有序、有索引、可存储重复元素的容器。List接口常用的实现类有3个：ArrayList、LinkedList和Vector(过时)。

List多了跟顺序(索引)有关的方法：

表9-2List接口中定义的方法

| 方法 | 说明 |
| --- | --- |
| void add (int index, Object element) | 在指定位置插入元素，以前元素全部后移一位 |
| Object set (int index,Object element) | 修改指定位置的元素 |
| Object get (int index) | 返回指定位置的元素 |
| Object remove (int index) | 删除指定位置的元素，后面元素全部前移一位 |
| int indexOf (Object o) | 返回第一个匹配元素的索引，如果没有该元素，返回-1. |
| int lastIndexOf (Object o) | 返回最后一个匹配元素的索引，如果没有该元素，返回-1 |

Java 8还为List接口添加了如下两个默认方法：

void replaceAll(UnaryOperator operator):根据operator指定的计算规则重新设置List集合的所有元素。

void sort(Comparator c):根据Comparator参数对List集合的元素排序。

List判断两个对象相等只要通过equals()方法比较返回true即可

- **ArrayList**

ArrayList是一个数组队列，底层用数组实现。封装了一个动态的、允许再分配的Object[]数组。initalCapacity参数可设置数组的长度。当向ArrayList或Vector中添加元素超过了该数组的长度时，它们的initalCapacity会自动增加。

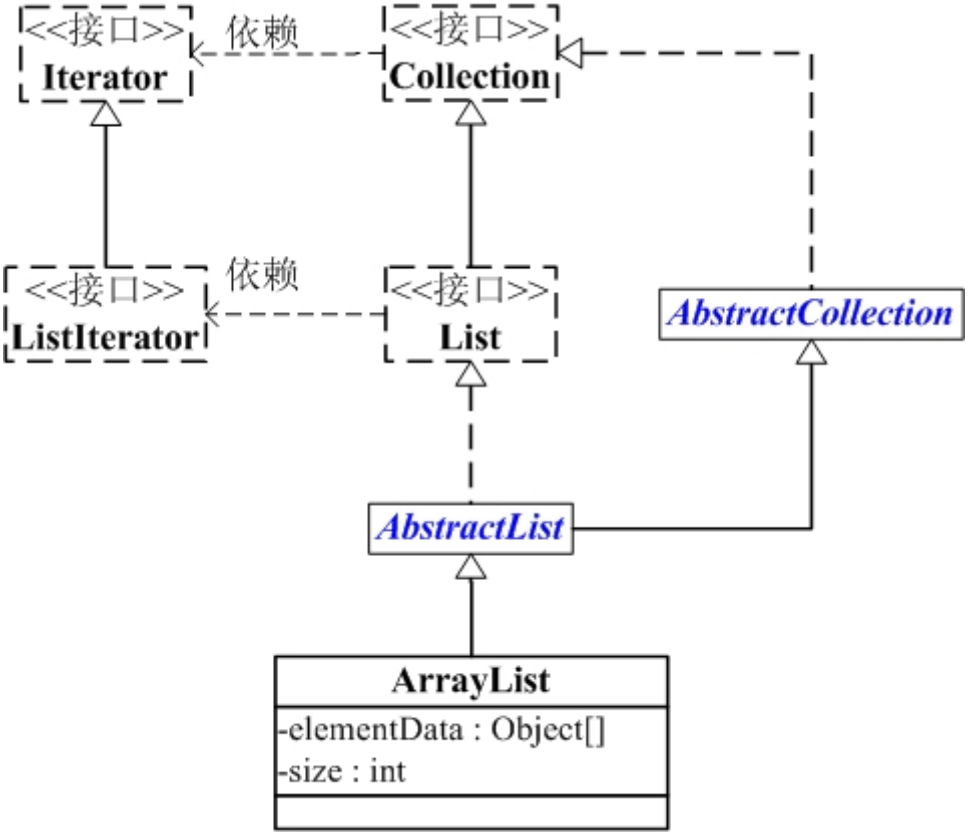
特点：查询效率高，增删效率低，线程不安全。一般使用它。

**遍历ArrayList**：使用随机访问(即，通过索引序号访问)效率最高，而使用迭代器的效率最低。

ArrayList *实现了RandmoAccess接口（*java中用来被List实现*）即提供了随机访问功能。通过元素序号快速获取元素对象。*

ArrayList 实现了Cloneable接口　　即覆盖了函数clone()，能被克隆。

ArrayList 实现java.io.Serializable接口　　支持序列化，能通过序列化去传输。当写入到输出流时，先写入"容量"，再依次写入"每一个元素"；当读出输入流时，先读取"容量"，再依次读取"每一个元素"。




