📖 青空的霞光B站 `JavaSE`

📅 2024年4月16日

集合类

⭐ 集合类和数学中的集合是差不多的概念，集合表示一组对象，每个对象都可以称其为元素

⭐ 集合也有很多不同的种类，比如一些集合可以有重复的元素，而有些的不行，有些的是无序，有些的是有序

⭐ 集合和数组也有，都可以表示同样的一组元素,但是它们也有不同

- 数组大小固定，集合大小可以改变
- 数数组可以存放基本数据类型，但是集合只能存放对象或者说引用类型
- 数组存放的类型只能是同一种，而集合可以存多种不同的类型的元素

集合根接口

⭐ 在Java中，它是直接将常用的集合类型都实现好了，包括顺序表，链表等等，是可以直接拿过来用的，不需要单独重写

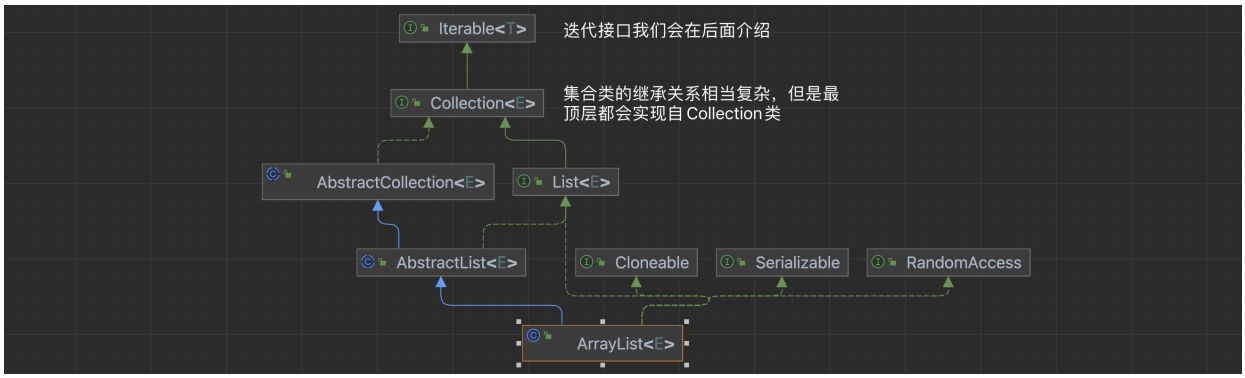
```java
//顺序表
import java.util.ArrayList; //导入顺序表，都是在java.util类型里面

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>(); //创建Arraylist
        list.add("我是ArrayList"); //同样的它也实现了添加的方法，只不过可能和之前在Java数据结构中使用的不一样
        System.out.println(list); //它也重写了toString
    }
}

//链表
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("我是LinkedList");
        System.out.println(list);
    }
}
```

⭐ 但是所有集合类都是由根接口实现的，这里拿`ArrayList`举例，它的祖先节点就是`Collection`接口



⭐ `Collection`接口中定义的基本操作

```java
public interface Collection<E> extends Iterable<E> {
    //-------这些是查询相关的操作----------

    //获取当前集合中的元素数量
    int size();

    //查看当前集合是否为空
    boolean isEmpty();

    //查询当前集合中是否包含某个元素
    boolean contains(Object o);

    //返回当前集合的迭代器，我们会在后面介绍
    Iterator<E> iterator();

    //将集合转换为数组的形式
    Object[] toArray();

    //支持泛型的数组转换，同上
    <T> T[] toArray(T[] a);

    //-------这些是修改相关的操作----------

    //向集合中添加元素，不同的集合类具体实现可能会对插入的元素有要求，
    //这个操作并不是一定会添加成功，所以添加成功返回true，否则返回false
    boolean add(E e);

    //从集合中移除某个元素，同样的，移除成功返回true，否则false
    boolean remove(Object o);


    //-------这些是批量执行的操作----------

    //查询当前集合是否包含给定集合中所有的元素
    //从数学角度来说，就是看给定集合是不是当前集合的子集
    boolean containsAll(Collection<?> c);

    //添加给定集合中所有的元素
    //从数学角度来说，就是将当前集合变成当前集合与给定集合的并集
```

```
    //添加成功返回true，否则返回false
    boolean addAll(Collection<? extends E> c);

    //移除给定集合中出现的所有元素，如果某个元素在当前集合中不存在，那么忽略这个元素
    //从数学角度来说，就是求当前集合与给定集合的差集
    //移除成功返回true，否则false
    boolean removeAll(Collection<?> c);

    //Java8新增方法，根据给定的Predicate条件进行元素移除操作
    default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();   //这里用到了迭代器，我们会在后面进行介绍
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }

    //只保留当前集合中在给定集合中出现的元素，其他元素一律移除
    //从数学角度来说，就是求当前集合与给定集合的交集
    //移除成功返回true，否则false
    boolean retainAll(Collection<?> c);

    //清空整个集合，删除所有元素
    void clear();


    //-------这些是比较以及哈希计算相关的操作----------

    //判断两个集合是否相等
    boolean equals(Object o);

    //计算当前整个集合对象的哈希值
    int hashCode();

    //与迭代器作用相同，但是是并行执行的，我们会在下一章多线程部分中进行介绍
    @Override
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, 0);
    }

    //生成当前集合的流，我们会在后面进行讲解
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }

    //生成当前集合的并行流，我们会在下一章多线程部分中进行介绍
```

```
    default Stream<E> parallelStream() {
        return StreamSupport.stream(spliterator(), true);
    }
}
```

