- o

**学习目标**

- **ArrayList集合底层数据结构介绍**
- **ArrayList继承关系**
- **ArrayList源码分析**
- **面试题讲解**
- **自定义ArrayList**

# 1. ArrayList集合底层数据结构

1. **ArrayList集合介绍**

   `List`接口的可调整大小的数组实现。

   数组:一旦初始化长度就不可以发生改变

2. **数组结构介绍**

- 增删慢：每次删除元素，都需要更改数组长度、拷贝以及移动元素位置。
- 查询快：由于数组在内存中是一块连续空间，因此可以根据地址+索引的方式快速获取对应位置上的元素。

# 2. ArrayList继承关系

## 2.1 Serializable标记性接口

1. **介绍** 类的序列化由实现java.io.Serializable接口的类启用。 不实现此接口的类将不会使任何状态序列化或反序列化。 可序列化类的所有子类型都是可序列化的。 序列化接口没有方法或字段，仅用于标识可串行化的语义。

   序列化:将对象的数据写入到文件(写对象)

   反序列化:将文件中对象的数据读取出来(读对象)

2. **Serializable**源码介绍

```
public interface Serializable {
}
```

> 案例: 通过序列化流序列化和反序列化集合

```
/**
 * @author itheima
 * @since JDK 1.8
 */
public class Student implements Serializable{
    private static final long serialVersionUID = 1014100089306623762L;

    //姓名
```

```
    private String name;
    //年龄
    private Integer age;

    public Student() {
    }

    public Student(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        //优化toString方法
        StringBuilder sb = new StringBuilder();
        sb.append("[").append("name = ").append(this.name).append(", ").append("age =
").append(this.age).append("]");
        return sb.toString();
    }
}

/**
 * @author itheima
 * @since JDK 1.8
 */
public class Test01 {
    public static void main(String[] args) throws Exception {
        Student s = new Student();
        System.out.println(s);
        //创建对象操作流 --> 序列化
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("MyTest\\obj.txt"));
        //创建集合,且添加学生对象
        ArrayList<Student>  list = new ArrayList<Student>();
        list.add(new Student("悔创阿里杰克马",51));

        list.add(new Student("会点一点长几颗",26));
```

```
        list.add(new Student("容颜老去蒋青青",32));
        list.add(new Student("将里最丑刘一飞",27));
        //将集合写入到文件
        oos.writeObject(list);

        //创建对象输入流 --> 反序列化
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("MyTest\\obj.txt"));
        //读取数据
        Object o = ois.readObject();
        //向下转型
        ArrayList<Student> al = (ArrayList<Student>) o;
        //遍历集合
        for (int i = 0; i < al.size(); i++) {
            //根据索引取出集合的每一个元素
            Student stu = al.get(i);
            System.out.println(stu);
        }
    }
}
```

## 2.2 Cloneable 标记性接口

1. **介绍** 一个类实现 `Cloneable` 接口来指示 `Object.clone()` 方法，该方法对于该类的实例进行字段的复制是合法的。在不实现 `Cloneable` 接口的实例上调用对象的克隆方法会导致异常 `CloneNotSupportedException` 被抛出。**简言之:克隆就是依据已经有的数据，创造一份新的完全一样的数据拷贝**
2. **Cloneable**源码介绍

```
public interface Cloneable {
}
```

3. 克隆的前提条件
   - 被克隆对象所在的类必须实现 `Cloneable` 接口
   - 必须重写 `clone` 方法
4. clone的基本使用

```
/**
 * @author 长沙黑马程序员
 * @since JDK 1.8
 *
 * 克隆的基本使用:
 *         将ArrayList集合的数据clone到另外一个集合
 */
public class ArrayList_Clone {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("人生就是旅途");
        list.add("也许终点和起点会重合");
        list.add("但是一开始就站在起点等待终点");

        list.add("那么其中就没有美丽的沿途风景和令人难忘的过往");
```

```
        //调用方法进行克隆
        Object o = list.clone();
        System.out.println(o == list);
        System.out.println(o);
        System.out.println(list);
    }
}
```

5. clone源码分析

```
public class ArrayList<E> {
    public Object clone() {
        try {
            ArrayList<?> v = (ArrayList<?>) super.clone();
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e);
        }
    }
}
```

> 案例:已知`A`对象的姓名为豹子头林冲，年龄30 。由于项目特殊要求需要将该对象的数据复制另外一个对象`B`中，并且此后`A`和`B`两个对象的数据不会相互影响

> 案例:已知`A`对象的姓名为鲁智深，年龄30,技能为倒拔垂杨柳 (技能为一个引用数据类型`Skill`)，由于项目特殊要求需要将该对象的数据复制另外一个对象`B`中，并且此后`A`和`B`两个对象的数据不会相互影响

**方式一:**创建两个对象模拟

1. 准备学生类

```
public class Student implements Cloneable{
    //姓名
    private String name;
    //年龄
    private Integer age;
    public Student() {
    }

    public Student(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }


    public void setName(String name) {
```

```
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

2. 准备测试类

```
public class Test01 {
    public static void main(String[] args) throws CloneNotSupportedException {

        //创建学生对象
        Student stu1 = new Student("豹子头林冲",29);
        //再次创建一个新的学生对象
        Student stu2 = new Student();
        //把stu1对象name的值取出来赋值给stu2对象的name
        stu2.setName(stu1.getName());
        //把stu1对象age的值取出来赋值给stu2对象的age
        stu2.setAge(stu1.getAge());

        System.out.println(stu1 == stu2);
        System.out.println(stu1);
        System.out.println(stu2);

        System.out.println("----此时不管修改哪个对象的内容,stu1和stu2都不会受到影响----");
        stu1.setName("扈三娘");
        System.out.println(stu1);
        System.out.println(stu2);
    }
}
```

3. 控制台效果

> false Student{name='豹子头林冲', age=29} Student{name='豹子头林冲', age=29} ----此时不管修改哪个对象的内容,stu1和stu2都不会受到影响---- Student{name='扈三娘', age=29} Student{name='豹子头林冲', age=29}

**方式二:使用克隆**

- **浅克隆**

  1.定义Javabean类

  学生技能类

```
//学生的技能类
public class Skill implements Cloneable{
    private String skillName;

    public Skill() {
    }

    public Skill(String skillName) {
        this.skillName = skillName;
    }

    public String getSkillName() {
        return skillName;
    }

    public void setSkillName(String skillName) {
        this.skillName = skillName;
    }

    @Override
    public String toString() {
        return "Skill{" +
                "skillName='" + skillName + '\'' +
                '}';
    }
}
```

  学生类

```
public class Student implements Cloneable{
    //姓名
    private String name;
    //年龄
    private Integer age;
    //技能
    private Skill skill;

    public Student() {
    }

    public Student(String name, Integer age, Skill skill) {
        this.name = name;
        this.age = age;
        this.skill = skill;
    }

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Skill getSkill() {
        return skill;
    }
    public void setSkill(Skill skill) {
        this.skill = skill;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", skill=" + skill +
                '}';
    }
    //浅克隆
    @Override
    public Student clone() throws CloneNotSupportedException {
        return (Student) super.clone();
    }
}
```

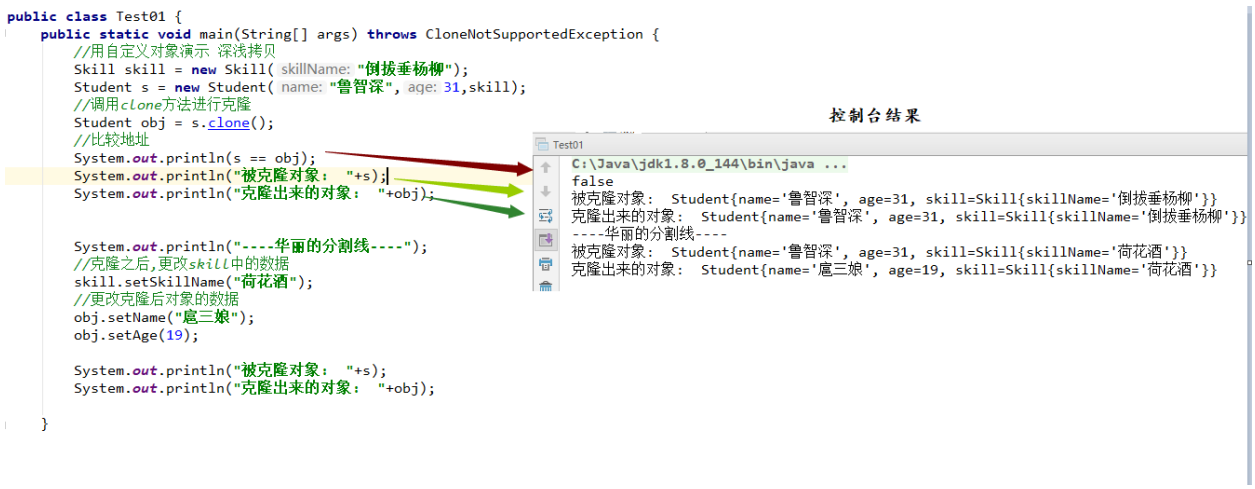
2.定义测试类

测试类

```
public class Test01 {
    public static void main(String[] args) throws CloneNotSupportedException {
        //用自定义对象演示 深浅拷贝
        Skill skill = new Skill("倒拔垂杨柳");
        Student s = new Student("鲁智深",31,skill);
        //调用clone方法进行克隆
        Student obj = s.clone();
        //比较地址
        System.out.println(s == obj);
        System.out.println("被克隆对象:  "+s);
        System.out.println("克隆出来的对象:  "+obj);

        System.out.println("----华丽的分割线----");
```

```
        //克隆之后,更改skill中的数据
        skill.setSkillName("荷花酒");
        //更改克隆后对象的数据
        obj.setName("扈三娘");
        obj.setAge(19);

        System.out.println("被克隆对象：  "+s);
        System.out.println("克隆出来的对象：  "+obj);

    }
}
```

控制台效果




**存在的问题:** 基本数据类型可以达到完全复制，引用数据类型则不可以

**原因:** 在学生对象s被克隆的时候，其属性skill(引用数据类型)仅仅是拷贝了一份引用，因此当skill的值发生改变时，被克隆对象s的属性skill也将跟随改变

- **深克隆**

1.定义Javabean类

学生技能类

```
/**
 * @author itheima
 * @since JDK 1.8
 */
public class Skill implements Cloneable{
    private String skillName;

    public Skill() {
    }

    public Skill(String skillName) {
        this.skillName = skillName;
    }

    public String getSkillName() {

        return skillName;
```

```
    }

    public void setSkillName(String skillName) {
        this.skillName = skillName;
    }

    @Override
    public String toString() {
        return "Skill{" +
                "skillName='" + skillName + '\'' +
                '}';
    }

    //重写克隆方法,将权限修饰符改成public
    @Override
    public Skill clone() throws CloneNotSupportedException {
        return (Skill) super.clone();
    }
}
```

学生类

```
/**
 * @author itheima
 * @since JDK 1.8
 */
public class Student implements Cloneable{
    //姓名
    private String name;
    //年龄
    private Integer age;
    //技能
    private Skill skill;

    public Student() {
    }

    public Student(String name, Integer age, Skill skill) {
        this.name = name;
        this.age = age;
        this.skill = skill;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


    public Integer getAge() {
```

```
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Skill getSkill() {
        return skill;
    }

    public void setSkill(Skill skill) {
        this.skill = skill;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", skill=" + skill +
                '}';
    }

    //深克隆
    @Override
    public Student clone() throws CloneNotSupportedException {
        //调用超类Object中方法clone进行对象克隆,得到一个新的学生对象
        Student newStu = (Student) super.clone();
        //调用学生类其属性skill的clone方法,对属性进行克隆
        Skill s = this.skill.clone();
        //再将克隆的Skill设置给克隆出来的学生对象
        newStu.setSkill(s);
        //返回克隆出来的学生对象
        return newStu;
    }

}
```

2.定义测试类

测试类

```
public class Test01 {
    public static void main(String[] args) throws CloneNotSupportedException {
        //用自定义对象演示 深浅拷贝
        Skill skill = new Skill("倒拔垂杨柳");
        Student s = new Student("鲁智深",31,skill);
        //调用clone方法进行克隆
        Student obj = s.clone();
        //比较地址
        System.out.println(s == obj);

        System.out.println("被克隆对象:  "+s);
```

```
        System.out.println("克隆出来的对象:  "+obj);
        System.out.println("----华丽的分割线----");
        //克隆之后,更改skill中的数据
        skill.setSkillName("荷花酒");
        //更改克隆后对象的数据
        obj.setName("扈三娘");
        obj.setAge(19);
        System.out.println("被克隆对象:  "+s);
        System.out.println("克隆出来的对象:  "+obj);
    }
}
```

控制台效果


## 2.3 RandomAccess标记接口

1. **介绍** 标记接口由`List`实现使用，以表明它们支持快速（通常为恒定时间）随机访问。

   此接口的主要目的是允许通用算法更改其行为，以便在应用于**随机访问列表**或**顺序访问列表**时提供良好的性能。

   用于操纵随机访问列表的最佳算法（例如`ArrayList`）可以在应用于顺序访问列表时产生二次行为（如`LinkedList`）。 鼓励通用列表算法在应用如果将其应用于顺序访问列表之前提供较差性能的算法时，检查给定列表是否为`instanceof`，并在必要时更改其行为以保证可接受的性能。

   人们认识到，随机访问和顺序访问之间的区别通常是模糊的。 例如，一些`List`实现提供渐近的线性访问时间，如果它们在实践中获得巨大但是恒定的访问时间。 这样的一个`List`实现应该通常实现这个接口。 根据经验，`List`实现应实现此接口，如果对于类的典型实例，此循环：

   ```
   for (int i=0, n=list.size(); i < n; i++)
               list.get(i);
   ```

   比这个循环运行得更快：

```
for (Iterator i=list.iterator(); i.hasNext(); )
            i.next();
```

2. 案例演示1

