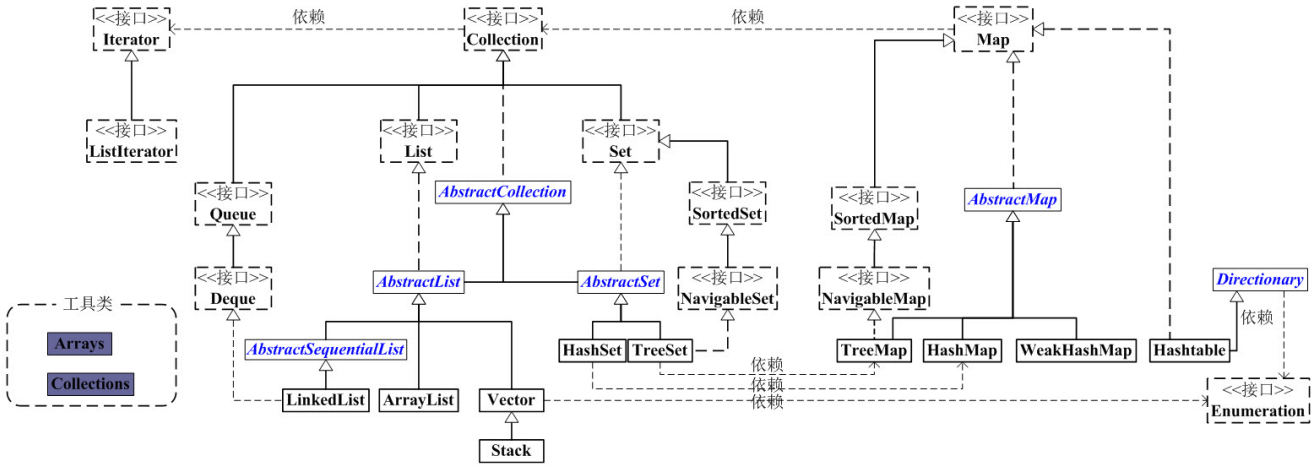
1. Collection接口
    1.1. Collection主要方法
    1.2. Collection对集合运算的支持
    1.3. Collection转化为数组toArray()
2. List接口
    2.1. List主要方法
3. Queue与Deque接口
    3.1. Queue主要方法
    3.2. Deque主要方法
4. Map接口
    4.1. Map的主要方法
    4.2. Map.Entry键值对
    4.3. SortedMap接口
    4.4. NavigableMap接口
5. Set接口
    5.1. SortedSet接口
    5.2. NavigableSet接口
6. 实现类小结
7. 迭代器
    7.1. Iterable
    7.2. Iterator
        7.2.1. Iterator主要方法
        7.2.2. 游标模式
        7.2.3. fail-fast机制
    7.3. ListIterator
        7.3.1. ListIterator主要方法

总体框架

Java容器主要可以划分为4个部分：`List`列表、`Set`容器、`Map`映射、工具类(`Iterator`迭代器、`Enumeration`枚举类、`Arrays`和`Collections`)



它们的关系如上图所示，**标蓝的为抽象类**，**实线全箭头**指的是extends（继承），**虚线全箭头**表示implement（实现），**虚线半箭头**依赖指的是这个类里面有依赖接口或者类的成员变量，比如HashSet类，继承AbstractSet抽象类，它里面又定义了HashMap的成员变量：

1. Collection接口

`Collection`是`Queue`,`List`,`Set`的根类，是一个高度抽象出来的接口，包含了容器的基本操作和属性。

1.1. Collection主要方法

```
boolean add(E e)                          //向Collection末尾中添加元素
boolean addAll(Collection<? extends E> c) //把Collectionc中的所有元素添加到指定的Collection里
void clear()                              //清除Collection中的元素，长度变为0
boolean contains(Object o)                //返回Collection中是否包含指定元素
int hashCode()                            //返回此Collection的哈希码值。
boolean isEmpty()                         //Collection是否为空
Iterator<E> iterator()                    //返回一个Iterator对象，用于遍历Collection中的元素
boolean remove(Object o)                  //删除Collection中指定的对象
default boolean removeIf(Predicate<? super E> filter) //删除此Collection中满足给定谓词的所有元素
int size()                                //返回Collection里元素的个数
Object[] toArray()                        //把Collection转化为一个数组
<T> T[] toArray(T[] a)                    //把Collection转化为一个指定类型的数组，推荐使用此种方式
```

- 其中`removeIf()`的源码

```
default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }
```

支持对容器的条件过滤。

1.2. Collection对集合运算的支持

```
boolean retainAll(Collection<?> c)          //得到调用Collection与c的交集
boolean addAll(Collection<? extends E> c)   //得到调用Collection与c的交集
boolean removeAll(Collection<?> c)          //得到调用Collection与c的差集
//被持有对象必须有正确的equals方法，也就是如果自定义类型需要重写equals方法
```

`Set`接口实现类更符合数学定义上的集合，因此我们常用`TreeSet`、`HashSet`去调用这些方法实现更符合数学定义的集合运算。当然`List`接口实现类也能调用：其中实现无重复元素并集的方式：

```
list2.removeAll(list1);
list1.addAll(list2);
```

再就是要注意以上方法，返回值是`boolean`类型，也就是集合运算会影响调用方法的`Collection`的结构，如果需要单独返回运算结果，可以通过`static Collections.copy()`静态方法或是对应的`clone()`方法，产生容器副本。

示例：

```
//返回ls、ls2的交集
public List intersect(List ls, List ls2) {
        List list = new ArrayList(Arrays.asList(new Object[ls.size()]));
//产生ls的副本
        Collections.copy(list, ls);
        list.retainAll(ls2);
        return list;
    }
  //返回ls、ls2的并集
    public List union(List ls, List ls2) {
```

```
        List list1 = new ArrayList(Arrays.asList(new Object[ls.size()]));
        List list2 = new ArrayList(Arrays.asList(new Object[ls2.size()]));
        Collections.copy(list1, ls);
        Collections.copy(list2, ls2);
        list2.removeAll(list1);          //剔除重复元素
        list1.addAll(list2)
        return list1;
    }
  //返回ls、ls2的差集，注意ls-ls2!=ls2-ls     -代表差集运算
    public List diff(List ls, List ls2) {
        List list = new ArrayList(Arrays.asList(new Object[ls.size()]));
        Collections.copy(list, ls);
        list.removeAll(ls2);
        return list;
    }
```

1.3. Collection转化为数组toArray()

当我们调用`Collection`中的`toArray()`，可能遇到过抛出“`java.lang.ClassCastException`”异常的情况。下面我们说说这是怎么回事。

`Collection`接口提供了2个`toArray()`函数：

```
Object[] toArray()
<T> T[] toArray(T[] contents)
```

调用`toArray()`函数会抛出“`java.lang.ClassCastException`”异常，但是调用`toArray(T[] contents)`能正常返回`T[]`。

`toArray()`会抛出异常是因为`toArray()`返回的是`Object[]`数组，将`Object[]`转换为其它类型(如，将`Object[]`转换为的`Integer[]`)则会抛出“`java.lang.ClassCastException`”异常，因为**Java不支持向下转型。** 解决该问题的办法是调用`<T> T[] toArray(T[] contents)`.

