第 5 章

Java 集合框架与泛型

Java 集合框架为有效地组织和管理数据，提供了一些数据结构和算法。Java 集合框架主要由包 java.util 内的两个接口（Collection 和 Map）来定义，而包 java.lang 内的接口 Iterable 用于迭代集合对象。Java 集合框架设计以接口为基础，符合“松耦合”机制，合理地使用集合 API 可以为程序员提供许多便利。Java 集合框架在后续课程（如 Java EE 和 Android）中经常会使用。本章学习要点如下：

- 掌握 Java 集合框架的顶层结构；
- 掌握 Java 集合框架的层次结构（接口—抽象实现类—实现类）；
- 掌握接口 Collection 的子接口 List 的实现类 ArrayList 的使用；
- 掌握接口 Map 的实现类 HashMap 的使用；
- 掌握接口 List 嵌套接口 Map 的用法；
- 了解接口 Collection 的子接口 Set 和 Queue 的实现类的使用。

5.1　Java 集合框架概述与泛型

5.1.1　Java 集合框架的主要接口

由第 2 章的内容可知，数组是用来存储对象（当然可以存储基本数据类型）的一种容器，但是数组的长度是固定的（一旦创建后，数组长度不可更改），不适合在对象数量未知的情况下使用。此外，数组元素类型必须相同。

集合是由具有相同性质的一类事物所组成的一个整体，Java 集合只能存储对象，其对象类型可以不一样，长度也可变。

在 Java 集合框架里，接口 java.util. Collection 和接口 java.util. Map 是两个独立的接口。其中，接口 Collection 继承了接口 Iterable，接口 Map 间接继承了接口 Iterable（参见 5.3 节）。接口 List、Set 和 Queue 又分别是接口 Collection 的子接口，而接口 Map 没有包含任何子接口。

Java 主要集合框架如图 5.1.1 所示。

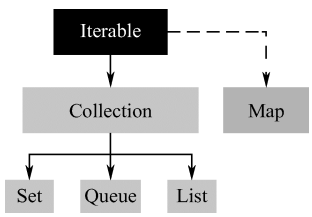

图 5.1.1　Java 主要集合框架的接口体系

注意：

（1）接口 Map 与接口 Collection 没有关系，是相互独立的。

（2）所有的集合类均直接或间接地实现了接口 Iterable。

5.1.2 迭代接口 Iterable 与迭代器 Iterator

为了遍历 Collection 集合元素，Java 引入了 Iterable 类型（Iterable 表示可迭代之意）。Java 接口 java.lang.Iterable 的抽象方法 iterator() 返回一个接口 Iterator 类型的对象。第一次调用接口 Iterator 的 next() 方法时，它返回序列的第一个元素。

Java 的迭代器接口 java.util.Iterator 定义了操作 Java 集合的方法，Iterator 用于遍历集合中元素，定义了以下三种方法：

- hasNext()：判断是否还有下一个元素。如果仍有元素可以迭代，则返回 true；
- next()：返回下一个元素；
- remove()：删除当前元素。

接口 java.lang.Iterable 和接口 java.util.Iterator 的定义（JDK 1.8 版本），如图 5.1.2 所示。

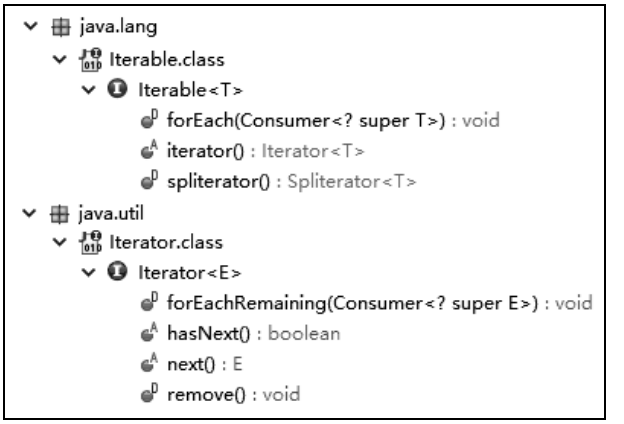

图 5.1.2 Java 可迭代接口与迭代器接口

接口 Iterable 和接口 Iterator 是两个相关联的接口。接口 Iterable 为所有 Collection 集合的遍历定义了抽象方法 iterator()，它的返回值是 Iterator 接口类型；而接口 Iterator 是一个迭代工具。

迭代器 Iterator 可以实现对 Collection 集合的迭代访问，即可以很方便地访问 Collection 集合中的每一个元素。Collection 接口提供了一个 iterator() 方法，用于获取集合中所有元素的迭代器，可以用此对象依次访问集合中的元素。

重复调用 next() 方法即可依次访问 Collection 集合中的元素，并在访问到达集合尾部时，抛出 NosuchElementExceptioin 异常。因此，调用 next() 方法前应先调用 hasNext() 方法判断集合中是否还有下一个元素未访问，如果还有此迭代器未访问到的元素，hasNext() 方法返回 true，否则返回 false。

如果需要访问集合中的所有元素，在满足 hasNext() 返回 true 的条件下，使用对应的 Iterator 对象反复调用 next() 方法，即可实现对所有元素的遍历，代码模板如下：

```
//先定义 Collection 集合对象 coll
Iterator    iter = coll.iterator();
while (iter.hasNext()){
      Object ob = iter.next();              //对 ob 的操作
}
```

对于 Collection 集合，可包含重复元素，并且元素是有顺序的。List 集合是一组允许重复且有序的元素，通过元素的 index 值（标明该元素在列表中的位置）来查找该元素。Set 集合不允许重复，并且对象之间没有指定的顺序。Queue 集合用于模拟队列这种数据结构，遵循“先进先出”的原则，不允许随机访问队列中的元素。

