第1章

集合框架和泛型

本章重点

※ List 接口及实现类
※ Map 接口及实现类
※ 泛型集合

本章目标

※ Iterator 接口
※ 泛型类、泛型接口

本章任务

学习本章，完成以下 2 个工作任务。记录学习过程中遇到的问题，可以通过自己的努力或访问 kgc.cn 解决。

任务 1：新闻标题查询功能

在新闻管理系统中，实现新闻标题信息的存储及查询功能。要求使用集合类存储新闻标题（包含 ID、名称、创建者）信息。

任务 2：改进新闻标题查询功能

改进任务 1，使用泛型集合存储新闻标题（包含 ID、名称、创建者）信息，输出新闻标题的总数量及每条新闻标题的名称。

任务 1　新闻标题查询功能

关键步骤如下：

- 创建集合对象，并添加数据。
- 统计新闻标题总数量。
- 输出新闻标题名称。

1.1.1　集合概述

开发应用程序时，如果想存储多个同类型的数据，可以使用数组来实现，但是使用数组存在如下一些明显缺陷：

- 数组长度固定不变，不能很好地适应元素数量动态变化的情况。
- 可通过数组名 .length 获取数组的长度，却无法直接获取数组中实际存储的元素个数。
- 数组采用在内存中分配连续空间的存储方式存储，根据元素信息查找时效率比较低，需要多次比较。

从以上分析可以看出数组在处理一些问题时存在明显的缺陷，针对数组的缺陷，Java 提供了比数组更灵活、更实用的集合框架，可大大提高软件的开发效率，并且不同的集合可适用于不同应用场合。

Java 集合框架提供了一套性能优良、使用方便的接口和类，它们都位于 java.util 包中，其主要内容及彼此之间的关系如图 1.1 所示。

从图 1.1 中可以看出，Java 的集合类主要由 Map 接口和 Collection 接口派生而来，其中 Collection 接口有两个常用的子接口，即 List 接口和 Set 接口，所以通常说 Java

集合框架由三大类接口构成（Map 接口、List 接口和 Set 接口）。本章讲解的主要内容就是围绕这三大类接口进行的。

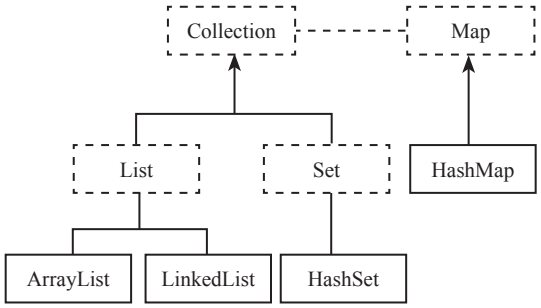

图 1.1　Java 集合框架图

注意：

虚线框表示接口或者抽象类，实线框表示开发中常用的实现类。

1.1.2　List 接口

Collection 接口是最基本的集合接口，可以存储一组不唯一、无序的对象。List 接口继承自 Collection 接口，是有序集合。用户可使用索引访问 List 接口中的元素，类似于数组。List 接口中允许存放重复元素，也就是说 List 可以存储一组不唯一、有序的对象。

List 接口常用的实现类有 ArrayList 和 LinkedList。

1．使用 ArrayList 类动态存储数据

针对数组的一些缺陷，Java 集合框架提供了 ArrayList 集合类，对数组进行了封装，实现了长度可变的数组，而且和数组采用相同的存储方式，在内存中分配连续的空间，如图 1.2 所示，所以，经常称 ArrayList 为动态数组。但是它不等同于数组，ArrayList 集合中可以添加任何类型的数据，并且添加的数据都将转换成 Object 类型，而在数组中只能添加同一数据类型的数据。

0	1	2	3	4	5	
aaaa	dddd	cccc	aaaa	eeee	dddd	

图 1.2　ArrayList 存储方式示意图

ArrayList 类提供了很多方法用于操作数据，如表 1-1 中列出的是 ArrayList 类的常用方法。

表 1-1　ArrayList 类的常用方法

方　法	说　明
boolean add(Object o)	在列表的末尾添加元素 o，起始索引位置从 0 开始
void add(int index,Object o)	在指定的索引位置添加元素 o，索引位置必须介于 0 和列表中元素个数之间
int size()	返回列表中的元素个数
Object get(int index)	返回指定索引位置处的元素，取出的元素是 Object 类型，使用前需要进行强制类型转换
void set(int index,Object obj)	将 index 索引位置的元素替换为 obj 元素
boolean contains(Object o)	判断列表中是否存在指定元素 o
int indexOf(Object obj)	返回元素在集合中出现的索引位置
boolean remove(Object o)	从列表中删除元素 o
Object remove(int index)	从列表中删除指定位置的元素，起始索引位置从 0 开始