1️⃣ 可以用代码`ArrayList`示范一下

```
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        ArrayList<String> list2 = new ArrayList<>();
        list.add("我是ArrayList");
        list.add("我是ArrayListTwo");
        list.add("我是ArrayListThree");
        list2.add("我是ArrayList");
        list2.add("我是ArrayListTwo");
        list2.add("我是ArrayListThree");
        System.out.println(list.equals(list2));  //在Java中，集合类（如ArrayList、HashSet、HashMap等）的equals()方法是比较两个集合的内容是否相等，而不是比较引用或者地址
        System.out.println(list.size()); //输出当前集合大小
        System.out.println(list.isEmpty()); //判断当前集合是否为空
        System.out.println(list.contains("AA")); //判读是否包含
        list.remove(1); //删除
        list.set(1,"我被修改了！"); //修改数据
        list.clear(); //清空全部元素
        System.out.println(list.isEmpty());
    }
```

2️⃣下面是一个详细的`ArrayList`重写列表

由于`Java` `ArrayList`的具体源代码细节可能因`JDK`版本而有所差异，我将提供一份基于`JDK 8`以后版本的`ArrayList`主要构造方法和成员方法的简要表格描述：

构造方法	作用
`ArrayList()`	创建一个默认初始容量为10的新`ArrayList`实例
`ArrayList(int initialCapacity)`	创建具有指定初始容量的新`ArrayList`实例

成员方法	作用
`boolean add(E element)`	在列表的末尾添加指定元素，如果需要则扩容，并返回true
`void add(int index, E element)`	在列表的指定索引处插入指定元素，原有元素向右移动，并可能触发扩容
`void ensureCapacity(int minCapacity)`	如果当前容量小于minCapacity，则扩容至至少minCapacity大小
`E get(int index)`	获取指定索引处的元素