对 Map 集合遍历时，先得到键名的 Set 集合，再对 Set 集合进行遍历，得到相应的键值（即键名 key 到键值 value 的映射），其代码模板如下：

```
//先定义 Map 集合对象 map
Set<String>    ks = map.keySet();        //假定 Map 集合里键名类型为 String，生成 Set 集合
for (String key : keySet) {
      //根据键名得到键值
      System.out.println("key= " + key + " and value= " + map.get(key));
}
//  也可使用 Collection 集合通用的迭代器进行迭代
```

注意：自 JDK 1.5 版本之后，Iterator 的应用逐渐淡出，取而代之的是 for 或 foreach（for 的简化版本）。

遍历 Map 集合的另一种方式是，先得到 Map 映射项（是键名和键值的统一体）的 Set 集合，再得到相应的键名的 Set 集合，进而遍历 Map 集合（详见 5.3.3 节）。

5.1.3　Java 泛型

泛型是 Java SE 1.5 版本之后的新特性。在 Java SE 1.5 版本之前，Java 通过对类型 Object 的引用来实现参数类型的“任意化”，特点则是需要进行显式的强制类型转换，但编译器无法发现强制类型转换可能引起的异常，异常只有在运行时才出现，这将成为系统的安全隐患。

如今，在 Java 集合框架的 API 中，绝大部分接口和类都已经泛型化。事实上，在 eclipse-jee 中，通过链接跟踪方式，可以打开接口 List 的源码如下：

```
public interface List<E> extends Collection<E> {        //使用 E 定义泛型类
      int size();
      boolean isEmpty();
      boolean contains(Object o);
      Iterator<E> iterator();                            //使用 E 声明泛型方法
      Object[ ] toArray();
      <T> T[ ] toArray(T[ ] a);                          //方法及其参数使用了泛型 T
```

```
    boolean add(E e);
    //其他成员略
}
```

上述代码表明：List<E>表示该集合由 E 类型的实例对象组成，即 E 是泛型。

泛型的本质是参数化类型，即所操作的数据类型被指定为一个参数，此参数类型可以用在类、接口和方法的声明及创建中，分别被称为泛型类、泛型接口及泛型方法。

注意：泛型能提高程序的通用性，定义泛型方法时需要使用一对尖括号<>来表示。

```
原数组：[1, 2, 3, 4, 5]
交换后：[1, 4, 3, 2, 5]
原数组：[aa, bb, cc, dd, ee]
交换后：[aa, dd, cc, bb, ee]
```

图 5.1.3　两个数组元素交换前后

【例 5.1.1】 使用 Java 泛型，分别输出数值数组与字符串数组中 2 号与 4 号元素交换的结果，如图 5.1.3 所示。

方法一：在成员方法声明及其方法内部分别使用泛型 T，而在类声明里没有使用泛型，程序代码如下：

```
public class TestRawTypeMethod {
    // 定义交换数组中两个元素的泛型方法
    public static <T> void changePosition(T[] arr, int index1, int index2) {
        T temp = arr[index1];arr[index1] = arr[index2];arr[index2] = temp;
    }
    public static void main(String[ ] args) {
        Integer[] arr1 = new Integer[] { 1, 2, 3, 4, 5 };      //对象数组
        //int[]arr1=new int[]{1,2,3,4,5};                      //调用时会报错，泛型不是值类型
        System.out.println("原数组：" + Arrays.toString(arr1));      //输出数组
        changePosition(arr1, 1, 3);                                  //交换位置
        System.out.println("交换后：" + Arrays.toString(arr1));      //输出数组
        String[] arr2 = new String[] { "aa", "bb", "cc", "dd", "ee" };
        System.out.println("原数组：" + Arrays.toString(arr2));      //输出数组
        changePosition(arr2, 1, 3);                                  //交换位置
        System.out.println("交换后：" + Arrays.toString(arr2));      //输出数组
    }
}
```

方法二：先定义数组实用工具类 ArrayUtils，在类及其成员方法声明和方法代码中分别使用泛型 T，程序代码如下：

```
import java.util.Arrays;
class ArrayUtils<T> {                                    //定义泛型类
    public void changePosition(T[] arr, int i, int j) {  //第一参数为对象数组
        T tem = arr[i];arr[i] = arr[j];arr[j] = tem;
    }
    public void reverse(T[] arr) {                       //倒序数组
        for (int i = 0; i < arr.length / 2; i++) {
            T tem = arr[i];arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = tem;
        }
    }
}
```

```
public class TestRawTypeClass {                          //主类
    public static void main(String[] args) {
        ArrayUtils<Integer> au1 = new ArrayUtils<Integer>();//创建泛型类对象
        Integer[] arr1 = new Integer[] { 1, 2, 3, 4, 5 };
        System.out.println("原数组："+Arrays.toString(arr1)); //输出数组
        au1.reverse(arr1);                                   //调用泛型方法：倒排数组
        System.out.println("反转后："+Arrays.toString(arr1)); //输出数组

        ArrayUtils<String> au2 = new ArrayUtils<String>();   //创建泛型类对象
        String[] arr2 = new String[] { "aa", "bb", "cc", "dd", "ee" };
        System.out.println("原数组："+Arrays.toString(arr2)); //输出数组
        au2.changePosition(arr2, 1, 3);                      //调用泛型方法：交换位置
        System.out.println("对换后："+Arrays.toString(arr2)); //输出数组
    }
}
```

注意：如果不使用泛型方法或泛型类，则本例需要针对不同数据类型写多个方法。

5.2　Collection 集合及其遍历

接口 Collection 定义一组抽象方法，用于实现集合元素的通用操作，如增加、删除（一个元素或所有元素）、查询、统计集合元素个数、转换为数组和遍历集合等，如图 5.2.1 所示。

