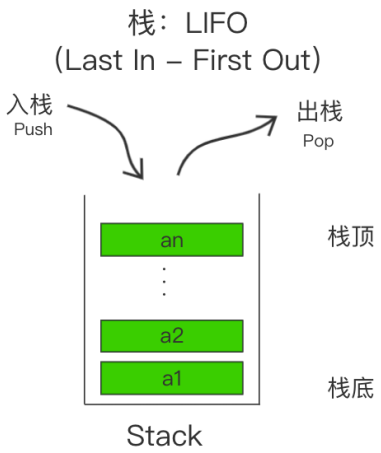
栈和队列的实现与特性

Stack（栈）和 Queue（队列）

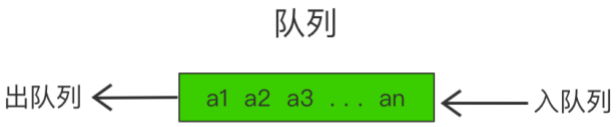
Stack（栈）

栈：后进先出（Last in – First out）



Queue（队列）

队列：先进先出 （First in – Fist out）



Stack（栈）& Queue（队列）关键点

- `Stack`：后入先出，增加、删除皆为 `O(1)`，查找为 `O(n)`
- `Queue`：先进先出，增加、删除皆为 `O(1)`，查找为 `O(n)`

源码解析（基于JDK12）

Java Stack源码解析

Stack是栈，它的特性是：先进后出（FILO，Fisrt In Last Out）。java工具包中的Stack是继承与Vector的，由于Vector是通过数组实现的，这意味着，Stack也是通过数字实现，而非链表。当然，也可以将LinkedList当作栈来使用后。

Stack的继承关系

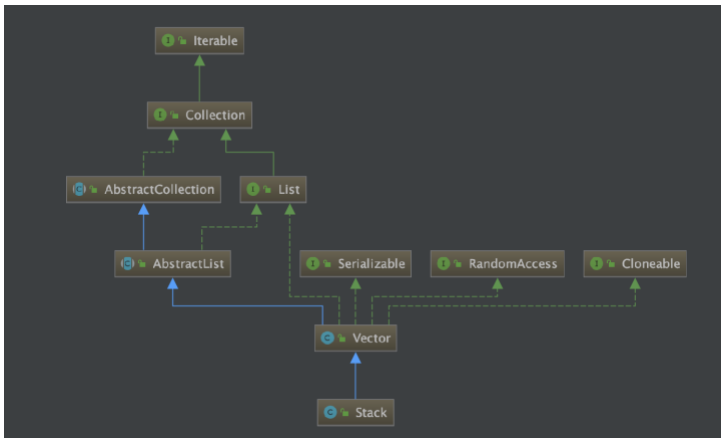
Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Stack 和 Coolection的关系如下：



Stack的源码

```
43
44          if (i >= 0) {
45              return size() - i;
46          }
47          return -1;
48      }
49
50      private static final long serialVersionUID = 1224463164541339165L;
51  }
```

总结：

1. Stack实际上也是通过数组去实现的。

   执行`push`时（即，将元素推入栈中），是通过将元素追加在数组的末尾。

   执行`peek`时（即，取出栈顶元素，不执行删除），是返回数组末尾的元素。

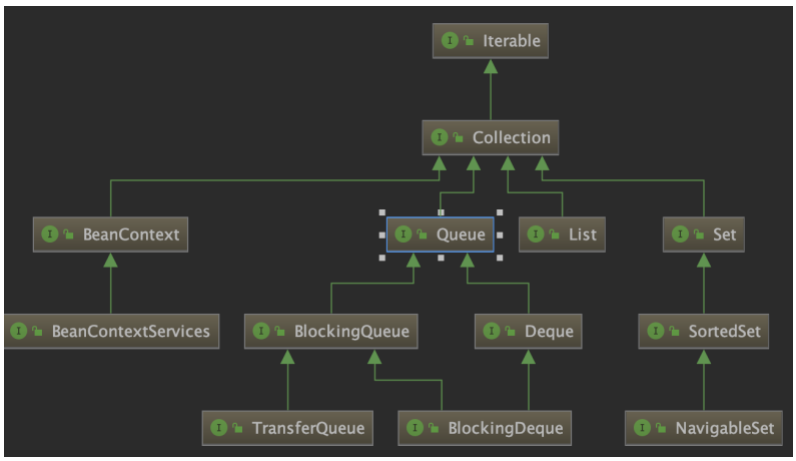
   执行`pop`时（即取出栈顶元素，并将该元素从栈中删除），是取出数组末尾元素，然后将该元素从数组中删除。