```
public class Test01 {
    public static void main(String[] args) {
        //创建ArrayList集合
        List<String> list = new ArrayList<>();
        //添加10W条数据

        for (int i = 0; i < 100000; i++) {
            list.add(i+"a");
        }

        System.out.println("----通过索引(随机访问:)----");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < list.size(); i++) {
            //仅仅为了演示取出数据的时间,因此不对取出的数据进行打印
            list.get(i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("随机访问: "+(endTime-startTime));

        System.out.println("----通过迭代器(顺序访问:)----");
        startTime = System.currentTimeMillis();
        Iterator<String> it = list.iterator();
        while (it.hasNext()){
            //仅仅为了演示取出数据的时间,因此不对取出的数据进行打印
            it.next();
        }
        endTime = System.currentTimeMillis();
        System.out.println("顺序访问: "+(endTime-startTime));
    }
}
```

控制台效果

> ----通过索引(随机访问:)---- 随机访问: 2 ----通过迭代器(顺序访问:)---- 顺序访问: 3

3. 案例演示二

```
public class Test02 {
    public static void main(String[] args) {
        //创建LinkedList集合
        List<String> list = new LinkedList<>();
        //添加10W条数据
        for (int i = 0; i < 100000; i++) {
            list.add(i+"a");

        }
```

```
        System.out.println("----通过索引(随机访问:)----");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < list.size(); i++) {
            //仅仅为了演示取出数据的时间,因此不对取出的数据进行打印
            list.get(i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("随机访问: "+(endTime-startTime));

        System.out.println("----通过迭代器(顺序访问:)----");
        startTime = System.currentTimeMillis();
        Iterator<String> it = list.iterator();
        while (it.hasNext()){
            //仅仅为了演示取出数据的时间,因此不对取出的数据进行打印
            it.next();
        }
        endTime = System.currentTimeMillis();
        System.out.println("顺序访问: "+(endTime-startTime));
    }
}
```

控制台效果

> ----通过索引(随机访问:)---- 随机访问: 33759 ----通过迭代器(顺序访问:)---- 顺序访问: 9

**为什么LinkedList随机访问比顺序访问要慢这么多?**

**源码分析**

随机访问

```
//每次LinkedList对象调用get方法获取元素,都会执行以下代码
list.get(i);

public class LinkedList<E> {
    public E get(int index) {
        //检验是否有效
        checkElementIndex(index);
        //调用node(index)
        return node(index).item;
    }

    //node方法
     Node<E> node(int index) {
         //node方法每次被调用的时候都会根据集合size进行折半动作
         //判断get方法中的index是小于集合长度的一半还是大于
        if (index < (size >> 1)) {
            //如果小于就从链表的头部一个个的往后找
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;

            return x;
```

```
        } else {
            Node<E> x = last;
            //如果大于就从链表的尾部一个个的往前找
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
}
```

顺序访问

```
//获取迭代器的时候,会执行以下代码
Iterator<String> it = list.iterator();

//AbstractList为LinkedList父类的父类
public abstract class AbstractList<E> {
    public ListIterator<E> listIterator() {
        //返回一个列表迭代器,且指定参数为0
        return listIterator(0);
    }
}

public class LinkedList<E>{
    public ListIterator<E> listIterator(int index) {
        //检查索引位置
        checkPositionIndex(index);
        //返回ListItr对象
        return new ListItr(index);
    }

    //LinkedList迭代器实现类
    private class ListItr implements ListIterator<E> {
        private Node<E> lastReturned;
        private Node<E> next;
        private int nextIndex;
        //将实际修改集合次数赋值给预期修改次数
        private int expectedModCount = modCount;

        ListItr(int index) {
            //判断 0 == size,实际上就是调用 node(index)方法
            next = (index == size) ? null : node(index);
            //将index的值赋值给 nextIndex,便于下次查找
            nextIndex = index;
        }
    }

    Node<E> node(int index) {
        //在获取迭代器的时候也会进行折半的动作
        //但是在获取迭代器的时候 index 一定是0,因此 if的条件成立
        if (index < (size >> 1)) {

            Node<E> x = first;
```

```
            //由于循环条件不成立,不会执行 x.next;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;   //返回第一个元素
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
}

//迭代器调用 hasNext()方法的时候,会执行以下代码
private class ListItr implements ListIterator<E> {
    public boolean hasNext() {
        //如果nextIndex < 集合的长度,就说明还有元素,可以进行next
        return nextIndex < size;
    }
}

//当迭代器调用it.next();方法的时候会执行以下代码
it.next();
public class LinkedList<E>{
    private class ListItr implements ListIterator<E> {
        public E next() {
            checkForComodification();   //检查集合实际修改次数和预期次数是否一样
            //再次判断是否有元素
            if (!hasNext())
                throw new NoSuchElementException();
            //将链表的第一个元素赋值给lastReturned
            lastReturned = next;
            //将下一个元素赋值给next
            next = next.next;
            //nextIndex++
            nextIndex++;
            //返回第一个元素
            return lastReturned.item;
        }
    }
}
```

**小结:** 由于随机访问的时候源码底层**每次**都需要进行折半的动作，再经过判断是从头还是从尾部一个个寻找。而顺序访问只会在获取迭代器的时候进行一次折半的动作,以后每次都是在上一次的基础上获取下一个元素。因此顺序访问要比随机访问快得多

4. **实际开发应用场景**

```
public class Test01 {
    //创建JdbcTemplate对象
    JdbcTemplate jt = new JdbcTemplate(JDBCUtils.getDataSource());

    //查询出基础班在读男学员的所有信息,且按成绩的降序输出到控制台上（利用JDBC）
```

```
    @Test
    public void fun2() throws Exception {
        //拼写SQL
        String sql = "select * from stutb where sex = ? and type like ? order by score
desc";
        //调用方法查询  将结果集的每一行都封装成一个Stutb对象,并把每一个对象都添加到集合
        List<Stutb> list = jt.query(sql, new BeanPropertyRowMapper<Stutb>(Stutb.class),
"男", "%基础班%");
        //在遍历集合取出结果集之前面临一个问题,使用普通for遍历好 还是使用迭代器(增强for)?
        //特别是数据量特别大的时候一定要考虑!
        //对返回的集合进行判断,如果返回的集合实现了 RandomAccess 就使用 普通for
        //否则使用迭代器(增强for)
        if(list instanceof RandomAccess){
            for (int i = 0; i < list.size(); i++) {
                System.out.println(list.get(i));
            }
        }else {
            for (Stutb stutb : list) {
                System.out.println(stutb);
            }
        }
    }
}
```

## 2.4 AbstractList抽象类

# 3. ArrayList源码分析

## 3.1 构造方法

| Constructor | Constructor描述 |
| --- | --- |
| ArrayList() | 构造一个初始容量为十的空列表。 |
| ArrayList(int initialCapacity) | 构造具有指定初始容量的空列表。 |
| ArrayList(Collection<? extends E> c) | 构造一个包含指定集合的元素的列表，按照它们由集合的迭代器返回的顺序。 |

## 3.2 案例演示

### 案例一:

1. 空参构造ArrayList()

```
public class Test01 {
    public static void main(String[] args) {
        //这行代码做了什么?
        //真的构造一个初始容量为十的空列表?
        ArrayList<String> list = new ArrayList<String>();
   }
}
```

2. 源码分析

```
public class ArrayList<E> {
    /**
     * 默认初始容量
     */
    private static final int DEFAULT_CAPACITY = 10;
     /**
     * 空数组
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};
    /**
     * 默认容量的空数组
     */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
    /**
     * 集合真正存储数组元素的数组
     */
    transient Object[] elementData;
     /**
     * 集合的大小
     */
    private int size;

    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
}
```

3. 结论

   通过空参构造方法创建集合对象并未构造一个初始容量为十的空列表，仅仅将`DEFAULTCAPACITY_EMPTY_ELEMENTDATA`的地址赋值给elementData

## 案例二:

1. 指定容量ArrayList(int initialCapacity)

```
public class Test01 {
    public static void main(String[] args) {
        //这行代码ArrayList底层做了什么?
        ArrayList<String> list = new ArrayList<String>(5);
   }
}
```

2. 源码分析

```
public class ArrayList<E> {
    public ArrayList(int initialCapacity) { //initialCapacity = 5
        //判断初始容量initialCapacity是否大于0
        if (initialCapacity > 0) {
            //创建一个数组,且指定长度为initialCapacity
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            //如果initialCapacity容量为0, 把EMPTY_ELEMENTDATA的地址赋值给elementData
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            //以上两个条件都不满足报错
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
}
```

3. 结论

根据`ArrayList`构造方法参数创建指定长度的数组

## 案例三:

1. ArrayList(Collection<? extends E> c)

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("aaa");
        list.add("bbb");
        list.add("ccc");
        //这行代码做了什么?
        ArrayList<String> list1 = new ArrayList<>(list);
        for (String s : list1) {
            System.out.println(s);
        }
   }
}
```

2. 源码分析

```
public class ArrayList<E> {
    public ArrayList(Collection<? extends E> c) {
        // 将集合构造中的集合对象转成数组,且将数组的地址赋值给elementData
        elementData = c.toArray();
        // 将elementData的长度赋值给 集合长度size,且判断是否不等于 0
        if ((size = elementData.length) != 0) {
            // 判断elementData 和 Object[] 是否为不一样的类型
            if (elementData.getClass() != Object[].class)
                //如果不一样,使用Arrays的copyOf方法进行元素的拷贝
                elementData = Arrays.copyOf(elementData, size, Object[].class);
        } else {
            // 用空数组代替
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }

    //将集合转数组的方法
    public Object[] toArray() {
        //调用数组工具类方法进行拷贝
        return Arrays.copyOf(elementData, size);
    }

}

//数组工具类
public class Arrays {
    public static <T> T[] copyOf(T[] original, int newLength) {
        //再次调用方法进行拷贝
        return (T[]) copyOf(original, newLength, original.getClass());
    }

    public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]>
newType) {
        //用三元运算符进行判断,不管结果如何都是创建一个新数组
        T[] copy = ((Object)newType == (Object)Object[].class)
            ? (T[]) new Object[newLength]
            : (T[]) Array.newInstance(newType.getComponentType(), newLength);
        //将数组的内容拷贝到 copy 该数组中
        System.arraycopy(original, 0, copy, 0,
                         Math.min(original.length, newLength));
       //返回拷贝元素成功后的数组
        return copy;
    }
}
```

## 3.3 添加方法

| 方法名 | 描述 |
| --- | --- |
| public boolean add(E e) | 将指定的元素追加到此列表的末尾。 |
| public void add(int index, E element) | 在此列表中的指定位置插入指定的元素。 |
| public boolean addAll(Collection<? extends E> c) | 按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的末尾。 |
| public boolean addAll(i nt index, Collection<? extends E> c) | 将指定集合中的所有元素插入到此列表中，从指定的位置开始。 |

- `public boolean add(E e)`添加单个元素

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("黑马程序员");
    }
}
```

源代码

```
public class ArrayList<E> {
    //将添加的数据传入给 e
    public boolean add(E e) {
        //调用方法对内部容量进行校验
        ensureCapacityInternal(size + 1);
        elementData[size++] = e;
        return true;
    }

    private void ensureCapacityInternal(int minCapacity) {
        //判断集合存数据的数组是否等于空容量的数组
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            //通过最小容量和默认容量 求出较大值（用于第一次扩容）
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        //将if中计算出来的容量传递给下一个方法,继续校验
        ensureExplicitCapacity(minCapacity);
    }

    private void ensureExplicitCapacity(int minCapacity) {
        //实际修改集合次数++（在扩容的过程中没用,主要是用于迭代器中）
        modCount++;
        //判断最小容量 - 数组长度是否大于 0
        if (minCapacity - elementData.length > 0)
            //将第一次计算出来的容量传递给 核心扩容方法
            grow(minCapacity);
    }
```

```
    private void grow(int minCapacity) {
        //记录数组的实际长度,此时由于木有存储元素,长度为0
        int oldCapacity = elementData.length;
        //核心扩容算法 原容量的1.5倍
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        //判断新容量 - 最小容量 是否小于 0, 如果是第一次调用add方法必然小于
        if (newCapacity - minCapacity < 0)
            //还是将最小容量赋值给新容量
            newCapacity = minCapacity;
        //判断新容量-最大数组大小 是否>0,如果条件满足就计算出一个超大容量
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // 调用数组工具类方法,创建一个新数组,将新数组的地址赋值给elementData
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

**总结**

- `public void add(int index, E element)` 在指定索引处添加元素

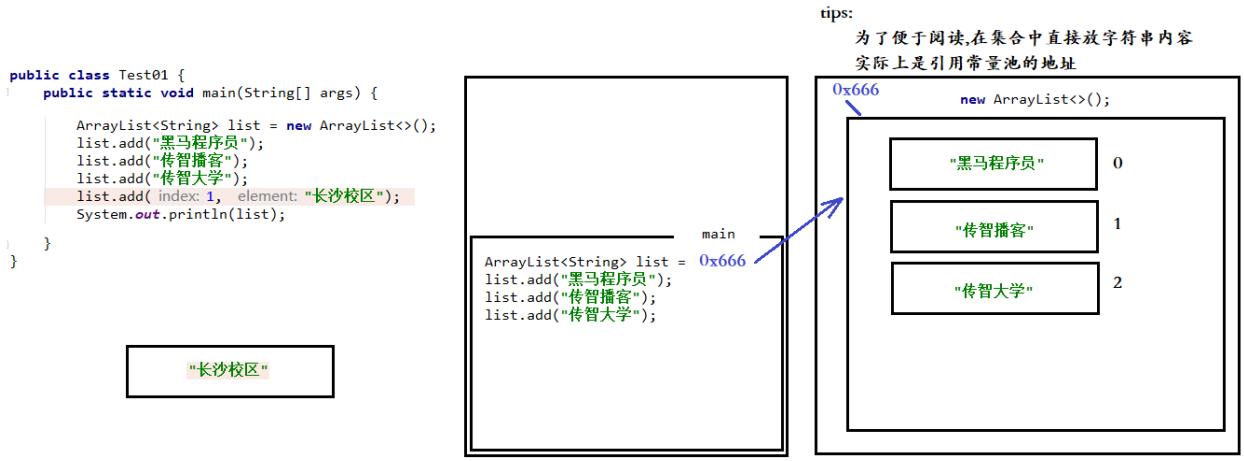
```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("黑马程序员");
        list.add("传智播客");
        list.add("传智大学");
        list.add(1, "长沙校区");
        System.out.println(list);
    }
}
```

