**JAVA框架集合之ArrayDeque深度解析（一）**

**摘要：**

队列是只能对头尾两个元素操作的数据结构，单向队列只能从头remove（poll），从尾add（offer），双端队列则头尾均可执行插入和删除操作。  
Java里有一个叫做Stack的类，却没有叫做Queue的类（它是个接口名字）。当需要使用栈时，Java已不推荐使用Stack，而是推荐使用更高效的ArrayDeque；既然Queue只是一个接口，当需要使用队列时也就首选ArrayDeque了（次选是LinkedList）。

**Queue**

1.ArrayDeque, （数组双端队列）  
2.PriorityQueue, （优先级队列）  
3.ConcurrentLinkedQueue, （基于链表的并发队列）  
4.DelayQueue, （延期阻塞队列）（阻塞队列实现BlockingQueue口）  
5.ArrayBlockingQueue, （基于数组的并发阻塞队列）  
6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）  
7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）  
8.PriorityBlockingQueue, （带优先级的无界阻塞队列）  
9.SynchronousQueue （并发同步阻塞队列）

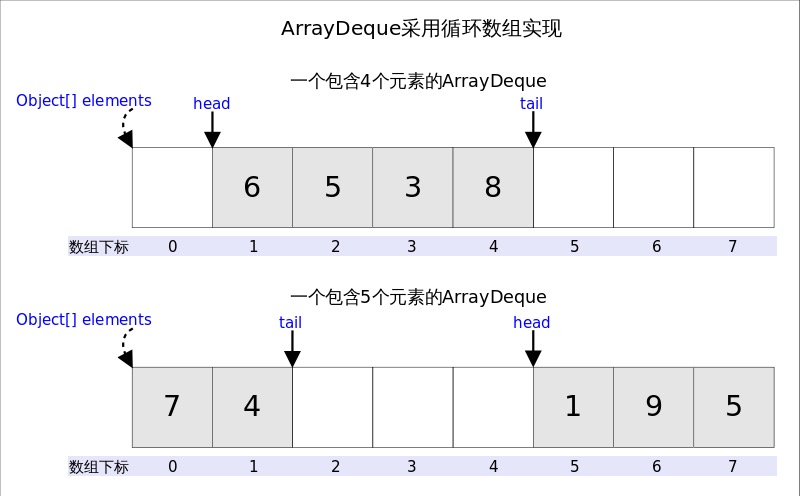
**接口介绍**

要讲栈和队列，首先要讲Deque接口。Deque的含义是“double ended queue”，即双端队列，它既可以当作栈使用，也可以当作队列使用。下表列出了Deque与Queue相对应的接口：  
下表列出了Deque与Stack对应的接口：

上面两个表共定义了Deque的12个接口。添加，删除，取值都有两套接口，它们功能相同，区别是对失败情况的处理不同。一套接口遇到失败就会抛出异常，另一套遇到失败会返回特殊值（false或null）。除非某种实现对容量有限制，大多数情况下，添加操作是不会失败的。虽然Deque的接口有12个之多，但无非就是对容器的两端进行操作，或添加，或删除，或查看。明白了这一点讲解起来就会非常简单。

ArrayDeque和LinkedList是Deque的两个通用实现，由于官方更推荐使用AarryDeque用作栈和队列，加之上一篇已经讲解过LinkedList，本文将着重讲解ArrayDeque的具体实现。

从名字可以看出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。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；另外，该容器不允许放入null元素。

上图中我们看到，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。

**addFirst(E e)**

*/\*\**

*\* Inserts the specified element at the front of this deque.*

*\**

*\* @param e the element to add*

*\* @throws NullPointerException if the specified element is null*

*\*/*

public void addFirst(E e) {

if (e == null)

throw new NullPointerException();

