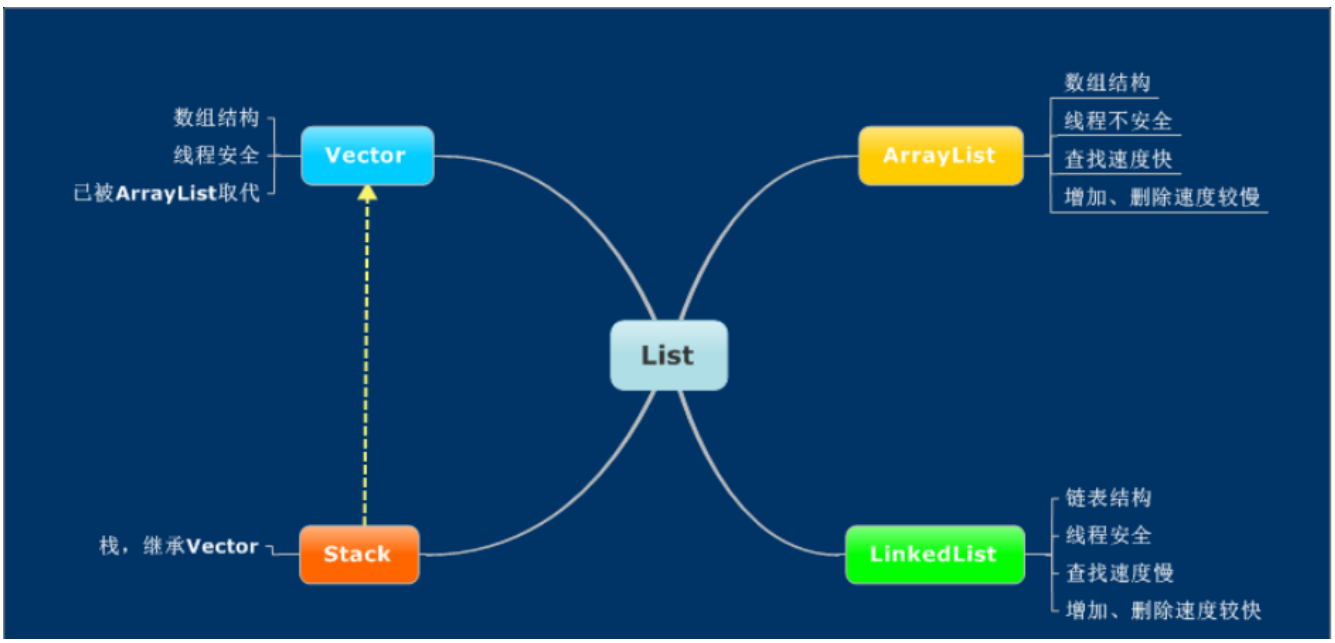
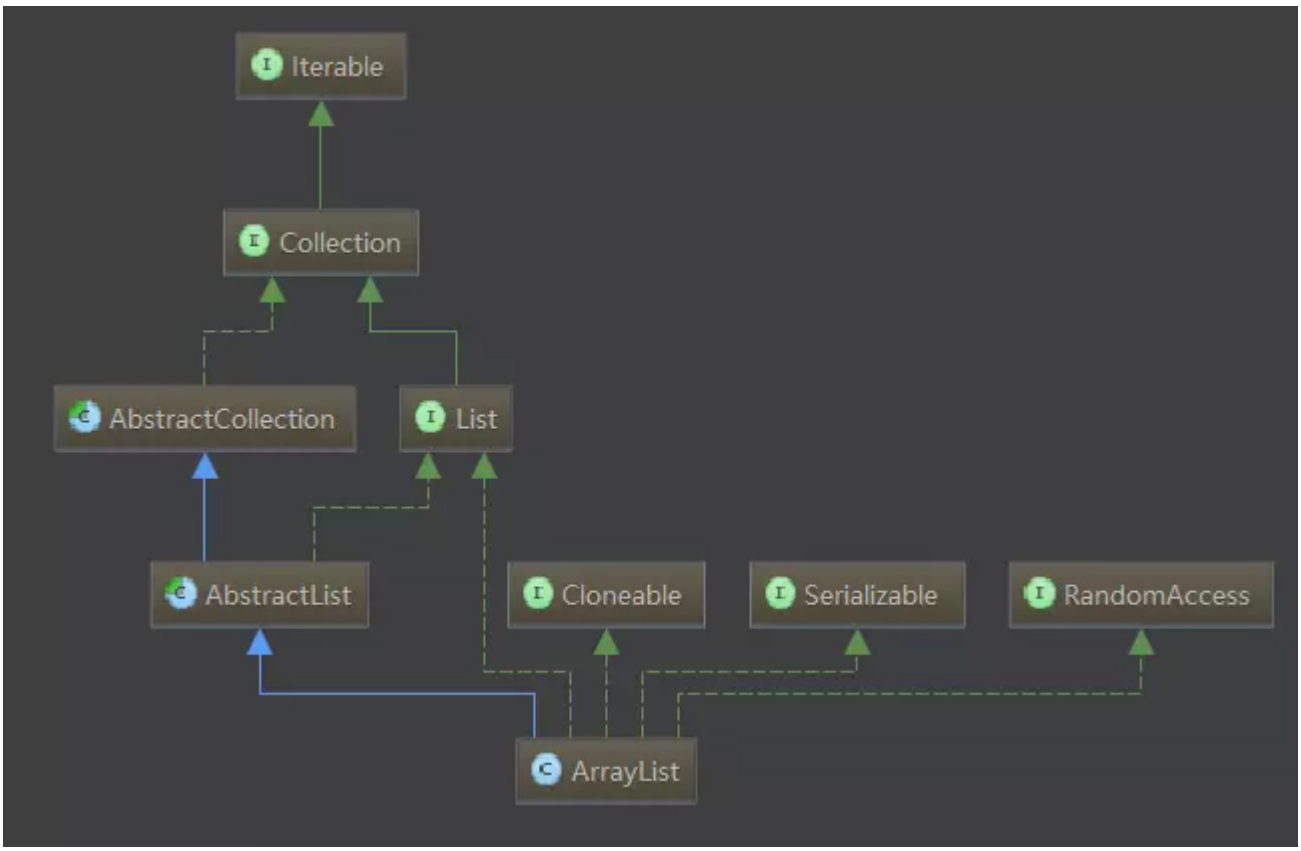
List实现类