源代码

```
public class ArrayList<E> {
    public void add(int index, E element) {
        //添加范围检查
        rangeCheckForAdd(index);
        //调用方法检验是否要扩容,且让增量++
        ensureCapacityInternal(size + 1);
        System.arraycopy(elementData, index, elementData, index + 1,
                         size - index);
        elementData[index] = element;
        size++;
    }

    private void rangeCheckForAdd(int index) {
        //超出指定范围就报错
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
```

```
    private void ensureCapacityInternal(int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        //确保明确的能力
        ensureExplicitCapacity(minCapacity);
    }

     private void ensureExplicitCapacity(int minCapacity) {
         //增量++ (也就是实际修改集合的次数)
        modCount++;
         //如果再调用 add(index,element) 方法之前已经扩容,那么源码跟踪到此结束
         //不会进行扩容
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }
}
```

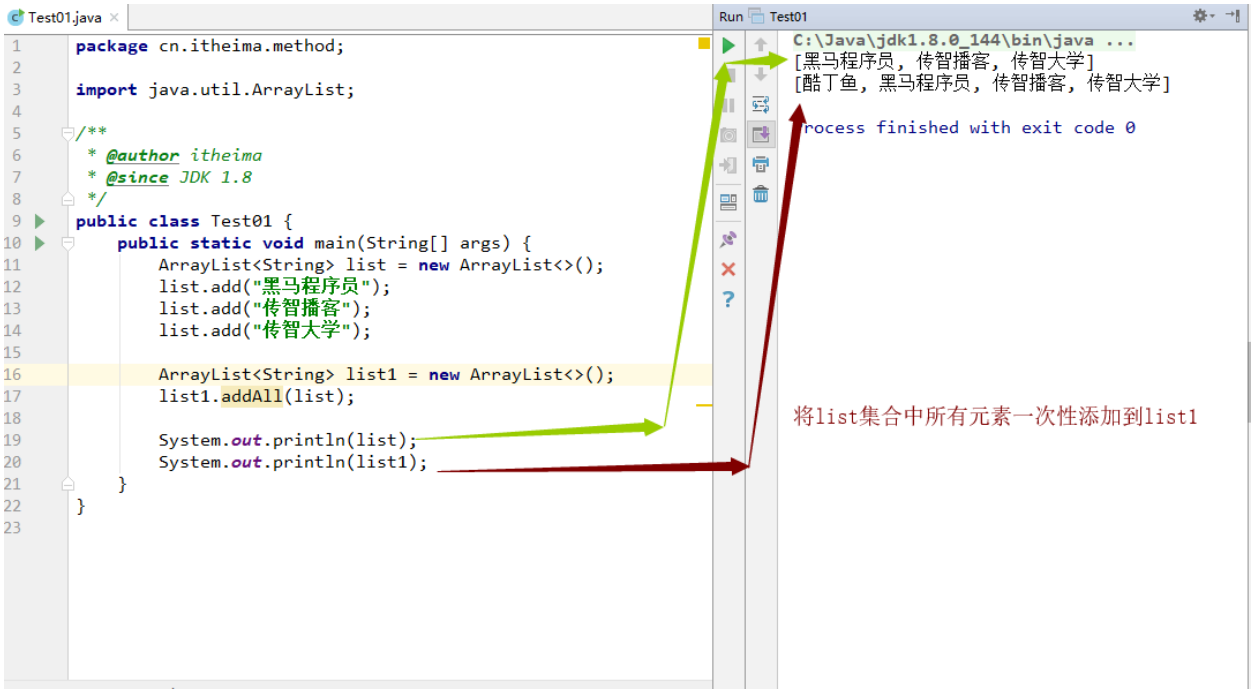
图解




- `public boolean addAll(Collection<? extends E> c)` 将集合的所有元素一次性添加到集合

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("黑马程序员");
        list.add("传智播客");
        list.add("传智大学");

        ArrayList<String> list1 = new ArrayList<>();
        list1.addAll(list);

        System.out.println(list);
        System.out.println(list1);
    }
}
```

效果




源码分析

```
public class ArrayList<E> {
    public boolean addAll(Collection<? extends E> c) {
        //把集合的元素转存到Object类型的数组中
        Object[] a = c.toArray();
        //记录数组的长度
        int numNew = a.length;
        //调用方法检验是否要扩容,且让增量++
        ensureCapacityInternal(size + numNew);
        //调用方法将a数组的元素拷贝到elementData数组中
        System.arraycopy(a, 0, elementData, size, numNew);
        //集合的长度+=a数组的长度
        size += numNew;
        //只要a数组的长度不等于0,即说明添加成功
        return numNew != 0;
    }
}
```

- `public boolean addAll(int index, Collection<? extends E> c)` 在指定的索引位置添加集合

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("黑马程序员");
        list.add("传智播客");
        list.add("传智大学");

        ArrayList<String> list1 = new ArrayList<>();
        list1.add("酷丁鱼");
        //在指定索引处添加一个集合
```

```
        list1.addAll(1,list);

        System.out.println(list);
        System.out.println(list1);
    }
}
```

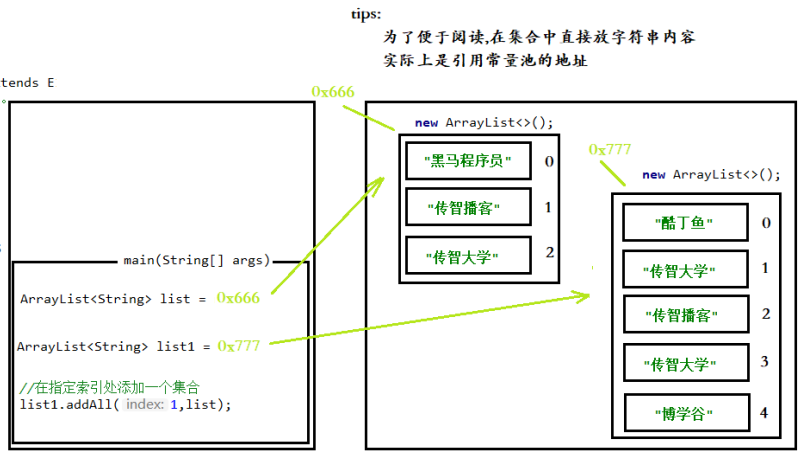
- 源码

```
public class ArrayList<E> {
    //长度为0的空数组
    private static final Object[] EMPTY_ELEMENTDATA = {};

    //默认容量为空的数组
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    //集合存元素的数组
    Object[] elementData;

    //集合的长度
    private int size;

    //默认的容量
    private static final int DEFAULT_CAPACITY = 10;

     public boolean addAll(int index, Collection<? extends E> c) {
        //校验索引
        rangeCheckForAdd(index);
        //将数据源转成数组
        Object[] a = c.toArray();
        //记录数据源的长度 3
        int numNew = a.length;
        //目的就是为了给集合存储数据的数组进行扩容
        ensureCapacityInternal(size + numNew);

        //numMoved:代表要移动元素的个数 --> 1个
        //numMoved: 数据目的(集合list1)的长度-调用addAll的第一个参数 (索引1)
        int numMoved = size - index;
        //判断需要移动的个数是否大于0
        if (numMoved > 0)
            //使用System中的方法arraycopy进行移动
            System.arraycopy(elementData, index, elementData, index + numNew,
                             numMoved);
        //才是真正将数据源(list)中的所有数据添加到数据目的(lsit1)
        System.arraycopy(a, 0, elementData, index, numNew);
        size += numNew;
        return numNew != 0;
    }

    private void rangeCheckForAdd(int index) {
        if (index > size || index < 0)

            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
```

```
    }
 }

public final class System {
    参数
            src - 源数组。
            srcPos - 源数组中的起始位置。
            dest - 目标数组。
            destPos - 目的地数据中的起始位置。
            length - 要复制的数组元素的数量。

    public static void arraycopy(Object src,int srcPos,Object dest,int destPos,int length)
}
```

- 图解

```
/**
 * @author 长沙黑马程序员
 * @since JDK 1.8
 * ArrayList添加方法
 *  public boolean addAll(int index, Collection<? extends E
 *  将指定集合中的所有元素插入到此列表中，从指定的位置开始。
 */
public class Test04 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("黑马程序员");
        list.add("传智播客");
        list.add("传智大学");

        ArrayList<String> list1 = new ArrayList<>();
        list1.add("酷丁鱼");
        list1.add("博学谷");
        //在指定索引处添加一个集合
        list1.addAll( index: 1,list);

        System.out.println(list);
        System.out.println(list1);
    }
}
```




- 如何计算元素移动的位置&数量

```
public class ArrayCopyMethodTest {
    public static void main(String[] args) {
        //数据源: list
        String[] a = {"黑马程序员","传智播客","传智大学"};
        //数据目的: list1
        String[] arr = {"酷丁鱼","博学谷",null,null,null,null,null,null,null,null};
        /*
        int numNew = a.length;
        int numMoved = size - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew,
                             numMoved);
         */
        //获取数据源的长度 3
        int numNew = a.length;
        //numMoved = 集合真实长度 - 要存的索引位置
```

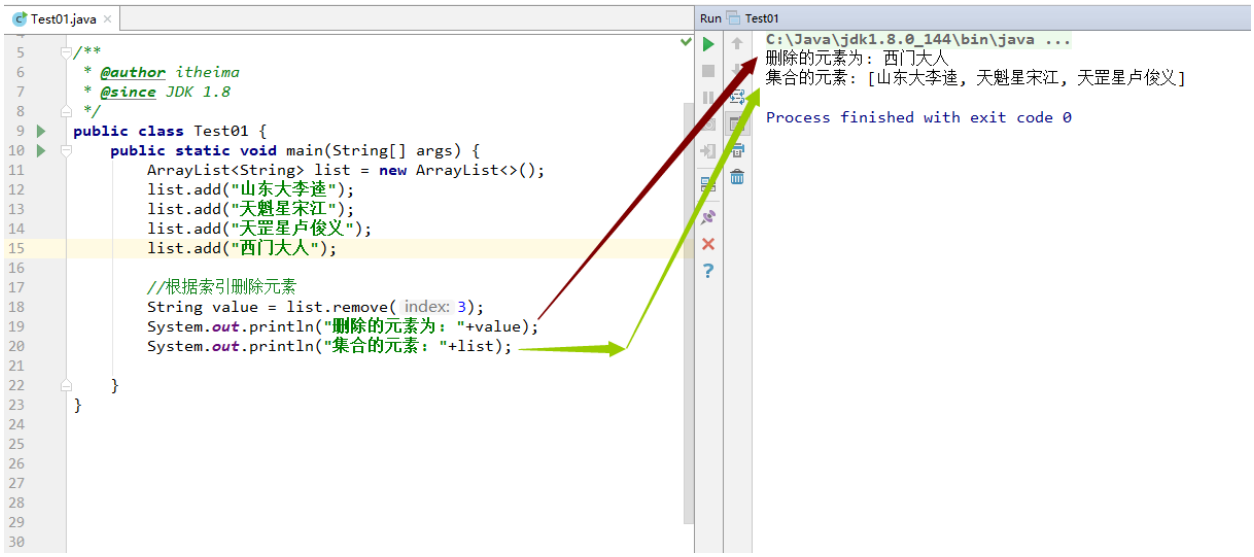
```
        //要移动元素的个数为:1
        int numMoved = 2 - 1;
        //判断是否需要移动元素
        if (numMoved > 0)
            //src - 源数组。
            //srcPos - 源数组中的起始位置。
            //dest - 目标数组。
            //destPos - 目的地数据中的起始位置。
            //length - 要复制的数组元素的数量
            System.arraycopy(arr, 1, arr, 4,
                    numMoved);

        System.out.println(Arrays.toString(arr));
    }
}
```

## 3.4 删除方法

- `public E remove(int index)` 根据索引删除元素

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("山东大李逵");
        list.add("天魁星宋江");
        list.add("天罡星卢俊义");
        list.add("西门大人");

        //根据索引删除元素
        String value = list.remove(3);
        System.out.println("删除的元素为: "+value);
        System.out.println("集合的元素: "+list);