调用`toArray(T[] contents)`返回`T[]`的可以通过以下几种方式实现（以`ArrayList`为例）。

```
// toArray(T[] contents)调用方式一
public static Integer[] vectorToArray1(ArrayList<Integer> v) {
    Integer[] newText = new Integer[v.size()];
    v.toArray(newText);
    return newText;
}

// toArray(T[] contents)调用方式二。最常用!
```

```java
public static Integer[] vectorToArray2(ArrayList<Integer> v) {
    Integer[] newText = (Integer[])v.toArray(new Integer[0]);
    return newText;
}

// toArray(T[] contents)调用方式三
public static Integer[] vectorToArray3(ArrayList<Integer> v) {
    Integer[] newText = new Integer[v.size()];
    Integer[] newStrings = (Integer[])v.toArray(newText);
    return newStrings;
}
```

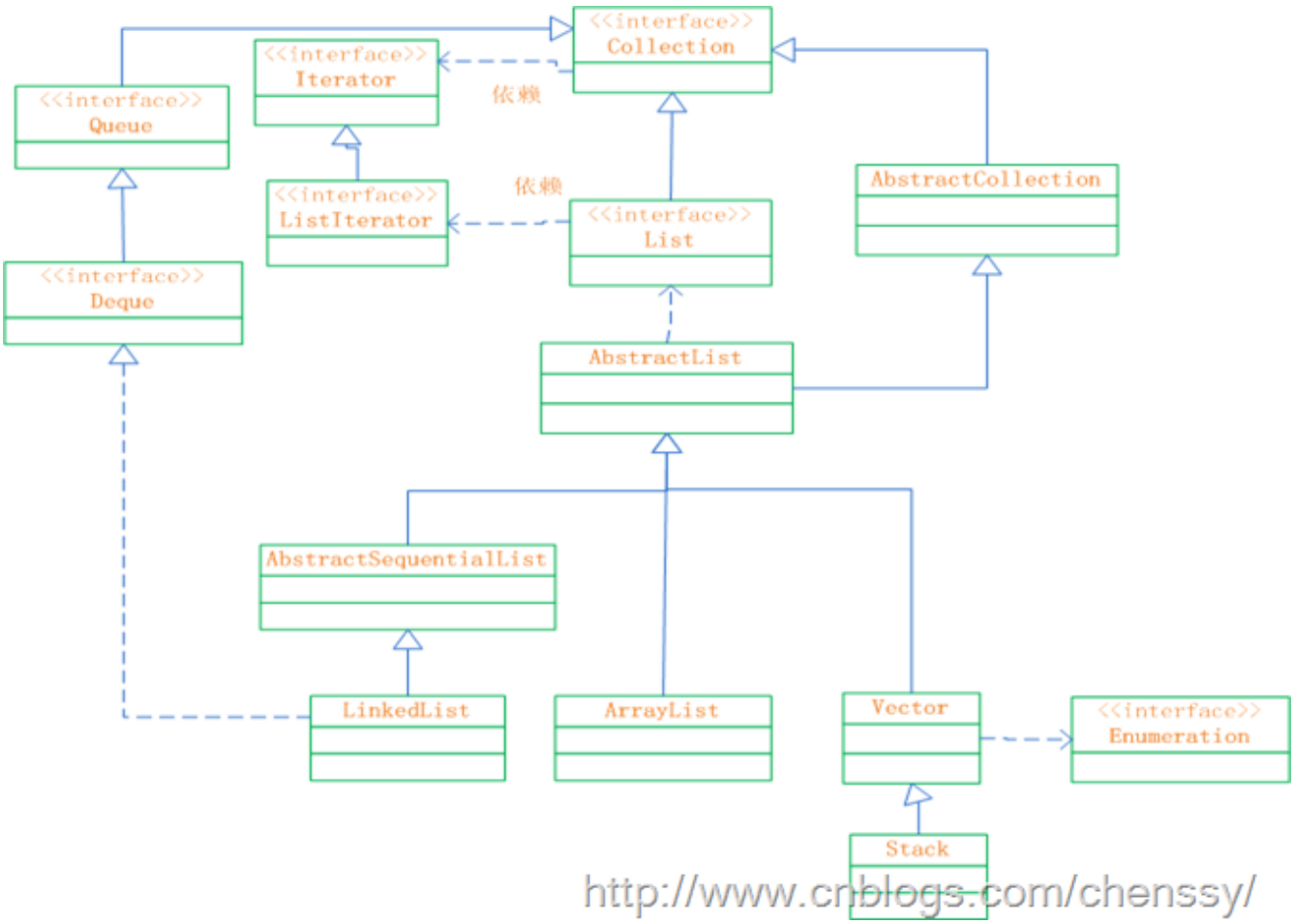
`toArray()` 、与`toArray(T[ ] a)`的源码

```java
    // 返回ArrayList的E数组
    public Object[] toArray() {
        return Arrays.copyOf(elementData, size);
    }

    // 返回ArrayList的模板数组。所谓模板数组，即可以将T设为任意的数据类型
    public <T> T[] toArray(T[] a) {
        // 若数组a的大小 < ArrayList的元素个数;
        // 则新建一个T[]数组，数组大小是“ArrayList的元素个数”，并将“ArrayList”全部拷贝到新数组中
        if (a.length < size)
            return (T[]) Arrays.copyOf(elementData, size, a.getClass());

        // 若数组a的大小 >= ArrayList的元素个数;
        // 则将ArrayList的全部元素都拷贝到数组a中。
        System.arraycopy(elementData, 0, a, 0, size);
        if (a.length > size)
            a[size] = null;
        return a;
    }
```