这里主要讲解`List`接口的主要实现类，了解它们的底层实现，数据结构以及一些常用方式。

1. ArrayList



蓝色线条：继承 绿色线条：接口实现

`ArrayList` 是一个**数组列表**，相当于 **动态数组**。与Java中的数组相比，它的容量能动态增长。

它继承于`AbstractList`，实现了`List`，`RandomAccess`，`Cloneable`，`java.io.Serializable`这些接口。

- `ArrayList` 继承了`AbstractList`，实现了`List`。它是一个数组列表，提供了相关的添加、删除、修改、遍历等功能。
- `ArrayList` 实现了`RandmoAccess`接口，即提供了随机访问功能。`RandmoAccess`是java中用来被List实现，为`List`**提供快速访问功能**的。在`ArrayList`中，我们即可以通过元素的序号快速获取元素对象（通过`get()`方法，注意：Java没有重载`[]`）；这就是快速随机访问。稍后，我们会比较List的“快速随机访问”和“通过`Iterator`迭代器访问”的效率。
- `ArrayList` 实现了`Cloneable`接口，即实现了函数`clone()`，能被克隆。
- `ArrayList` 实现`java.io.Serializable`接口，这意味着`ArrayList`支持序列化，能通过序列化去传输。
- 和`Vector`不同，`ArrayList`**中的操作不是线程安全的**！所以，建议在单线程中才使用`ArrayList`，而在多线程中可以选择`Vector`或者`CopyOnWriteArrayList`。

1.1. ArrayList的主要成员变量

`ArrayList`底层时数组，数据结构就是动态数组。 因此其中包含了两个重要对象一个数组，以及数组的元素数量

```
// 保存ArrayList中数据的数组，会动态增长
private transient E[] elementData;
// ArrayList中实际数据的数量
private int size;
```

