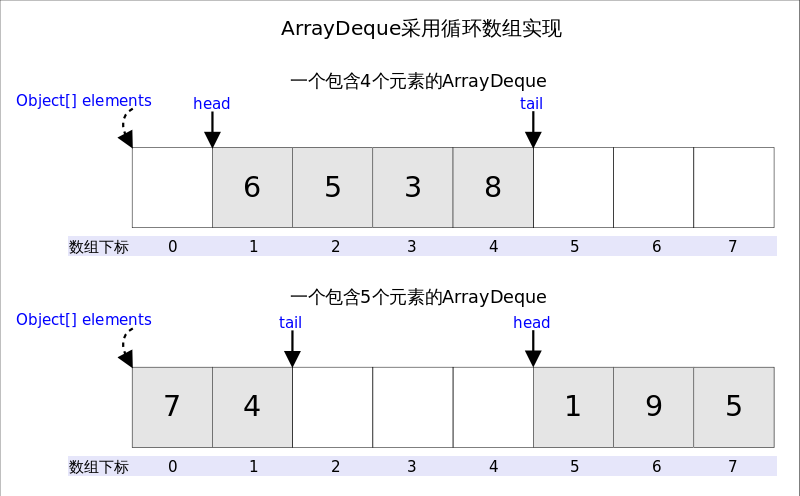
1. 底层实现

4.1 基本介绍

(1) **ArrayDeque底层通过数组实现**，为了满足可以同时在数组两端插入或删除元素的需求，该数组还必须是循环的，即**循环数组**（circular array），也就是说数组的任何一点都可能被看作起点或者终点。

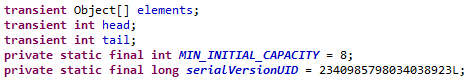
(2) ArrayDeque是非线程安全的（not thread-safe），当多个线程同时使用的时候，需要程序员手动同步。容器不允许放入null元素。

4.2．原理图



1. 构造原理

5.1 类型属性



1. elements是指底层实现的数组，head指向首端第一个有效元素，tail指向尾端第一个可以插入元素的空位。因为是循环数组，所以head不一定总等于0，tail也不一定总是比head大。
2. ArrayDeque的高效来源于head和tail这两个变量，它们使得物理上简单的从头到尾的数组变为了一个逻辑上循环的数组，避免了在头尾操作时的移动。

5.2 构造方法（关键点：数组的扩容）

(1) 无参构造方法（默认构造方法）

public ArrayDeque()

{ //默认创建一个长度为16的底层数组

elements = new Object[16];

}

1. 传参构造方法 int
2. public ArrayDeque(int paramInt)

{

allocateElements(paramInt);

}

1. private void allocateElements(int paramInt)

{

elements = new Object[calculateSize(paramInt)];

}

1. private static int calculateSize(int paramInt)// 扩容数组--比原本大且为2的幂次方的最小数

{

int i = 8;

if (paramInt >= i)

{ //见下面注解

i = paramInt;

i |= i >>> 1;

i |= i >>> 2;

i |= i >>> 4;

i |= i >>> 8;

i |= i >>> 16;

i++;

if (i < 0) {

i >>>= 1;

}

}

return i;