示例 1

使用 ArrayList 常用方法动态操作数据。

实现步骤：

（1）导入 ArrayList 类。

（2）创建 ArrayList 对象，并添加数据。

（3）判断集合中是否包含某元素。

（4）移除索引为 0 的元素。

（5）把索引为 1 的元素替换为其他元素。

（6）输出某个元素所在的索引位置。

（7）清空 ArrayList 集合中的数据。

（8）判断 ArrayList 集合中是否包含数据。

关键代码：

```
public static void main(String[] args){
    ArrayList list=new ArrayList();                    // ①
    list.add(" 张三丰 ");
    list.add(" 郭靖 ");
    list.add(" 杨过 ");
    // 判断集合中是否包含 " 李莫愁 "，对应步骤（3）
    System.out.println(list.contains(" 李莫愁 "));     // 输出 false
    // 把索引为 0 的数据移除，对应步骤（4）
    list.remove(0);                                    // ②
    System.out.println("----------------------");
    list.set(1," 黄蓉 ");                              // ③
```

```
    for (int i=0; i<list.size();i++){
        String name=(String)list.get(i);          // ④
        System.out.println(name);
    }
    System.out.println("----------------------");
    System.out.println(list.indexOf(" 小龙女 "));          // ⑤
    list.clear(); // 清空 list 中的数据
    System.out.println("----------------------");
    for(Object obj:list){
        String name=(String)obj;                  // ⑥
        System.out.println(name);
    }
System.out.println(list.isEmpty());                  // ⑦
}
```

输出结果如下所示：

```
false
----------------------
郭靖
黄蓉
----------------------
-1
----------------------
true
```

在示例 1 中，①的代码调用 ArrayList 的无参构造方法，创建集合对象。常用的 ArrayList 类的构造方法还有一个带参数的重载版本，即 ArrayList(int initialCapacity)，它构造一个具有指定初始容量的空列表。

② 的代码将 list 集合中索引为 0 的元素删除，list 集合的下标是从 0 开始，也就是删除了“张三丰”，集合中现有元素为“郭靖”和“杨过”。

③ 的代码将 list 集合中索引为 1 的元素替换为“黄蓉”，即将“杨过”替换为“黄蓉”，集合中现有元素为“郭靖”和“黄蓉”。

④ 的代码是使用 for 循环遍历集合，输出集合中的所有元素。list.get(i) 取出集合中索引为 i 的元素，并强制转换为 String 类型。

⑤ 的代码为输出元素“小龙女”所在的索引位置，因集合中没有该元素，所以输出结果为 -1。

⑥ 的代码是使用增强 for 循环遍历集合，输出集合中的所有元素。增强 for 循环的语法在 Java 基础课程中讲过，这里不再赘述。可以看出，遍历集合时使用增强 for 循环比普通的 for 循环在写法上更加简单方便，而且不用考虑下标越界的问题。

⑦ 的代码用来判断 list 集合是否为空，因为前面执行了 list.clear() 操作，所以集合已经为空，输出为 true。

注意：

① 调用 ArrayList 类的 add(Object obj) 方法时，添加到集合当中的数据将被转换为 Object 类型。

② 使用 ArrayList 类之前，需要导入相应的接口和类，代码如下：

```
import java.util.ArrayList;
import java.util.List;
```

示例 2

使用 ArrayList 集合存储新闻标题信息（包含 ID、名称、创建者），输出新闻标题的总数量及每条新闻标题的名称。

实现步骤：

（1）创建 ArrayList 对象，并添加数据。

（2）获取新闻标题的总数。

（3）遍历集合对象，输出新闻标题名称。

关键代码：

```
// 创建新闻标题对象，NewTitle 为新闻标题类
NewTitle car=new NewTitle(1, " 汽车 ", " 管理员 ");
NewTitle test=new NewTitle(2, " 高考 ", " 管理员 ");
// 创建存储新闻标题的集合对象
List newsTitleList=new ArrayList();
// 按照顺序依次添加新闻标题
newsTitleList.add(car);
newsTitleList.add(test);
// 获取新闻标题的总数
System.out.println(" 新闻标题数目为："+newsTitleList.size()+" 条 ");
// 遍历集合对象
System.out.println(" 新闻标题名称为：");
for(Object obj:newsTitleList){
   NewTitle title=(NewTitle)obj;
   System.out.println(title.getTitleName());
}
```

输出结果如图 1.3 所示。

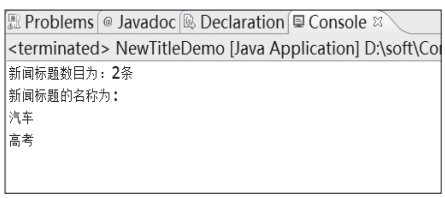

图 1.3　输出新闻标题信息

在示例 2 中，ArrayList 集合中存储的是新闻标题对象。在 ArrayList 集合中可以存储任何类型的对象。其中，代码 List newsTitleList=new ArrayList(); 是将接口 List 的引用指向了实现类 ArrayList 的对象，在编程中将接口的引用指向实现类的对象是 Java 实现多态的一种形式，也是软件开发中实现低耦合的方式之一，这样的用法可以大大提高程序的灵活性。随着编程经验的积累，对这个用法的理解会逐步加深。

ArrayList 因为可以使用索引来直接获取元素，所以其优点是遍历元素和随机访问元素的效率比较高。但是由于 ArrayList 采用了和数组相同的存储方式，在内存中分配连续的空间，因此在添加和删除非尾部元素时会导致后面所有元素的移动，这就造成在插入、删除等操作频繁的应用场景下用 ArrayList 会性能低下。所以数据操作频繁时，最好使用 LinkedList 存储数据。