elements[head = (head - 1) & (elements.length - 1)] = e;*//校验越界*

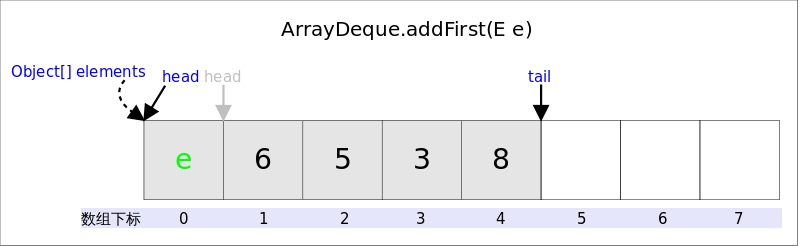
if (head == tail)*//空间是否够用*

doubleCapacity();*//扩容*

}

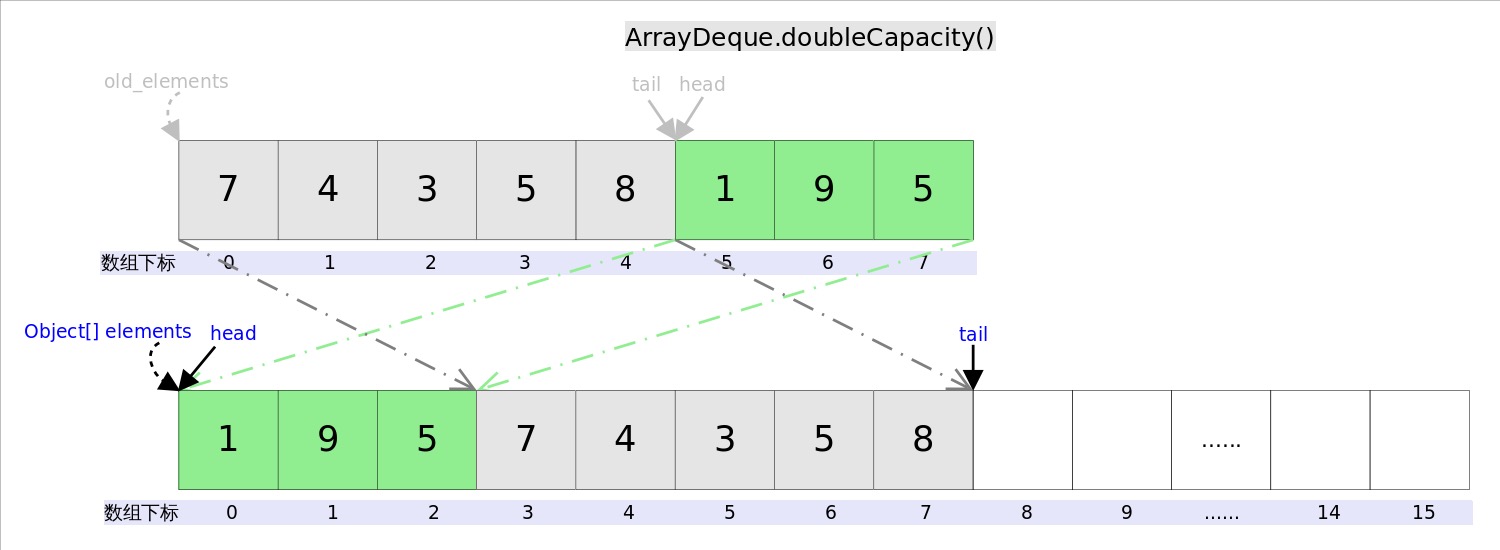
实际需要考虑：1.空间是否够用，以及2.下标是否越界的问题。上图中，如果head为0之后接着调用addFirst()，虽然空余空间还够用，但head为-1，下标越界了。  
上述代码我们看到，空间问题是在插入之后解决的，因为tail总是指向下一个可插入的空位，也就意味着elements数组至少有一个空位，所以插入元素的时候不用考虑空间问题。

下标越界的处理解决起来非常简单，head = (head - 1) & (elements.length - 1)就可以了，这段代码相当于取余，同时解决了head为负值的情况。因为elements.length必需是2的指数倍，elements.length - 1就是二进制低位全1，跟head - 1相与之后就起到了取模的作用，如果head - 1为负数（其实只可能是-1），则相当于对其取相对于elements.length的补码。



**doubleCapacity()**

下面再说说扩容函数doubleCapacity()，其逻辑是申请一个更大的数组（原数组的两倍），然后将原数组复制过去。过程如下图所示：



*/\*\**

*\* Doubles the capacity of this deque. Call only when full, i.e.,*

*\* when head and tail have wrapped around to become equal.*

*\*/*

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;*//扩容2倍*

if (newCapacity < 0)

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

Object[] a = new Object[newCapacity];