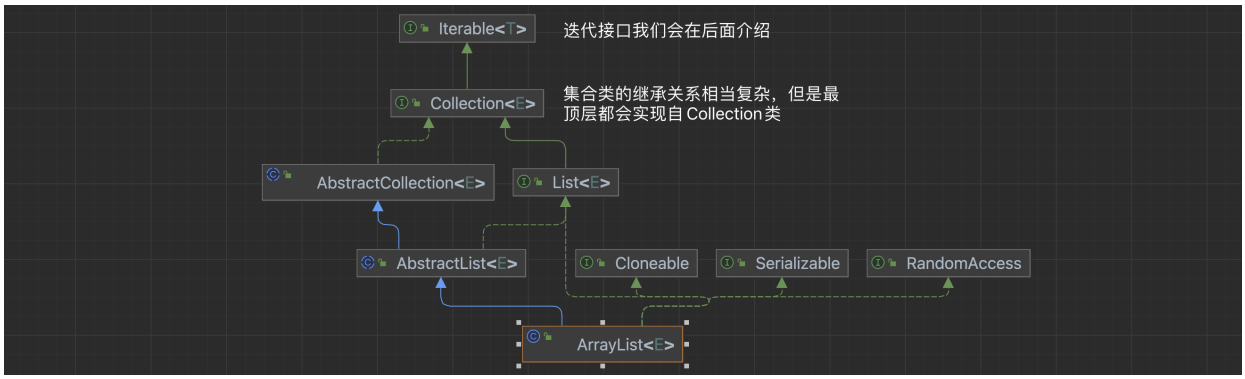
成员方法	作用
`E remove(int index)`	移除指定索引处的元素，并返回被移除的元素
`boolean remove(Object o)`	从列表中移除首次出现的指定元素，如果存在则返回true
`void clear()`	清空列表中的所有元素
`E set(int index, E element)`	用新元素替换指定索引处的旧元素，并返回被替换的旧元素
`int size()`	返回列表中的元素数量
`boolean isEmpty()`	判断列表是否为空
`boolean contains(Object o)`	判断列表是否包含指定元素
`int indexOf(Object o)`	返回列表中首次出现指定元素的索引，不存在则返回-1
`int lastIndexOf(Object o)`	返回列表中最后一次出现指定元素的索引，不存在则返回-1
`void trimToSize()`	调整列表容量为实际存储的元素数量，减少空间占用
`Object[] toArray()`	将列表转换为对象数组
`<T> T[] toArray(T[] a)`	将列表转换为指定类型的数组
`Iterator<E> iterator()`	返回一个迭代器，用于遍历列表中的元素
`ListIterator<E> listIterator()`	返回一个列表迭代器，支持双向遍历
`ListIterator<E> listIterator(int index)`	返回一个从指定索引开始的列表迭代器
`boolean addAll(Collection<? extends E> c)`	将指定集合中的所有元素添加到此列表的结尾
`boolean addAll(int index, Collection<? extends E> c)`	将指定集合中的所有元素插入到此列表的指定位置

以上并非全部方法，仅列出了`ArrayList`最常用的方法。同时，由于`ArrayList`实现了`List`接口，它还继承了一系列其他方法，如`equals()`、`hashCode()`等。并且，由于`ArrayList`也实现了`RandomAccess`接口，这意味着它可以高效地进行随机访问（通过索引）。

List列表

⭐ `List`列表，也就线性表，支持随机访问，相比之前的`Collection`接口定义的功能，还会多一些，会发现`ArrayList`它也是实现了`List`接口

⭐ `ArrayList`顺序表，它的底层是用数组实现的，内部是一个可动态扩容的数组，在之前数据结构中实现了，但是很简陋，`Java`团队帮我们定义的要比我们的规范得多，而且功能更多，而且他也是实现了`List`接口



1 `List`是集合类的一个分支，它的特性有:

- 是一个有序的集合，插入元素默认是插入到尾部，按顺序从前往后存放，每个元素都有一个自己的下标位置
- 列表中允许存在重复元素

`List接口Java`源代码，这里去除了实现`Collection`接口功能的方法，值保留了新的功能

```
//List是一个有序的集合类，每个元素都有一个自己的下标位置
//List中可插入重复元素
//针对于这些特性，扩展了Collection接口中一些额外的操作
public interface List<E> extends Collection<E> {
    ...

    //将给定集合中所有元素插入到当前结合的给定位置上（后面的元素就被挤到后面去了，跟我们之前顺序表的插入是一样的）
    boolean addAll(int index, Collection<? extends E> c);

    ...

    //Java 8新增方法，可以对列表中每个元素都进行处理，并将元素替换为处理之后的结果
    default void replaceAll(UnaryOperator<E> operator) {
        Objects.requireNonNull(operator);
        final ListIterator<E> li = this.listIterator();  //这里同样用到了迭代器
        while (li.hasNext()) {
            li.set(operator.apply(li.next()));
        }
    }

    //对当前集合按照给定的规则进行排序操作，这里同样只需要一个Comparator就行了
    @SuppressWarnings({"unchecked", "rawtypes"})
    default void sort(Comparator<? super E> c) {
```

```
        Object[] a = this.toArray();
        Arrays.sort(a, (Comparator) c);
        ListIterator<E> i = this.listIterator();
        for (Object e : a) {
            i.next();
            i.set((E) e);
        }
    }

    ...

    //-------- 这些是List中独特的位置直接访问操作 --------

    //获取对应下标位置上的元素
    E get(int index);

    //直接将对应位置上的元素替换为给定元素
    E set(int index, E element);

    //在指定位置上插入元素，就跟我们之前的顺序表插入是一样的
    void add(int index, E element);

    //移除指定位置上的元素
    E remove(int index);


    //------- 这些是List中独特的搜索操作 -------

    //查询某个元素在当前列表中的第一次出现的下标位置
    int indexOf(Object o);

    //查询某个元素在当前列表中的最后一次出现的下标位置
    int lastIndexOf(Object o);


    //------- 这些是List的专用迭代器 -------

    //迭代器我们会在下一个部分讲解
    ListIterator<E> listIterator();

    //迭代器我们会在下一个部分讲解
    ListIterator<E> listIterator(int index);

    //------- 这些是List的特殊转换 -------

    //返回当前集合在指定范围内的子集
    List<E> subList(int fromIndex, int toIndex);

    ...
}
```