    }
}
```

效果图




源代码

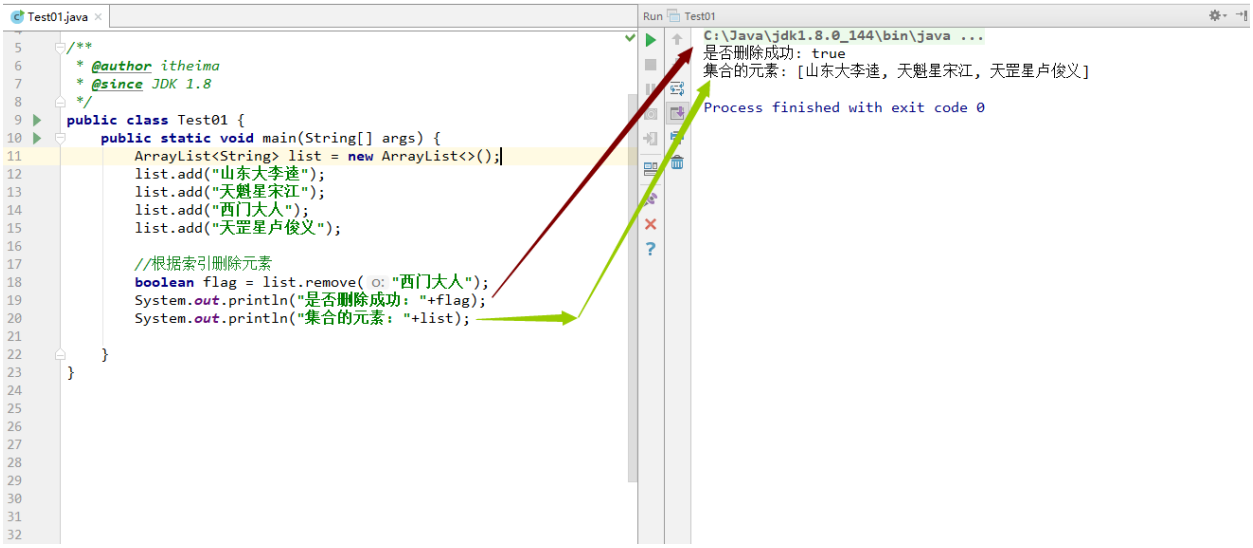
```
public class ArrayList<E> {
    public E remove(int index) {
        //范围校验
        rangeCheck(index);
       //增量++
        modCount++;
        //将index对应的元素赋值给 oldValue
        E oldValue = elementData(index);
        //计算集合需要移动元素个数
        int numMoved = size - index - 1;
        //如果需要移动元素个数大于0,就使用arrayCopy方法进行拷贝
        //注意:数据源和数据目的就是elementData
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        //将源集合最后一个元素置为null,尽早让垃圾回收机制对其进行回收
        elementData[--size] = null;
       //返回被删除的元素
        return oldValue;
    }
}
```

- `public boolean remove(Object o)` 根据元素删除元素

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("山东大李逵");
        list.add("天魁星宋江");
        list.add("西门大人");
        list.add("天罡星卢俊义");

        //根据索引删除元素
```

```
        boolean flag = list.remove("西门大人");
        System.out.println("是否删除成功: "+flag);
        System.out.println("集合的元素: "+list);

    }
}
```

效果




源码分析

```
public class ArrayList<E> {
    public boolean remove(Object o) {
        //判断要删除的元素是否为null
        if (o == null) {
            //遍历集合
            for (int index = 0; index < size; index++)
                //判断集合的元素是否为null
                if (elementData[index] == null) {
                    //如果相等,调用fastRemove方法快速删除
                    fastRemove(index);
                    return true;
                }
        } else {
            //遍历集合
            for (int index = 0; index < size; index++)
                //用o对象的equals方法和集合每一个元素进行比较
                if (o.equals(elementData[index])) {
                    //如果相等,调用fastRemove方法快速删除
                    fastRemove(index);
                    return true;
                }
        }
        //如果集合没有o该元素,那么就会返回false
        return false;
    }

    private void fastRemove(int index) {
```

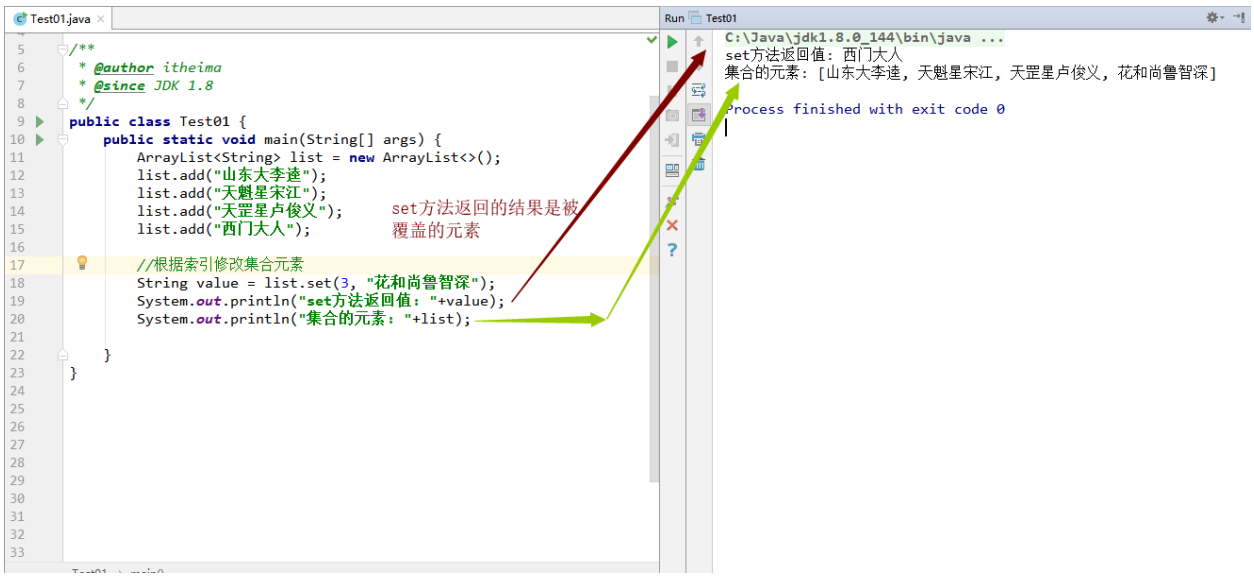
```
            //增量++
            modCount++;
            //计算集合需要移动元素的个数
            int numMoved = size - index - 1;
            //如果需要移动的个数大于0,调用arrayCopy方法进行拷贝
            if (numMoved > 0)
                System.arraycopy(elementData, index+1, elementData, index,
                                 numMoved);
            //将集合最后一个元素置为null,尽早被释放
            elementData[--size] = null;
        }
    }
```

## 3.5 修改方法

- `public E set(int index, E element)` 根据索引修改集合元素

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("山东大李逵");
        list.add("天魁星宋江");
        list.add("天罡星卢俊义");
        list.add("西门大人");

        //根据索引修改集合元素
        String value = list.set(3, "花和尚鲁智深");
        System.out.println("set方法返回值: "+value);
        System.out.println("集合的元素: "+list);

    }
}
```

效果




源码分析

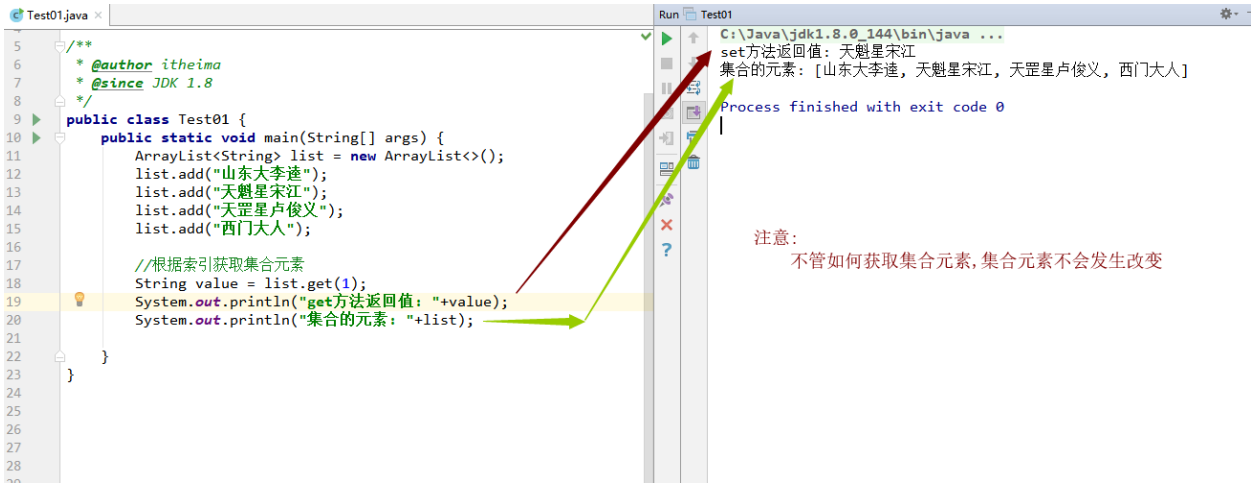
```
public class ArrayList<E> {
    public E set(int index, E element) {
        //范围校验
        rangeCheck(index);
        //先取出index对应的元素,且赋值给oldValue
        E oldValue = elementData(index);
        //将element直接覆盖index对应的元素
        elementData[index] = element;
        //返回被覆盖的元素
        return oldValue;
    }

    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
}
```

## 3.6 获取方法

- `public E get(int index)` 根据索引获取元素

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("山东大李逵");
        list.add("天魁星宋江");
        list.add("天罡星卢俊义");
        list.add("西门大人");

        //根据索引获取集合元素
        String value = list.get(1);
        System.out.println("get方法返回值: "+value);
        System.out.println("集合的元素: "+list);
    }
}
```

效果




源码分析

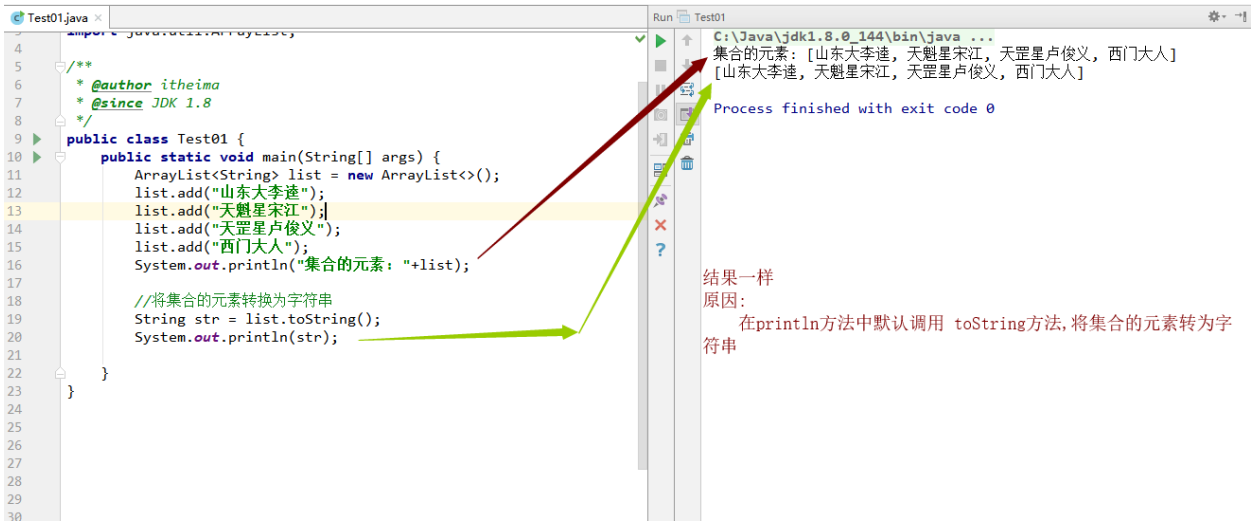
```
public class ArrayList<E> {
     public E get(int index) {
         //范围校验
        rangeCheck(index);
        //直接根据索引取出集合元素
        return elementData(index);
    }

    private void rangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
}
```

## 3.7 转换方法

- `public String toString()` 把集合所有数据转换成字符串

```
public class Test01 {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("山东大李逵");
        list.add("天魁星宋江");
        list.add("天罡星卢俊义");
        list.add("西门大人");
        System.out.println("集合的元素: "+list);

        //将集合的元素转换为字符串
        String str = list.toString();
        System.out.println(str);
    }
}
```

效果




源码分析

```
public class ArrayList<E> extends AbstractList<E>{
    public Iterator<E> iterator() {
        return new Itr();
    }
    //ArrayList集合内部类
    private class Itr implements Iterator<E> {
        int cursor;
        int lastRet = -1;
        //将实际修改集合次数赋值给预期修改次数 ,注意只会赋值一次
        //以后在迭代器获取元素的时候,每次都会判断集合实际修改次数是否和预期修改次数一致
        //如果不一致就会产生并发修改异常
        int expectedModCount = modCount;

        //判断光标 和 集合的大小 是否不相等
        public boolean hasNext() {
            return cursor != size;
        }

        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }

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

}
```

```
//ArrayList亲爹
public abstract class AbstractList<E> extends AbstractCollection<E> {}
//ArrayList亲爷爷
public abstract class AbstractCollection<E> {
    public String toString() {
        //注意:此时相当于用ArrayList对象在调用iterator()方法 获取迭代器
        //那么这个时候需要先看看ArrayList中的iterator()方法
        Iterator<E> it = iterator();
        //调用ArrayList中hasNext方法判断是否有元素,如果hasNext()方法返回false
        //那么就toString方法就返回一个 "[]"
        if (! it.hasNext())
            return "[]";
        //创建StringBuilder,对集合的内容进行拼接,避免字符串频繁拼接产生很多无效对象
        StringBuilder sb = new StringBuilder();
        sb.append('[');
        //无限循环
        for (;;) {
            //调用ArrayList中next方法取出元素
            E e = it.next();
            sb.append(e == this ? "(this Collection)" : e);
            if (! it.hasNext())
                return sb.append(']').toString();
            sb.append(',').append(' ');
        }
    }
}
```

## 3.8 迭代器

- `public Iterator<E> iterator()` 普通迭代器

  源码同上(在讲toString方法的时候已经讲过基本操作,通过以下两个案例进行一步分析源码)

  **案例一: 已知集合：List list = new ArrayList();里面有三个元素："hello"、"Java"、"PHP"，使用迭代器遍历获取集合的每一个元素**

```
/**
 * @author 长沙黑马程序员
 * @since JDK 1.8
 *
 * 案例一: 已知集合: List<String> list = new ArrayList<String>();里面有三个元
素: "hello"、"Java"、"PHP",
 * 使用迭代器遍历获取集合的每一个元素
 */
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");

        list.add("Java");
```

```
        list.add("PHP");
        //获取迭代器
        Iterator<String> it = list.iterator();
        //遍历集合
        while (it.hasNext()) {
            String s = it.next();
            System.out.println(s);
        }
    }
}
```

**案例二: 已知集合：List list = new ArrayList();里面有三个元素："hello"、"Java"、"PHP"，使用迭代器遍历集合看有没有"PHP"这个元素，如果有，就使用集合对象删除该元素**

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("Java");
        list.add("PHP");
        //获取迭代器
        Iterator<String> it = list.iterator();
        //遍历集合
        while (it.hasNext()) {
            String s = it.next();
            if(s.equals("PHP")) {
                list.remove("PHP");
            }
        }
    }
}
```

控制台结果:**并发修改异常**

> Exception in thread "main" java.util.ConcurrentModificationException at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901) at java.util.ArrayList$Itr.next(ArrayList.java:851) at cn.itheima.method.Test01.main(Test01.java:24)

源码分析:**(应该从获取迭代器的时候就进入到源代码中)**