System.arraycopy(elements, p, a, 0, r);*//复制右半部分，对应上图中绿色部分*

System.arraycopy(elements, 0, a, r, p);*//复制左半部分，对应上图中灰色部分*

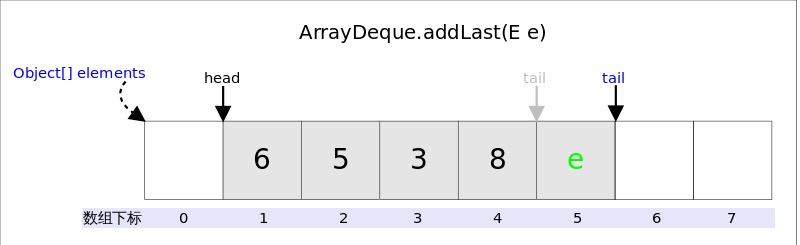
elements = a;

head = 0;

tail = n;

}

**addLast(E e)**

addLast(E e)的作用是在Deque的尾端插入元素，也就是在tail的位置插入元素，由于tail总是指向下一个可以插入的空位，因此只需要elements[tail] = e;即可。插入完成后再检查空间，如果空间已经用光，则调用doubleCapacity()进行扩容。  


*/\*\**

*\* Inserts the specified element at the end of this deque.*

*\**

*\* &lt;p>This method is equivalent to {@link #add}.*

*\**

*\* @param e the element to add*

*\* @throws NullPointerException if the specified element is null*

*\*/*

public void addLast(E e) {

if (e == null)

throw new NullPointerException();

elements[tail] = e;

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

doubleCapacity();

}

**pollFirst()**

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);

return result;

}

pollFirst()的作用是删除并返回Deque首端元素，也即是head位置处的元素。如果容器不空，只需要直接返回elements[head]即可，当然还需要处理下标的问题。由于ArrayDeque中不允许放入null，当elements[head] == null时，意味着容器为空。

**pollLast()**

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;

}

pollLast()的作用是删除并返回Deque尾端元素，也即是tail位置前面的那个元素。

**peekFirst()**

@SuppressWarnings("unchecked")

public E peekFirst() {

*// elements[head] is null if deque empty*

return (E) elements[head];

}

`peekFirst()`的作用是返回但不删除`Deque`首端元素，也即是`head`位置处的元素，直接返回`elements[head]`即可。

**peekLast()**

@SuppressWarnings("unchecked")

public E peekLast() {

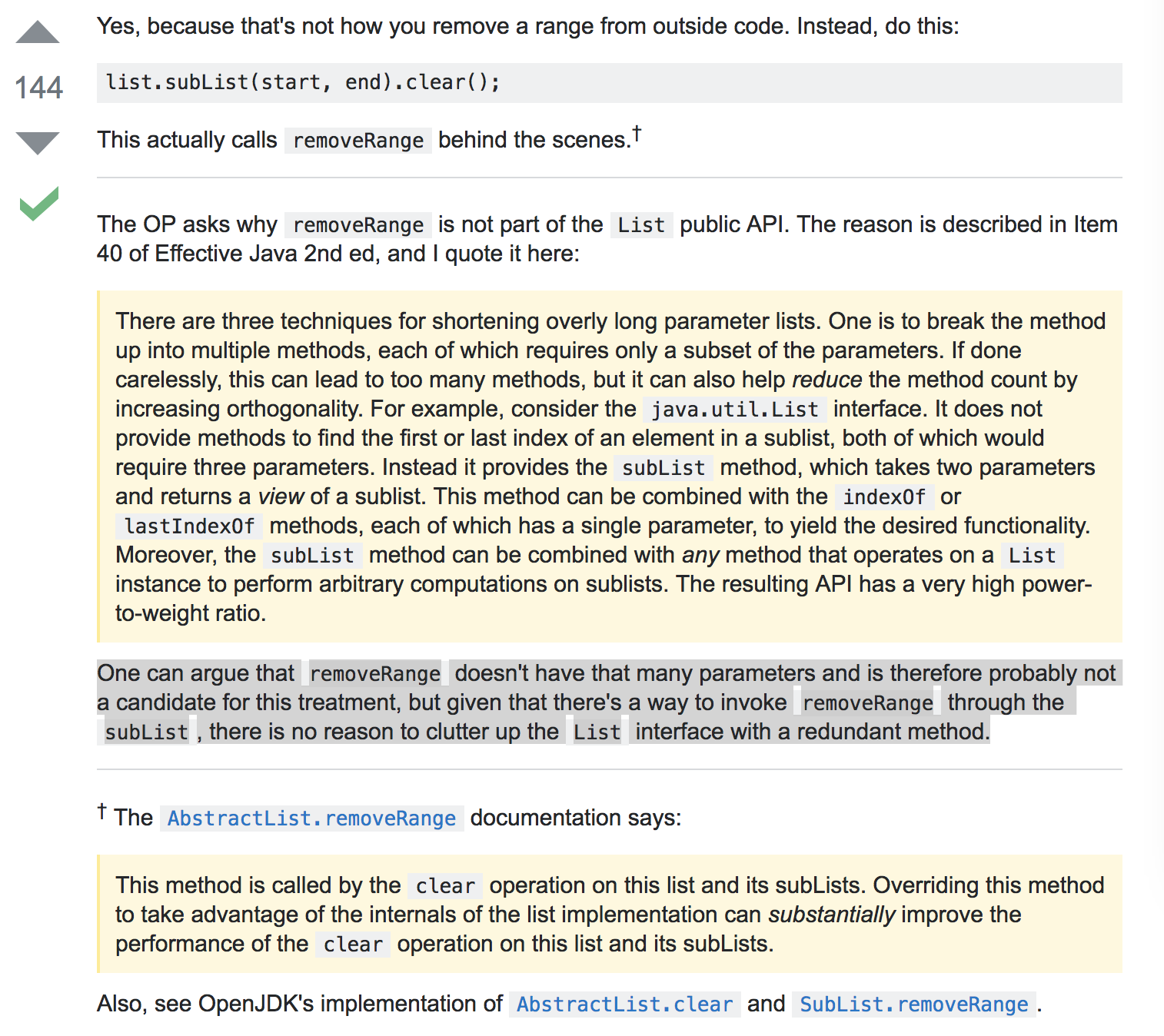
return (E) elements[(tail - 1) & (elements.length - 1)];

