第5章 集合

5.1 集合概述

在前面的章节中学习了数组，数组可以存储同一种类型的对象，并且数组的长度是不可变的。为了使程序能够方便地存储和操作不定数量、不同类型的对象，JDK 中提供了 Java 集合类来存储对象（实际上是对象的引用），这些集合类位于 java. util 包中。

Java 中集合类可以分为两大类，分别是单列集合 Collection 和双列集合 Map。

（1）Collection。Collection 为单列集合的根接口，用来存储一组集合元素。Collection 有两个重要的子接口，即 List 和 Set。List 是一个元素排列有序、可重复的集合，List 中的每个元素都有索引，类似于 Java 的数组。List 接口有 3 个主要的实现类，分别为 ArrayList、LinkedList 和 Vector。Set 集合的特点是元素排列无序、不可重复，该接口主要有两个实现类，即 HashSet 和 TreeSet。

（2）Map。Map 为双列集合的根接口，用来存储具有键（key）、值（value）映射关系的一对元素。一个 Map 中 key 唯一不可重复，每个 key 只能映射一个 value。Map 有两个主要的实现类，即 HashMap 和 TreeMap。

接下来通过一张图来描述集合的继承关系，如图 5-1 所示。

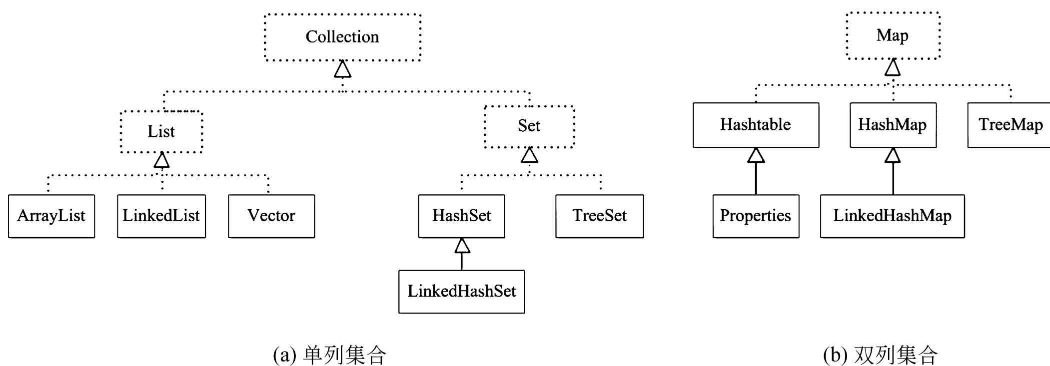

图 5-1　集合的继承关系

图 5-1 中列举出了 Java 中常用的集合类。其中，虚线框表示的是接口类型，实线框表示的是具体的实现类。

5.2 List 集合

5.2.1 List 接口介绍

List 接口是一个有序的 Collection,使用此接口可以控制每个元素插入的位置,也可以通过索引来访问 List 中指定的元素。List 集合中第一个元素的索引为 0,并且允许存储相同的元素。List 接口常用方法如表 5-1 所示。

表 5-1 List 接口常用方法

方法声明	功能描述
boolean add(Object element)	将指定元素 element 追加到 List 集合的末尾
void add(int index,Object element)	将指定元素 element 插入到 List 集合中指定位置
Object get(int index)	返回集合中指定位置的元素
Object remove(int index)	删除指定位置的元素
Object set(int index,Object element)	将指定位置 index 处元素替换成 element 元素,并返回替换后的元素
int indexOf(Object o)	返回对象 o 在 List 集合中首次出现的位置索引
int lastIndexOf(Object o)	返回对象 o 在 List 集合中最后一次出现的位置索引
List<E> subList(int fromIndex,int toIndex)	返回此集合中指定的 fromIndex(含)到 toIndex 之间所有元素的集合
Object[] toArray()	返回一个包含此集合中所有元素的数组

5.2.2 ArrayList 集合

ArrayList 是最常用的一种 List 实现类,其内部的存储结构是通过数组实现的。数组的缺点是每个元素之间不能有间隔,当数组大小不满足时需要增加存储能力,就要将已经有数组的数据复制到新的存储空间中。ArrayList 通过索引的方式来访问集合中的元素,因此它适合做随机查找和遍历操作。当从 ArrayList 的中间位置插入或者删除元素时,需要对数组进行复制、移动,代价比较高,因此,它不适合插入和删除。