1.2. ArrayList的构造方法

```
public ArrayList(int initialCapacity)  //ArrayList指定初始容量大小的构造函数。
public ArrayList()                     // ArrayList无参构造函数。默认容量是10。
ArrayList(Collection<? extends E> collection)  // 创建一个包含collection的ArrayList
```

1.3. ArrayList的主要方法

Collection与List接口方法

```
//ArrayList新增的方法
// 由于ArrayList支持随机存取，因此增加了许多支持索引的放啊。
void ensureCapacity( int minCapacity)        //数组容量检查，不够时则进行扩容
void forEach(Consumer<? super E> action)     //对ArrayList的每个元素执行给定的操作，直到所有元素都被处理或动作引发异常。
void     removeRange(int fromIndex, int toIndex)//删除索引范围[fromIndex,toIndex)的元素
void     trimToSize()                          //将底层数组的容量调整为当前实际元素的大小，来释放空间。
```

注意：

- `ArrayList`在添加元素时会进行数组越界检查，如果当前容量不够会自动扩容，扩容的过程就是数组拷贝`System.arraycopy()`的过程，每一次扩容就会开辟一块新的内存空间和数据的复制移动，这样势必对性能造成影响。 那么在这种以写为主（**写会扩容，删不会缩容**）场景下，**提前预知性的设置一个大容量——`ArrayList(int initialCapacity)`，或是主动进行容量扩充——`ensureCapacity( int minCapacity)`便可减少扩容的次数，提高了性能。**
- 如果担心因扩容而造成的内存浪费行为，可以通过`trimToSize()`将底层数组的容量调整为当前实际元素的大小，来释放空间。

1.4. 遍历效率分析

1.4.1. 迭代器遍历

通过`Iterator/ListIterator`遍历

```
Integer value = null;
Iterator iter = list.iterator();
while (iter.hasNext()) {
    value = iter.next();
}
```

1.4.2. for循环随机访问，通过索引值去遍历。

```
Integer value = null;
int size = list.size();
for (int i=0; i<size; i++) {
    value = list.get(i);
}
```

1.4.3. for-each循环遍历

```
Integer value = null;
for (Integer integ:list) {
    value = integ;
}
```

1.4.4. 效率分析

遍历`ArrayList`时，使用for循环随机访问(即，通过索引序号访问)效率最高，而使用迭代器的效率最低！，并且尽量减少函数调用使用`size`常量代替`size()`方法

时间差异比较

```
compare loop performance of ArrayList

\-------------------------------------------------------------------

list size              | 10,000    | 100,000   | 1,000,000 | 10,000,000

\-for (Integer j : list)-----------------------------------------------

                     | 1 ms      | 3 ms      | 14 ms     | 152 ms

\-for (Iterator<Integer> iterator = list.iterator(); iterator.hasNext();)-

                     | 0 ms      | 1 ms      | 12 ms     | 114 ms

\--for (int j = 0; j < list.size(); j++)----------------------👆迭代器，👇索引器
```