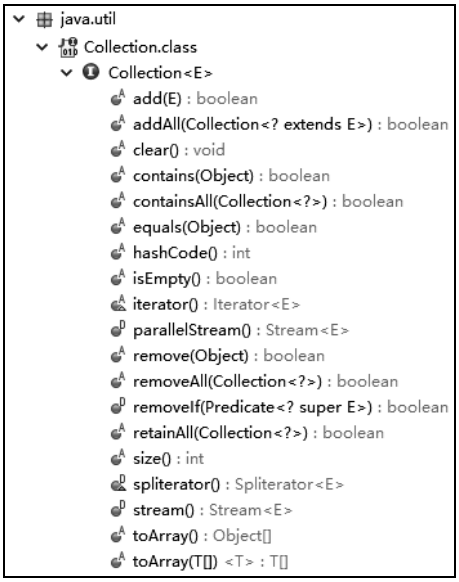

图 5.2.1　接口 Collection 的定义

注意：

（1）接口 Collection 定义的抽象方法，在其实现类 ArrayList、HashSet 和 LinkedList 中以不同的方式实现。

（2）Collection 集合中的 size() 方法对应于数组中的 length 属性，用于统计元素个数。

（3）工具类 java.util.Collections 提供了各种有关集合操作的静态方法，如排序方法 sort() 等。

5.2.1 List 接口及其常用实现类

List 集合是一组能包含重复元素的有序集合。与数组一样，List 集合的首元素的索引也是 0。List 集合有如下特征：

- 元素有序排列；
- 可以有重复元素；
- 可以随机访问，即通过元素索引来快速访问元素。

List 接口作为 Collection 接口的子接口，大部分是继承 Collection 接口的方法，此外，少量提供了一些方法，如表 5.2.1 所示。

表 5.2.1 List 接口的常用方法

方法参数及返回值类型	方法功能描述
E get (int)	获取集合中指定位置的元素
E set(int,E)	用指定的元素替换指定位置上的元素
void add(int,E)	把新元素插入到集合中指定位置
int indexOf(object)	获取指定元素在集合中第一次出现的位置
int lastIndexOf(object)	获取指定元素在集合中最后出现的位置
E remove(int)	删除指定位置的元素
List<E> subList(int, int)	获取集合中起止位置元素所组成的子列表

List 接口有不同的实现类，抽象类 AbstractList 提供 List 接口的骨干实现，从而最大限度地减少了实现由“随机访问”数据存储（如数组）支持的接口所需的工作。另外，List 接口定义了一些抽象方法用以对这些实现类方法进行抽象。List 接口的实现类有 ArrayList、LinkedList 和 Vector 等。

List 接口的实现类及其继承关系，如图 5.2.2 所示。

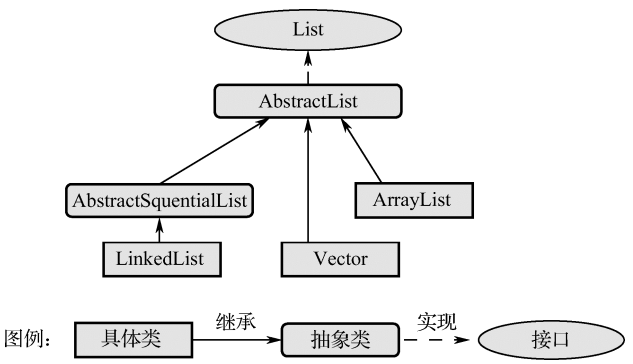

图 5.2.2 List 接口的实现类

注意：

（1）当需要保留存储顺序，并且保留重复元素时，使用 List 集合。

（2）具体类的选取原则是：查询较多时，使用 ArrayList 类；存取较多时，使用 LinkedList 类；需要线程安全时，使用 Vector 类。

1．ArrayList

ArrayList 类的内部实现基于内部数组 Object[]，类似于可变长的数组。在 ArrayList 的前面或中间插入数据时，必须将其后的所有元素顺序地后移，需花费较多时间，因此，当程序主要是在后面添加元素，并且需要随机地访问其中的元素时，使用 ArrayList 能得到较好的性能。

注意：

（1）ArrayList 创建的集合，相应于线性表中的顺序表，优点是适合随机查找，不适合插入和删除。

（2）java.util.Arrays 类主要提供了用于操作 List 集合的静态方法，如 toString() 和 sort() 等。

（3）定义 List 集合时，应用泛型才能保证类型安全。

【例 5.2.1】 测试 List 接口及其实现类 ArrayList 和工具类 Arrays。

测试程序的源代码如下：

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class ArrayAndList {
    public static void main(String[] args) {
        //创建一个只能插入 String 类型元素的 List 对象
        List<String> names = new ArrayList<String>();   // List 集合不必指定大小
        names.add("张三");
        System.out.println(names.get(0));              //获取
        // 下一行代码无法编译通过，使用泛型后编译器将进行类型检查
        // names.add(new Integer(4));                  //只能添加 String 类型的对象
        List list = new ArrayList();                   //未使用泛型，类型不安全
        list.add("测试"); list.add(20);                //List 集合长度可变
        System.out.println(list.get(0));               //获取 List 集合元素，使用方法 get()
        System.out.println(list.get(1));
        Object[] array = list.toArray();               //List 集合转数组
        System.out.println("对应的 Object[]数组：" + Arrays.toString(array));
        // 调用类 Arrays 输出数组全部元素
        System.out.println("输出数组的第一个元素：" + array[0]); //使用[]和下标
        System.out.println("数组 array 不能再添加元素，但 List 集合可以");
        list.add(30);
        System.out.println("List 集合又添加了一个元素");
        System.out.println("集合长度：" + list.size());
```