接下来通过一个案例来介绍 ArrayList 集合的创建和元素的存取,如例 5-1 所示。

例 5-1 Example01.java。

```
import java.util.ArrayList;
public class Example01 {
        public static void main(String[] args) {
            //创建一个 ArrayList 集合
            ArrayList arrayList = new ArrayList();
            //集合中可以存储不同类型的对象和相同的元素
            arrayList.add(new Integer(10));
            arrayList.add("element1");
            arrayList.add("element1");
```

```
            System.out.println("集合中的元素有: " + arrayList);
            System.out.println("集合的长度: " + arrayList.size());
            System.out.println("第 1 个元素是: " + arrayList.get(0));
        }
}
```

例 5-1 的运行结果如图 5-2 所示。

图 5-2　例 5-1 的运行结果

在例 5-1 中，首先创建了 ArrayList 集合，然后调用 add(Object element)方法向集合中添加了 3 个元素，最后通过使用 ArrayList 的引用变量名、size()方法和 get(int index)方法分别打印出了集合中存放的元素、集合的长度和指定位置的元素。

从运行结果(图 5-2)可以看出，一个 ArrayList 集合中可以存储不同类型的对象和相同的元素，并且集合中第一个元素的索引为 0。

5.2.3　LinkedList 集合

LinkedList 是一种用链表结构存储数据的双向循环链表。LinkedList 类每一个节点用内部类 Node 表示，每一个节点都使用引用的方式来指向它的前一个节点和后一个节点，当向链表中插入或者删除元素时，只要修改元素之间的引用关系即可。此外，在 LinkedList 中也定义了两个 Node 类型的 first 和 last 属性分别指向链表的第一个元素和最后一个元素，当链表为空时，first 和 last 都为 null 值。链表中的 LinkedList 很适合数据的动态插入和删除，随机访问和遍历速度比较慢。LinkedList 集合添加元素的过程如图 5-3 所示。

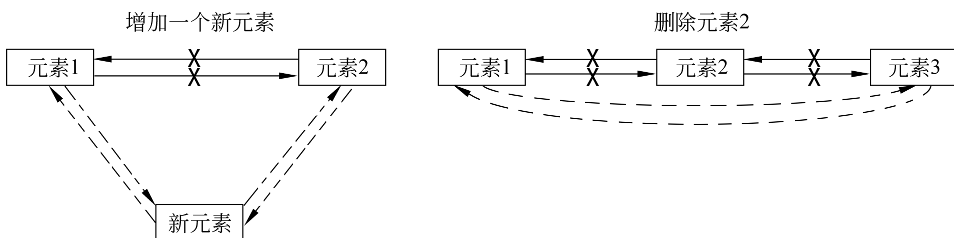

图 5-3　LinkedList 向链表中添加元素和删除元素

当向元素 1 和元素 2 之间新增一个元素时，只需要将新增元素的前后引用分别指向元素 1 和元素 2，同时让元素 1 的后引用及元素 2 的前引用指向新元素即可；当删除位于元素 1 和元素 3 之间的元素 2 时，需要分别删除元素 1 和元素 2 之间的引用以及元素 2 和元素 3 之间的引用，并且建立元素 1 和元素 3 之间的引用即可。

另外，它还提供了 List 接口中没有定义的方法，专门用于操作表头元素和表尾元素，可以当作堆栈、队列和双向队列使用，如表 5-2 所示。

表 5-2　LinkedList 特有方法

方 法 声 明	功 能 描 述
void add(int index,element)	将指定的元素插入到集合中指定的位置
void addFirst(Object o)	将指定元素插入到集合的开头
void addLast(Object o)	将指定元素添加到集合的结尾
Object get(int index)	返回集合中指定位置的元素
Object getFirst()	返回集合中的第一个元素
Object getLast()	返回集合中的最后一个元素
Object removeFirst()	移除并返回集合中的第一个元素
Object removeLast()	移除并返回集合中的最后一个元素
boolean offer(Object o)	将指定元素添加到集合的结尾
boolean offerFirst(Object o)	将指定元素添加到集合的开头
boolean offerLast(Object o)	将指定元素添加到集合的结尾
Object peek()	获取集合中的顶部元素
Object peekFirst()	获取集合中的第一个元素
Object peekLast()	获取集合中的最后一个元素
Object poll()	移除并返回集合中的顶部元素
Object pollFirst()	移除并返回集合中的第一个元素
Object pollLast()	移除并返回集合中的最后一个元素
void push(Object o)	将指定元素添加到集合的开头
Object pop()	移除并返回集合中的第一个元素