elementData 是"Object[]类型的数组"，它保存添加到ArrayList中的元素。

size 则是动态数组的实际大小。默认为10。容量不足时：

新的容量="(原始容量x3)/2 + 1"。

## 源码分析

**toArray异常**：

ArrayList提供了2个toArray()函数：

```
Object[] toArray()    //用到了copyOf()方法
<T> T[] toArray(T[] contents)
```

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.ClassCast Exception”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)， 而不是 Object[] toArray()。

**删除元素**

调用 System.arraycopy() 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)。代价高。在add（）方法中也使用了。

```
    /**
     * 在此列表中的指定位置插入指定的元素。
     *先调用 rangeCheckForAdd 对index进行界限检查；然后调用 ensureCapacityInternal 方法保证capacity足够大；
     *再将从index开始之后的所有成员后移一个位置；将element插入index位置；最后size加1。
     */
    public void add(int index, E element) {
        rangeCheckForAdd(index);

        ensureCapacityInternal(size + 1);  // Increments modCount!!
        //arraycopy()方法实现数组自己复制自己
        //elementData:源数组;index:源数组中的起始位置;elementData：目标数组；index + 1：目标数组中的起始位置；  size - index：要复制的数组元素的数量;
        System.arraycopy(elementData, index, elementData, index + 1, size - index);
        elementData[index] = element;
        size++;
    }
```

copyOf()内部也调用了System.arraycopy()方法 区别：

1. arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置
2. copyOf()是系统自动在内部新建一个数组，并返回该数组。

Fail-Fast

modCount 记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，或者是调整内部数组的大小，仅仅只是设置元素的值不算结构发生变化。

在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。

序列化

ArrayList 基于数组实现，并且具有动态扩容特性，因此保存元素的数组 elementData不一定都会被使用，那就没必要全部进行序列化。

```
transient Object[] elementData; // non-private to simplify nested class access
```

ArrayList 实现了 writeObject() 和 readObject() 来控制只序列化数组中有元素填充那部分内容。

- **LinkedList**

LinkedList底层用双向链表实现的存储。使LinkedList变成线程安全，可以调用静态类Collections类中的synchronizedList方法：

```
List list=Collections.synchronizedList(new LinkedList(...));
```