2. Stack 继承于 `Vector`，意味这Vector拥有的属性和功能，stack都拥有。

Java Queue源码解析

Queue, 是一个队列先进先出（FIFO，First In First Out）的数据结构；

在Java工具包中，Queue是一个接口，与List、Set同一级别，都是继承了Collection接口



Queue的源码

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

```java
public interface Queue<E> extends Collection<E> {
    // 将元素插入队列，容量不够则抛出异常
    //如果队列已满，抛出IllegalStateException异常
    boolean add(E e);
    // 将元素插入队列，与add相比，在容量受限时应该使用这个
    // 如果队列已满，返回false
    boolean offer(E e);
    // 将对首的元素删除，队列为空则抛出异常
    // 队列为空，抛出NoSuchElementException异常
    E remove();
    // 将对首的元素删除，队列为空则返回null
    // 队列为空，返回null
    E poll();
    // 获取队首元素，单不移除，队列为空则抛出异常
    // 如果队列为空，抛出NoSuchElementException异常
    E element();
    // 获取队首元素，但不移除，队列为空则返回null
    // 队列为空，返回null
    E peek();
}
```

Deque：Double-End-Queue（双端队列）

问题

（1）什么是双端队列?

（2）ArrayDeque 是这么实现双端队列的?

（3）ArrayDeque是线程安全的吗?

（4）ArrayDeque是有界的吗?

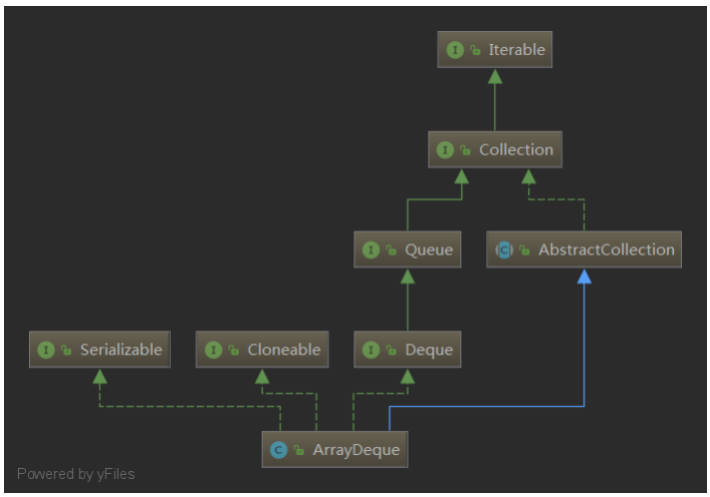
理解

简单理解：两端都可以进出的Queue（Deque – double end queue），双端队列是一种特殊的队列，它的两端都可以进出元素，故而得名双端队列。

插入和删除都是O(1)操作，查找是O(n)操作。

ArrayDeque是一种以数组方式实现的双端队列，它是非线程安全的。

继承体系



`Deque` 接口继承自 `Queue` 接口，但 Qeque 支持同时从两端添加或移除元素，因此被成为双端队列。鉴于此，Deque 接口的实现可以被当作 FIFO 队列使用，也可以当作 FILO队列（栈）来使用。官方也是推荐使用Deque的实现来替代Stack。