下面通过一个案例来学习 LinkedList 中元素的增加、获取和删除的操作，如例 5-2 所示。

例 5-2　Example02.java。

```
import java.util.LinkedList;
public class Example02 {
    public static void main(String[] args) {
        LinkedList linkList = new LinkedList();
        linkList.add("obj1");
        linkList.add("obj2");
        System.out.println(linkList);
        linkList.push("push");                    //向集合头部添加元素
        linkList.offer("offer");                  //向集合尾部追加元素
        System.out.println(linkList);
        System.out.println(linkList.peek());      //获取集合顶部元素
        linkList.poll();                          //删除集合顶部元素
        linkList.removeLast();                    //删除集合最后一个元素
        System.out.println(linkList);
    }
}
```

例 5-2 的运行结果如图 5-4 所示。

图 5-4 例 5-2 的运行结果

例 5-2 中,首先使用 void add(int index,element) 方法向集合中添加了两个元素,接着分别使用 push(Object o)和 offer(Object o)方法向集合的头部和尾部各添加了一个元素,然后使用 peek()方法获取了集合中的顶部元素,最后使用 poll()和 removeLast()删除了集合中顶部元素和最后一个元素。

5.3 Collection 集合遍历

Collection 集合框架提供了 3 种遍历集合的方法,分别是 Iterator 遍历集合、for-each 遍历集合和 forEach 遍历集合,接下来分别对这 3 种方法进行介绍。

5.3.1 Iterator 遍历集合

Iterator 接口又称为迭代器,是 Java 集合框架中的一员,主要用来遍历集合中的元素,可以使用 Collection 接口中的 iterator()方法获取集合的迭代器。接下来通过一个案例来学习 Iterator 遍历集合的使用,如例 5-3 所示。

例 5-3 Example03.java。

```
import java.util.ArrayList;
import java.util.Iterator;
public class Example03 {
    public static void main(String[] args) {
        //创建 ArrayList 集合
        ArrayList list = new ArrayList();
        list.add("obj1");
        list.add("obj2");
        list.add("obj3");
        //获取迭代器
        Iterator iterator = list.iterator();
        //迭代集合中所有的元素
        while(iterator.hasNext()) {
          //取出集合中的下一个元素
          Object obj = iterator.next();
          System.out.println(obj);
        }
    }
}
```

例 5-3 的运行结果如图 5-5 所示。

图 5-5　例 5-3 的运行结果

如例 5-3 所示，遍历 ArrayList 集合中的元素时，首先调用 ArrayList 的 iterator()方法获取集合的迭代器；然后调用迭代器的 hasNext()方法判断集合中是否存在下一个元素，如果返回 true，则证明集合中存在下一个元素，否则返回 false；最后调用迭代器的 next()方法返回集合中的下一个元素。值得注意的是，在调用 next()方法之前需要调用 hasNext()方法；否则会抛出 NoSuchElementException 异常。

5.3.2　for-each 遍历集合

for-each 循环也称为增强 for 循环，也可以使用 for-each 循环遍历集合的元素。for-each 循环的一般语法如下：

```
for(集合中元素的类型 临时变量: 集合变量) {
}
```

接下来通过一个案例来学习 for-each 循环遍历集合中元素的使用，如例 5-4 所示。

例 5-4　Example04.java。

```
import java.util.ArrayList;
public class Example04 {
    public static void main(String[] args) {
        //创建 ArrayList 集合
        ArrayList list = new ArrayList();
        list.add("obj1");
        list.add("obj2");
        list.add("obj3");
        for(Object obj: list) {
            System.out.println(obj);
        }
    }
}
```