```
public class ArrayList<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    //ArrayList内部类
    //一定要注意观察 Itr 类中的几个成员变量
    private class Itr implements Iterator<E> {
        int cursor;       // 下一个要返回元素的索引
        int lastRet = -1; // 最后一个返回元素的索引
```

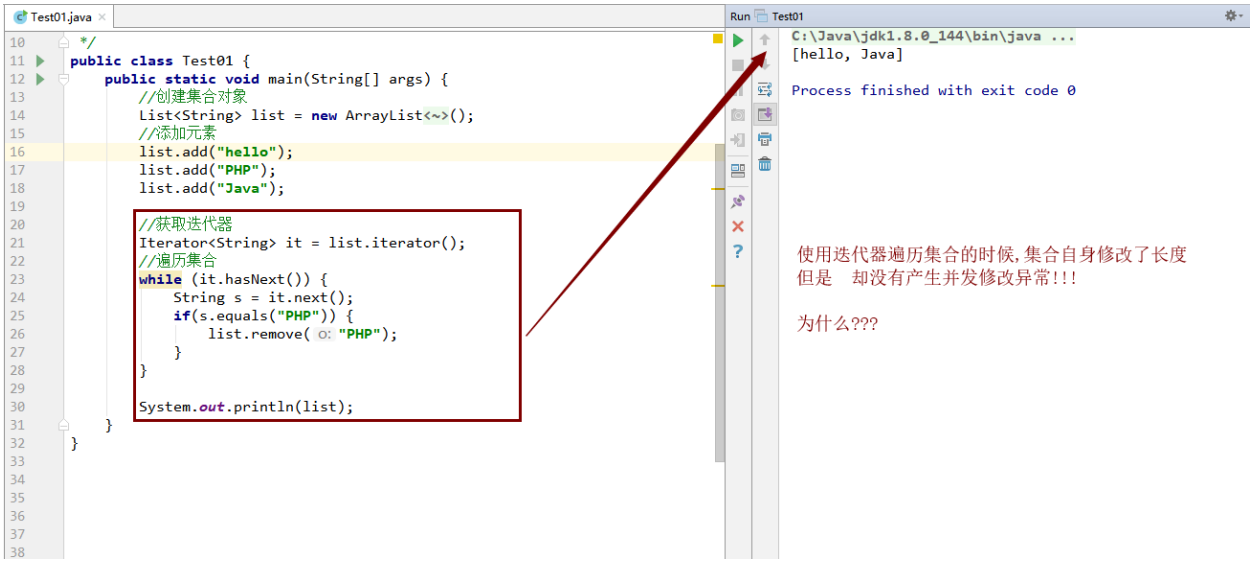
```
        //将实际修改集合次数 赋值 给预期修改次数
        //在迭代的过程中,只要实际修改次数和预期修改次数不一致就会产生并发修改异常
        //由于expectedModCount是Itr的成员变量,那么只会被赋值一次!!!
        //同时由于集合调用了三次add方法,那么实际修改集合次数就是 3,因此expectedModCount的值也是 3
        int expectedModCount = modCount;

        public boolean hasNext() {
            return cursor != size;
        }

        //获取元素的方法
        public E next() {
            //每次获取元素,会先调用该方法校验 预期修改次数是否 == 实际修改次数
            /*
            tips:
                if(s.equals("hello")) {
                    list.remove("hello");
                 }
                 当if表达式的结果为true,那么集合就会调用remove方法
            */
            checkForComodification();
            //把下一个元素的索引赋值给i
            int i = cursor;
            //判断是否有元素
            if (i >= size)
                throw new NoSuchElementException();
            //将集合底层存储数据的数组赋值给迭代器的局部变量 elementData
            Object[] elementData = ArrayList.this.elementData;
            //再次判断,如果下一个元素的索引大于集合底层存储元素的长度 并发修改异常
            //注意,尽管会产生并发修改异常,但是这里显示不是我们要的结果
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            //每次成功获取到元素,下一个元素的索引都是当前索引+1
            cursor = i + 1;
            //返回元素
            return (E) elementData[lastRet = i];
        }

        final void checkForComodification() {
            //如果预期修改次数 和 实际修改次数不相等 就产生并发修改异常
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }

    //集合的remove方法
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;

                }
```

```
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }

    //快速删除方法
    private void fastRemove(int index) {
        //最最最关键的一个操作,集合实际修改次数++,那么这个时候由原来的3变成4
        //but迭代器的预期修改次数还是3!!!
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        //还有一个很关键的操作,集合的长度也发生了改变
        elementData[--size] = null;
    }
}
```

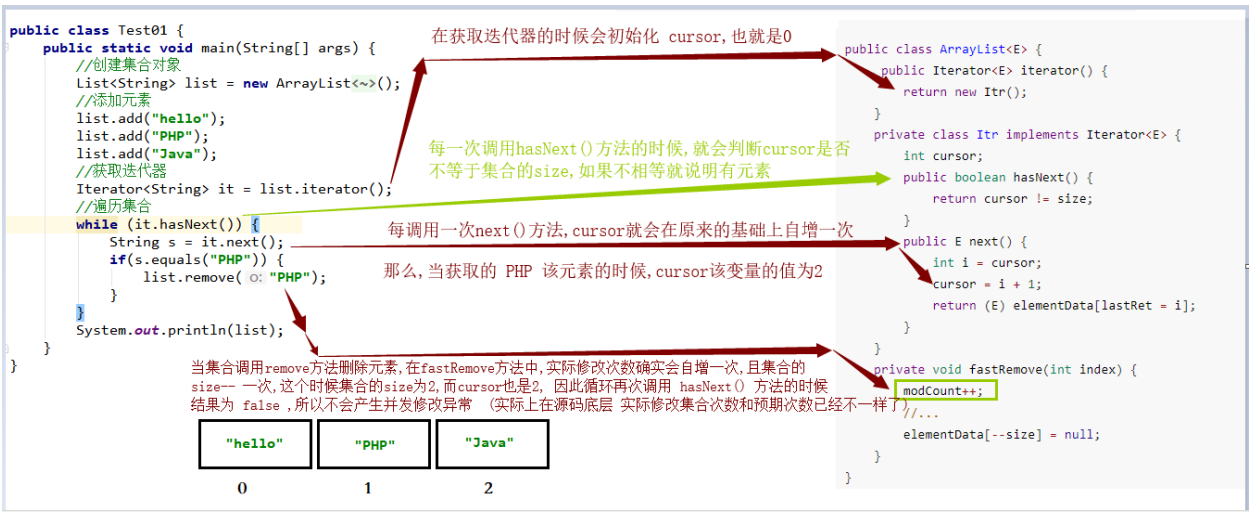
**案例三:已知集合：List list = new ArrayList();里面有三个元素："hello"、"PHP"、"JavaSE"，使用迭代器遍历集合看有没有"PHP"这个元素，如果有，就使用集合对象删除该元素**

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");

        //获取迭代器
        Iterator<String> it = list.iterator();
        //遍历集合
        while (it.hasNext()) {
            String s = it.next();
            if(s.equals("PHP")) {
                list.remove("PHP");
            }
        }
        System.out.println(list);
    }
}
```

结果图




图解

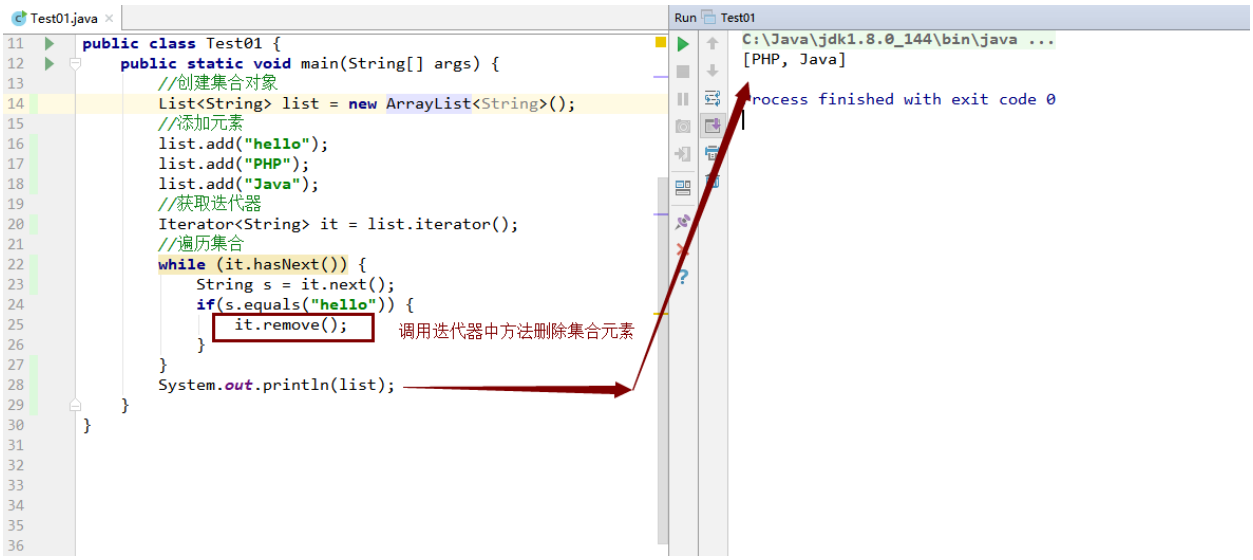



- `default void remove()` 迭代器中的remove方法,删除集合中的元素

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");
        //获取迭代器
        Iterator<String> it = list.iterator();
        //遍历集合
        while (it.hasNext()) {
            String s = it.next();
            if(s.equals("hello")) {
                it.remove();
            }
        }
        System.out.println(list);
```

```
    }
}
```

结果图




源码分析**(应该从获取迭代器的时候就进入到源代码中)**

```
public class ArrayList<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    //ArrayList内部类
    //一定要注意观察 Itr 类中的几个成员变量
    private class Itr implements Iterator<E> {
        int cursor;       // 下一个要返回元素的索引
        int lastRet = -1; // 最后一个返回元素的索引
        //将实际修改集合次数 赋值 给预期修改次数
        //在迭代的过程中,只要实际修改次数和预期修改次数不一致就会产生并发修改异常
        //由于expectedModCount是Itr的成员变量,那么只会被赋值一次!!!
        //同时由于集合调用了三次add方法,那么实际修改集合次数就是 3,因此expectedModCount的值也是 3
        int expectedModCount = modCount;

        public boolean hasNext() {
            return cursor != size;
        }

        //获取元素的方法
        public E next() {
            checkForComodification();
            //把下一个元素的索引赋值给i
            int i = cursor;
            //判断是否有元素
            if (i >= size)
                throw new NoSuchElementException();
            //将集合底层存储数据的数组赋值给迭代器的局部变量 elementData
```

```
            Object[] elementData = ArrayList.this.elementData;
            //再次判断,如果下一个元素的索引大于集合底层存储元素的长度 并发修改异常
            //注意,尽管会产生并发修改异常,但是这里显示不是我们要的结果
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            //每次成功获取到元素,下一个元素的索引都是当前索引+1
            cursor = i + 1;
            //返回元素,且将i的值 赋值给 lastRet /*0*/
            return (E) elementData[lastRet = i];
        }

        //迭代器删除元素方法
        public void remove() {
            //判断最后返回元素的索引是否小于0,满足条件就产生 非法状态异常
            if (lastRet < 0)
                throw new IllegalStateException();
            //校验是否会产生并发修改异常,第一次调用不会,因为与其修改次数和实际修改次数一致
            checkForComodification();
            try {
                //真正删除集合元素的方法,调用方法为ArrayList的方法remove,且将0作为参数进行传递
                ArrayList.this.remove(lastRet);
                //将lastRet赋值给cursor
                cursor = lastRet;
                //再次等于-1
                lastRet = -1;
                //再次将集合实际修改次数赋值给预期修改次数,那么这个时候不管集合自身是否删除成功
                //那么实际修改次数和预期修改次数又一致了,所以并不会产生并发修改异常
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        final void checkForComodification() {
            //如果预期修改次数 和 实际修改次数不相等 就产生并发修改异常
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }

    //集合的remove方法
    public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);

                    return true;
```

```
                }
            }
            return false;
        }

        //快速删除方法
        private void fastRemove(int index) {
            //最最最关键的一个操作,集合实际修改次数++,那么这个时候由原来的3变成4
            //but迭代器的预期修改次数还是3!!!
            modCount++;
            int numMoved = size - index - 1;
            if (numMoved > 0)
                System.arraycopy(elementData, index+1, elementData, index,
                                 numMoved);
            //还有一个很关键的操作,集合的长度也发生了改变
            elementData[--size] = null;
        }
    }
```

结论: 1，迭代器remove方法底层调用的还是集合自身的remove方法删除元素

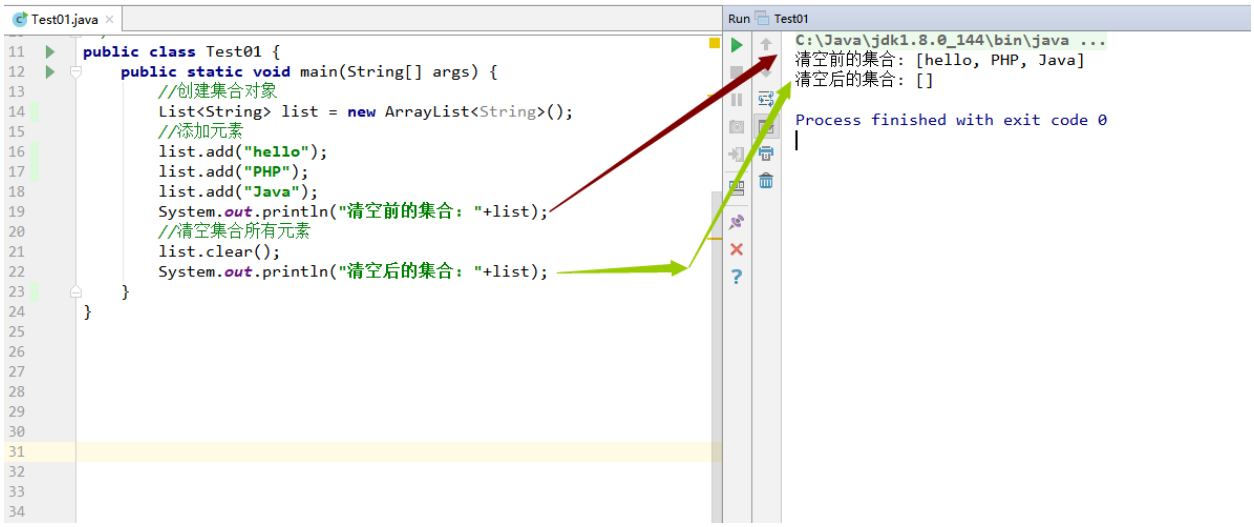
2，之所以不会产生并发修改异常，其原因是因为在迭代器的remove方法中会再次将 集合时机修改次数赋值给预期修改次数

## 3.9 清空方法

- `public void clear()` 清空集合所有数据

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");
        System.out.println("清空前的集合: "+list);
        //清空集合所有元素
        list.clear();
        System.out.println("清空后的集合: "+list);
    }
}
```