2. List接口

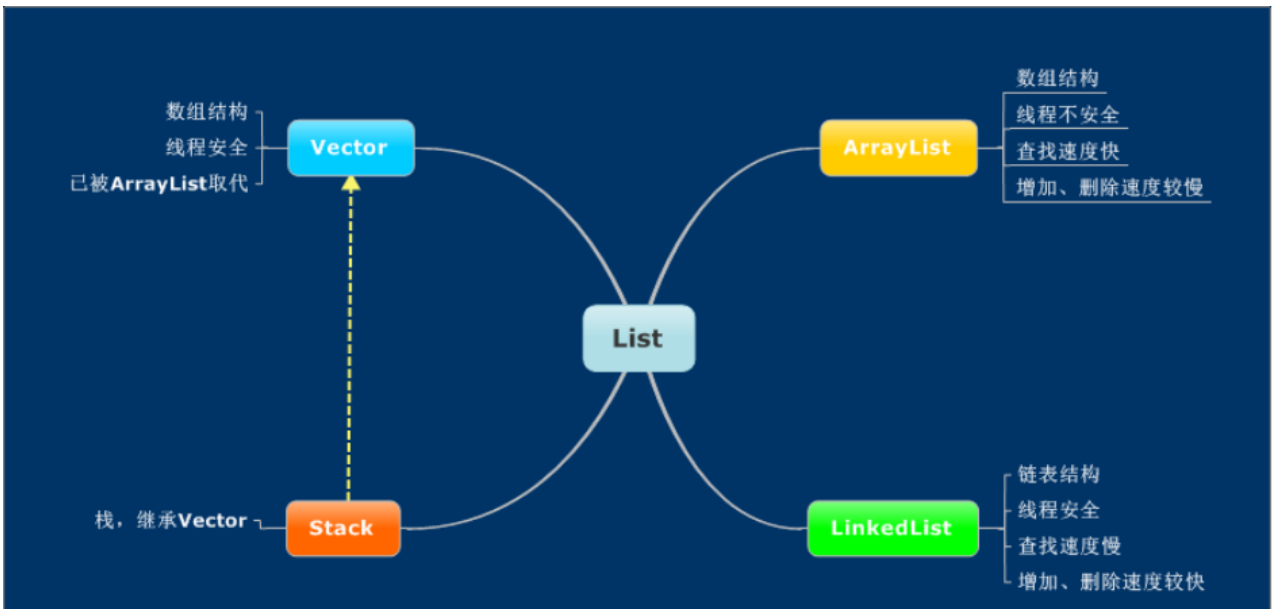


`List` 接口，称为**有序的** `Collection` 也就是列表。该接口可以对列表中的每一个元素的插入位置进行精确的控制，同时用户**可以根据元素的整数索引访问元素，并搜索列表中的元素。**

简要说明上面框架各主要元素功能：

- `ListIterator`：列表迭代器，允许**按任一方向遍历列表、迭代期间修改列表，并获得迭代器在列表中的当前位置。**这个迭代器只能被实现了`List`接口的容器类使用，并且有很多实用的方法，我们可以在下面关于迭代器——ListIterator的介绍部分了解其用法。
- `Queue`：队列接口。提供队列基本的插入、获取、检查操作。
- `Deque`：一个线性容器接口，**支持在两端插入和移除元素。**大多数 Deque 实现对于它们能够包含的元素数没有固定限制，但此接口既支持有容量限制的双端队列，也支持没有固定大小限制的双端队列。
- `LinkedList`：`List` 接口的链表实现。`LinkedList`可被用作堆栈（`stack`），队列（`queue`）或双向队列（`deque`）。
- `ArrayList`：`List` 接口的大小可变数组的实现。它实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 `List` 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。
- `Vector`：有可变数组实现（已被`ArrayList`取代）。与数组一样，它包含可以使用整数索引进行访问的组件。

- `Stack`：后进先出（LIFO）的对象堆栈（可以被`ArrayDeque`接口替代）。它通过五个操作对类 Vector 进行了扩展，允许将向量视为堆栈。



2.1. List主要方法

`List` 接口的方法除了继承自 `Collection` 接口的操作：**基础操作**、**批量操作**、**数组操作**、以及 JDK1.8 后提供的**聚合操作** 还包括一些基于索引的额外操作：**位置访问**、**搜索**、**迭代**、**范围视图**

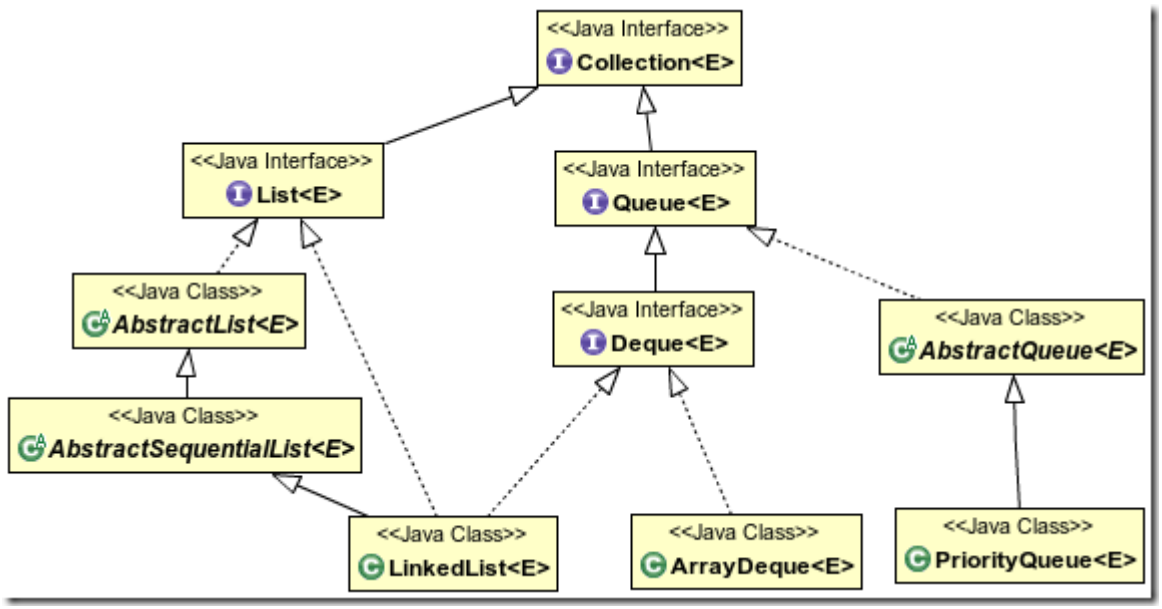
```
void add(int index, E element)                          //在指定位置插入元素
void addAll(int index, Collcetion<? extends E> c)      //在指定位置插入c中的所有元素
E get(int index)                                        //返回指定位置的元素
int indexOf(E element)                                  //返回指定元素第一次出现的索引，若不包含此元素返回-1
int lastIndexOf(E element)                              //返回指定元素最后一次出现的索引
ListIterator<E> listIterator();                         //返回从初始位置开始的ListIterator
ListIterator<E> listIterator(int index);                //返回从指定位置开始的ListIterator
E remove(int index)                                     //删除指定位置的元素
void replaceAll(UnaryOperator<E> operator)      //将该ArrayList的每个元素替换为将该operator应用于该元素的结果。
E set(int index, E element)                             //用指定元素替换指定位置的元素，并返回被替换的元素
void sort(Comparator<? super E> c)                      //根据指定的comparator进行排序
List<E> subList(int fromIndex,int toIndex)              //返回下标范围[fromIndex,toIndex)的视图
```