```
                System.out.println("数组长度：" + array.length);
            }
        }
```

程序包含了 List 集合长度可变、使用泛型时类型安全和工具类 Arrays 的使用，其运行结果如图 5.2.3 所示。

```
张三
------------------
测试
20
对应的Object[]数组：[测试, 20]
输出数组的第一个元素：测试
数组array不能再添加元素，但List集合可以
List集合又添加了一个元素
集合长度：3
数组长度：2
```

图 5.2.3　程序的运行结果

2．LinkedList

LinkedList 类的内部实现基于一组连接的记录，类似于一个链表结构。访问 LinkedList 集合中的某个元素时，就必须从链表的一端开始沿着链接方向逐个元素地查找，直到找到所需的元素为止，但将元素添加到原有元素中间时效率很高。因此，当程序需要经常在指定位置添加元素，并且按照顺序访问其中的元素时，优先使用 LinkedList。

ArrayList 类常用的方法与 List 接口基本相同，LinkedList 类则比 List 接口多了一些方便操作头元素和尾元素的方法。增加的常用方法如表 5.2.2 所示。

表 5.2.2　LinkedList 增加的常用方法

方　法	描　述
void　addFirst(E e)	把新元素插入到列表中的最前位置
void　addLast(E e)	把新元素插入到列表中的最后位置
E　getFirst( )	获取列表中最前位置的元素
E　getLast( )	获取列表中最后位置的元素
E　peek( )	获取列表中最前位置的元素，但此元素仍保留在列表中
E　peekFirst( )	获取列表中最前位置的元素，但此元素仍保留在列表中
E　peekLast( )	获取列表中最后位置的元素，但此元素仍保留在列表中
E　poll( )	获取列表中最前位置的元素，同时把此元素从列表中删除
E　pollFirst( )	获取列表中最前位置的元素，同时把此元素从列表中删除
E　pollLast( )	获取列表中最后位置的元素，同时把此元素从列表中删除
E　pop( )	从栈中弹出栈顶元素
void　push(E e)	把指定元素压入到栈顶

注意：

（1）使用 LinkedList 创建的集合，相应于线性表中的链表，它的优缺点与顺序表正好相反。

（2）实际编程中，使用像 List 这样的通用接口，不用关心具体的实现，即性能由具体的实现来保证。

3．Vector

Vector 的方法都是同步、线程安全的，即某一时刻只有一个线程能够写 Vector；而 ArrayList 的方法不是。

如果有多个线程会访问到集合，那么最好使用 Vector，因为此时不需要再去考虑和编写线程安全的代码。但是，为实现线程同步，Vector 的性能比 ArrayList 和 LinkedList 要低。

注意：Vector 的用法，参见 10.1 节（打坦克游戏）。

5.2.2　Set 集合接口及实现类

Set 集合是一种无重复、无指定顺序的元素的集合。Set 接口是最常用的基本接口之一，它继承 Collection 接口，如果不需要保留存储顺序，并且需要去掉重复元素，则应使用 Set 接口。

接口 Set 的定义如图 5.2.4 所示。

就常用方法而言，Set 接口与 Collection 接口基本一致。

```
public interface Set<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    ……

    @Override
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, Spliterator.DISTINCT);
    }
}
```

```
Set<E>
  size() : int
  isEmpty() : boolean
  contains(Object) : boolean
  iterator() : Iterator<E>
  toArray() : Object[]
  toArray(T[]) <T> : T[]
  add(E) : boolean
  remove(Object) : boolean
  containsAll(Collection<?>) : boolean
  addAll(Collection<? extends E>) : boolean
  retainAll(Collection<?>) : boolean
  removeAll(Collection<?>) : boolean
  clear() : void
  equals(Object) : boolean
  hashCode() : int
  spliterator() : Spliterator<E>
```

图 5.2.4　接口 Set 的定义示意图

注意：相对于接口 Collection，子接口 Set 没有定义特有的方法。

Set 接口有多种实现类，主要有 HashSet、TreeSet、SortedSet 及 LinkedHashSet。

Set 接口的三种常用实现类的特点如下：

- 如果不需要排序，则使用 HashSet 类，其效率比 TreeSet 类高；
- 如果需要将元素排序，那么使用 TreeSet 类；
- 如果需要保留存储顺序，又要过滤重复元素，则使用 LinkedHashSet 类。

1．HashSet

HashSet 存储元素的顺序并不是按照存入时的顺序（与 List 不同），元素是按照哈希码（hashCode）值来存入的，取数据也是按照哈希码值取的（即 HashSet 类是将元素存储在散列表中），因此，HashSet 适合用于不需要有序的元素序列，并能快速查找特定元素。

【例 5.2.2】 测试 Set 接口及其实现类 HashSet。

测试程序的源代码如下：

```
import java.util.HashSet;
import java.util.Iterator;
public class TestHashSet2 {
    private static String[] stuNames = { "张三", "李四", "王五", "陈六", "赵七", "李四" };
    public static void main(String[] args) {
        HashSet<String> names = new HashSet<String>(10);    //指定容量
        for (int i = 0; i < stuNames.length; i++) {
            names.add(stuNames[i]);                         //添加元素
        }
        Iterator<String> iter = names.iterator();
        while (iter.hasNext()) {                            //使用迭代器遍历
            System.out.print(iter.next() + "    ");
        }
        System.out.println();

        // 容量不同，则元素存储顺序发生改变
        names = new HashSet<String>(100);                   //指定不同容量
        for (int i = 0; i < stuNames.length; i++) {
            names.add(stuNames[i]);                         //添加元素
        }
        for(String ele:names) {                             //另一种遍历方式
            System.out.print(ele + "    ");
        }