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

`ArrayDeque` 是 `Deque` 接口的一种具体实现，是依赖于可变数组来实现的。ArrayDeque没有容量现在，可根据需求自动进行扩容。ArrayDeque不支持值为null的元素。

```
1 public interface Deque<E> extends Queue<E> {
2     // 添加元素到队列头，容量不够会抛出异常
3     void addFirst(E e);
```

```java
    // 添加元素到队列尾，容量不够会抛出异常
    void addLast(E e);
    // 添加元素到队列头，容量不够返回false
    boolean offerFirst(E e);
    // 添加元素到队列尾，容量不够返回false
    boolean offerLast(E e);
    // 从队列头移除元素，不存在抛出异常
    E removeFirst();
    // 从队列尾移除元素，不存在抛出异常
    E removeLast();
    // 从队列头移除元素，不存在返回null
    E pollFirst();
    // 从队列尾移除元素，不存在返回null
    E pollLast();
    // 获取队列头元素，不存在抛出移除
    E getFirst();
    // 获取队列尾元素，不存在抛出移除
    E getLast();
    // 获取队列头元素，不存在返回null
    E peekFirst();
    // 获取队列尾元素，不存在返回null
    E peekLast();
    // 从队列头向后遍历移除指定元素
    boolean removeFirstOccurrence(Object o);
    // 从队列尾向前遍历移除指定元素
    boolean removeLastOccurrence(Object o);

    // *** Queue methods ***
    // 添加元素，等于addLast(e)
    boolean add(E e);
    // 添加元素，等于offerLast(e)
    boolean offer(E e);
    // 移除元素，等于removeFirst()
    E remove();
    // 移除元素，等于pollFirst()
    E poll();
    // 查看元素，等于getFirst()
    E element();
    // 查看元素，等于peekFirst()
    E peek();


    // *** Stack methods ***
    // 入栈，等于addFirst(e)
    void push(E e);
    // 出栈，等于removeFirst()
    E pop();


    // *** Collection methods ***
    // 删除指定元素，等于removeFirstOccurrence(o)
    boolean remove(Object o);
    // 检查是否包含某个元素
```

```
    boolean contains(Object o);
    // 元素个数
    public int size();
    // 迭代器
    Iterator<E> iterator();
    // 反向迭代器
    Iterator<E> descendingIterator();
}
```

Deque 提供来双端的插入和移除操作，如下：

	First Element（Head）		Last Element（Tail）	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst(e)	pollFirst(e)	removeLast(e)	pollLast(e)
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Deque 和 Queue 方法的对应关系，如下：

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	pkkeFist()

Deque 和 Stack 方法的对应关系，如下：

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	getFirst()

源码分析

主要属性

```
// 存储元素的数组
transient Object[] elements; // non-private to simplify nested class access
// 队列头的位置
transient int head;
// 队列尾的位置
transient int tail;
// 最小初始容量
private static final int MIN_INITIAL_CAPACITY = 8;
```

从属性可以看出，ArrayDeque使用数组存储元素，并使用头尾指针标识队列的头和尾，其最小容量为8。

构造方法

```
    // 默认构造方法，初始容量为16
    public ArrayDeque() {
         elements = new Object[16];
    }
     // 指定元素个数初始化，最小为8
     public ArrayDeque(int numElements) {
         allocateElements(numElements);
     }
     // 将集合c中的元素初始化到数组中
     public ArrayDeque(Collection<? extends E> c) {
         allocateElements(c.size());
         addAll(c);
     }
     // 初始化数组
     private void allocateElements(int numElements) {
         elements = new Object[calculateSize(numElements)];
     }
     // 计算容量，这段代码的逻辑是算出大于numElements的最接近的2的n次方且不小于8
     // 比如，3算出来是8，9算出来是16，33算出来是64
     private static int calculateSize(int numElements) {
         int initialCapacity = MIN_INITIAL_CAPACITY;
         // Find the best power of two to hold elements.
         // Tests "<=" because arrays aren't kept full.
         if (numElements >= initialCapacity) {
             initialCapacity = numElements;
             initialCapacity |= (initialCapacity >>>  1);
             initialCapacity |= (initialCapacity >>>  2);
             initialCapacity |= (initialCapacity >>>  4);
             initialCapacity |= (initialCapacity >>>  8);
             initialCapacity |= (initialCapacity >>> 16);
             initialCapacity++;

             if (initialCapacity < 0)   // Too many elements, must back off
                 initialCapacity >>>= 1;// Good luck allocating 2 ^ 30 elements
         }
         return initialCapacity;
     }
```