2 同样的也是可以用`ArrayList`来进行操作，因为它也继承于`List`，这里只是一些常用好记的

```
public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("AAA");
        list.add("BBB");
        list.add("AAA");
        System.out.println(list.get(0)); //获取下标0的集合数据
        list.set(1,"CCC"); //修改下标为1的数据
        System.out.println(list.indexOf("AAA")); //获取当前列表中第一次一次出现的位置
        System.out.println(list.lastIndexOf("AAA")); //获取当前列表中最后一次出现的位置
    }
}
```

3 接口只定义了这些方法，具体是如何实现的是在实现类中，`ArrayList`就是`List接口`的实现类之一，下面是`ArrayList`的源码，可以很清楚的看到和之前顺序表定义差不多，底层也是数组

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{

    //默认的数组容量
    private static final int DEFAULT_CAPACITY = 10;

    ...

    //存放数据的底层数组，这里的transient关键字我们会在后面I/O中介绍用途
    transient Object[] elementData;

    //记录当前数组元素数的
    private int size;

    //这是ArrayList的其中一个构造方法
    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];   //根据初始化大小，创建当前列表
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }

    ...

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // 这里会判断容量是否充足，不充足需要扩容
        elementData[size++] = e;
        return true;
```

```
    }

    ...

    //默认的列表最大长度为Integer.MAX_VALUE - 8
    //JVM都C++实现中，在数组的对象头中有一个_length字段，用于记录数组的长
    //度，所以这个8就是存了数组_length字段（这个只做了解就行）
        private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

    private void grow(int minCapacity) {
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);    //扩容规则跟我们之前的是一样的，也是1.5倍
        if (newCapacity - minCapacity < 0)    //要是扩容之后的大小还没最小的大小大，那么直接扩容到最小的大小
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)    //要是扩容之后比最大的大小还大，需要进行大小限制
            newCapacity = hugeCapacity(minCapacity);  //调整为限制的大小
        elementData = Arrays.copyOf(elementData, newCapacity);    //使用copyOf快速将内容拷贝到扩容后的新数组中并设定为新的elementData底层数组
    }
}
```

4 使用的一些问题

1. 如果要使用一个集合类，使用接口的引用，而不是直接创建匹配的集合类，但是只能可以使用`List`接口中定义的方法，有些在实现类中的方法使用不了

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();    //使用接口的引用来操作具体的集合类实现，是为了方便日后如果我们想要更换不同的集合类实现，而且接口中本身就已经定义了主要的方法，所以说没必要直接用实现类
    list.add("科技与狠活");    //使用add添加元素
    list.add("上头啊");
    System.out.println(list);    //打印集合类，可以得到一个非常规范的结果
}
```

2. 在使用`Integer`时，要注意传参问题，集合的接口是泛型的，所以说使用`add`操作时添加的是一个引用类型，但是在使用`remove`的时候(可以上去看看源代码，上面是传入的`index`),传入的是一个`int`，而下面传入的是`Object`一个对象

```
    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])) { //这里在删除元素是，会使用equals方法判断是否为指定元素，而不是用等号，如果两个对象使用equlas方法比较是否相等，如果相等，就表示在集合中这两个就是相同的两个对象
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