    }
}
```

程序开始就指定了 HashSet 的容量，并重复添加了元素“李四”，但 HashSet 并不存储重复元素。因此，运行输出时，“李四”只出现了一次。第二次创建 HashSet 时，添加

了相同的元素，但由于 HashSet 的容量改变，元素存储的顺序就发生了变化。程序运行结果如图 5.2.5 所示。

```
李四 张三 王五 陈六 赵七
陈六 赵七 李四 张三 王五
```

图 5.2.5　程序的运行结果

2．TreeSet

TreeSet 将元素存储在树中，但元素按有序方式存储，可以按任何次序向 TreeSet 中添加元素，但遍历 TreeSet 时，元素出现的序列是有序的。在 TreeSet 中插入元素的效率要低于在 HashSet 中插入元素的效率，但是比把元素插入到数组或链表的合适位置要快。

TreeSet 增加的常用方法如表 5.2.3 所示。

表 5.2.3　TreeSet 增加的常用方法

方　法	功 能 描 述
E　first( )	获得当前第一个（最低）元素
E　floor(E e)	获得当前集合中小于等于指定元素的最大元素
E　higher(E e)	获得当前集合中大于指定元素的最小元素
E　last( )	获得最后（最大）元素
E　pollFirst( )	获得第一个（最低）元素，并把此元素从集合中删除
E　pollLast( )	获得最后（最高）元素，并把此元素从集合中删除

注意：TreeSet 底层是一个二叉树结构，需要数据结构课程的相关知识，此处不再详述。

3．LinkedHashSet

LinkedHashSet 集合同样是根据元素的哈希码值来决定元素的存储位置。但是，它同时使用链表来维护元素的次序，这样使得元素看起来像是以插入顺序保存的。也就是说，当遍历该集合时，LinkedHashSet 将会以元素的添加顺序访问集合的元素。

LinkedHashSet 在迭代访问 Set 集合中的全部元素时，性能要优于 HashSet，但是插入性能稍微逊色于 HashSet。

5.2.3　队列接口 Queue 及实现类

队列接口 Queue 用来表示先入先出（FIFO）的数据结构，接口 Queue 继承 Collection 接口，其定义如图 5.2.6 所示。

前面介绍的 LinkedList 类是 List 接口的实现类，同时也是 Queue 接口的实现类，因此，可以使用 LinkedList 类来创建队列。LinkedList 类的定义如图 5.2.7 所示。

注意：在数据结构中，链式队列和链栈是特殊的链式线性表。

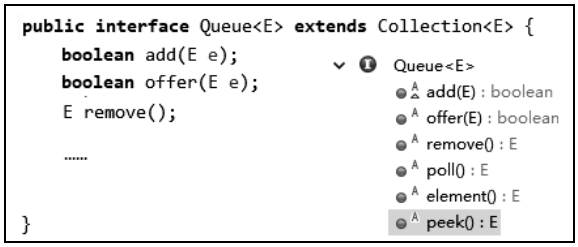

图 5.2.6　接口 Queue 的定义

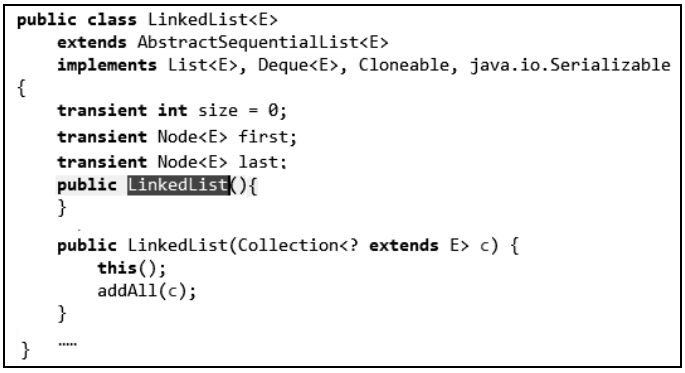

图 5.2.7　LinkedList 类的定义

【例 5.2.3】 测试 Queue 接口及其实现类 LinkedList。

测试链队程序的源代码如下：

```
import java.util.LinkedList;
import java.util.Queue;
public class TestQueue{
    public static void main(String[] args) {
        //创建链队
        Queue<String> queue = new LinkedList<String>();
        //添加元素
        queue.offer("a");
        queue.offer("b"); queue.offer("c");
        queue.offer("d"); queue.offer("e");
        for(String q : queue){
            System.out.print(q);
        }
        System.out.println();
        //返回首个元素并从队列中删除
        System.out.print("poll="+queue.poll()+"    ");
        for(String q : queue){
            System.out.print(q);
        }
        System.out.println();
```

```
            //add() 和 remove() 方法在失败的时候会抛出异常（不推荐）
            queue.add("a");
            //返回队列首个元素//返回第一个元素，队列空时抛出异常
            System.out.print("element="+queue.element()+"    ");
            for(String q : queue){
                System.out.print(q);
            }
            System.out.println();

            //返回队列首个元素，队列空时返回 null
            System.out.print("peek="+queue.peek()+"    ");
            for(String q : queue){
                System.out.print(q);
            }
            System.out.println();
        }
    }
```

程序开始创建一个链队，包含 5 个元素，分别使用 poll()、add()、element() 和 peek() 方法后，输出队列的效果，如图 5.2.8 所示。