2．使用 LinkedList 类动态存储数据

LinkedList 类是 List 接口的链接列表实现类。它支持实现所有 List 接口可选的列表的操作，并且允许元素值是任何数据，包括 null。

LinkedList 采用链表存储方式存储数据，如图 1.4 所示，优点在于插入、删除元素时效率比较高，但是 LinkedList 的查找效率很低。

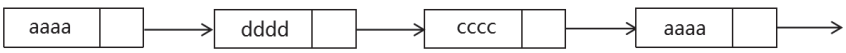

图 1.4　LinkedList 存储示意图

它除了包含 ArrayList 类所包含的方法外，还提供了如表 1-2 所示的一些方法，可以在 LinkedList 的首部或尾部进行插入、删除操作。

表 1-2　LinkedList 类的常用方法

方　法	说　明
void addFirst(Object obj)	将指定元素插入到当前集合的首部
void addLast(Object obj)	将指定元素插入当前集合的尾部
Object getFirst()	获得当前集合的第一个元素
Object getLast()	获得当前集合的最后一个元素
Object removeFirst()	移除并返回当前集合的第一个元素
Object removeLast()	移除并返回当前集合的最后一个元素

示例 3

使用 LinkedList 集合存储新闻标题（包含 ID、名称、创建者），实现获取、添加及删除头条和末条新闻标题信息功能，并遍历集合。

实现步骤：

（1）创建 LinkedList 对象，并添加数据。

（2）添加头条和末尾标题。

（3）获取头条和末条新闻标题信息。

（4）删除头条和末条新闻标题。

关键代码：

```
// 创建多个新闻标题对象
NewTitle car=new NewTitle(1, " 汽车 ", " 管理员 ");
NewTitle medical=new NewTitle(2, " 医学 ", " 管理员 ");
NewTitle fun=new NewTitle(3, " 娱乐 ", " 管理员 ");
NewTitle gym=new NewTitle(4, " 体育 ", " 管理员 ");
// 创建存储新闻标题的集合对象并添加数据
LinkedList newsTitleList=new LinkedList();
newsTitleList.add(car);
newsTitleList.add(medical);
// 添加头条新闻标题和末尾标题
newsTitleList.addFirst(fun);
newsTitleList.addLast(gym);
System.out.println(" 头条和末条新闻已添加 ");
// 获取头条以及最末条新闻标题
NewTitle first=(NewTitle) newsTitleList.getFirst();
System.out.println(" 头条的新闻标题为 :"+first.getTitleName());
NewTitle last=(NewTitle) newsTitleList.getLast();
System.out.println(" 排在最后的新闻标题为 :"+last.getTitleName());
// 删除头条和最末条新闻标题
newsTitleList.removeFirst();
newsTitleList.removeLast();
System.out.println(" 头条和末条新闻已删除 ");
System.out.println(" 遍历所有新闻标题： ");
for(Object obj:newsTitleList){
   NewTitle newTitle=(NewTitle)obj;
   System.out.println(" 新闻标题名称： "+newTitle.getTitleName());
}
```

输出结果如图 1.5 所示。

图 1.5　使用 LinkedList 存储并操作新闻标题信息

除了表 1-2 中列出的 LinkedList 类提供的方法外，LinkedList 类和 ArrayList 类所包含的大部分方法是完全一样的，这主要是因为它们都是 List 接口的实现类。由于

ArrayList 采用和数组一样的连续的顺序存储方式，当对数据频繁检索时效果较高，而 LinkedList 采用链表存储方式，当对数据添加、删除或修改比较多时，建议选择 LinkedList 存储数据。

1.1.3　Set 接口

1. Set 接口概述

Set 接口是 Collection 接口的另外一个常用子接口，Set 接口描述的是一种比较简单的集合，集合中的对象并不按特定的方式排序，并且不能保存重复的对象，也就是说 Set 接口可以存储一组唯一、无序的对象。

Set 接口常用的实现类有 HashSet。

2. 使用 HashSet 类动态存储数据

假如现在需要在很多数据中查找某个数据，LinkedList 就无需考虑了，它的数据结构决定了它的查找效率低下。如果使用 ArrayList，在不知道数据的索引，且需要全部遍历的情况下，效率一样很低下。为此 Java 集合框架提供了一个查找效率高的集合类 HashSet。HashSet 类实现了 Set 接口，是使用 Set 集合时最常用的一个实现类。HashSet 集合的特点如下：

- 集合内的元素是无序排列的。
- HashSet 类是非线程安全的。
- 允许集合元素值为 null。

表 1-3 中列举了 HashSet 类的常用方法。

表 1-3　HashSet 类的常用方法

方　法	说　明
boolean add(Object o)	如果此 Set 中尚未包含指定元素 o，则添加指定元素 o
void clear()	从此 Set 中移除所有元素
int size()	返回此 Set 中的元素的数量（Set 的容量）
boolean isEmpty()	如果此 Set 不包含任何元素，则返回 true
boolean contains(Object o)	如果此 Set 包含指定元素 o，则返回 true
boolean remove(Object o)	如果指定元素 o 存在于此 Set 中，则将其移除