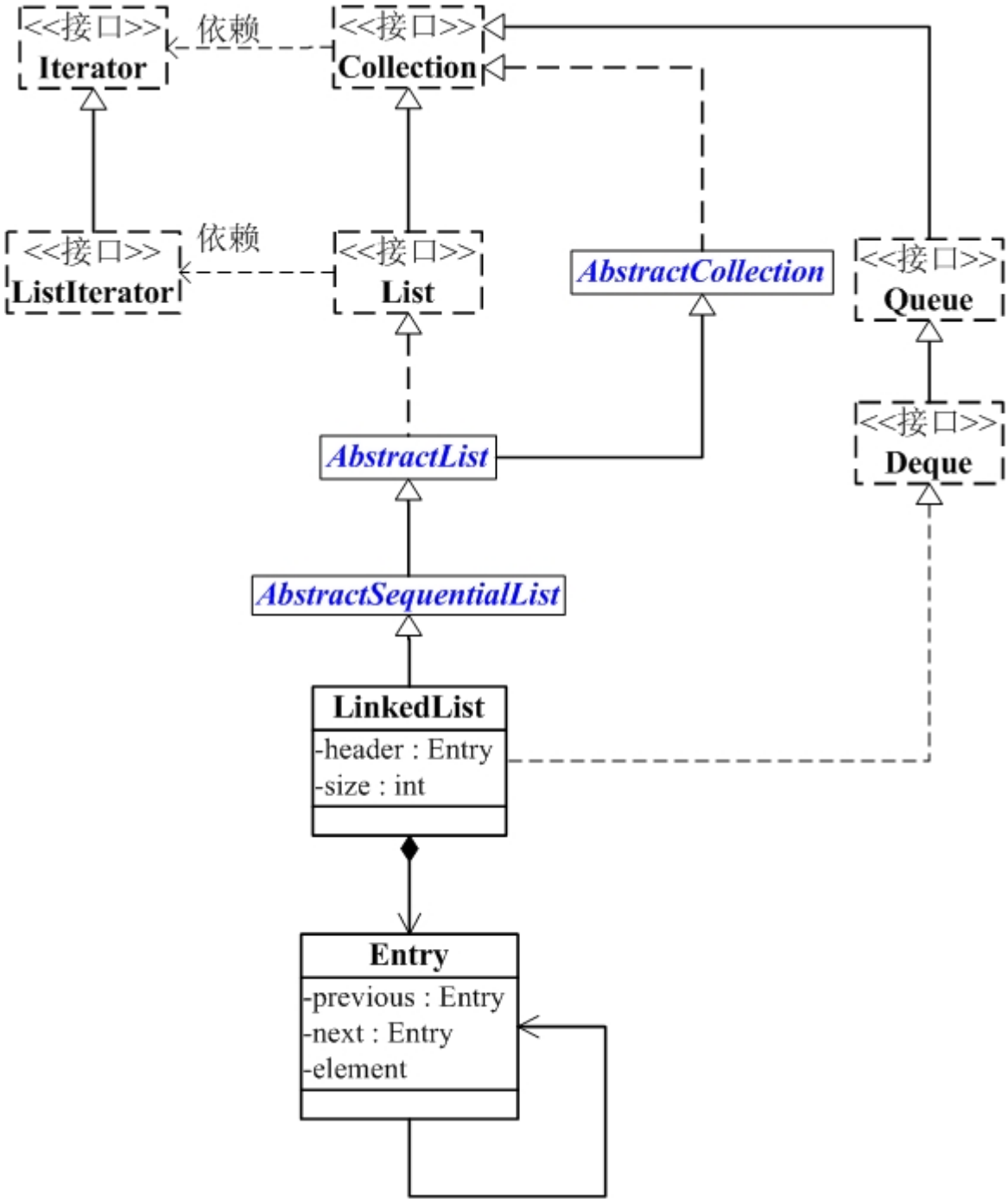
```
                      | 1 ms      | 1 ms      | 13 ms     | 128 ms

\---for (int j = 0; j < size; j++)--------------------------------

                     | 0 ms      | 0 ms      | 6 ms      | 62 ms

\--for (int j = list.size() - 1; j >= 0; j--) ----------------------

                       | 0 ms      | 1 ms      | 6 ms      | 63 ms
```

1.5. ArrayList和Vector的区别

- `Vector`和`ArrayList`几乎是完全相同的,唯一的区别在于`Vector`是同步类(synchronized)，属于强同步类。因此开销就比`ArrayList`要大，访问要慢。正常情况下,大多数的Java程序员使用`ArrayList`而不是`Vector`,因为同步完全可以由程序员自己来控制。
- `Vector`每次扩容请求其大小的2倍空间，而`ArrayList`是1.5倍。
- `Vector`还有一个子类`Stack`.