例 5-4 的运行结果如图 5-6 所示。

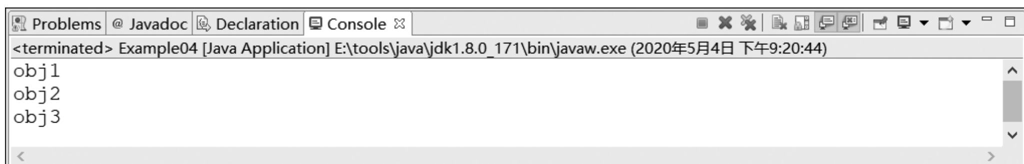

图 5-6　例 5-4 的运行结果

从例 5-4 可以看出，for-each 循环遍历集合的语法非常简洁，没有循环条件，不必担心在遍历的过程中会超出集合的长度。

需要注意的是，在使用 for-each 循环时，不能从集合中删除元素，否则会抛出 ConcurrentModificationException 异常，如例 5-5 所示。

例 5-5　Example05.java。

```
import java.util.ArrayList;
public class Example05 {
    public static void main(String[] args) {
        //创建 ArrayList 集合
        ArrayList list = new ArrayList();
        list.add("obj1");
        list.add("obj2");
        list.add("obj3");
        for(Object obj: list) {
            list.remove(obj);
        }
    }
}
```

例 5-5 的运行结果如图 5-7 所示。

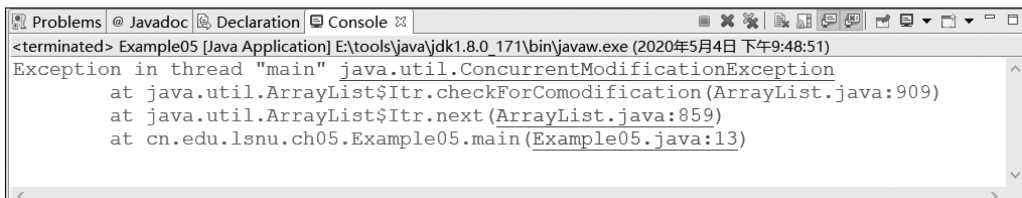

图 5-7　例 5-5 的运行结果

在例 5-5 中，使用了 ArrayList 的 remove(Object o)方法来删除集合中的元素，结果抛出了 ConcurrentModificationException 异常，实际上 Java 的 for-each 循环就是将 List 对象的遍历托管给了迭代器 Iterator，从集合中删除元素导致迭代器预期的迭代次数发生改变，导致迭代器的结果不准确。

5.3.3　forEach 遍历集合

在 JDK 8 中，可以使用 Iterable 接口的 forEach(Consumer action)方法来遍历 Collection 集合中的元素，该方法的入参是一个函数式接口，Iterable 是 Collection 的父接口。接下来通过一个案例来介绍如何使用 forEach(Consumer action)方法来遍历集合中的元素，如例 5-6 所示。

例 5-6　Example06.java。

```
import java.util.ArrayList;
public class Example06 {
    public static void main(String[] args) {
        //创建 ArrayList 集合
```

```
        ArrayList list = new ArrayList();
        list.add("obj1");
        list.add("obj2");
        list.add("obj3");
        //使用 forEach(Consumer action)方法遍历集合
        list.forEach(System.out::println);
    }
}
```

例 5-6 的运行结果如图 5-8 所示。

图 5-8　例 5-6 的运行结果

在例 5-6 中，使用 forEach(Consumer action)方法来遍历集合中的元素，方法的入参是一个 Lambda 表达式形式的函数式接口，代码非常简洁。

5.4　Set 集合

5.4.1　Set 接口介绍

Set 接口继承自 Collection 接口，Set 体系集合用于存储无序（存入和取出的顺序不一定相同）元素，并且值不可重复。Set 接口有两个主要的实现类，分别是 HashSet 和 TreeSet。HashSet 根据对象的哈希值来确定元素在集合中的存储位置，具有良好的存取和查找性能。TreeSet 以二叉树结构来存储元素，能够对集合中的元素进行排序。