示例 4

使用 HashSet 类的常用方法存储并操作新闻标题信息，并遍历集合。

实现步骤：

（1）创建 HashSet 对象，并添加数据。

（2）获取新闻标题的总数。

（3）判断集合中是否包含汽车新闻标题。

（4）移除对象。

（5）判断集合是否为空。

（6）遍历集合。

关键代码：

```
// 创建多个新闻标题对象
NewTitle car=new NewTitle(1, " 汽车 ", " 管理员 ");
NewTitle test=new NewTitle(2, " 高考 ", " 管理员 ");
// 创建存储新闻标题的集合对象
Set newsTitleList=new HashSet();
// 按照顺序依次添加新闻标题
newsTitleList.add(car);
newsTitleList.add(test);
// 获取新闻标题的总数
System.out.println(" 新闻标题数目为："+newsTitleList.size()+" 条 ");
// 判断集合中是否包含汽车新闻标题
System.out.println(" 汽车新闻是否存在："+newsTitleList.contains(car));          // 输出 true
newsTitleList.remove(test);        // 移除对象
System.out.println(" 汽车对象已删除 ");
System.out.println(" 集合是否为空："+newsTitleList.isEmpty());                  // 判断是否为空
// 遍历所有新闻标题
System.out.println(" 遍历所有新闻标题：");
for(Object obj:newsTitleList){
   NewTitle title=(NewTitle)obj;
   System.out.println(title.getTitleName());
}
```

输出结果如图 1.6 所示。

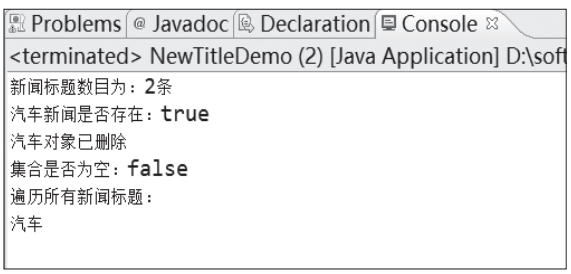

图 1.6　使用 HashSet 存储并操作新闻标题信息

注意：

使用 HashSet 类之前，需要导入相应的接口和类，代码如下：

```
import java.util.Set;
import java.util.HashSet;
```

在示例 4 中，通过增强 for 循环遍历 HashSet，前面讲过 List 接口可以使用 for 循环和增强 for 循环两种方式遍历，使用 for 循环遍历时，通过 get() 方法取出每个对象，但 HashSet 类不存在 get() 方法，所以 Set 接口无法使用普通 for 循环遍历。其实遍历集合还有一种比较常用的方式，即使用 Iterator 接口。

1.1.4　Iterator 接口

1. Iterator 接口概述

Iterator 接口表示对集合进行迭代的迭代器。Iterator 接口为集合而生，专门实现集合的遍历。此接口主要有如下两个方法：

- hasNext()：判断是否存在下一个可访问的元素，如果仍有元素可以迭代，则返回 true。
- next()：返回要访问的下一个元素。

凡是由 Collection 接口派生而来的接口或者类，都实现了 iterate() 方法，iterate() 方法返回一个 Iterator 对象。

2. 使用 Iterator 遍历集合

下面通过示例来学习使用迭代器 Iterator 遍历 Arraylist 集合。

➲ 示例 5

使用 Iterator 接口遍历 ArrayList 集合。

实现步骤：

（1）导入 Iterator 接口。

（2）使用集合的 iterate() 方法返回 Iterator 对象。

（3）while 循环遍历。

（4）使用 Iterator 的 hasNext() 方法判断是否存在下一个可访问的元素。

（5）使用 Iterator 的 next() 方法返回要访问的下一个元素。

关键代码：

```
public static void main(String[] args){
   ArrayList list=new ArrayList();
   list.add(" 张三 ");
   list.add(" 李四 ");
   list.add(" 王五 ");
   list.add(2, " 杰伦 ");
   System.out.println(" 使用 Iterator 遍历，分别是：");
   Iterator it=list.iterator();          // 获取集合迭代器 Iterator
   while(it.hasNext()){                  // 通过迭代器依次输出集合中所有元素的信息
      String name=(String)it.next();
      System.out.println(name);
   }
}
```

输出结果如下所示：

```
使用 Iterator 遍历，分别是：
张三
李四
杰伦
王五
```

示例 5 中是以 ArrayList 为例使用 Iterator 接口，其他由 Collection 接口直接或间接派生的集合类如已经学习的 LinkedList、HashSet 等。同样可以使用 Iterator 接口进行遍历，遍历方式与示例 5 遍历 ArrayList 集合的方式相同。例如，将示例 5 改为使用 Iterator 对象遍历，关键代码如下：

```
// 使用 Iterator 遍历 HashSet 集合
while(iterator.hasNext()){
    NewTitle title=(NewTitle) iterator.next();
    System.out.println(title.getTitleName());
}
```

1.1.5 Map 接口

1. Map 接口概述

Map 接口存储一组成对的键（key）- 值（value）对象，提供 key 到 value 的映射，通过 key 来检索。Map 接口中的 key 不要求有序，不允许重复。value 同样不要求有序，但允许重复。表 1-4 中列举了 Map 接口的常用方法。

表 1-4 Map 接口的常用方法

方 法	说 明
Object put(Object key, Object value)	将相互关联的一个 key 与一个 value 放入该集合，如果此 Map 接口中已经包含了 key 对应的 value，则旧值将被替换
Object remove(Object key)	从当前集合中移除与指定 key 相关的映射，并返回该 key 关联的旧 value。如果 key 没有任何关联，则返回 null
Object get(Object key)	获得与 key 相关的 value。如果该 key 不关联任何非 null 值，则返回 null
boolean containsKey(Object key)	判断集合中是否存在 key
boolean containsValue(Object value)	判断集合中是否存在 value
boolean isEmpty()	判断集合中是否存在元素
void clear()	清除集合中的所有元素
int size()	返回集合中元素的数量
Set keySet()	获取所有 key 的集合
Collection values()	获取所有 value 的集合