特点：查询效率低，增删效率高，线程不安全。

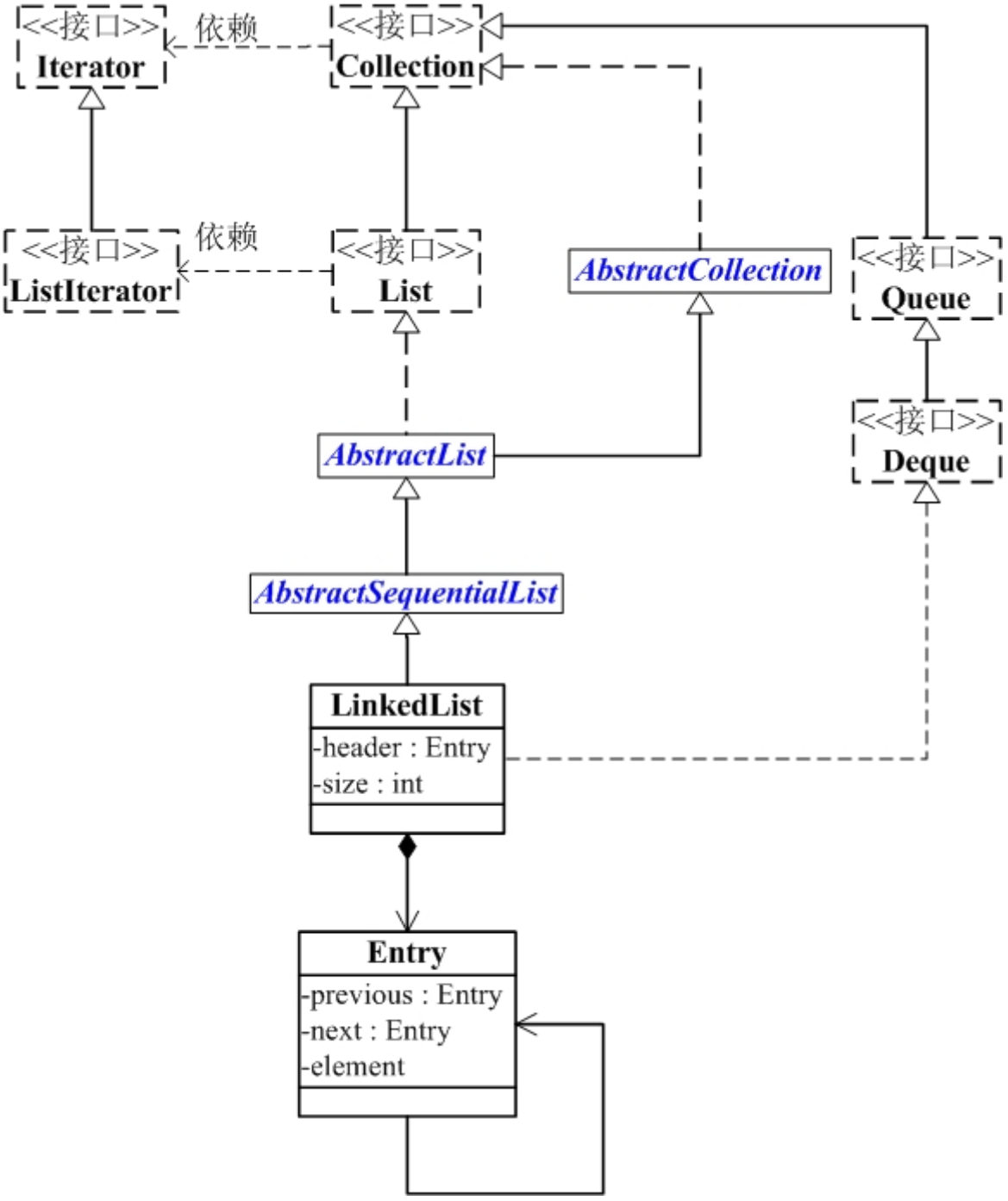
**遍历LinkedList**：使用removeFist()或removeLast()效率最高。但会删除原始数据；若单纯只读取，而不删除，应该使用加强for循环遍历方式。不支持随机访问。遍历：使用索引的随机访问方式最快，使用迭代器最慢。

继承于AbstractSequentialList实现List，Deque，Cloneable，java.io.Serializable接口。

*Deque接口*定义了在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（null 或 false，具体取决于操作）。

若需要通过*AbstractSequentialList*自己实现一个列表，只需要扩展此类，并提供 listIterator() 和 size() 方法的实现即可。

LinkedList可以作为**FIFO**(先进先出)的队列，也可以作为**LIFO**(后进先出)的栈。




**header**是双向链表的表头，它是双向链表节点所对应的类Entry的实例。

内部类：**Entry**。Entry是**双向链表节点所对应的数据结构**

## 源码分析

### add方法

add(E e)　方法：将元素添加到链表尾部

```
public boolean add(E e) {
        linkLast(e);//这里就只调用了这一个方法
```

```
        return true;
    }
    /**
     * 链接使e作为最后一个元素。
     */
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;//新建节点
        if (l == null)
            first = newNode;
        else
            l.next = newNode;//指向后继元素也就是指向下一个元素
        size++;
        modCount++;
    }
```

## add(int index,E e)：在指定位置添加元素

```
public void add(int index, E element) {
        checkPositionIndex(index); //检查索引是否处于[0-size]之间

        if (index == size)//添加在链表尾部
            linkLast(element);
        else//添加在链表中间
            linkBefore(element, node(index));
    }
```