通过构造方法，我们指定默认初始容量是16，最小容量是8，计算出的容量一定是2的次幂。

入队

入队有很多方法，这里主要分析两个，`addFirst(e)` 和 `addLast(e)`。

```
// 从队列头入队
public void addFirst(E e) {
        // 不允许null元素
        if (e == null)
            throw new NullPointerException();
        // 将head指针减1并与数组长度减1取模
        // 这是为来防止数组到头来边界溢出
        // 如果到头了就从尾再向前
        // 相当于循环利用数组
        elements[head = (head - 1) & (elements.length - 1)] = e;
        // 如果头尾挨在一起了，就扩容
        // 扩容规则也很简单，直接两倍
        if (head == tail)
            doubleCapacity();
    }

// 从队列尾入队
public void addLast(E e) {
        // 不允许null元素
        if (e == null)
            throw new NullPointerException();
        // 在尾指针的位置放入元素
        // 可以看到tail指针指向的是队列最后一个元素的下一个位置
        elements[tail] = e;
        // tail指针加1，如果到数组尾了就从头开始
        if ( (tail = (tail + 1) & (elements.length - 1)) == head)
            doubleCapacity();
    }
```

(1) 入队有两种方式，从队列头或者从队列尾；
(2) 如果容量不够了，直接扩大为两倍；
(3) 通过取模的方式让头尾指针在数组范围内循环；
(4) x & (len – 1) = x % len，使用&的方式更快；

扩容

```
private void doubleCapacity() {
        assert head == tail;
        // 头指针的位置
        int p = head;
        // 旧数组的长度
        int n = elements.length;
        //获取头节点右侧的元素个数
```

```
        int r = n - p; // number of elements to the right of p
        //新容量等于原容量左移一位，转换为十进制计算就是：newCapacity=n*(2^1)
        int newCapacity = n << 1;
        //如果新容量小于0,这里肯定又是超过int值上限之后变成负数的情况了,从这里能够看出来原容量大于等于2^
        if (newCapacity < 0)
            throw new IllegalStateException("Sorry, deque too big");
        // 新建数组
        Object[] a = new Object[newCapacity];
        /*拷贝原数组的数据到新数组*/
        //拷贝原头节点右侧的数据,索引为[p,elements.length-1],到新数组索引为[0,r-1]
        System.arraycopy(elements, p, a, 0, r);
        // 拷贝原头节点左侧的数据,索引为[0,p-1],到新数组索引为[r,p-1]
        System.arraycopy(elements, 0, a, r, p);
        //通过上面的拷贝,将头节点索引重新变成了0,下一个尾节点索引变成了n(即原数组的容量,因为此时数组能够
        // 赋值为新数组
        elements = a;
        // head指向0，tail指向旧数组长度表示的位置
        head = 0;
        tail = n;
    }
```

总结一些扩容的主要步骤：

（1）首先计算出新容量，使用 `<<` 运算计算出理论新容量应该为原容量的两倍或者为负数，然后进入步骤2；实际上原容量大于等于2^30之后，扩容后的容量值就会小于0，即ArryDeque最大容量为2^30。

（2）然后判断容量是否小于0，如果小于0那说明新容量超过了int值上线，抛出一次程序结束；如果大于0，那么说明可以扩容，进入步骤3。

（3）新建扩容后大小的空数组，拷贝原数组的数据到新数组，通过两次拷贝，将头节点索引重新变成了0，下一个尾节点索引变成了原数组长度。然后改变数组引用和相关值，扩容结束。

出队

出队同样有很多方法，我们主要看两个，`pollFirst()` 和 `pollLast()`。

```
    // 从队列头出队
    public E pollFirst() {
        int h = head;
        @SuppressWarnings("unchecked")
        // 取出队列头元素
        E result = (E) elements[h];
        // Element is null if deque empty
        // 如果队列为空，就返回null
        if (result == null)
            return null;
        // 将队列头置为空
        elements[h] = null;     // Must null out slot
        // 队列头指针右移一位
        head = (h + 1) & (elements.length - 1);
        // 返回取得的元素
        return result;
```

```
    }
    // 从队列尾出队
    public E pollLast() {
        // 尾指针左移一位
        int t = (tail - 1) & (elements.length - 1);
        @SuppressWarnings("unchecked")
        // 取出当前尾指针处元素
        E result = (E) elements[t];
        // 如果队列为空返回null
        if (result == null)
            return null;
        // 将当前尾指针处置为空
        elements[t] = null;
        // tail指向新的尾指针处
        tail = t;
        // 返回取得的元素
        return result;
    }
```