效果图




源码分析

```
public class ArrayList<E> {
    public void clear() {
        //实际修改集合次数++
        modCount++;
        //遍历集合,将集合每一个索引对应位置上的元素都置为null,尽早让其释放
        for (int i = 0; i < size; i++)
            elementData[i] = null;
        //集合长度更改为0
        size = 0;
    }
}
```

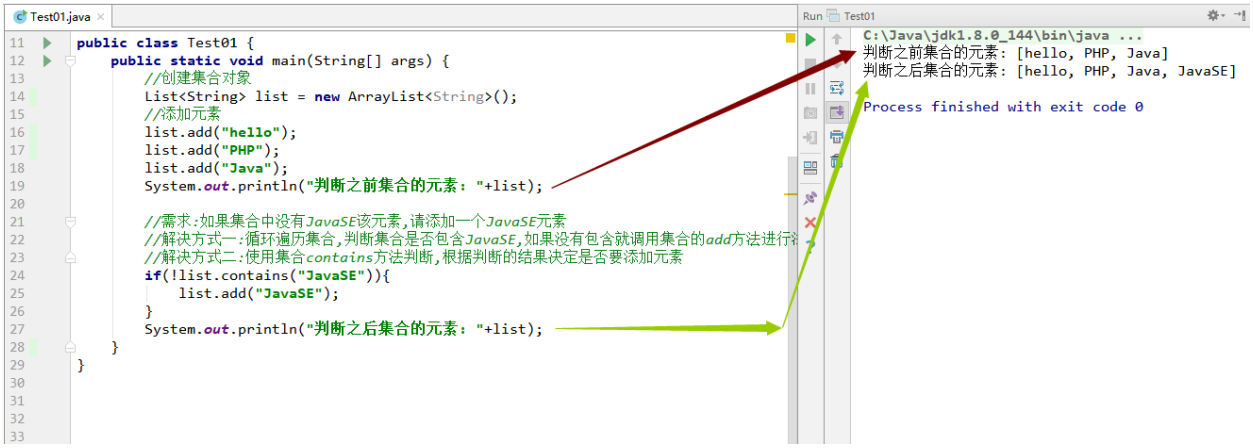
## 4.0 包含方法

- `public boolean contains(Object o)` 判断集合是否包含指定元素

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");
        System.out.println("判断之前集合的元素："+list);

        //需求:如果集合中没有JavaSE该元素,请添加一个JavaSE元素
        //解决方式一:循环遍历集合,判断集合是否包含JavaSE,如果没有包含就调用集合的add方法进行添加操
作
        //解决方式二:使用集合contains方法判断,根据判断的结果决定是否要添加元素
        if(!list.contains("JavaSE")){
            list.add("JavaSE");

        }
```

```
        System.out.println("判断之后集合的元素: "+list);
    }
}
```

效果头




源码分析

```
public class ArrayList<E> {
    //源码contains方法
    public boolean contains(Object o) {
        //调用indexOf方法进行查找
        return indexOf(o) >= 0;
    }

    public int indexOf(Object o) {
        //如果元素是null,也进行遍历操作
        //因为集合中有可能够会存储null
        if (o == null) {
            for (int i = 0; i < size; i++)
                if (elementData[i]==null)
                    return i;
        } else {
            for (int i = 0; i < size; i++)
                if (o.equals(elementData[i]))
                    return i;
        }
        //如果没有走if,也没有走else,那么就说明o该元素在集合中不存在
        return -1;
    }
}
```

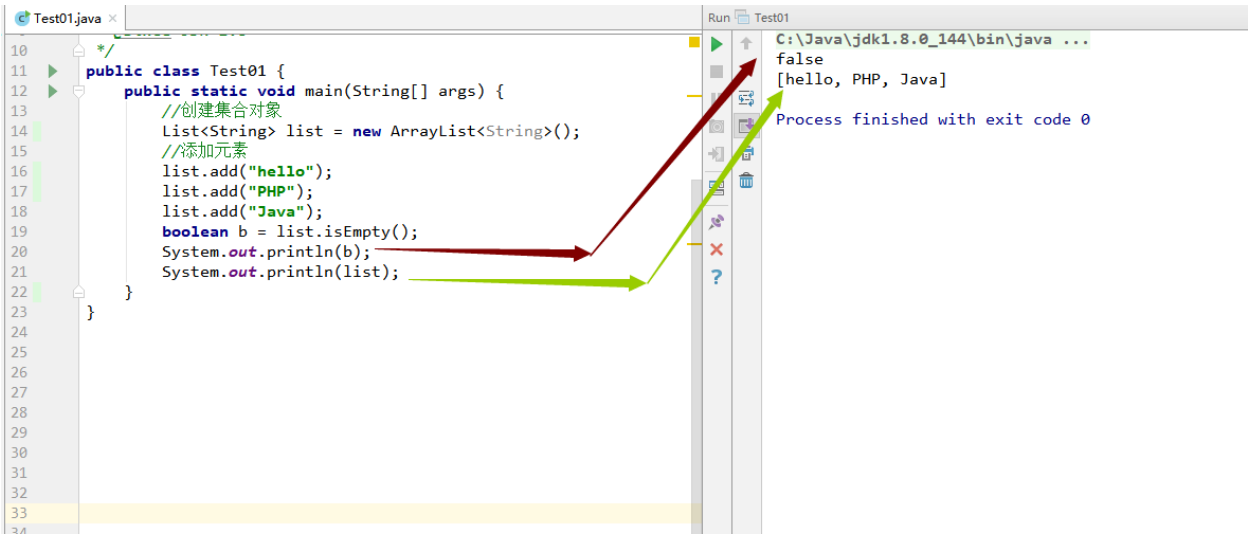
结论:底层也是通过循环遍历集合,取出一个个的元素和要找的元素进行比较

## 4.1 判断集合是否为空

- `public boolean isEmpty()`

```java
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");
        boolean b = list.isEmpty();
        System.out.println(b);
        System.out.println(list);
    }
}
```

效果图




源码分析

```java
public class ArrayList<E> {
    public boolean isEmpty() {
        return size == 0;
    }
}
```

# 4. 面试题

## 4.1 ArrayList是如何扩容的?

源码分析过程中已经讲解

第一次扩容10

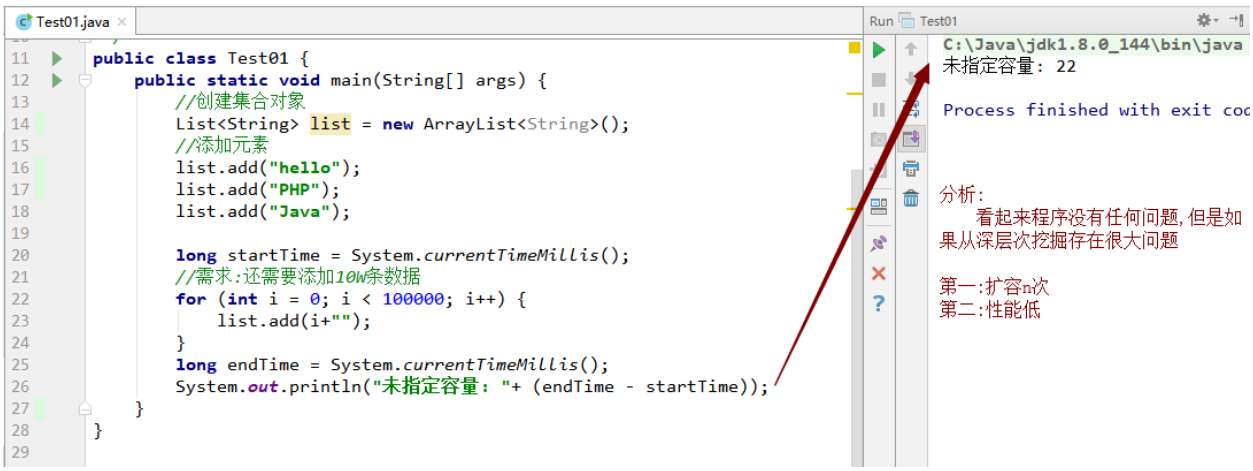
以后每次都是原容量的1.5倍

## 4.2 ArrayList频繁扩容导致添加性能急剧下降，如何处理?

- 案例

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");

        long startTime = System.currentTimeMillis();
        //需求:还需要添加10W条数据
        for (int i = 0; i < 100000; i++) {
            list.add(i+"");
        }
        long endTime = System.currentTimeMillis();
        System.out.println("未指定容量: "+ (endTime - startTime));
    }
}
```

- 效果图

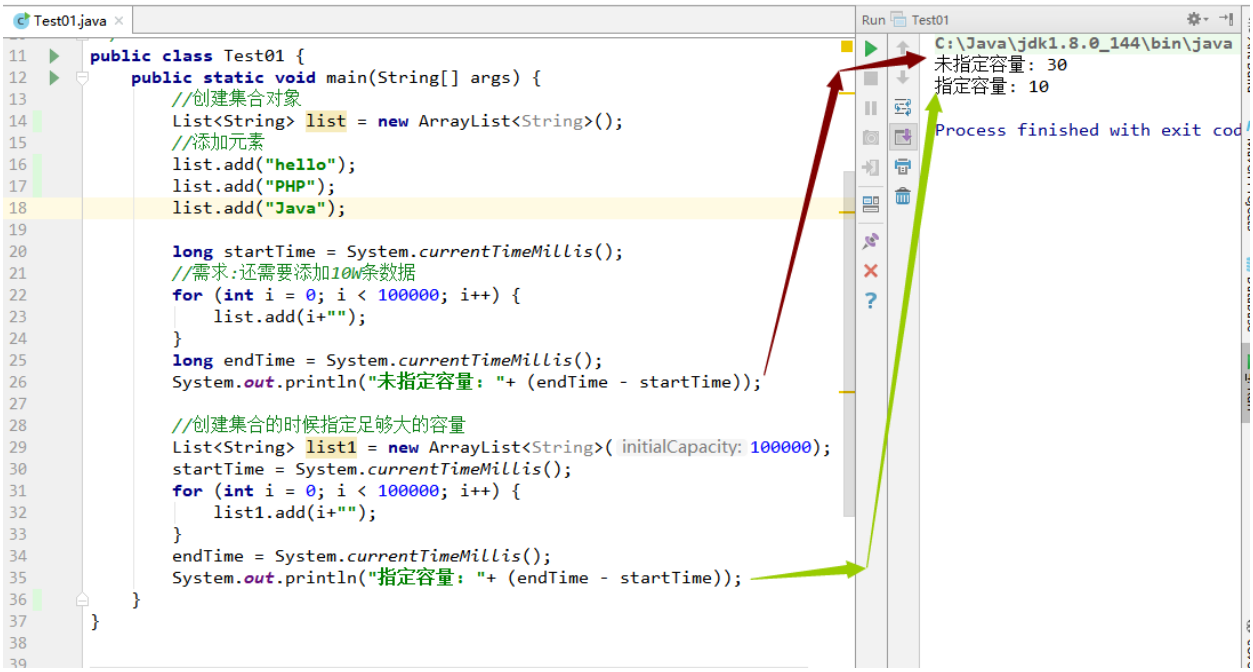



- 解决方案

```
public class Test01 {
    public static void main(String[] args) {
        //创建集合对象
        List<String> list = new ArrayList<String>();
        //添加元素
        list.add("hello");
        list.add("PHP");
        list.add("Java");

        long startTime = System.currentTimeMillis();
        //需求:还需要添加10W条数据
        for (int i = 0; i < 100000; i++) {
```

```
            list.add(i+"");
        }
        long endTime = System.currentTimeMillis();
        System.out.println("未指定容量: "+ (endTime - startTime));

        //创建集合的时候指定足够大的容量
        List<String> list1 = new ArrayList<String>(100000);
        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            list1.add(i+"");
        }
        endTime = System.currentTimeMillis();
        System.out.println("指定容量: "+ (endTime - startTime));
    }
}
```

- 优化效果图




注意:这种优化方式只针对特定的场景，如果添加的元素是少量的、未知的，不推荐使用

## 4.3 ArrayList插入或删除元素一定比LinkedList慢么?

- 根据索引删除

  案例：ArrayList和LinkedList对比

```
public class Test02 {
    public static void main(String[] args) {
        //创建ArrayList集合对象
        ArrayList<String> arrayList = new ArrayList<String>();
        //添加500W个元素
        for (int i = 0; i < 5000000; i++) {

            arrayList.add(i+"黑马");
```

```
        }
        //获取开始时间
        long startTime = System.currentTimeMillis();
        //根据索引删除ArrayList集合元素
        //删除索引5000对应的元素
        String value = arrayList.remove(50000);
        System.out.println(value);
        //获取结束时间
        long endTime = System.currentTimeMillis();
        System.out.println("ArrayList集合删除元素的时间: "+(endTime-startTime));

        //创建LinkedList集合对象
        LinkedList<String> linkedList = new LinkedList<String>();
        //添加500W个元素
        for (int i = 0; i < 5000000; i++) {
            linkedList.add(i+"黑马");
        }
        //获取开始时间
        startTime = System.currentTimeMillis();
        //根据索引删除LinkedList集合元素
        //删除索引5000对应的元素
        value = arrayList.remove(50000);
        System.out.println(value);
        endTime = System.currentTimeMillis();
        System.out.println("LinkedList集合删除元素的时间: "+(endTime-startTime));
    }
}
```