```
abcde
poll=a  bcde
element=b  bcdea
peek=b  bcdea
```

图 5.2.8　程序的运行结果

5.3　Map 集合及其遍历

5.3.1　Map 接口

Map 接口是另一个重要的集合接口，用来存储一组成对的对象。Map 集合中存储的是键值对，键名不能重复，键值可以重复。Map 表示“键-值”成对的一组对象（key-value），它不能有重复的 key，但可以有重复的 value。

与接口 Collection 类似，接口 Map 也提供了一组抽象方法，用于对 Map 集合中元素进行增加、删除、查询和统计等，它在 HashMap 等实现类中得到了实现。

接口 Map 的几个重要方法分别是 keySet()、put()、get() 和 size()，分别用于获取键名集合、存放键-值对、获取键名所对应的键值及统计元素个数。Map 接口的定义如图 5.3.1 所示。

接口 Map 并未继承接口 Interable，这与接口 Collection 不同。接口 Map 提供了返回值类型为 Set< Entry<K,V>>的方法 entrySet()。Entry<K,V>是接口 Map 的内部接口，表示 Map 集合项，并提供了从集合项中获取键名的方法 getKey() 和获取键值的方法 getValue() 等。

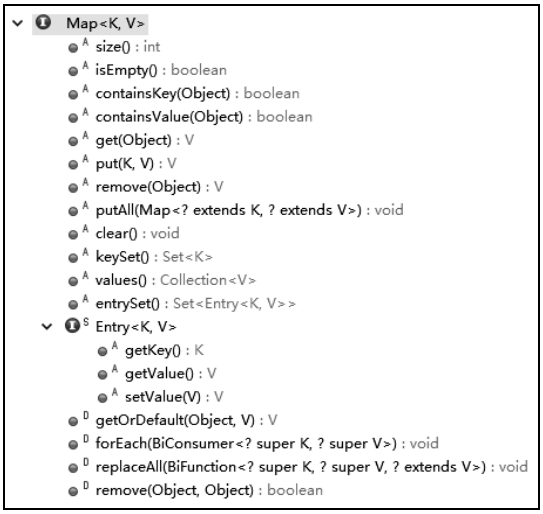

图 5.3.1　Map 接口的定义

注意：因为是对 Set< Entry<K,V>>应用迭代器，所以，可以认为 Map 接口间接实现了 Iterable 接口。

5.3.2　Map 实现类 HashMap 及其他实现类

集合是一种数据结构，可以包含其他对象的引用，相当于装载其他对象的容器。合理地使用集合 API 可以为程序员提供多方面的便利，使程序开发人员能将注意力集中到程序的重要部分而无须过度关注底层设计，减少程序设计中为转换对象类型而编写代码的工作量。集合 API 通过提供对数据结构和算法的高性能和高质量实现，保证了程序的执行速度和质量。

集合接口声明的是可以对每种集合类型所执行的各种方法，集合的实现类以特殊的方式执行这些方法。绝大部分处理集合的类与接口位于 java.util 包。

集合 API 分为两大类：以 Collection 为接口的元素集合类型和以 Map 为接口的映射集合类型。Collection 类型又分为 Set 和 List，Collection 接口包含大量方法用于添加、删除、比较集合中的元素，Collection 集合也可以转换成数组。

集合框架中 Set 的特征是其元素无重复且无序，因此 Set 接口及其实现类没有按下标进行添加、删除、访问的方法。Set 接口的实现类有 HashSet、TreeSet 及子类 LinkedHashSet，这三个类是非线程安全的。TreeSet 是基于树结构的集合，LinkedHashSet 具备按照插入先后顺序访问的功能，HashSet 访问元素的顺序是不确定的，TreeSet 的访问顺序是按照树接口的顺序访问。

Map 接口的实现类有 HashMap、IdentityHashMap、WeakHashMap、TreeMap，以及 LinkedHashMap 子类，这些类都是非线程安全的。WeakHashMap 是一种改进的 HashMap，如果一个 key 不再被外部所引用，那么该 key 可以被垃圾回收器回收。HashTable 是线程安全的，HashTable 不能插入 null 空元素。

Map 接口是映射类的顶层接口，SortedMap 接口提供了排序功能，最经常使用到的已实现 Map 接口的类有 HashMap 和 TreeMap。HashMap 对“键”进行散列；TreeMap 实现了 SortedMap 接口，通过用排序方法根据元素的键的排序结果把元素组织到树中。

Map 接口及其实现类，如图 5.3.2 所示。

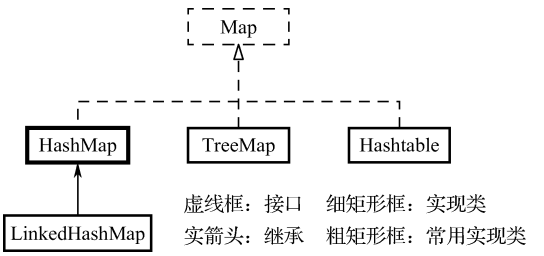

图 5.3.2　Map 接口及其实现类

注意：

（1）Map 接口不能包含重复的 key，但是可以包含相同的 value。

（2）跟踪 Java 集合框架的相关接口与类可知，HashMap 实际上继承抽象类 AbstractMap，而 AbstractMap 实现接口 Map。因此，HashMap 间接实现接口 Map。

（3）Hashtable 和 HashMap 在性能方面的比较类似 Vector 和 ArrayList，例如，Hashtable 的方法是同步的，而 HashMap 的方法不是。

【例 5.3.1】 测试 Map 集合及其实现类 HashMap。

测试程序的源代码如下：

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class TestMap {
    public static void main(String[] args) {
        // 创建 Map 集合对象，根据课程查学分
        Map<String, Float> course = new HashMap<String, Float>();
        course.put("Java", new Float(3.0));
        course.put("Java EE", new Float(2.5));
        course.put("Android", new Float(2.5));

        // Map 集合的相关方法

        //获取 Map 集合对象 course 中所有 key 对象的集合
        Set<String> set = course.keySet();
        //HashSet<String> set = (HashSet<String>) course.keySet();
        System.out.println("Map 集合大小：" + set.size());
        System.out.println("集合中包含课程 JavaEE 吗？：" + set.contains("JavaEE"));
        System.out.println("集合中包含课程 Java EE 吗？：" + set.contains("Java EE"));
```

```
            System.out.println("Java EE 课程的学分：" + course.get("Java EE"));
            // 迭代 Map 集合
            System.out.println("使用迭代器遍历结果：");
            Iterator<String> it = set.iterator();
            while (it.hasNext()) {
                String key = it.next();
                System.out.println(key + "---" + course.get(key));
            }
            // 迭代 Map 集合
            System.out.println("使用 for 循环遍历结果：");
            for (String key : set) {
                System.out.println(key + "---" + course.get(key));
            }
        }
    }
```