```
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(10);   //添加Integer的值10
        list.remove((Integer)10);   //注意，不能直接用10，默认情况下会认为传入的是int类型值，删除的是下标为10的元素，我们这里要删除的是刚刚传入的值为10的Integer对象
        System.out.println(list);   //可以看到，此时元素成功被移除
    }


//也可以这样
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(new Integer(10));   //添加的是一个对象
        list.remove(new Integer(10));   //删除的是另一个对象
        System.out.println(list);
    }
```

3. 依照上面的问题，如果有两个`new Integer(10)`怎么办，通常是会删除排在前面的第一个元素

```
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(new Integer(10));   //添加的是一个对象
        list.add(new Integer(10));   //添加的是一个对象
        list.remove(new Integer(10));   //删除的是另一个对象
        System.out.println(list);
    }
}
```

> 在这段Java代码中，`list.remove(new Integer(10))`尝试从列表中删除一个新创建的`Integer(10)`对象。然而，列表中已经存在两个通过`new Integer(10)`创建的对象。
>
> 虽然它们都表示相同的数值10，但由于它们是通过两次独立的`new Integer(10)`创建的，因此它们是两个不同的对象，具有不同的内存地址。Java集合类（如`ArrayList`）在执行`remove`操作时，默认调用的是对象的`equals`()方法来判断是否为同一个对象。
>
> Integer类重写了`equals`()和`hashCode`()方法，使得两个具有相同数值的Integer对象在用equals()方法比较时会被认为相等。所以在这个例子中，`list.remove(new Integer(10))`将会删除列表中第一个遇到的数值为10的Integer对象。

> 但由于列表中添加了两个相同的`Integer`对象，我们不能确定`remove`操作会删除列表中的哪个对象。在实际情况中，这可能会删除列表中的任意一个数值为10的对象。运行这段代码后，输出结果可能是`[10]`或者是空列表`[]`，具体取决于`ArrayList`内部实现的细节。但在实践中，我们不推荐这样依赖于不可预测的行为，应当尽量避免在集合中添加多个相同的可变对象（除非明确知道`equals`和`hashCode`方法已正确重写）。

4. 列表允许存在相同的元素

```
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        Integer num = 10;
        list.add(num);
        list.add(num);
        System.out.println(list);
    }
```

5. 集合类支持嵌套，甚至可以在集合类中套一个集合类

```
public class Main {
    public static void main(String[] args) {
        List<List<String>> list = new LinkedList<>();
        List<Integer> list2 = new LinkedList<>();
        list.add(new LinkedList<>());   //集合中的每一个元素就是一个集合，这个套娃是可以一直套下去的
        System.out.println(list.get(0).isEmpty()); //可以看到可以继续调用集合类中集合类的isEmpty方法
    }
}
```

6. 使用`Arrays`工具类的`asList`方法可以快速生成一个只读的`list`

```
    public static void main(String[] args) {
        List<String> list = Arrays.asList("A", "B", "C");    //非常方便
        list.remove(0); //删除会报错,因为asList无法被修改
        System.out.println(list);
    }
```

```
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint
    at java.util.AbstractList.remove(AbstractList.java:161)
    at fun.tanc.Main.main(Main.java:12)
```

7. 如果将`asList`作为参数传入一个`list`就还是可以修改

```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
        list.remove(0); //可以修改
        System.out.println(list);
    }
//输出
[B, C]
```

5️⃣ 链表，`LinkedList`同样是`List`的实现类，只不过它是采用的链式实现，也就是我们之前讲解的链表，只不过它是一个双向链表，也就是同时保存两个方向

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;

    //引用首结点
    transient Node<E> first;

    //引用尾结点
    transient Node<E> last;

    //构造方法，很简单，直接创建就行了
    public LinkedList() {
    }

    ...

    private static class Node<E> {   //内部使用的结点类
        E item;
        Node<E> next;   //不仅保存指向下一个结点的引用，还保存指向上一个结点的引用
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    ...
}
```

`LinkedList`的使用和`ArrayList`的使用几乎相同，各项操作的结果也是一样的，在什么使用使用`ArrayList`和`LinkedList`，我们需要结合具体的场景来决定，尽可能的扬长避短。

只不过`LinkedList`不仅可以当做`List`来使用，也可以当做双端队列使用