## addAll(Collection c )：将集合插入到链表尾部

```
public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
    }
```

addAll(int index, Collection c)：  将集合从指定位置开始插入

1. 检查index范围是否在size之内
2. toArray()方法把集合的数据存到对象数组中
3. 得到插入位置的前驱和后继节点
4. 遍历数据，将数据插入到指定位置

## addFirst(E e)：  将元素添加到链表头部

```
public void addFirst(E e) {
        linkFirst(e);
    }
```

```
private void linkFirst(E e) {
        final Node<E> f = first;
        final Node<E> newNode = new Node<>(null, e, f);//新建节点，以头节点为后继节点
        first = newNode;
        //如果链表为空，last节点也指向该节点
        if (f == null)
            last = newNode;
        //否则，将头节点的前驱指针指向新节点，也就是指向前一个元素
        else
            f.prev = newNode;
        size++;
        modCount++;
    }
```

addLast(E e)： 将元素添加到链表尾部，与 add(E e) 方法一样

```
public void addLast(E e) {
        linkLast(e);
    }
```

## 根据位置取数据的方法

get(int index)： 根据指定索引返回数据

```
public E get(int index) {
        //检查index范围是否在size之内
        checkElementIndex(index);
        //调用Node(index)去找到index对应的node然后返回它的值
        return node(index).item;
    }
```

获取头节点（index=0）数据方法:

```
public E getFirst() {
        final Node<E> f = first;
        if (f == null)
            throw new NoSuchElementException();
        return f.item;
    }
public E element() {
        return getFirst();
    }
public E peek() {
        final Node<E> f = first;
        return (f == null) ? null : f.item;
    }

public E peekFirst() {
        final Node<E> f = first;
```

```
        return (f == null) ? null : f.item;
    }
```

**区别：** getFirst(),element(),peek(),peekFirst() 这四个获取头结点方法的区别在于对链表为空时的处理，是抛出异常还是返回null，其中getFirst() 和element() 方法将会在链表为空时，抛出异常。

获取尾节点（index=-1）数据方法:

```
public E getLast() {
        final Node<E> l = last;
        if (l == null)
            throw new NoSuchElementException();
        return l.item;
    }
public E peekLast() {
        final Node<E> l = last;
        return (l == null) ? null : l.item;
    }
```