Map 接口中存储的数据都是键 - 值对，例如，一个身份证号码对应一个人，其中身份证号码就是 key，与此号码对应的人就是 value。

2. 使用 HashMap 类动态存储数据

最常用的 Map 实现类是 HashMap，其优点是查询指定元素效率高。

示例 6

使用 HashMap 存储学生信息，要求可以根据英文名检索学生信息。

实现步骤：

（1）导入 HashMap 类。

（2）创建 HashMap 对象。

（3）调用 HashMap 对象的 put() 方法，向集合中添加数据。

（4）输出学员个数。

（5）输出键集。

（6）判断是否存在“Jack”这个键，如果存在，则根据键获取相应的值。

（7）判断是否存在“Rose”这个键，如果存在，则根据键获取相应的值。

关键代码：

```
// 创建学员对象
Student student1=new Student(" 李明 ", " 男 ");
Student student2=new Student(" 刘丽 ", " 女 ");
// 创建保存 " 键 - 值对 " 的集合对象
Map students=new HashMap();
// 把英文名称与学员对象按照“键 - 值对”的方式存储在 HashMap 中
students.put("Jack", student1);
students.put("Rose", student2);
// 输出学员个数
System.out.println(" 已添加 "+students.size()+" 个学员信息 ");
// 输出键集
System.out.println(" 键集："+students.keySet());
String key="Jack";
// 判断是否存在 "Jack" 这个键，如果存在，则根据键获取相应的值
if(students.containsKey(key)){
    Student student=(Student)students.get(key);
    System.out.println(" 英文名为 "+key+" 的学员姓名："+student.getName());
}
String key1="Rose";
// 判断是否存在 "Rose" 这个键，如果存在，则删除此键 - 值对
if(students.containsKey(key1)){
    students.remove(key1);
    System.out.println(" 学员 "+key1+" 的信息已删除 ");
}
```

输出结果如图 1.7 所示。

图 1.7　使用 HashMap 存储并检索学生信息

注意：

① 数据添加到 HashMap 集合后，所有数据的数据类型将转换为 Object 类型，所以从其中获取数据时需要进行强制类型转换。

② HashMap 类不保证映射的顺序，特别是不保证顺序恒久不变。

遍历 HashMap 集合时可以遍历键集和值集。

示例 7

改进示例 6，遍历所有学员的英文名及学员详细信息。

实现步骤：

（1）遍历键集。

（2）遍历值集。

关键代码：

```
// 创建学员对象
Student student1=new Student(" 李明 ", " 男 ");
Student student2=new Student(" 刘丽 ", " 女 ");
// 创建保存 " 键 - 值对 " 的集合对象
Map students=new HashMap();
// 把英文名称与学员对象按照“键 - 值对”的方式存储在 HashMap 中
students.put("Jack", student1);
students.put("Rose", student2);
// 输出英文名
System.out.println(" 学生英文名 :");
for(Object key:students.keySet()){
   System.out.println(key.toString());
}
// 输出学生详细信息
System.out.println(" 学生详细信息 :");
for(Object value:students.values()){
   Student student=(Student)value;
   System.out.println(" 姓名："+student.getName()+", 性别："+student.getSex());
}
```

输出结果如图 1.8 所示。

图 1.8　使用 HashMap 遍历学生英文名及详细信息

在示例 7 中，使用增强 for 循环遍历 HashMap 的键集和值集，当然也可以使用前面的普通 for 循环或者迭代器 Iterator 来遍历，视个人习惯而选择。

1.1.6　使用 Collections 类操作集合

Collections 类是 Java 提供的一个集合操作工具类，它包含了大量的静态方法，用于实现对集合元素的排序、查找和替换等操作。

注意：

Collections 和 Collection 是不同的，前者是集合的操作类，后者是集合接口。

1. 对集合元素排序与查找

排序是针对集合的一个常见需求。要排序就要知道两个元素哪个大哪个小。在 Java 中，如果想实现一个类的对象之间比较大小，那么这个类就要实现 Comparable 接口。此接口强行对实现它的每个类的对象进行整体排序，这种排序被称为类的自然排序，类的 compareTo() 方法被称为它的自然比较方法，此方法用于比较此对象与指定对象的顺序，如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。

compareTo() 方法的定义语法格式如下：

```
int compareTo(Object obj);
```

在语法中：

- 参数：obj 即要比较的对象。
- 返回值：负整数、零或正整数，根据此对象是小于、等于还是大于指定对象返回不同的值。

实现此接口的对象列表（和数组）可以通过 Collections.sort() 方法（和 Arrays.sort() 方法）进行自动排序。示例 8 通过实现 Comparable 接口对集合进行排序。

示例 8

学生类 Student 实现了 Comparable 接口，重写了 compareTo() 方法，通过比较学号实现对象之间的大小比较。

实现步骤：

（1）创建 Student 类。

（2）添加属性学号 number(int)、姓名 name(String) 和性别 gender(String)。

（3）实现 Comparable 接口、compareTo() 方法。

关键代码：

```
public class Student implements Comparable{
   private int number=0;                    // 学号
   private String name="";                  // 姓名
   private String gender="";                // 性别
   public int getNumber(){
       return number;
   }
   public void setNumber(int number){
       this.number=number;
   }
   public String getName(){
       return name;
   }
   public void setName(String name){
       this.name=name;
   }
   public String getGender(){
       return gender;
   }
   public void setGender(String gender){
       this.gender=gender;
   }
   public int compareTo(Object obj){
    Student student=(Student)obj;
    // 如果学号相同，那么两者就是相等的
    if(this.number==student.number){
       return 0;
    // 如果这个学生的学号大于传入学生的学号
    }else if(this.number>student.getNumber()){
       return 1;
    // 如果这个学生的学号小于传入学生的学号
    }else{
       return -1;
    }
   }
}
```