}

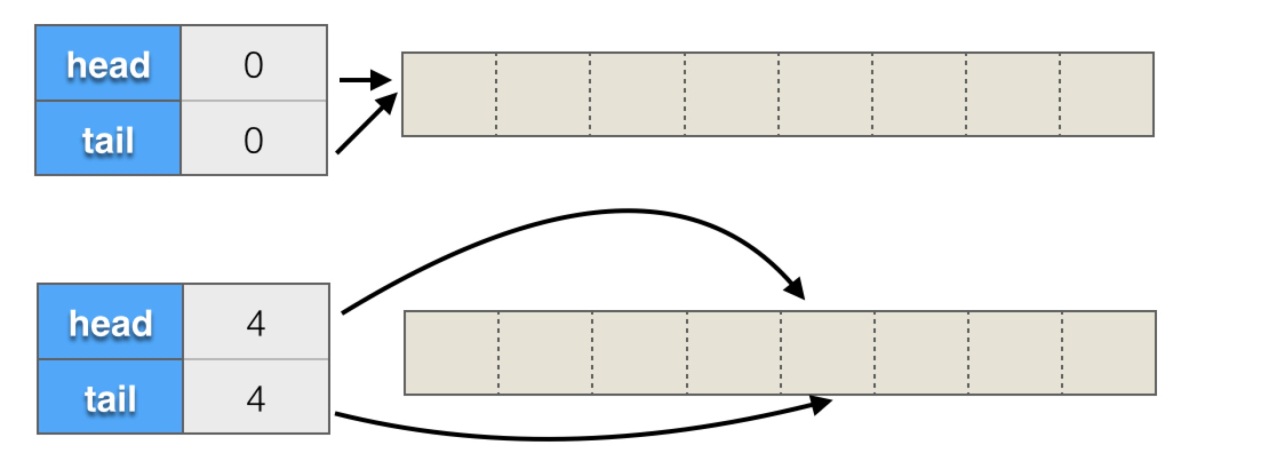
//在将i左边最高位的1复制到右边的每一位，这种复制类似于病毒复制，是1传2、2传4、4传8式的指数级复制，最后再执行i++就可以得到比i大且为2的幂次方的最小的数。

5.3 循环数组的实现

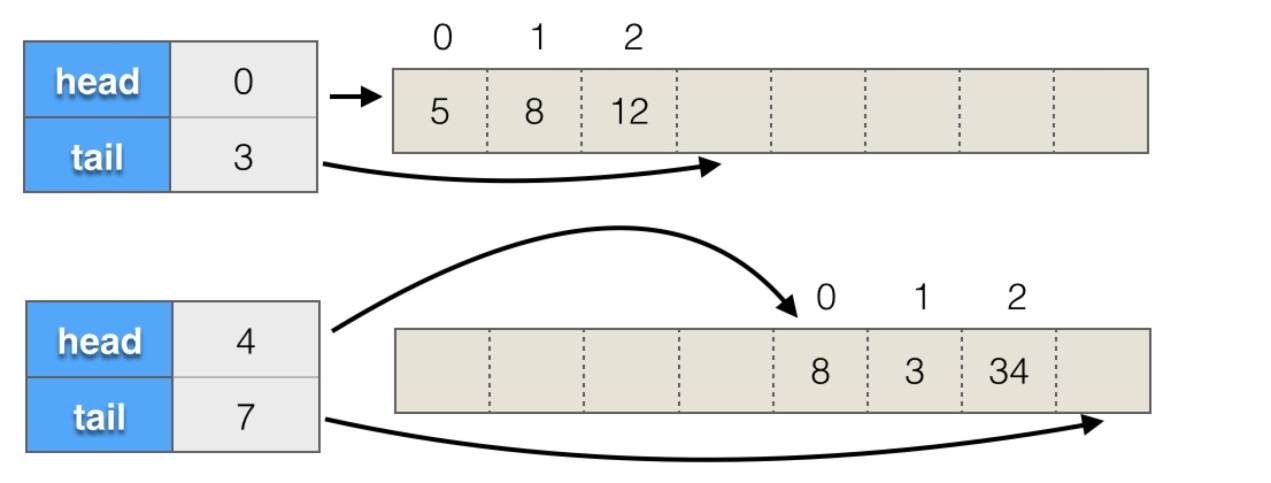
5.3.1 循环数组原理

(1) 所谓循环是指元素到数组尾之后可以接着从数组头开始，数组的长度、第一个和最后一个元素都与head和tail这两个变量有关。head指向数组开头，tail指向最后一个元素的下一个null。