**区别：** getLast() 方法在链表为空时，会抛出NoSuchElementException，而peekLast() 则不会，只是会返回 null。

**根据对象得到索引的方法**

int indexOf(Object o)： 从头遍历找

int lastIndexOf(Object o)： 从尾遍历找

**检查链表是否包含某对象的方法：**

contains(Object o)： 检查对象o是否存在于链表中

**删除方法**

remove() ,removeFirst(),pop()： 删除头节点

removeLast(),pollLast()： 删除尾节点

remove(Object o)： 删除指定元素

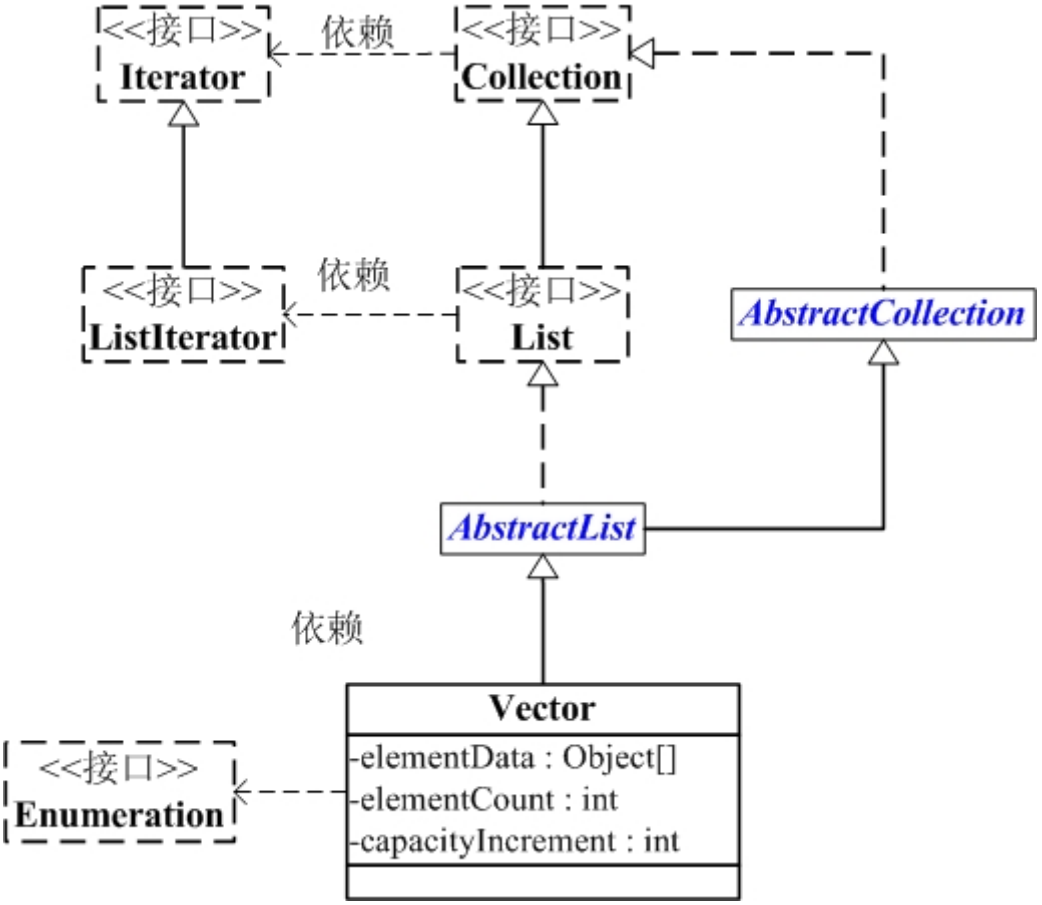
remove(int index)：删除指定位置的元素

如何将“**双向链表和索引值联系起来的**”？（随机访问）
通过一个**计数索引值**来实现的。例如，当我们调用get(int location)时，首先会比较“location”和“双向链表长度的1/2”；若前者大，则从链表头开始往后查找，直到location位置；否则，从链表末尾开始先前查找，直到location位置。

- **Vector**

“线程安全,效率低”。底层是用数组实现，是矢量队列，Vector的数据结构和ArrayList差不多，但使用了 synchronized 进行同步 。
eg：indexOf方法就增加了synchronized同步标记。它还有一种迭代器通过vector.elements()获取，判断是否有元素和取元素的方法为：hasMoreElements()，nextElement()。

继承于AbstractList，实现了List, RandomAccess, Cloneable这些接口。




elementData 是"Object[]类型的数组"，elementCount 是动态数组的实际大小，capacityIncrement 是动态数组的增长系数。

- **Stack类**

Vector的一个子类，意味Stack也是数组实现，而非链表。一个标准的先进后出（FILO）的栈。

常用API：（Stack包含Vector的全部API和属性）Stack只定义了一个默认构造函数，用来创建一个空栈。

| 序号 | 方法描述 |
| --- | --- |
| 1 | boolean empty()<br>测试堆栈是否为空。 |
| 2 | Object peek( )<br>查看堆栈顶部的对象，但不从堆栈中移除它。 |
| 3 | Object pop( )<br>移除堆栈顶部的对象，并作为此函数的值返回该对象。 |
| 4 | Object push(Object element)<br>把项压入堆栈顶部。 |
| | |

| | |
|---|---|
| 5 | int search(Object element)<br>返回对象在堆栈中的位置，以 1 为基数。 |

执行**push**时(即，**将元素推入栈中**)，是通过将元素追加的数组的末尾中。

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**)，是取出数组末尾的元素，然后将该元素从数组中删除。

**总结：**

1. 需要线程安全时，用Vector或者CopyOnWriteArrayList。
2. 不存在线程安全问题时，并且查找较多用ArrayList(一般使用它)。
3. 不存在线程安全问题时，增加或删除元素较多用LinkedList。