3. Queue与Deque接口

java中的队列接口`Queue`是一个被设计用来按一定的优先级处理元素的Deque，**仅支持队尾插入队首读取，又称为“先进先出”（FIFO—first in first out）的线性表。**与`List`、`Set`同一级别，都是继承了`Collection`接口。

`Deque`继承自`Queue`，它是支持双端读写的**双端队列。**`Deque`继承了Queue接口因此完全支持队列行为，除此之外还支持双端队列，以及堆栈的行为——我们应该优先使用此接口而不是使用遗留类`Stack`。

结构框架



3.1. Queue主要方法

虽然`Queue`接口实现了`Collection`接口但是，为了保持队列的特性，我们应该尽可能实用`Queue`接口提供的方法。

队列，它主要分为两大类，一类是阻塞式队列（与线程相关），队列满了以后再插入元素或是空了移除元素则会抛出异常 。另一种队列则是双端队列，不会返回异常。

由于第一套方法的名字，能够见名知意因此介绍第一套方法，它们操作失败会抛出异常。

`Queue`对队列行为的支持：

```
boolea add(E e)                 //在队首添加一个元素
E remove()                     //移除并返回队列头部的元素
E element()                    //返回队列头部的元素

```

队列不允许随机访问队列中的元素，因此`Queue`、`Deque`接口没有提供随机读写的方法

3.2. Deque主要方法

插入失败返回`false`，返回元素失败返回`null`

`Deque`支持对双端队列行为的支持，失败会抛出异常

```
boolean addFirst(E e)        //在队首插入元素
boolean addLast(E e)         //在队尾插入元素
E removeFirst()              //返回并移除队首元素
E removeLast()               //返回并移除队尾元素
E getFirst()                 //返回队首元素
E getLast()                  //返回队尾元素
int size()                   //返回Deque中的元素数

```

`Deque`支持堆栈行为：

```
void push(E e)             //向栈顶添加元素
E pop()                    //弹出栈顶元素
E peek()                   //获取但不弹出栈顶

```

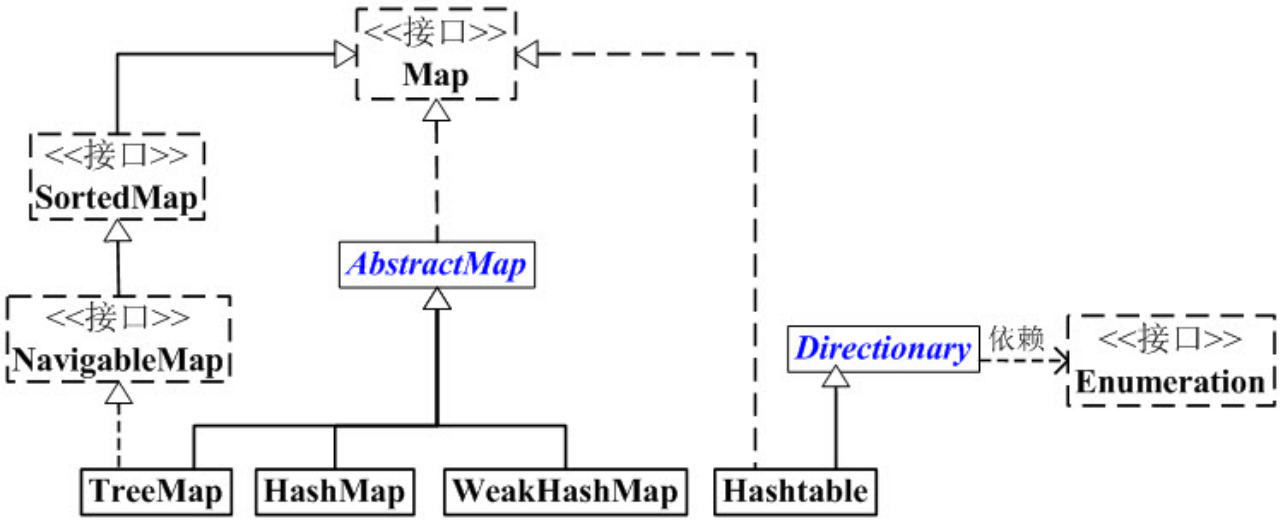
虽然我们可以将Dqueue窄化为队列，堆栈。但是我们仍然丰富的方法操控实现了Dqueue的对象，对其使用一些不符合当前数据结构的方法。——要灵活使用。

主要实现类

- `PriorityQueue`： **优先队列**，本质维护一个**有序列表**。可自然排序亦可传递`comparator`构造函数实现自定义排序，是很常用的数据结构。
- `LinkedList`底层是链表，实现了`Deque`接口(不推荐)。
- `ArrayDeque`底层是数组，实现了`Deque`接口。当用作栈时，性能优于`stack`；用作队列时，性能优于`LinkedList`

4. Map接口

我们先了解`Map`再了解`Set`，因为`Set`**的实现类都是基于**`Map`**来实现的**(如，`HashSet`是通过`HashMap`实现的，`TreeSet`是通过`TreeMap`实现的。