(1) 出队有两种方式，从队列头或者从队列尾；
(2) 通过取模的方式让头尾指针在数组范围内循环；
(3) 出队之后没有缩容。

出队

前面介绍Deque的时候说过，Deque可以直接作为栈来使用，那么ArrayDeque是怎么实现的呢?

```

public void push(E e) {
    addFirst(e);
}

public E pop() {
    return removeFirst();
```

是不是很简单，入栈出栈只要都操作队列头就可以了

总结

(1) `ArrayDeque` 是采用数组方式实现的双端队列；
(2) `ArrayDeque` 的出队入队是通过头尾指针循环利用数组实现的；
(3) `ArrayDeque` 容量不足时是会扩容的，每次扩容容量增加一倍；
(4) `ArrayDeque` 可以直接作为栈使用。

拓展：ArrayDeque 与 LinkedList 的区别与使用建议

相同点：

1. ArrayDeque 和 LinkedList 都实现了Deque接口，都是双端队列的实现，都具有操作队列头尾的一系列方法。
2. 都是非线程安全的集合。

不同点：

1. ArrayDeque来自JDK1.6，底层是采用数组实现的双端队列，而LinkedList来自JDK1.2，底层则是采用链表实现的双端队列；
2. ArrayDeque不允许null元素，而LinkedList允许null元素。
3. 如果仅仅使用Deque的方法，即从队列两端操作元素，用作队列或者栈，并且如果数据量比较大，一般来说ArrayDeque的效率要高于LinkedList，其效率更高的元素可能是ArrayDeque不需要创建节点对象，添加的元素可直接作为节点对象，LinkedList则需要队添加的元素进行包装为Node节点，并且还具有其它引用的赋值操作。
4. LinkedList还同时实现了List接口，具有通过“索引”操作队列数据的方法，虽然这里的“索引”只是自己维护的索引，并发数组的索引，但该功能这是ArrayDeque所不具备的。如果需要队队列中间的进行元素的增、删、改操作，那么你智能使用LinkedList，因此LinkedList的应用（或者说可用方法）更加广泛。

附，使用Deque的方法时的ArrayDeque和LinkedList的性能对比：

```
public class ArrayDequeTest2 {
    static ArrayDeque<Integer> arrayDeque = new ArrayDeque<Integer>();
    static LinkedList<Integer> linkedList = new LinkedList<Integer>();


    public static long arrayDequeAdd() {
        //开始时间
        long startTime = System.currentTimeMillis();
        for (int i = 1; i <= 5000000; i++) {
            arrayDeque.addLast(i);
            arrayDeque.addFirst(i);
        }
        //结束时间
        long endTime = System.currentTimeMillis();
        //返回所用时间
        return endTime - startTime;
    }

    public static long arrayDequeDel() {
        //开始时间
        long startTime = System.currentTimeMillis();
        for (int i = 1; i <= 5000000; i++) {
            arrayDeque.pollFirst();
            arrayDeque.pollLast();
        }
        //结束时间
        long endTime = System.currentTimeMillis();
        //返回所用时间
        return endTime - startTime;
    }
```

```
31
32      public static long linkedListAdd() {
33          //开始时间
34          long startTime = System.currentTimeMillis();
35          for (int i = 1; i <= 5000000; i++) {
36              linkedList.addLast(i);
37              linkedList.addFirst(i);
38          }
39          //结束时间
40          long endTime = System.currentTimeMillis();
41          //返回所用时间
42          return endTime - startTime;
43      }
44
45      public static long linkedListDel() {
46          //开始时间
47          long startTime = System.currentTimeMillis();
48          for (int i = 1; i <= 5000000; i++) {
49              linkedList.pollFirst();
50              linkedList.pollLast();
51          }
52          //结束时间
53          long endTime = System.currentTimeMillis();
54          //返回所用时间
55          return endTime - startTime;
56      }
57
58
59      public static void main(String[] args) throws InterruptedException {
60          Thread.sleep(100);
61          long time1 = arrayDequeAdd();
62          long time3 = arrayDequeDel();
63          System.out.println("arrayDeque添加元素所用时间====>" + time1);
64          System.out.println("arrayDeque删除元素所用时间====>" + time3);
65          System.gc();
66          Thread.sleep(100);
67          long time2 = linkedListAdd();
68          long time4 = linkedListDel();
69          System.out.println("linkedList添加元素所用时间====>" + time2);
70          System.out.println("linkedList删除元素所用时间====>" + time4);
71      }
72  }
```