2. LinkedList

`LinkedList`与`ArrayList`一样实现List接口，只是`ArrayList`是`List`接口的大小可变数组的实现，`LinkedList`是`List`接口**链表**的实现。基于链表实现的方式使得`LinkedList`在插入和删除时更优于`ArrayList`，而随机访问则比`ArrayList`逊色些。



蓝色线条：继承 绿色线条：接口实现

`LinkedList`的定义

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

`LinkedList` 是一个继承于`AbstractSequentialList`的**双向循环链表。**由于实现了`Deque`接口因此它也可以被当作堆栈、队列或双端队列进行操作，但是并不推荐，更好的选择是`ArrayDeque`类。

- `LinkedList` 实现 `List` 接口，能对它进行列表操作。

- `LinkedList`是`AbstractSequentialList`的子类，该抽象类提供了随机访问的方法，但是由于`LinkedList`是链表随机访问的效率不高。
- `LinkedList` 实现 `Deque` 接口，即能将`LinkedList`当作双端队列使用。
- `LinkedList` 实现了`Cloneable`接口，即覆盖了函数`clone()`，能克隆。
- `LinkedList` 实现`java.io.Serializable`接口，这意味着`LinkedList`支持序列化，能通过序列化去传输。
- `LinkedList` 是非同步的。

2.1. LinkedList的成员变量

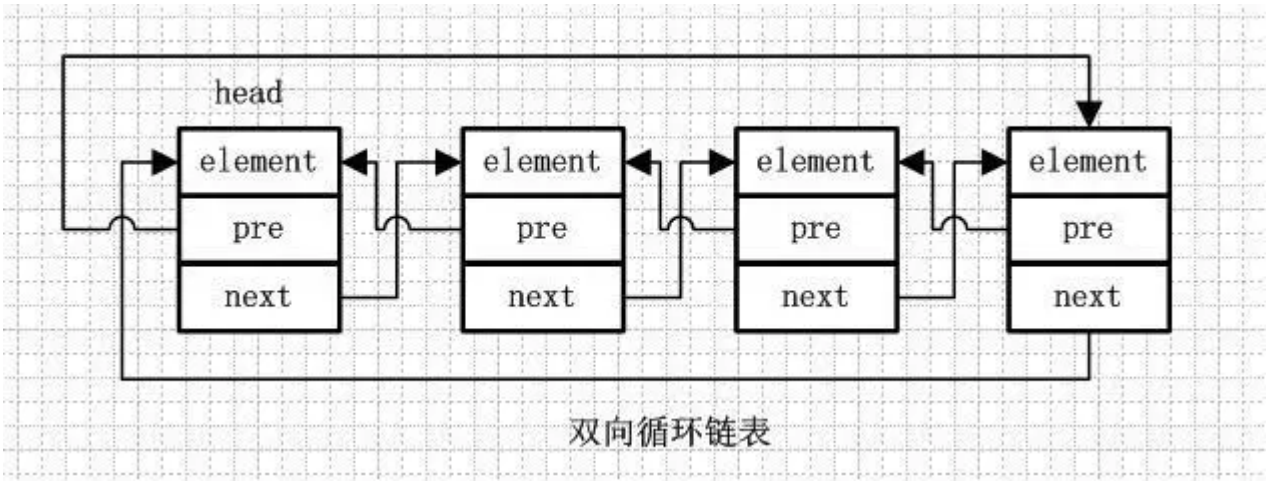
```
private transient Entry<E> header = new Entry<E>(null, null, null);//链表头节点
private transient int size = 0;                 //记录元素数目
private static class Entry<E> {                 //内部类，链表节点
    E element;  // 当前存储元素
    Entry<E> next;  // 下一个元素节点
    Entry<E> previous;  // 上一个元素节点
    Entry(E element, Entry<E> next, Entry<E> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
```

典型的双向链表。

2.2. LinkedList的构造方法

```
/**
* 构造一个空的LinkedList .
*/
public LinkedList() {
    //将header节点的前一节点和后一节点都设置为自身
    header.next = header. previous = header ;
}
/**
* 构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列
*/
public LinkedList(Collection<? extends E> c) {
    this();
    addAll(c);
}
```

需要注意的是空的`LinkedList`构造方法，它将`header`节点的前一节点和后一节点都设置为自身，这里便说明`LinkedList`是一个**双向循环链表**