- `Map` 是**映射接口**，`Map`中存储的内容是**键值对**(key-value)。`Map`**映射中不能包含重复的键；每个键最多只能映射到一个值。**
- `AbstractMap` 是**继承于Map的抽象类，它实现了**`Map`**中的大部分API**。其它`Map`的实现类可以通过继承`AbstractMap`来减少重复编码。
- `SortedMap` 是继承于`Map`的接口。`SortedMap`中的内容是**排序的键值对**，排序的方法是通过比较器(`Comparator`)。
- `NavigableMap` 是继承于`SortedMap`的接口。相比于`SortedMap`，`NavigableMap`有一系列的搜索方法；如"获取大于/等于某对象的键值对"、“获取小于/等于某对象的键值对”等等。
- `TreeMap` 继承于`AbstractMap`，且实现了`NavigableMap`接口；因此，`TreeMap`中的内容是“**有序的键值对**”！
- `HashMap` 继承于`AbstractMap`，但没实现`NavigableMap`接口；因此，HashMap的内容是“**键值对，但不保证次序**”！
- `Hashtable` 虽然不是继承于`AbstractMap`，但它继承于`Dictionary`(`Dictionary`也是键值对的接口)，而且也实现`Map`接口；因此，`Hashtable`的内容也是“**键值对，也不保证次序**”——被淘汰，可以使用`HashMap`替代。
- `WeakHashMap` 继承于`AbstractMap`。它和`HashMap`的键类型不同，`WeakHashMap`**的键是“弱键”。**

4.1. Map的主要方法

```
1 V get(Object key)                          //返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
2 V put(K key, V value)                       //将指定的值与此Map中的指定键关联（可选操作）。并返回原来键，若没有返回null
3  void putAll(Map<? extends K,? extends V> m) //将m中的所有映射关系复制到此Map中（可选操作）。
```

```
V remove(Object key)                          //如果存在一个键key的映射关系，则将其从此Map中移除（可选操作）。
V replace(K key, V value)                     //只有当指定键映射到某个值时，才能替换指定键的条目。
boolean containsKey(Object key)               //如果此Map包含指定键的映射关系，则返回true。
boolean containsValue(Object value)           //如果此Map将一个或多个键映射到指定值，则返回 true。
 int size()                                   //返回此Map中的Map.Entry数。
 void clear()                                 //从此Map中移除所有映射关系（可选操作）。
 boolean isEmpty()                            //如果此Map未包含Map.Entry，则返回true。
 boolean equals(Object o)                     //比较指定的对象与此映射是否相等。
 Collection<V> values()                       //返回此Map中包含的值的 Collection 对象。
 Set<Map.Entry<K,V>> entrySet()               //返回此Map中包含的映射关系的 Set 对象。
 Set<K> keySet()                              //返回此Map中包含的键的 Set 对象。
void forEach(BiConsumer<? super K,? super V> action) //对此Map中的每项（键值对）执行给定的操作，直到所有项都被处理或操作引发异常。 --常用来遍历

```

`Map`提供接口分别用于返回 键集、值集或`Map.Entry`集。

- `entrySet()`用于返回**键-值集**的**Set集合**
- `keySet()`用于返回**键集**的**Set集合**
- `values()`用户返回**值集**的**Collection集合**

  因为`Map`中不能包含重复的键；每个键最多只能映射到一个值。所以，**键-值集、键集都是Set，值集时Collection。**

4.2. Map.Entry键值对

`Map.Entry`的定义如下类似于C++的pair：

```
interface Entry<K,V> {
    //...
}

```

`Map.Entry`是`Map`中内部的一个接口，`Map.Entry`是**键值对**，`Map`通过`entrySet()`获取`Map.Entry`的键值对集合，从而**通过该集合实现对键值对的操作。**提供了直接操控映射关系的键，值得方法

```
boolean  equals(Object object)           //如果obj是一个Map.Entry返回true
K  getKey()                              //返回此键值对项的键。
V  getValue()                            //返回此映射项的值。
V  setValue(V value)                     //用指定得值替换该键值对的值，并返回原值
static <K extends Comparable<? super K>,V>  Comparator<Map.Entry<K,V>>
comparingByKey() //返回一个比较器 ，按键的自然顺序比较Map.Entry 。
static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super
K> cmp)  //返回一个比较器，比较Map.Entry按键使用给定的Comparator 。
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V
comparingByValue() //返回一个比较器，按值的自然顺序比较Map.Entry。
static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super
V> cmp) //返回一个比较器 ，使用给定的Comparator比较Map.Entry的值。

```

通过`Map.entrySet()`返回`Map.Entry`对象的`Set`容器对象，再通过`Map.Entry`的方法，我们可以间接且高效的遍历`Map`，查找元素。

```
import java.util.*;

public class HashMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      HashMap hm = new HashMap();
      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();
      // Get an iterator
      Iterator i = set.iterator();
      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();
      // 修改键值关系
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
```

```
        System.out.println("Zara's new balance: " +
        hm.get("Zara"));
    }
}
```

4.3. SortedMap接口

`SortedMap`的定义如下：

```
public interface SortedMap<K,V> extends Map<K,V> {
//...
}
```

`SortedMap`是一个继承于`Map`接口的接口。它是一个**有序的**`SortedMap`键值映射。

`SortedMap`的排序方式有两种：**自然排序** 或者 **用户指定比较器。** 插入 `SortedMap` 的所有元素都必须实现 `Comparable` 接口（或者被指定的比较器所接受）。

另外，所有`SortedMap` 实现类都提供 4 个“标准”构造方法：

1. **void（无参数）构造方法**，它创建一个空的有`SortedMap`，**按照键的自然顺序进行排序。**
2. 带有一个 `Comparator` 类型 参数的构造方法，**它创建一个空的**`SortedMap`，**根据指定的比较器进行排序。**
3. 带有一个 `Map` 类型参数的构造方法，**它创建一个新的SortedMap，其键值对参数相同，按照键的自然顺序进行排序。**
4. 带有一个 `SortedMap` 类型参数的构造方法，**它创建一个新的**`SortedMap`，**其键值对的排序方法与输入的**`SortedMap`**相同。**

除了继承自Map的主要方法`SortedMap`有以下方法

```
 Comparator<? super K>  comparator()              //返回当前SortedMap的比较器。如果当前SortedMap按键的自然顺序排序，则返回null。
 K  firstKey()                                   //返回SortedMap的第一个键
 K  lastKey()                                    //返回SortedMap的最后一个键
 SortedMap<K, V>    headMap(K endKey)            //返回当前SortedMap中键小于endKey的子集
 SortedMap<K, V>    subMap(K startKey, K endKey)//返回当前SortedMap中键属于[startKey,endKey)的子集
 SortedMap<K, V>    tailMap(K startKey)          ////返回当前SortedMap中键大于等于startKey的子集