元素之间可以比较大小之后，就可以使用 Collections 类的 sort() 方法对元素进行排序操作了。前面介绍过 List 接口和 Map 接口，Map 接口本身是无序的，所以不能对 Map 接口做排序操作；但是 List 接口是有序的，所以可以对 List 接口进行排序。注意

List 接口中存放的元素，必须是实现了 Comparable 接口的元素才可以。

示例 9

使用 Collections 类的静态方法 sort() 和 binarySearch() 对 List 集合进行排序与查找。

实现步骤：

（1）导入相关类。

（2）初始化数据。

（3）遍历排序前集合并输出。

（4）使用 Collections 类的 sort() 方法排序。

（5）遍历排序后集合并输出。

（6）查找排序后某元素的索引。

关键代码：

```
// 省略声明 Student 对象代码
public static void main(String[] args){
    Student student1=new Student();
    student1.setNumber(5);
    Student student2=new Student();
    student2.setNumber(2);
    Student student3=new Student();
    student3.setNumber(1);
    Student student4=new Student();
    student4.setNumber(4);
    ArrayList list=new ArrayList();
    list.add(student1);
    list.add(student2);
    list.add(student3);
    list.add(student4);
    System.out.println("------- 排序前 -------");
    Iterator iterator=list.iterator();
    while(iterator.hasNext()){
        Student stu=(Student)iterator.next();
        System.out.println(stu.getNumber());
    }
    // 使用 Collections 类的 sort() 方法对 List 集合进行排序
    System.out.println("------- 排序后 -------");
    Collections.sort(list);
    iterator=list.iterator();
    while(iterator.hasNext()){
        Student stu=(Student)iterator.next();
        System.out.println(stu.getNumber());
    }
    // 使用 Collections 类的 binarySearch() 方法对 List 集合进行查找
    int index=Collections.binarySearch(list,student3);  // ①
    System.out.println("student3 的索引是："+index);
}
```

输出结果如下所示：

```
------- 排序前 -------
5
2
1
4
------- 排序后 -------
1
2
4
5
student3 的索引是：0
```

示例 9 中，①的代码是使用 Collections 类的 binarySearch() 方法对 List 集合进行查找，因 student3 的学号为 1，排序后索引变为 0。

2. 替换集合元素

若有一个需求，需要把一个 List 集合中的所有元素都替换为相同的元素，则可以使用 Collections 类的静态方法 fill() 来实现。下面通过一个示例来学习使用 fill() 方法替换元素。

➲ 示例 10

使用 Collections 类的静态方法 fill() 替换 List 集合中的所有元素为相同的元素。

实现步骤：

（1）导入相关类，初始化数据。

（2）使用 Collections 类的 fill() 方法替换集合中的元素。

（3）遍历输出替换后的集合。

关键代码：

```
public static void main(String[] args){
   ArrayList list=new ArrayList();
   list.add(" 张三丰 ");
   list.add(" 杨过 ");
   list.add(" 郭靖 ");
   Collections.fill(list, " 东方不败 ");                    // 替换元素
   Iterator iterator=list.iterator();
   while(iterator.hasNext()){
      String name=(String)iterator.next();
      System.out.println(name);
   }
}
```

输出结果如下所示：

```
东方不败
东方不败
东方不败
```

至此，任务 1 已经全部完成。

任务 2　改进新闻标题查询功能

关键步骤如下：

- 修改任务 1，将集合改为泛型形式。
- 修改遍历集合的代码。

1.2.1　泛型介绍

泛型是 JDK 1.5 的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数，使代码可以应用于多种类型。简单说来，Java 语言引入泛型的好处是安全简单，且所有强制转换都是自动和隐式进行的，提高了代码的重用率。

1. 泛型的定义

将对象的类型作为参数，指定到其他类或者方法上，从而保证类型转换的安全性和稳定性，这就是泛型。泛型的本质就是参数化类型。

泛型的定义语法格式如下：

类 1 或者接口 < 类型实参 > 对象 =new 类 2< 类型实参 >();

注意：

首先，“类 2”可以是“类 1”本身，可以是“类 1”的子类，还可以是接口的实现类；其次，“类 2”的类型实参必须与“类 1”中的类型实参相同。

例如：ArrayList<String> list=new ArrayList<String>();

上述代码表示创建一个 ArrayList 集合，但规定该集合中存储的元素类型必须为 String 类型。

2. 泛型在集合中的应用

前面学习 List 接口时已经提到，其 add() 方法的参数是 Object 类型，不管把什么对象放入 List 接口及其子接口或实现类中，都会被转换为 Object 类型。在通过 get() 方法取出集合中元素时必须进行强制类型转换，不仅繁琐而且容易出现 ClassCastException 异常。Map 接口中使用 put() 方法和 get() 方法存取对象时，以及使用 Iterator 的 next() 方法获取元素时存在同样问题。JDK 1.5 中通过引入泛型有效地解决了这个问题。JDK 1.5 中已经改写了集合框架中的所有接口和类，增加了对泛型的支持，也就是泛型集合。