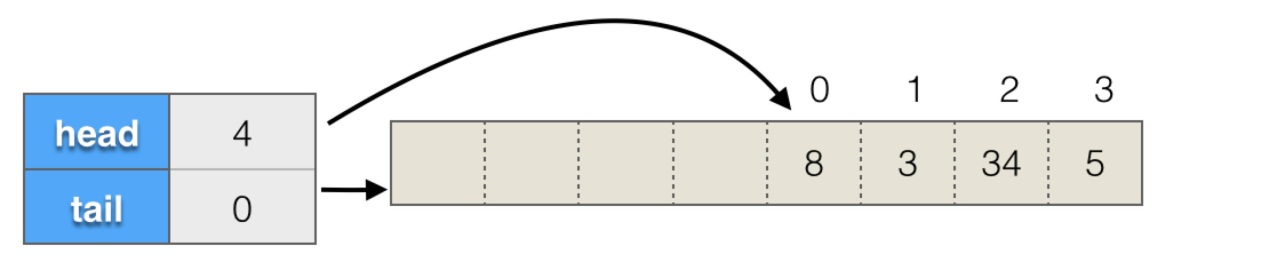
a. 如果head和tail相同，则数组为空，长度为0。



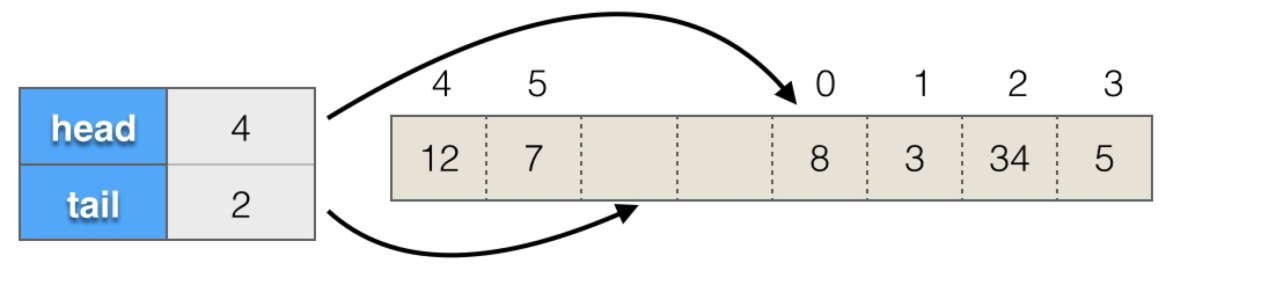
b. 如果tail大于head，则第一个元素为elements[head]，最后一个为elements[tail-1]，长度为tail-head，元素索引从head到tail-1。



c. 如果tail小于head，且为0，则第一个元素为elements[head]，最后一个为elements[elements.length-1]，元素索引从head到elements.length-1。



1. 如果tail小于head，且大于0，则会形成循环，第一个元素为elements[head]，最后一个是elements[tail-1]，元素索引从head到elements.length-1，然后再从0到tail-1。



5.3.2循环数组的长度

(1) 默认长度

private static final int MIN\_INITIAL\_CAPACITY = 8;

(2) 扩充长度

a. 如果numElements小于MIN\_INITIAL\_CAPACITY，则分配的数组长度就是MIN\_INITIAL\_CAPACITY，它是一个静态常量，值为8。

b. 在numElements大于等于8的情况下，分配的实际长度是**严格大于numElements并且为2的整数次幂的最小数**。比如，如果numElements为10，则实际分配16，如果numElements为32，则为64。

c. 因为循环数组必须时刻至少留一个空位，tail变量指向下一个空位，为了容纳numElements个元素，至少需要numElements+1个位置。

1. 具体操作方法
2. ArrayDeque当做队列使用（Queue）
3. public boolean offer(E paramE) -- 核心方法：addLast()

在此队列的末尾插入指定的元素。

1. public E peek()

检索但不删除由此deque表示的队列的头部，如果此deque为空，则返回 null。

1. public E poll() -- 核心方法：pollFirst()

检索并删除由此deque（换句话说，该deque的第一个元素）表示的队列的 null如果此deque为空，则返回 null 。

1. ArrayDeque当做栈使用
2. public void push(E paramE) -- 核心方法：addFirst()

将元素推送到由此deque表示的堆栈上，相当于在头部添加一个元素。

1. public E peek()

检索但不删除由此deque表示的头部元素，如果此deque为空，则返回 null。

1. public E pop() -- 核心方法：removeFirst()调用pollFirst()