```

除了 `Map` 接口中的方法外， `SortedMap` 接口提供了如下面的操作：

- 范围查看 – 允许在 `SortedMap` 上进行任意的范围操作： `headMap、subMap、tailMap` 。
- 访问端点的键– 从 `SortedMap` 中返回第一个或最后一个键值对： `firstKey、lastKey` 。
- 访问 `Comparator` – 如果有，返回用于 `SortedMap` 的 `Comparator` 。

注意：几种方法返回限制范围的视图。 这种范围是*半开放的*，也就是说，**它们包括其低端点，但不包括其高端点。**

4.4. NavigableMap接口

`NavigableMap` 继承了 `SortedMap` ，定义如下：

```
public interface NavigableMap<K,V> extends SortedMap<K,V> {
//...
}

```

`NavigableMap` 扩展了 `SortedMap` ，除了 `SortedMap` 提供的方法外，还提供了针对给定搜索目标返回最接近匹配项的搜索方法。

```
Map.Entry<K, V> ceilingEntry(K key)//返回一个Map.Entry，它与大于等于给定键的最小键关联；如果不存在这样的键，则返回 null。

K   ceilingKey(K key)//返回大于等于给定键的最小键；如果不存在这样的键，则返回 null。

Map.Entry<K, V> firstEntry()//返回一个与此Map中的最小键关联的Map.Entry；如果映射为空，则返回 null。

Map.Entry<K, V> floorEntry(K key)//返回一个Map.Entry，它与小于等于给定键的最大键关联；如果不存在这样的键，则返回 null。

K   floorKey(K key)//返回小于等于给定键的最大键；如果不存在这样的键，则返回 null。

Map.Entry<K, V> higherEntry(K key)//返回一个Map.Entry，它与严格大于给定键的最小键关联；如果不存在这样的键，则返回 null。

K   higherKey(K key)//返回严格大于给定键的最小键；如果不存在这样的键，则返回 null。

Map.Entry<K, V> lastEntry()// 返回与此Map中的最大键关联的Map.Entry；如果映射为空，则返回 null。

Map.Entry<K, V> lowerEntry(K key)//返回一个Map.Entry，它与严格小于给定键的最大键关联；如果不存在这样的键，则返回 null。
```

```
18
19  Map.Entry<K, V> pollFirstEntry()//移除并返回与此Map中的最大键关联的Map.Entry;
    如果映射为空，则返回 null。
20
21  Map.Entry<K, V> pollLastEntry()//移除并返回与此Map中的最大键关联的Map.Entry; 如
    果映射为空，则返回 null。
22
23  K    lowerKey(K key)//返回严格小于给定键的最大键；如果不存在这样的键，则返回 null。
```

说明:

`NavigableMap`除了继承`SortedMap`的特性外，它的提供的常用功能可以分为2类:

- **提供搜索键-值对的方法。** `lowerEntry`、`floorEntry`、`ceilingEntry`和`higherEntry`方法，它们分别返回与小于、小于等于、大于等于、大于给定键的键关联的`Map.Entry`对象。`firstEntry`、`pollFirstEntry`、`lastEntry`和`pollLastEntry`方法，它们返回和/或移除最小和最大的`Map.Entry`（如果存在），否则返回`null`。
- **提供搜索键的方法。**这个和第1类比较类似`lowerKey`、`floorKey`、`ceilingKey`和`higherKey`方法，它们分别返回与小于、小于等于、大于等于、大于给定键的键。

5. Set接口

`Set`**具有与**`Collection`**完全一样的接口**（Collection接口），因此没有任何额外的功能

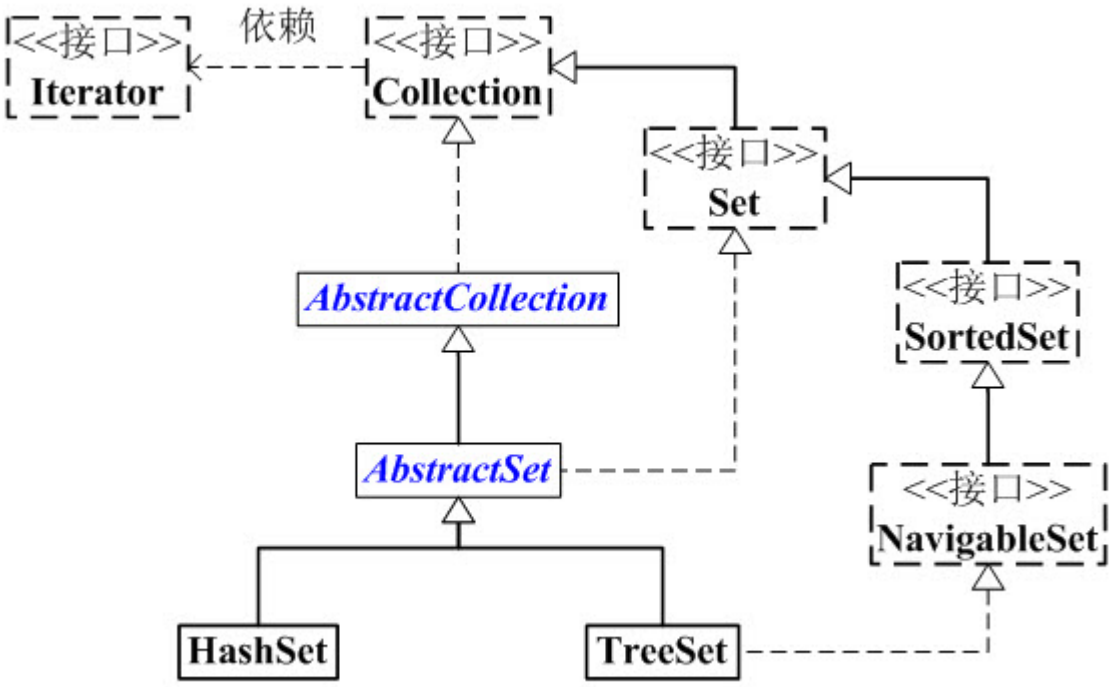
实际上`Set`就是`Collection`,只是行为不同。(这是继承与多态思想的典型应用：表现不同的行为。)`Set`**不保存重复的元素，就像数学定义中的集合保证元素的互异性**(至于如何判断元素相同则较为复杂，在具体实现部分在讲解)