使用泛型集合在创建集合对象时指定集合中元素的类型，从集合中取出元素时无

需进行类型强制转换，并且如果把非指定类型对象放入集合，会出现编译错误。

List 和 ArrayList 的泛型形式是 List<E> 和 ArrayList<E>，ArrayList<E> 与 ArrayList 类的常用方法基本一样，示例 11 演示了 List<E> 和 ArrayList<E> 的用法。

➲ 示例 11

使用 ArrayList 的泛型形式改进示例 2。

实现步骤：

（1）实现步骤同示例 2。

（2）创建集合对象时，使用的是 ArrayList<NewTitle>。

（3）遍历集合时不需要进行类型转换。

关键代码：

```
// 省略与示例 2 相同部分的代码
List<NewTitle> newsTitleList=new ArrayList<NewTitle>();
// 按照顺序依次添加新闻标题
newsTitleList.add(car);
newsTitleList.add(test);
// 根据位置获取相应新闻标题，逐条输出每条新闻标题的名称
System.out.println(" 新闻标题的名称为 :");
for (NewTitle title:newsTitleList) {
    System.out.println(title.getTitleName());
}
```

输出结果与示例 2 相同，如图 1.3 所示。

示例 11 中通过 <NewTitle> 指定了 ArrayList 中元素的类型，代码中指定了 ArrayList 中只能添加 NewTitle 类型的数据，如果添加其他类型数据，将会出现编译错误，这在一定程度上保证了代码安全性。并且数据添加到集合中后不再转换为 Object 类型，保存的是指定的数据类型，所以在集合中获取数据时也不再需要进行强制类型转换。

同样的，Map 与 HashMap 也有它们的泛型形式，即 Map<K,V> 和 HashMap<K,V>。因为它们的每一个元素都包含两个部分，即 key 和 value，所以，在应用泛型时，要同时指定 key 的类型和 value 的类型，K 表示 key 的类型，V 表示 value 的类型。

HashMap<K,V> 操作数据的方法与 HashMap 基本一样，示例 12 演示了 Map<K,V> 和 HashMap<K,V> 的用法。

➲ 示例 12

使用 HashMap 的泛型形式改进示例 7。

实现步骤：

（1）实现步骤同示例 7。

（2）创建集合对象时，使用的是 HashMap<String,Student>。

（3）遍历集合时不需要进行类型转换。

关键代码：

```
// 省略与示例 7 相同部分的代码
Map<String,Student> students=new HashMap<String,Student>();
```

```
// 把英文名称与学员对象按照“键 - 值对”的方式存储在 HashMap 中
students.put("Jack", student1);
students.put("Rose", student2);
// 输出英文名
System.out.println(" 学生英文名 :");
for(String key:students.keySet()){
    System.out.println(key);
}
// 输出学生详细信息
System.out.println(" 学生详细信息 :");
for(Student value:students.values()){
    System.out.println(" 姓名："+value.getName()+", 性别："+value.getSex());
}
```

输出结果与示例 7 相同，如图 1.8 所示。

在示例 12 中，通过 <String,Student> 指定了 Map 集合的数据类型，在使用 put() 方法存储数据时，Map 集合的 key 必须为 String 类型，value 必须为 Student 类型的数据，而在遍历键集的 for 循环中，变量 key 的类型不再是 Object，而是 String；在遍历值集的 for 循环中，变量 value 的类型不再是 Object，而是 Student，同样，Map.get(key) 得到的值也是 Student 类型数据，不再需要进行强制类型转换。

当然，其他的集合类，如前面讲到的 LinkedList、HashSet 等也都有自己的泛型形式，用法和 ArrayList、HashMap 的泛型形式类似，这里不再赘述。

泛型使集合的使用更方便，也提升了安全：

- 存储数据时进行严格类型检查，确保只有合适类型的对象才能存储在集合中。
- 从集合中检索对象时，减少了强制类型转换。

1.2.2　深入理解泛型

在集合中使用泛型只是泛型多种应用的一种，在接口、类、方法等方面也有着泛型的广泛应用。泛型的本质就是参数化类型，参数化类型的重要性在于允许创建一些类、接口和方法，其所操作的数据类型被定义为参数，可以在真正使用时指定具体的类型。