测试程序包含了 Map 集合的创建、得到键名集合和键值的遍历，其运行结果如图 5.3.3 所示。

```
Map集合大小：3
集合中包含课程JavaEE吗？：false
集合中包含课程Java EE吗？：true
Java EE课程的学分：2.5
使用迭代器遍历结果：
Java EE---2.5
Java---3.0
Android---2.5
使用for循环遍历结果：
Java EE---2.5
Java---3.0
Android---2.5
```

图 5.3.3　程序的运行结果

【例 5.3.2】 显示二维表数据的两种实现方式。

程序的源代码如下：

```
import java.util.ArrayList;
/*
 * 存储二维表数据的两种实现方式：
 * （1）在 List 集合里嵌套 Map 集合
 * （2）定义与使用实体类，作为 List 元素类型
 */
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class MapInList {
    public static void main(String[] args) {
        //二维表数据实现方式一
        List<Map<String, Object>> persons = new ArrayList<Map<String, Object>>();
        Map<String, Object> person1 = new HashMap<String, Object>();    //记录
        person1.put("id", 1);
```

```
            person1.put("name", "张三");
            person1.put("salary", 5000);
            persons.add(person1);
            Map<String, Object> person2 = new HashMap<String, Object>();
            person2.put("id", 2);
            person2.put("name", "李四");
            person2.put("salary", 5800);
            persons.add(person2);
            Map<String, Object> person3 = new HashMap<String, Object>();
            person3.put("id", 3);
            person3.put("name", "王五");
            person3.put("salary", 5500);
            persons.add(person3);
            System.out.println("遍历结果如下：");
            for(int i=0;i<persons.size();i++) {
                System.out.println(persons.get(i));
            }
             /* List<Person>    persons=new ArrayList<Person>();       //实现方式二
             Person person1=new Person(1, "张三", 5000);
             Person person2=new Person(2, "李四", 5800);
             Person person3=new Person(3, "王五", 5500);
             persons.add(person1);
             persons.add(person2);
             persons.add(person3);
             System.out.println("遍历结果如下：");
             for(Person person:persons) {
                 System.out.println(person);
             }*/
        }
    }
    class Person {   //本实体类包含了构造方法、Setter/Getter
        int id;
        String name;
        int salary;
        //以下方法都是在 eclipse 中自动生成，右键菜单 Source-Generate Getters/Setters
        public Person(int id, String name, int salary) {
            this.id = id;
            this.name = name;
            this.salary = salary;
        }
        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Person{" + "id=" + id + ", name='" + name + '\'' + ", salary=" + salary + '}';
    }
}
```

程序的运行结果如图 5.3.4 所示。

```
遍历结果如下：
{name=张三, id=1, salary=5000}
{name=李四, id=2, salary=5800}
{name=王五, id=3, salary=5500}
```

图 5.3.4　程序的运行结果

注意：实体类方式将在后续课程（如 Android 和 Java Web）中广泛使用。

5.3.3　使用 Map.Entry 遍历 Map 集合

有时进行元素查找时，希望通过某些关键信息来查找与之相关的对象，如在地址簿中通过姓名查找相应的地址。映射类就是解决此类问题的数据结构之一，映射类储存的数据是“键-值”对，将“键”与“值”关联起来，给出键（key）就可以查找到与之相关的值（value）。

Map 接口的方法 entrySet() 是将 Map 集合里的每一个“键-值”对取出来封装成一个 Entry 类型的对象，并存到一个 Set 集合里。

接口 Map.Entry 是 Map 集合中的一个内部接口，表示一个映射项（里面有 key 和 value），Map.entrySet() 方法的结果类型是 Set<Map.Entry<K,V>>。

内部接口 Map.Entry 提供了 getKey() 和 getValue() 方法，从一个 Entry 项中取出 key 和 value。

【例 5.3.3】 测试 Map 集合的内部接口 Entry。

测试程序的源代码如下：

```
import java.util.HashMap;
import java.util.Iterator;
```

```
import java.util.Map;
import java.util.Map.Entry; //内部接口
import java.util.Set;
public class MapEntry {
    static Map<String, String> map;
    static Set<String> keySet;
    static Set<Entry<String, String>> entrySet;    //Map 映射项的集合
    public static void main(String[] args) {
        map = new HashMap<String, String>();
        map.put("1", "value1");
        map.put("2", "value2");
        map.put("3", "value3");
        entrySet = map.entrySet(); //Map 映射项的集合
        //使用 iterator 遍历 key 和 value
        Iterator<Entry<String, String>> it = entrySet.iterator();
        while (it.hasNext()) {
            Map.Entry<String, String> entry = it.next(); //一个映射项
            System.out.println("key= " + entry.getKey() +
                                                    " and value= " + entry.getValue());
        }
        /*    for (Map.Entry<String, String> entry : entrySet) { //另一种遍历方式
            System.out.println("key= " + entry.getKey() +
                                                    " and value= " + entry.getValue());
        }*/
    }
}
```