`Set`中如何判断俩个对象是否相等（俩个条件必须同时满足）：

1）：俩个对象的`hashcode`方法返回值相等。

2）：俩个对象的`equals`比较相等，返回`true`，则说明是相同对象。

`Set`**框架结构** 实线是继承关系，虚线是实现关系



- `AbstractSet`是一个抽象类，它继承于`AbstractCollection`，实现了`Set`接口。`AbstractCollection`提供了Set接口的骨干实现，从而最大限度地减少了实现此接口所需的工作。
- `SortedSet`进一步提供关于元素的总体排序的`Set`。**这些元素使用其自然顺序进行排序，或者根据用户在创建`SortedSet`时提供的`Comparator`进行排序。**该`Set`的迭代器将按元素升序遍历`Set`。提供了一些附加的操作来利用这种排序。
- `NavigableSet`扩展的`SortedSet`，提供了了为给定搜索目标报告最接近匹配项的搜索方法。
- `HashSet`依赖于`HashMap`的实现，实际上是个`HashMap`的实例。`HashSet`**中的元素是无序的。**；特别是它不保证该顺序恒久不变。此类允许使用`null`元素。
- `TreeSet`基于`TreeMap`的`NavigableSet`实现。**使用元素的自然顺序对元素进行排序，或者根据用户创建`Set`时提供的`Comparator`进行排序**，具体取决于使用的构造方法。

5.1. SortedSet接口

`SortedSet` 是 `Set` 的一个子类，它支持 `Set` 中的元素排序，排序规则按照自然排序或者按照创建 `SortedSet` 时设置的 `Comparator` 进行排序。

```
public interface SortedSet<E> extends Set<E> {
//...
}
```

插入到排序集中的所有元素必须实现`Comparable`接口（或被指定的比较器`Comparator`）。 因此，所有这些元素都必须是可*相互比较*：`e1.compareTo(e2)`（或`comparator.compare(e1, e2)`）

所有通用排序集实现类应提供四个“标准”构造函数：

- 一个void（无参数）构造函数，它创建一个**根据其元素的自然顺序排序**的空排序集。
- 具有`Comparator`类型参数的构造函数，**它创建一个空的`SortedSet`，根据指定的比较器进行排序。**
- 具有类型为`Collection`的单个参数的构函数，它创建一个具有与其参数元素相同的的新的SortedSet，并**根据元素的自然顺序进行排序。**
- 具有类型为`SortedSet`的单个参数的构造函数，其创建具有与输入`SortedSet`相同的元素和相同排序行为的新排序集。

除了继承自（Collection接口的方法`有以下方法

```
Comparator<? super E> comparator( )//返回调用SortedSet的比较比较器。如果SortedSet的元素按照自然顺序排序，则返回null。
E first( )                          //返回该SortedSet的第一个元素。
E last( )                           //返回调用SortedSet的最后一个元素。
```

除了 `Set` 接口中的方法外，`SortedSet` 接口提供了如下面的操作：

- 范围查看 – 允许在 `SortedSet` 上进行任意的范围操作：`headSet、subSet、tailSet`。
- 访问端点 – 从 `SortedSet` 中返回第一个或最后一个元素：`first、last`。
- 访问 `Comparator` – 如果有，返回用于 SortedSet 的 `Comparator`。

注意：几种方法返回限制范围的视图。 这种范围是*半开放的*，也就是说，**它们包括其低端点，但不包括其高端点。**

5.2. NavigableSet接口

`NavigableSet`继承了`SortedSet`，定义如下：

```
public interface NavigableSet<E> extends SortedSet<E> {
//...
}
```

`NavigableSet`扩展了 `SortedSet`，除了`SortedSet`提供的方法外，还提供了针对给定搜索目标返回最接近匹配项的搜索方法。

```
 E  ceiling(E e) //返回此 Set 中大于等于给定元素的最小元素；如果不存在这样的元素，则返回 null。

Iterator<E> descendingIterator() //返回在此 Set 的元素上的逆序迭代器。

E   floor(E e) //返回此 Set 中小于等于给定元素的最大元素；如果不存在这样的元素，则返回 null。

E   higher(E e) //返回此 Set 中严格大于给定元素的最小元素；如果不存在这样的元素，则返回 null。

E   lower(E e) //返回此 Set 中严格小于给定元素的最大元素；如果不存在这样的元素，则返回 null。

E   pollFirst() //获取并移除第一个（最低）元素；如果此 Set 为空，则返回 null。

E   pollLast() //获取并移除最后一个（最高）元素；如果此 Set 为空，则返回 null。
```

说明:

`NavigableSet`除了继承`SortedSet`的特性外，它主要提供了最佳匹配目标元素的搜索方法:

`lower, floor, ceiling、`和`higher`分别返回`Set`中小于，小于等于，大于等于，大于给定元素的的元素，如果不存在这样的元素返回`null`.

6. 实现类小结

接口	哈希表实现	可变数组实现	树实现	链表实现	哈希表+链表实现
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList PriorityQueue	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap
SortedSet			TreeSet		
SortedMap			TreeMap		

7. 迭代器

7.1. Iterable

`Collection` 继承了 `Iterable< E >` 接口，`Iterable` 接口内只有一个 `iterator` 方法，返回一个 `Iterator` 迭代器:

```
public interface Iterable<T> {
/**
* Returns an {@link Iterator} for the elements in this object.
*
* @return An {@code Iterator} instance.
*/
Iterator<T> iterator();
```

7.2. Iterator

由于`Collection`依赖`Iterator`接口，因此所有实现了`Collection`的容器类都拥有了`Iterator`的方法，并且都有一个`iterator()`方法用来返回一个实现了`Iterator`的对象。可以通过其方法遍历容器中的元素。