}

peekLast()的作用是返回但不删除Deque尾端元素，也即是tail位置前面的那个元素。

**总结：**

该篇文章讲解了部分ArrayDeque方法，结合了图解和部分源码，让大家更加清晰的认识了它的使用方法，在今后的日常工作中很可能需要它的，高并发的编程中用的集合不仅仅是常用的ArrayList,HashMap,HashSet`,这三个是日常工作用得做多的了吧，不过这个在java这个集合大家庭中是远远不够的，集合框架的其它兄弟扮演的角色也是特别重要的，后续文章会继续分析其它集合。

个人能力有限，只能找到网上这样的晦涩的答案

**译文：**  
有三种缩短过长参数列表的技巧。 一种是将方法分解成多种方法，每种方法只需要一部分参数。 如果不小心做到了，这会导致太多的方法，但它也可以通过增加正交性来帮助减少方法数量。 例如，考虑java.util.List接口。 它没有提供方法来查找子列表中元素的第一个或最后一个索引，这两个索引都需要三个参数。 相反，它提供了subList方法，它接受两个参数并返回一个子列表视图。 该方法可以与indexOf或lastIndexOf方法结合使用，每个方法都有一个参数，以产生所需的功能。 而且，subList方法可以与在List实例上运行的任何方法组合，以对子列表执行任意计算。 由此产生的API具有非常高的功率重量比。

[参考链接](https://stackoverflow.com/questions/2289183/why-is-javas-abstractlists-removerange-method-protected)

**总结：**

ArrayList部分的方法实在太博大精深了，时间和精力有限，只能分析到这个地方了，其他的方法不太常用的就没有提出来分析，可能遗漏的是移除所有removeAll(Collection<?> c)和批量移除batchRemove(Collection<?> c, boolean complement)这2个比较常用的方法吧，如果大家有兴趣的话可以自行去分析，其实分析集合框架对自己的能力提升是非常有帮助的，可以学习大神们的思维和写法，java中的集合框架毫无疑问也堪称是一门艺术！

**参考文献**

[Collections Framework Overview](https://docs.oracle.com/javase/6/docs/technotes/guides/collections/overview.html)  
[The For-Each Loop](https://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html)

**参考本系列文章**

[Java 集合框架之概览](http://www.spring4all.com/article/1042)  
[java集合框架之ArrayList深度解析（一）](http://www.spring4all.com/article/1045)  
[java集合框架之ArrayList深度解析（二）](http://www.spring4all.com/article/1050)  
[java集合框架之ArrayList深度解析（三）](http://www.spring4all.com/article/1055)  
[java集合框架之ArrayList深度解析（四）](http://www.spring4all.com/article/1056)