测试程序主要包含了从 Map 集合创建它的映射项 Entry 的 Set 集合，进而遍历该集合，其运行结果如图 5.3.5 所示。

```
key= 1 and value= value1
key= 2 and value= value2
key= 3 and value= value3
```

图 5.3.5　程序的运行结果

习题 5

一、判断题

1．Java 集合框架中不同的容器存储不同结构的数据。
2．Java 集合和迭代器支持使用泛型。
3．Map 集合对象具有 iterator() 方法。
4．接口 List 对接口 Collection 的扩展在于增加了面向位置的操作。
5．接口 Collection 和接口 Map 都定义了抽象方法 iterator()。

二、选择题

1．接口 List 定义的下列方法中，不是继承父接口 Collection 的是____。
A．size　　B．iterator　　C．get　　D．contains
2．向 Map 集合添加元素所使用的方法是____。
A．add　　B．put　　C．insert　　D．get
3．下列关于接口 Collection 及其子接口的说法中，不正确的是____。
A．Collection 表示一组允许重复的对象，对象之间没有指定的顺序关系
B．List 表示允许重复且有顺序关系的一组对象
C．Set 表示不允许重复且没有顺序关系的一组对象
D．Queue 表示先进后出的一组对象
4．关于 Java 集合框架，下列说法中不正确的是____。
A．接口 Collection 和 Map 均继承接口 Iterable
B．接口 Iterable 定义了抽象方法 iterator()
C．接口 List 和 Set 都是 Collection 的子接口
D．ArrayList 是 List 接口的实现类
5．关于 Java 集合框架，下列说法中不正确的是____。
A．List、Set 和 Queue 均是接口 Collection 的子接口
B．ArrayList 是接口 List 的实现类
C．获取 Map 集合大小的方法是 length()
D．HashMap 是 Map 接口的实现类

三、填空题

1．处理 Java 集合的绝大部分接口和类位于软件包____中。
2．Java 的____接口定义集合元素不允许重复且没有指定的顺序关系。
3．在元素的插入与删除方面，LinkedList 的速度比 ArrayList____。
4．向 List、Set 和 Queue 三种集合添加元素，都是使用___方法。
5．取出队列首个元素并从队列中删除该元素的方法是___。
6．迭代器游标只能单向移动，得到下一个元素的方法是___。

实验 5

一、实验目的

1．掌握 Java 泛型方法（泛型类）的使用。

2．掌握集合 Collection（List 集合）与数组的区别和联系，以及常用 List 实现类的使用。

3．掌握 Map 集合的使用，以及与 List 集合的综合运用。

4．了解 Set 集合和 Queue 集合（队列）的使用。

二、实验内容及步骤

访问上机实验网站（http://www.wustwzx.com/java），单击“5. Java 集合框架与泛型”的超链接，下载本实验内容的源代码（含素材）并解压，得到文件夹 Java_ch05。

1．泛型方法（泛型类）的定义与使用

（1）在 eclipse 中导入项目 Java_ch05。

（2）打开源程序 Ex5_1.java，查看两个非泛型方法 changePosition() 的定义（同名方法但参数类型不同，用于重载）。

（3）查看 main() 方法中对上面两个非泛型方法的调用。

（4）打开源程序 Ex5_1a.java，查看泛型方法 changePosition() 的定义。

（5）查看 main() 方法中对泛型方法的调用（参数不能为值类型）。

（6）打开源程序 Ex5_1b.java，查看泛型类 ArrayUtils<T>的定义。

（7）查看测试类 Ex5_1b 的 main() 中泛型类 ArrayUtils<T>对象的创建及使用。

2．集合 Collection（List 集合）与数组的区别和联系、常用 List 实现类的使用

（1）打开源程序 Ex5_2.java，查看泛型集合 ArrayList<String> names 的创建。

（2）查看为集合对象 names 增加元素的代码。

（3）验证不能为集合对象 names 增加 String 类型以外的元素。

（4）查看集合对象 List 的创建。

（5）验证可以向 list 对象添加除基本数据类型以外的任意类型的数据。

（6）总结访问 List 集合元素的方法（使用下标）。

（7）了解 List 作为动态数组的用法。

3．Set 集合的创建及遍历

（1）打开源程序 Ex5_3.java，查看根据字符串数组创建 Set 集合的方法。其中，数组包含了重复的元素。

（2）查看向 Set 集合增加元素的方法。

（3）查看使用迭代器遍历 Set 集合的方法。观察输出元素的顺序及个数（重复的元素只输出一次）。

（4）运行程序，查验增加元素后再输出的顺序变化。

4．Map 集合与 Set 集合的使用

（1）打开源程序 Ex5_5.java，查看创建 Map 集合对象 course 的创建过程。

（2）查看从 Map 集合生成 Set 集合对象 set 的方法。

（3）查看使用迭代器 Iterator 遍历 Map 集合的方法。

（4）查看使用 for 循环遍历 Set 集合的方法。

（5）打开源程序 Ex5_6.java，查看 Map 集合的内部接口 Entry 的使用。

（6）打开源程序 Ex5_7.java，查看在 List 集合中嵌套 Map 集合的用法。

（7）注释实体类 Person 的 get/set 方法，使用 eclipse 右键快捷菜单，自动生成实体类 Person 的 get/set 方法。

（8）查看使用实体类 Person 和 List 集合表示二维表数据的方法。

三、实验小结及思考

（由学生填写，重点填写上机实验中遇到的问题。）