使用 `Iterator` 遍历Deque是这样的:

```
Iterator iterator = list.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

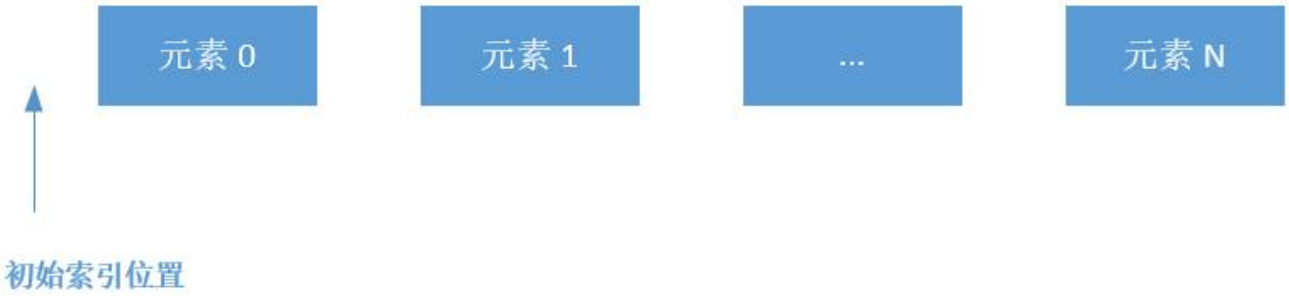
7.2.1. Iterator主要方法

```
default void forEachRemaining(Consumer<? super E> action) //对每个剩余元素执行给定的操作，直到所有元素都被处理或抛出异常。
boolean hasNext()                                //判断是否有下一个可迭代元素
E next()                                         //返回游标当前位置右边的元素并将游标移动到下一个位置
default void remove ()                           //删除迭代器最后一次操作的元素为 E，也就是更新最近一次调用 next() 或者 previous() 返回的元素。
```

注意`next() ,remove ()` 遵守fail-fast机制

7.2.2. 游标模式

迭代器模式也称游标模式，因为迭代器 没有当前所在元素一说，它只有一个游标(cursor)的概念，这个游标总是在元素之间，比如这样:



初始状态下`cursor=-1`，游标位置右边是第一个元素。调用`next()`游标会右移一位：（调用`ListIterator`提供的`previous()`游标就会回到之前位置。）

遍历完元素后，游标会在最后一个元素右边：

7.2.3. fail-fast机制

fail-fast 机制是java集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。 例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。

由于**fail-fast**机制，对使用迭代器的`remove()`增加了限制。

先看一下关于`Iterator`的官方文档

> Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

`Iterators`允许使用well-defined的语法删除元素，但是事实上我们在使用 Iterator 对容器进行迭代时如果修改容器 可能会报 *ConcurrentModificationException* 的异常。官方称这种情况下的迭代器是 *fail-fast* 迭代器。

以 `ArrayList` 为例，在调用迭代器的 `next, remove` 方法时：

```
public E next() {
        if (expectedModCount == modCount) {
            try {
                E result = get(pos + 1);
                lastPosition = ++pos;
                return result;
            } catch (IndexOutOfBoundsException e) {
                throw new NoSuchElementException();
            }
        }
        throw new ConcurrentModificationException();
    }

    public void remove() {
        if (this.lastPosition == -1) {
            throw new IllegalStateException();
        }

        if (expectedModCount != modCount) {
            throw new ConcurrentModificationException();
        }

        try {
            AbstractList.this.remove(lastPosition);
        } catch (IndexOutOfBoundsException e) {
            throw new ConcurrentModificationException();
        }

        expectedModCount = modCount;
        if (pos == lastPosition) {
            pos--;
        }
        lastPosition = -1;
    }
```

可以看到在调用迭代器的 `next, remove` 方法时都会比较 `expectedModCount` 和 `modCount` 是否相等，如果不相等就会抛出 `ConcurrentModificationException` ，也就是 fail-fast。

继续查看源码发现 `modCount` 在 `add, clear, remove` 中都会被修改：

```
public boolean add(E object) {
    //...
    modCount++;
    return true;
}

public void clear() {
    if (size != 0) {
        //...
        modCount++;
    }
}

public boolean remove(Object object) {
    Object[] a = array;
    int s = size;
    if (object != null) {
        for (int i = 0; i < s; i++) {
            if (object.equals(a[i])) {
                //...
                modCount++;
                return true;
            }
        }
    } else {
        for (int i = 0; i < s; i++) {
            if (a[i] == null) {
                //...
                modCount++;
                return true;
            }
        }
    }
    return false;
}
```

- **因此在使用 `Iterator` 进行遍历时，不能调用容器修改的操作——增删改查等，会令迭代器失效。**
- 但同时我们可以仅通过 `Iterator.next(), Iterator.remove()` 方法在遍历时删除元素，因为并不会影响 `modCount` 。

- 而且每调用一次`Iterator.next()`方法，`Iterator.remove()`方法只能被调用一次，如果违反这个规则将抛出一个异常`IllegalStateException`。

7.3. ListIterator

`ListIterator`是一个功能更加强大的，它继承于`Iterator`接口,**只能用于各种`List`类型的访问。**

- 可以通过调用`listIterator()`方法产生一个指向`List`开始处的`ListIterator`。
- 还可以调用`listIterator(n)`方法创建一个一开始就指向列表索引为`n`的元素处的`ListIterator`。

根据官方文档介绍，`ListIterator` 有以下功能：

1. 允许我们向前、向后两个方向遍历 List;
2. 在遍历时修改 List 的元素；
3. 遍历时获取迭代器当前游标所在位置。

7.3.1. ListIterator主要方法

```
void add(E e)                  //在游标左边插入一个元素，注意，是左边
boolean hasNext()              //判断是否有下一个可迭代元素
E next()                      //返回游标当前位置右边的元素并将游标移动到下一个位置
int nextindex()                //返回游标右边元素的索引位置，初始为 0 ；遍历 N 个元素结束时为 N
boolean hasPrevious()          //判断游标前面是否有元素
E previous()                    //返回游标当前位置右边的元素并将游标移动到上一个位置
int previousindex()            //返回游标前面元素的位置，初始时为-1游标没发生移动的话会抛出异常
void remove ()                 //删除迭代器最后一次操作的元素为 E，也就是更新最近一次调用 next() 或者 previous() 返回的元素。
void set(E e)                  //更新迭代器最后一次操作的元素为e，也就是更新最近一次调用 next() 或者 previous() 返回的元素。
```

注意：

- 由于初始状态下`cursor=-1`， 在没有开始迭代之前调用 `previous()` 、`previousIndex()` 、`set(E)` 、`remove()` 会抛出异常。
- `add()`、`next()`、`previous()`、`remove()`、`set()` 都遵循fail-fast机制 因此，若实现列表元素的修改请使用ListIterator提供的修改方法。