2.3. LinkedList的主要方法

Collection与List接口方法

Queue与Deque接口方法

由于`LinkedList`实现了`Collection`、`List`、`Queue`、`Deque`等接口，这些接口方法不再赘述。

我们主要将`LinkedList`作为链表使用，因此应该首先调用`List`接口定义的方法。

```
Iterator<E>     descendingIterator()            //返回反向遍历的Iterator
boolean         removeFirstOccurrence(E e)      //从LinkedList开始向后查找，删除第一个值为元素(o)的节点
boolean         removeLastOccurrence(E e)       // 从LinkedList末尾向前查找，删除第一个值为元素(o)的节点
```

2.4. 遍历效率分析

`LinkedList`使用迭代器的方式遍历。 不过一般使用`LinkedList`尽量避免遍历

```
compare loop performance of LinkedList
-------------------------------------------------------------------------
list size               | 100       | 1,000     | 10,000    | 100,000
\-for (Integer j : list)-----------------------------------------------
                        | 0 ms      | 1 ms      | 1 ms      | 2 ms
\-for (Iterator<Integer> iterator = list.iterator(); iterator.hasNext();)-
                        | 0 ms      | 0 ms      | 0 ms      | 2 ms
\--for (int j = 0; j < list.size(); j++)----------------------👆迭代器，👇索引器
                        | 0 ms      | 1 ms      | 73 ms     | 7972 ms
```

```
\---for (int j = 0; j < size; j++)--------------------------------
                     | 0 ms      | 0 ms      | 67 ms     | 8216 ms
\--for (int j = list.size() - 1; j >= 0; j--) ----------------------
                     | 0 ms      | 1 ms      | 67 ms     | 8277 ms
-----------------------------------------------------------------------
```

注意，当数据足够大的时候迭代器的效率会优于foreach

3. Collection与List接口方法

```
//继承自Collection的方法

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(E o)                   //返回Collection中是否包含指定元素
boolean containsAll(Collection<?> c)      //返回Collection中是否包含Collectionc中的所有元素
int hashCode()                            //返回此Collection的哈希码值。
boolean isEmpty()                         //返回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(E o)                        //删除Collection中与指定元素匹配的第一个元素
boolean removeAll(Collection<?> c)         //删除c中的所有对象
default boolean removeIf(Predicate<? super E> filter) //删除此Collection中满足给定谓词的所有元素
boolean retainAll(Collection<?> c)         //仅保留c中的对象
int size()                                //返回Collection里元素的个数
E[] 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

 //************************************************************************

//继承自AbstractCollection的方法
String toString()                         //返回此容器的字符串表示形式。

 //************************************************************************

 //继承自List接口的方法
void    add(int index, E e)                  //在指定位置添加元素
boolean addAll(int index, Collection<? extends E> collection)//在指定位置添加collection的所有元素
```

```
E   get(int index)                          //获取指定位置的元素
int indexOf(E e)                            //遍历数组返回第一个目标元素的下标，若未找到返回-1
int lastIndexOf(E e)                        //遍历数组返回最后一个目标元素的下标，未找到返回-1
ListIterator<E> listIterator()              //返回起始位置开始的ListIterator对象
ListIterator<E> listIterator(int index)     //返回指定位置开始的ListIterator对象
E   remove(int index)                       //根据索引位置删除元素
void replaceAll(UnaryOperator<E> operator)  //将该List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 e)                     //将指定位置的元素更新为新元素
void sort(Comparator<? super E> c)          //使用提供的 Comparator对此List进行排序。
List<E> subList(int fromIndex, int toIndex) //返回下标范围[fromIndex,toIndex)的元素视图
```

4. Queue与Deque接口方法

注意，下列方法失败会抛出异常

`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
```

`Deque`支持对双端队列行为的支持（未介绍线程相关的方法）：

```
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
```

5. 源码参考

Java 集合深入理解（5）： AbstractCollection

Java集合干货系列-（一）ArrayList源码解析

Java 集合系列05之 LinkedList详细介绍(源码解析)和使用示例