效果图


源码分析

- ArrayList根据索引删除元素源码

```
public class ArrayList<E> {
    public E remove(int index) {
        //范围校验
```

```
        rangeCheck(index);
       //增量++
        modCount++;
        //将index对应的元素赋值给 oldValue
        E oldValue = elementData(index);
        //计算集合需要移动元素个数
        int numMoved = size - index - 1;
        //如果需要移动元素个数大于0,就使用arrayCopy方法进行拷贝
        //注意:数据源和数据目的就是elementData
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        //将源集合最后一个元素置为null,尽早让垃圾回收机制对其进行回收
        elementData[--size] = null;
       //返回被删除的元素
        return oldValue;
    }
}
```

- LinkedList根据索引删除元素源码

```
public class LinkedList<E> {
    public E remove(int index) {
        //调用方法校验元素的索引
        checkElementIndex(index);
        //先调用node(index)方法,找到需要删除的索引
        //再调用unlink方法解开链条
        return unlink(node(index));
    }

    //校验索引是否在合法范围之内,不再就报错
    private void checkElementIndex(int index) {
        if (!isElementIndex(index))
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }

    private boolean isElementIndex(int index) {
        return index >= 0 && index < size;
    }

    //获取要删除的元素
    Node<E> node(int index) {
        //不管索引是多少,在源码底层都会对整个链表上的元素进行折半的动作
        //如果要删除元素的索引小于集合长度的一半,那么就从头节点一个个的往后找
        //如果要删除元素的索引大于集合长度的一半,那么就从尾节点一个个的往后找
        //(注:这个查找的效率相对于ArrayList集合来说较低)
        if (index < (size >> 1)) {
            Node<E> x = first;
            //如果循环条件不满足,那么first就是要删除的元素
            //否则,要删除的元素就是first的下一个
            for (int i = 0; i < index; i++)

                x = x.next;
```

```
            return x;
        } else {
            Node<E> x = last;
            //如果循环条件不满足,那么last就是要删除的元素
            //否则,要删除的元素就是last的前一个
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }

    //解开链表,让前后节点相互记录地址
    E unlink(Node<E> x) {
        //获取要删除的元素
        final E element = x.item;
        //获取被删除节点下一个节点的地址
        final Node<E> next = x.next;
        //获取被删除节点上一个节点的地址
        final Node<E> prev = x.prev;

        //如果被删除节点的上一个节点为null,就让被删除节点的下一个节点成为首节点
        if (prev == null) {
            first = next;
        } else {
            //否则,被删除元素上一个节点的 下一个节点 变成 被删除元素的下一个节点
            prev.next = next;
            //被删除元素的上一个节点置为null
            x.prev = null;
        }
        //如果被删除元素的下一个节点为null,最后一个节点就等于被删除元素的上一个节点
        if (next == null) {
            last = prev;
        } else {
            //否则,被删除节点的下一个节点 等于被删除节点的前一个节点
            next.prev = prev;
            //被删除元素的下一个节点置为null
            x.next = null;
        }

        //被删除元素的内容置为null
        x.item = null;
        //集合长度--
        size--;
        //实际修改次数++
        modCount++;
        //返回被删除的元素
        return element;
    }
}
```

- 结论

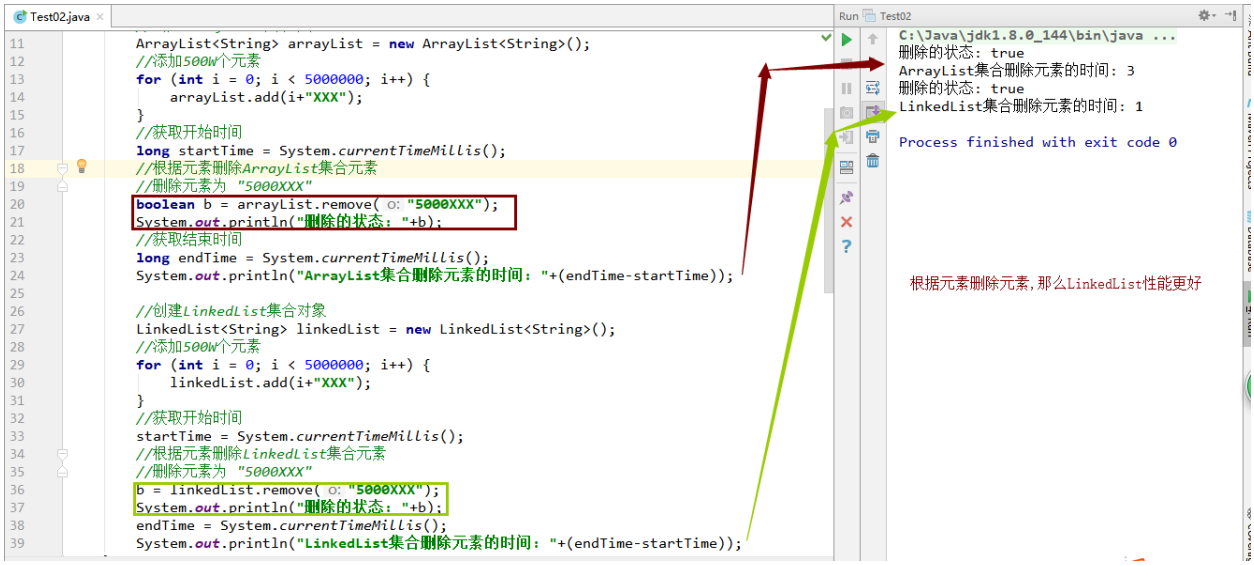
1. 数组删除元素确实要比链表慢，慢在需要创建新数组，还有比较麻烦的数据拷贝，但是在ArrayList底层不是每次删除元素都需要扩容，因此在这个方面相对于链表来说数组的性能更好
2. LinkedList删除元素之所以效率并不高，其原理在于底层先需要对整个集合进行折半的动作，然后又需要对集合进行遍历一次，这些操作导致效率变低

- 根据元素删除

  案例：ArrayList和LinkedList对比

```
public class Test02 {
    public static void main(String[] args) {

        //创建ArrayList集合对象
        ArrayList<String> arrayList = new ArrayList<String>();
        //添加500W个元素
        for (int i = 0; i < 5000000; i++) {
            arrayList.add(i+"XXX");
        }
        //获取开始时间
        long startTime = System.currentTimeMillis();
        //根据元素删除ArrayList集合元素
        //删除元素为 "5000XXX"
        boolean b = arrayList.remove("5000XXX");
        System.out.println("删除的状态: "+b);
        //获取结束时间
        long endTime = System.currentTimeMillis();
        System.out.println("ArrayList集合删除元素的时间: "+(endTime-startTime));

        //创建LinkedList集合对象
        LinkedList<String> linkedList = new LinkedList<String>();
        //添加500W个元素
        for (int i = 0; i < 5000000; i++) {
            linkedList.add(i+"XXX");
        }
        //获取开始时间
        startTime = System.currentTimeMillis();
        //根据元素删除LinkedList集合元素
        //删除元素为 "5000XXX"
        b = linkedList.remove("5000XXX");
        System.out.println("删除的状态: "+b);
        endTime = System.currentTimeMillis();
        System.out.println("LinkedList集合删除元素的时间: "+(endTime-startTime));
    }
}
```

  效果图




源码分析

- ArrayList根据元素删除元素

```
public class ArrayList<E> {
    public boolean remove(Object o) {
        //判断要删除的元素是否为null
        if (o == null) {
            //遍历集合
            for (int index = 0; index < size; index++)
                //判断集合的元素是否为null
                if (elementData[index] == null) {
                    //如果相等,调用fastRemove方法快速删除
                    fastRemove(index);
                    return true;
                }
        } else {
            //遍历集合
            for (int index = 0; index < size; index++)
                //用o对象的equals方法和集合每一个元素进行比较
                if (o.equals(elementData[index])) {
                    //如果相等,调用fastRemove方法快速删除
                    fastRemove(index);
                    return true;
                }
        }
        //如果集合没有o该元素,那么就会返回false
        return false;
    }

    private void fastRemove(int index) {
        //增量++
        modCount++;
        //计算集合需要移动元素的个数
        int numMoved = size - index - 1;
        //如果需要移动的个数大于0,调用arrayCopy方法进行拷贝
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
```

```
                                numMoved);
        //将集合最后一个元素置为null,尽早被释放
        elementData[--size] = null;
    }
}
```

- LinkedList根据元素删除元素

```
public class LinkedList<E> {
    //LinkedList集合底层删除源码
    public boolean remove(Object o) {
        //判断要删除的元素是否为null
        //不管是否为null都从第一个元素开始,从头部往后找
        //找到之后,调用unlink方法进行解绑,更改节点和节点之间记录的地址
        if (o == null) {
            for (Node<E> x = first; x != null; x = x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node<E> x = first; x != null; x = x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
        return false;
    }

    E unlink(Node<E> x) {
        final E element = x.item;
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;

        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        if (next == null) {
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }


        x.item = null;
```

```
            size--;
            modCount++;
            return element;
        }
    }
```

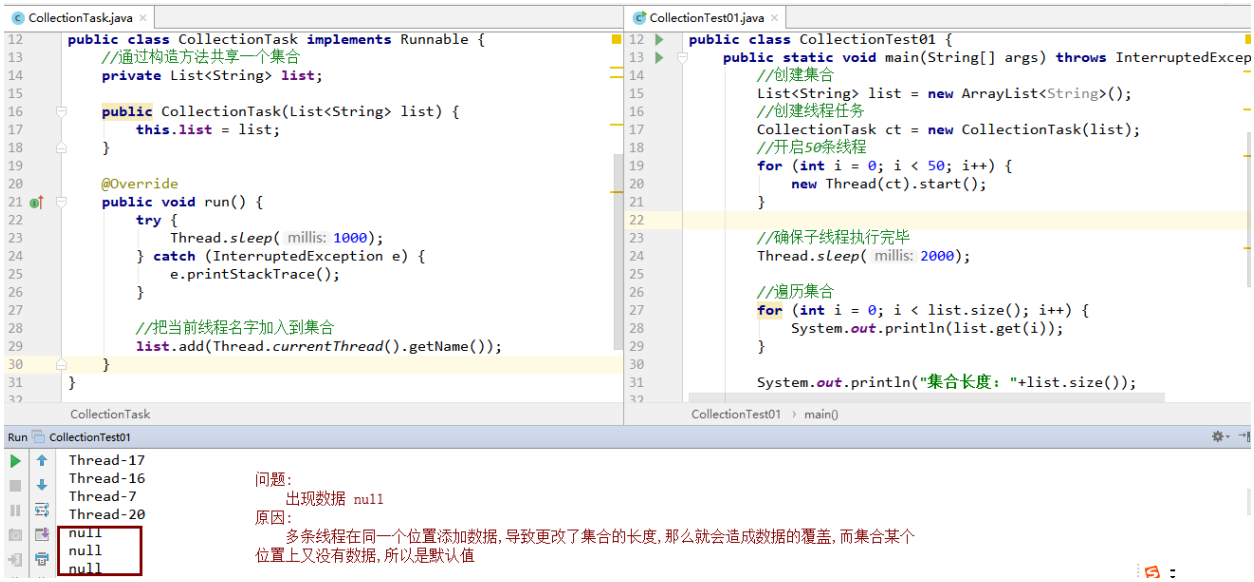
## 4.4 ArrayList是线程安全的么?

- ArrayList不是线程安全的,使用一个案例演示

```
//线程任务类
public class CollectionTask implements Runnable {
    //通过构造方法共享一个集合
    private List<String> list;

    public CollectionTask(List<String> list) {
        this.list = list;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //把当前线程名字加入到集合
        list.add(Thread.currentThread().getName());
    }
}

//测试类
public class CollectionTest01 {
    public static void main(String[] args) throws InterruptedException {
        //创建集合
        List<String> list = new ArrayList<String>();
        //创建线程任务
        CollectionTask ct = new CollectionTask(list);
        //开启50条线程
        for (int i = 0; i < 50; i++) {
            new Thread(ct).start();
        }

        //确保子线程执行完毕
        Thread.sleep(1000);
        //遍历集合
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));

        }
```

```
        System.out.println("集合长度: "+list.size());
    }
}
```

效果图



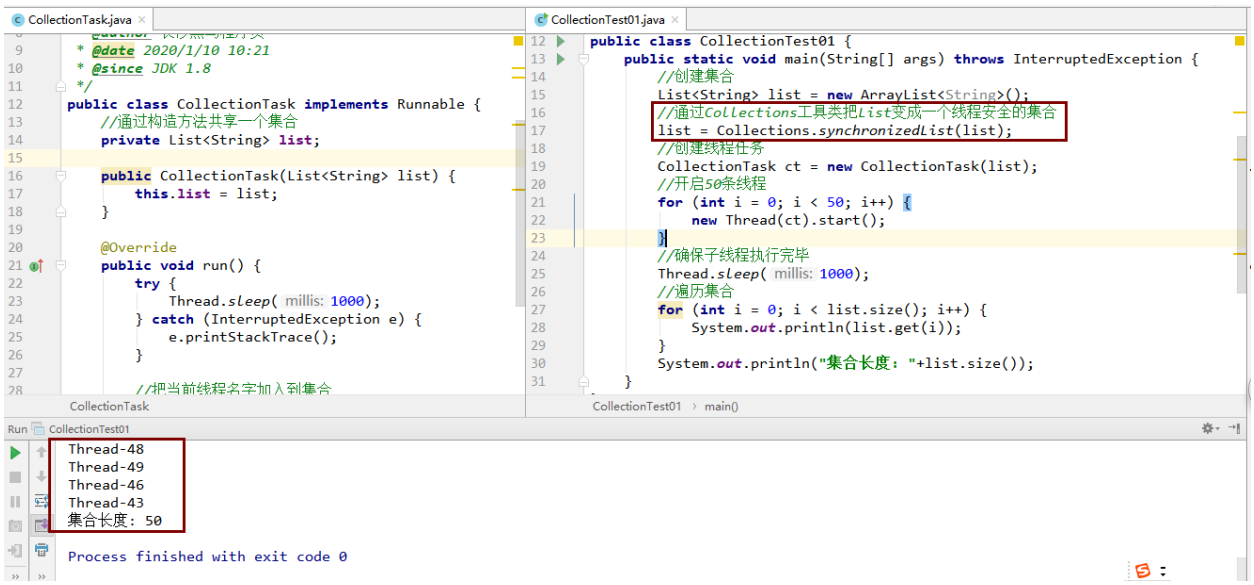

- 需要线程安全怎么办?