5.4.2　HashSet 集合

HashSet 是 Set 接口的一个实现类，其内部封装了 HashMap。HashSet 中存储的元素无序并且不可重复，元素存放的位置根据元素的哈希值来确定。当向 HashSet 中添加一个元素时，首先会调用该元素的 hashCode()方法来获取对象的哈希值，如果该哈希值对应的存储位置上没有元素，则直接将该元素存入；如果该位置已存有元素，则接着调用元素的 equals()方法，如果 equals 结果为 true，HashSet 则认为该元素已重复，将该元素舍弃；如果 equals 结果为 false，则将该元素存入集合。整个存储流程如图 5-9 所示。

接下来通过一个案例来演示 HashSet 的使用方法，如例 5-7 所示。

例 5-7　Example07.java。

```
import java.util.HashSet;
public class Example07 {
    public static void main(String[] args) {
        HashSet set = new HashSet();
```

```
            set.add("obj1");
            set.add("obj2");
            set.add("obj3");
            set.add("obj3");//相同的元素不会被添加到 Set 集合中
            //输出集合中的元素
            set.forEach(System.out::println);
        }
    }
```

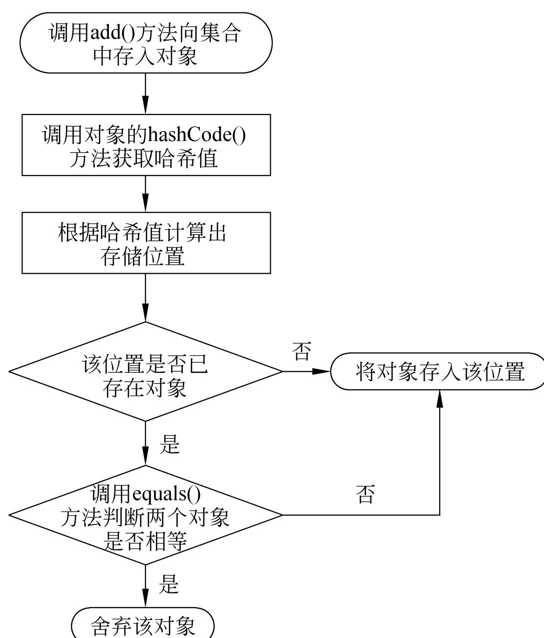

图 5-9 HashSet 存储对象流程

例 5-7 的运行结果如图 5-10 所示。

图 5-10 例 5-7 的运行结果

在例 5-7 中，向 HashSet 集合中添加了 4 个元素，其中添加了一个重复的元素 obj3，从运行结果可以看出，重复的元素 obj3 没有被添加到 Set 集合中，并且存入和取出元素的顺序并不相同。

我们知道为了保证 HashSet 中的元素不可重复，在向 HashSet 中存入对象时，需要重写 Object 类中的 hashCode()和 equals()方法。在 Java 中，一些基本数据包装类、String 类等都已经默认重写了 hashCode()和 equals()方法。但是如果开发者向 HashSet 集合中添加自定义的数据类型，必须增加重写的 hashCode()和 equals()方法，才能保证数据的唯一

性,如例 5-8 所示。

例 5-8 Example08.java。

```
import java.util.HashSet;
public class Example08 {
    public static void main(String[] args) {
        HashSet set = new HashSet();
        Order order1 = new Order("1", "fruit");
        Order order2 = new Order("2", "book");
        Order order3 = new Order("2", "book1");
        Order order4 = new Order("3", "book");
        set.add(order1);
        set.add(order2);
        set.add(order3);
        set.add(order4);
        System.out.println(set);
    }
}

class Order {
    String orderId;
    String goods;
    //构造方法
    public Order(String orderId,String goods) {
        this.orderId=orderId;
        this.goods = goods;
    }
    public String toString() {
        return orderId + ":" + goods;
    }
    //重写 hashCode()方法
    @Override
    public int hashCode() {
        return orderId.hashCode();              //返回 orderId 的哈希值
    }
    //重写 equals()方法
    @Override
    public boolean equals(Object obj) {
        //定义一个 boolean 类型的返回标识
        boolean flag = false;
        if(this == obj) {                       //判断是否是同一个对象
            flag = true;
        }else if (!(obj instanceof Order)) {    //判断对象是否为 Order 类型
            flag = false;
        }else {
            Order order = (Order) obj;          //将对象强制转换为 Order 类型
            flag = this.orderId.equals(order.orderId);          //判断 id 值是否相同
        }
        return flag;                            //返回判断结果
    }
}
```