从这个deque表示的堆栈中弹出一个元素，从头部删除一个元素。

1. 添加元素add()
2. public boolean add(E paramE)

{

addLast(paramE); //在末尾添加元素，调用addLast()方法

return true;

}

1. public void addFirst(E paramE)

{ //当添加元素为null时，抛出异常

if (paramE == null) {

throw new NullPointerException();

}//将head往左移动一位，当head为0时，取数组最后一位。

elements[(head = head - 1 & elements.length - 1)] = paramE;

if (head == tail) {

doubleCapacity(); //将数组大小扩充为两倍，并将数组从头开始排版。

}

}

private void doubleCapacity() {

assert head == tail;

int p = head;

int n = elements.length;

int r = n - p; // number of elements to the right of p

int newCapacity = n << 1;

if (newCapacity < 0)

throw new IllegalStateException("Sorry, deque too big");

Object[] a = new Object[newCapacity];

System.arraycopy(elements, p, a, 0, r);

System.arraycopy(elements, 0, a, r, p);//从新拷贝数组

elements = a;

head = 0;//将head设为0

tail = n;

}

1. public void addLast(E e) {

if (e == null)

throw new NullPointerException();

elements[tail] = e; //直接在tail处添加元素，并将tail增加1

if ( (tail = (tail + 1) & (elements.length - 1)) == head)

doubleCapacity();

}

1. 删除元素
2. remove() -- 删除第一个元素removeFirst()

public E remove()

1. removeFirst() -- 核心方法pollFirst()

public E removeFirst() {

E x = pollFirst();

if (x == null)

throw new NoSuchElementException();

return x;

}//当删除的元素为空则报错

1. removeLast() -- 核心方法pollLast()

public E removeLast() {

E x = pollLast();

if (x == null)

throw new NoSuchElementException();

return x;

}//当删除的元素为空则报错

1. pollFirst() -- 删除第一个元素

public E pollFirst() {

int h = head;

@SuppressWarnings("unchecked")

E result = (E) elements[h];

// Element is null if deque empty

if (result == null)

return null;

elements[h] = null; // Must null out slot

head = (h + 1) & (elements.length - 1); //处理临界情况（当h为elements.length - 1

时），与后的结果为0。

return result;

}

★elements.length始终为2的幂次方，所以始终是1111取模。当head为element.length-1时，head和element.length-1的按位与等于0，相当于从最后一个元素跳到第一个元素。

1. pollLast() -- 删除最后一个元素

public E pollLast() {

int t = (tail - 1) & (elements.length - 1);

@SuppressWarnings("unchecked")

E result = (E) elements[t];

if (result == null)

return null;

elements[t] = null;

tail = t;

return result;

}

1. 判断元素是否存在
2. contain()

public boolean contains(Object o) {

if (o == null)

return false; //当元素为空时，返回false

int mask = elements.length - 1; //标记最后一个元素

int i = head;

Object x;

while ( (x = elements[i]) != null) {

if (o.equals(x))

return true;

i = (i + 1) & mask; //往后跳转一位

}

return false;

}

1. 转换成数组
2. toArray()

public Object[] toArray() {

return copyElements(new Object[size()]);

} //创建一个size大小的数组

private <T> T[] copyElements(T[] a) {

if (head < tail) {

System.arraycopy(elements, head, a, 0, size());

} else if (head > tail) {

int headPortionLen = elements.length - head;

System.arraycopy(elements, head, a, 0, headPortionLen);

System.arraycopy(elements, 0, a, headPortionLen, tail);

}

return a;

}

1. 总结
2. 特点
3. 在两端添加、删除元素的效率很高，动态扩展需要的内存分配以及数组拷贝开销可以被平摊，具体来说，添加N个元素的效率为O(N)。
4. 根据元素内容查找和删除的效率比较低，为O(N)。
5. 与ArrayList和LinkedList不同，没有索引位置的概念，不能根据索引位置进行操作。
6. ArrayDeque实现了双端队列接口，可以作为队列、栈、或双端队列使用，相比LinkedList效率要更高一些，实现原理上，它采用动态扩展的循环数组，使用高效率的位操作。