  **方式一:**使用Collections.synchronizedList(list)

```
//线程任务类
public class CollectionTask implements Runnable {
    //通过构造方法共享一个集合
    private List<String> list;

    public CollectionTask(List<String> list) {
        this.list = list;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //把当前线程名字加入到集合
        list.add(Thread.currentThread().getName());
    }
}

//测试类
public class CollectionTest01 {
    public static void main(String[] args) throws InterruptedException {
```

```
        //创建集合
        List<String> list = new ArrayList<String>();
        //通过Collections工具类把List变成一个线程安全的集合
        list = Collections.synchronizedList(list);
        //创建线程任务
        CollectionTask ct = new CollectionTask(list);
        //开启50条线程
        for (int i = 0; i < 50; i++) {
            new Thread(ct).start();
        }

        //确保子线程执行完毕
        Thread.sleep(1000);
        //遍历集合
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }

        System.out.println("集合长度: "+list.size());
    }
}
```

效果图

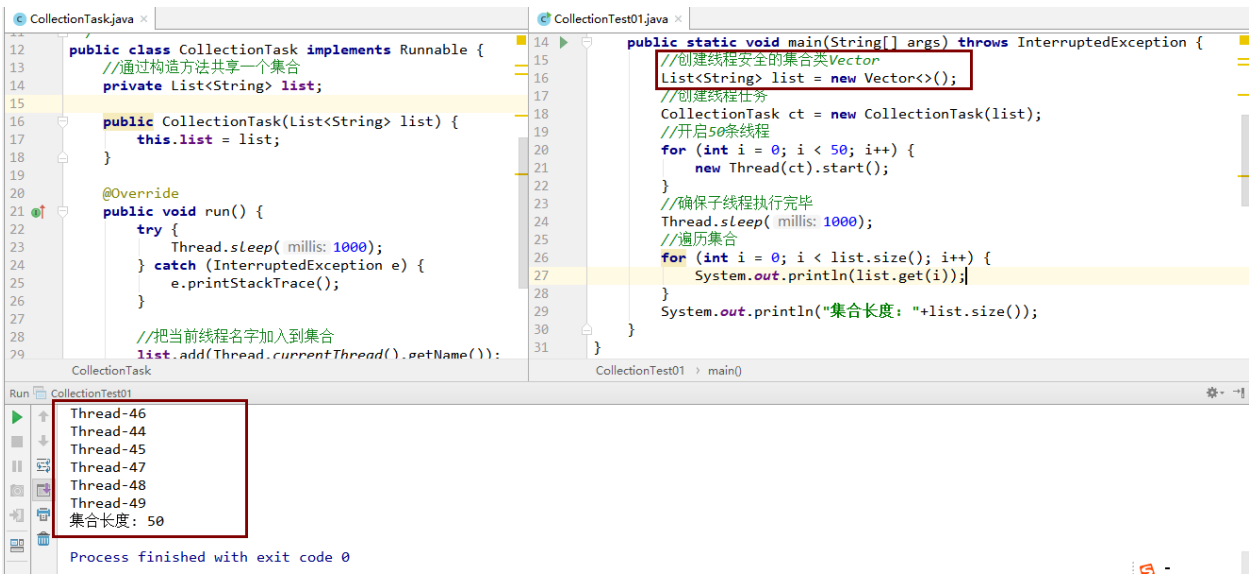



**方式二:**使用

```
//线程任务类
public class CollectionTask implements Runnable {
    //通过构造方法共享一个集合
    private List<String> list;

    public CollectionTask(List<String> list) {
        this.list = list;
    }

    @Override
    public void run() {
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            //把当前线程名字加入到集合
            list.add(Thread.currentThread().getName());
        }
    }

    //测试类
    public class CollectionTest01 {
        public static void main(String[] args) throws InterruptedException {
            //创建线程安全的集合类Vector
            List<String> list = new Vector<>();
            //通过Collections工具类把List变成一个线程安全的集合
            list = Collections.synchronizedList(list);
            //创建线程任务
            CollectionTask ct = new CollectionTask(list);
            //开启50条线程
            for (int i = 0; i < 50; i++) {
                new Thread(ct).start();
            }

            //确保子线程执行完毕
            Thread.sleep(1000);
            //遍历集合
            for (int i = 0; i < list.size(); i++) {
                System.out.println(list.get(i));
            }

            System.out.println("集合长度: "+list.size());
        }
    }
```

效果图



- 实际开发场景

  **案例:**使用JdbcTemplate查询数据库返回一个List集合是否需要保证线程安全?

```
public class Test01 {
    //创建JdbcTemplate对象
    JdbcTemplate jt = new JdbcTemplate(JDBCUtils.getDataSource());

    //3.利用JDBC查询出基础班在读的男学员的所有信息按成绩的降序输出到控制台上（利用JDBC）
    @Test
    public void fun1() throws Exception {
        //拼写SQL
        String sql = "select * from stutb where sex = ? and type like ? order by score
desc";
        //调用方法查询  将结果集的每一行都封装成一个Stutb对象,并把每一个对象都添加到集合
        //查询的结果是否需要保证线程安全???
        List<Stutb> list = jt.query(sql, new BeanPropertyRowMapper<Stutb>(Stutb.class),
"男", "%基础班%");
        //在遍历集合取出结果集之前面临一个问题,使用普通for遍历好 还是使用迭代器(增强for)?
        //特别是数据量特别大的时候一定要考虑!
        //对返回的集合进行判断,如果返回的集合实现了 RandomAccess 就使用 普通for
        //否则使用迭代器(增强for)
        if(list instanceof RandomAccess){
            for (int i = 0; i < list.size(); i++) {
                System.out.println(list.get(i));
            }
        }else {
            for (Stutb stutb : list) {
                System.out.println(stutb);
            }
        }
    }
}
```

## 4.5 如何复制某个ArrayList到另一个ArrayList中去?

- 使用clone()方法
- 使用ArrayList构造方法
- 使用addAll方法

  以上三种方式都在前面有讲解

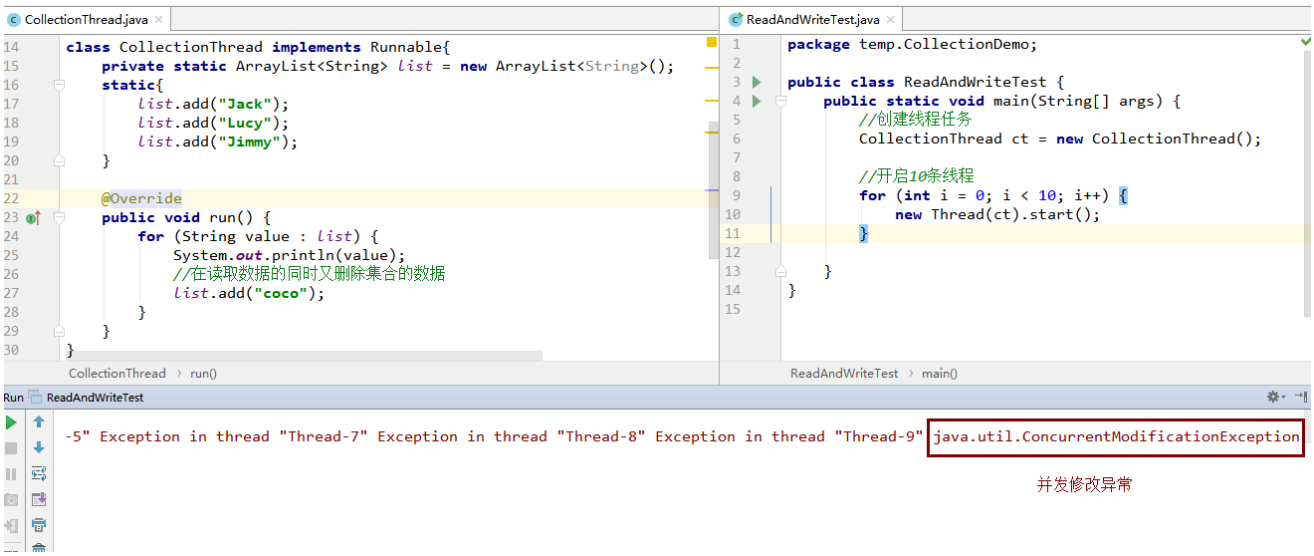
## 4.6 已知成员变量集合存储N多用户名称,在多线程的环境下,使用迭代器在读取集合数据的同时如何保证还可以正常的写入数据到集合?

普通集合`ArrayList`

```
//线程任务类
class CollectionThread implements Runnable{
    private static ArrayList<String> list = new ArrayList<String>();
```

```
    static{
        list.add("Jack");
        list.add("Lucy");
        list.add("Jimmy");
    }

    @Override
    public void run() {
        for (String value : list) {
            System.out.println(value);
            //在读取数据的同时又向集合写入数据
            list.add("coco");
        }
    }
}

//测试类
public class ReadAndWriteTest {
    public static void main(String[] args) {
        //创建线程任务
        CollectionThread ct = new CollectionThread();

        //开启10条线程
        for (int i = 0; i < 10; i++) {
            new Thread(ct).start();
        }
    }
}
```

效果图

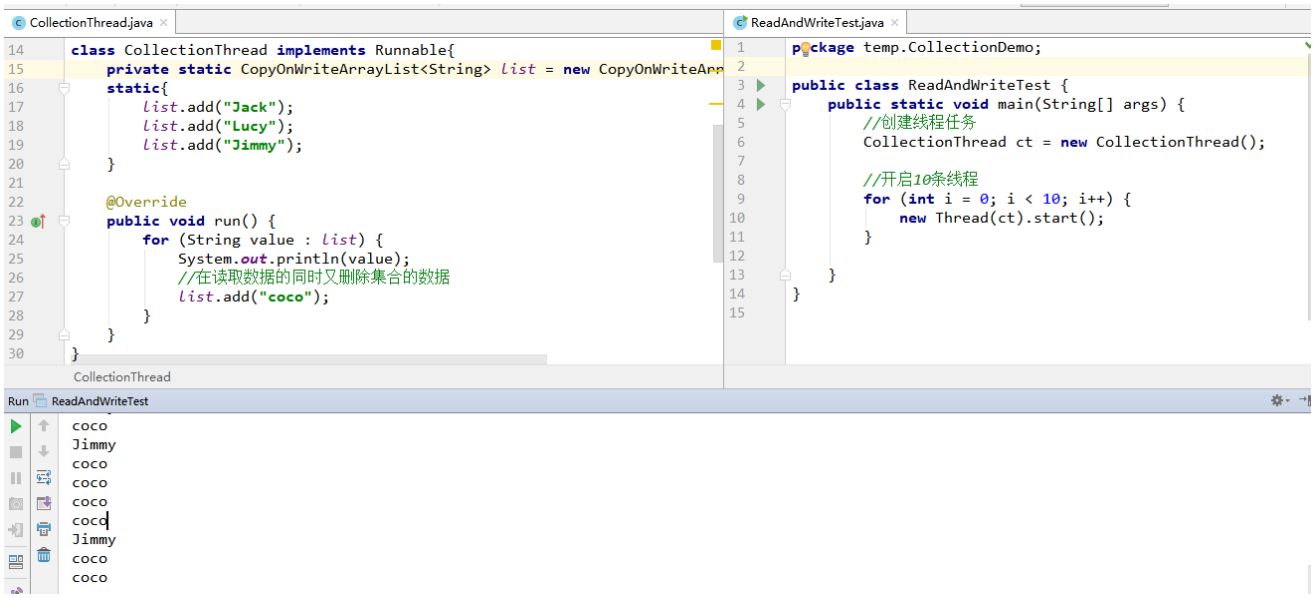



读写分离集合

```
//线程任务类
class CollectionThread implements Runnable{
    private static CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

    static{
```

```
            list.add("Jack");
            list.add("Lucy");
            list.add("Jimmy");
        }

        @Override
        public void run() {
            for (String value : list) {
                System.out.println(value);
                //在读取数据的同时又向集合写入数据
                list.add("coco");
            }
        }
    }

    //测试类
    public class ReadAndWriteTest {
        public static void main(String[] args) {
            //创建线程任务
            CollectionThread ct = new CollectionThread();

            //开启10条线程
            for (int i = 0; i < 10; i++) {
                new Thread(ct).start();
            }
        }
    }
```

效果图




## 4.7 ArrayList 和 LinkList区别?

- **ArrayList**
  - 基于动态数组的数据结构

    - 对于随机访问的get和set，ArrayList要优于LinkedList
    - 对于随机操作的add和remove，ArrayList不一定比LinkedList慢 (ArrayList底层由于是动态数组，因此并不是每次add和remove的时候都需要创建新数组)
- **LinkedList**
    - 基于链表的数据结构
    - 对于顺序操作，LinkedList不一定比ArrayList慢
    - 对于随机操作，LinkedList效率明显较低

# 5. 自定义ArrayList

略，详见代码