在学习如何使用泛型之前，还需要了解以下两个重要的概念：

- 参数化类型：参数化类型包含一个类或者接口，以及实际的类型参数列表。
- 类型变量：是一种非限定性标识符，用来指定类、接口或者方法的类型。

1. 定义泛型类、泛型接口和泛型方法

对于一些常常处理不同类型数据转换的接口或者类，可以使用泛型定义，如 Java 中的 List 接口。定义泛型接口或类的过程，与定义一个接口或者类相似。

（1）泛型类

泛型类简单地说就是具有一个或者多个类型参数的类。

定义泛型类的语法格式如下：

访问修饰符 class className<TypeList>

TypeList 表示类型参数列表，每个类型变量之间以逗号分隔。

例如：

```
public class GenericClass<T>{……}
```

创建泛型类实例的语法格式如下：

```
new className<TypeList>(argList);
```

- TypeList 表示定义的类型参数列表，每个类型变量之间以逗号分隔。
- argList 表示实际传递的类型参数列表，每个类型变量之间同样以逗号分隔。

例如：

```
new GenericClass<String>("this is String object")
```

（2）泛型接口

泛型接口就是拥有一个或多个类型参数的接口。泛型接口的定义方式与定义泛型类类似。

定义泛型接口的语法格式如下：

访问修饰符 interface interfaceName<TypeList>

TypeList 表示由逗号分隔的一个或多个类型参数列表。

例如：

```
public interface TestInterface<T>{
    public T print(T t);
}
```

泛型类实现泛型接口的语法格式如下：

访问修饰符 class className<TypeList> implements interfaceName<TypeList>

➲ 示例 13

定义泛型接口、泛型类，泛型类实现泛型接口，在泛型类中添加相应的泛型方法。

实现步骤：

1）定义泛型接口 TestInterface<T>，添加方法 getName()，并设置返回类型为 T。

2）定义泛型类 Student<T>，并实现接口 TestInterface<T>，声明类型为 T 的字段 name，添加构造方法。

3）使用 Student<T> 实例化 TestInterface<T>。

关键代码：

```
// 定义泛型接口
interface TestInterface<T>{
    public T getName();                          // 设置的类型由外部决定
}
// 定义泛型类
class Student<T> implements TestInterface<T>{    // 实现接口 TestInterface<T>
    private T name;                              // 设置的类型由外部决定
    public Student(T name){
        this.setName(name);
    }
```

```
    public void setName(T name){
        this.name=name;
    }
    public T getName(){                                    // 返回类型由外部决定
        return this.name;
    }
}
public class GenericesClass{
    public static void main(String[] args){
        TestInterface<String> student=new Student<String>(" 张三 ");      // ①
        System.out.println(student.getName());
    }
}
```

输出结果如下所示：

```
张三
```

在示例 13 中，①的代码用来创建 Student 对象，Student 泛型类的泛型参数定义为 String 类型，执行此代码后，TestInterface 接口和 Student 类中的泛型参数类型都为 String。并会通过 Student 类中只有一个 String 参数的构造方法来创建对象，则 name 属性的值为“张三”。

（3）泛型方法

一些方法常常需要对某一类型数据进行处理，若处理的数据类型不确定，则可以通过泛型方法的方式来定义，达到简化代码、提高代码重用性的目的。

泛型方法实际上就是带有类型参数的方法。需要特别注意的是，定义泛型方法与方法所在的类、或者接口是否是泛型类或者泛型接口没有直接的联系，也就是说无论是泛型类还是非泛型类，如果需要就可以定义泛型方法。

定义泛型方法的语法格式如下：

```
访问修饰符 < 类型参数 > 返回值 方法名（类型参数列表）
```

例如：

```
public <String> void showName(String s){ }
```

注意在泛型方法中，类型变量是放置在访问修饰符与返回值之间。

示例 14

定义泛型方法并调用。

实现步骤：

1）定义泛型方法。

2）调用泛型方法。

关键代码：

```
public class GenericMethod {
    // 定义泛型方法
    public <Integer> void showSize(Integer o){
```

```
        System.out.println(o.getClass().getName());
    }
    public static void main(String[] args) {
        GenericMethod gm=new GenericMethod();
        gm.showSize(10);
    }
}
```

输出结果如下所示：

```
java.lang.Integer
```

2. 多个参数的泛型类

前面的示例中，泛型类的类型参数都只有一个，实际上类型参数可以有多个，如 HashMap<K,V> 就有两个类型参数，一个指定 key 的类型，一个指定 value 的类型。下面介绍如何自定义一个包含多个类型参数的泛型类。

➲ 示例 15

定义泛型类，并设置两个类型参数。

实现步骤：

（1）定义泛型类。

（2）实例化泛型类。

关键代码：

```
// 创建泛型类
class GenericDemo<T,V>{
    private T a;
    private V b;
    public GenericDemo(T a,V b){
        this.a=a;
        this.b=b;
    }
    public void showType(){
        System.out.println("a 的类型是 "+a.getClass().getName());
        System.out.println("b 的类型是 "+b.getClass().getName());
    }
}
// 实例化泛型类
public class Demo{
    public static void main(String[] args) {
        GenericDemo<String,Integer> ge1=new GenericDemo<String,Integer>("Jack",23);
        ge1.showType();
    }
}
```

输出结果如下所示：

a 的类型是 java.lang.String
b 的类型是 java.lang.Integer

在示例 15 中，GenericDemo<T,V> 类定义了两个类型参数，分别是 T 和 V。定义时这两个类型变量的具体类型并不知道。注意，当在一个泛型中，需要声明多个类型参数时，只需要在每个类型参数之间使用逗号将其隔开即可。在实例化泛型类时，就需要传递两个类型参数，这里分别使用了 String 和 Integer 代替了 T 和 V。

3. 从泛型类派生子类

面向对象的特性同样适用于泛型类，所以泛型类也可以继承。不过，继承了泛型类的子类，必须也是泛型类。

继承泛型类的语法格式如下：

```
class 子类 <T> extends 父类 <T>{ }
```

示例 16

定义泛型父类，同时定义一个泛型子类继承泛型父类。

实现步骤：

（1）定义父类 Farm<T>，并添加整型字段 plantNum、方法 plantCrop(T crop)。

（2）定义子类 FruitFarm <T>，重写方法 plantCrop (List<T> list)。

关键代码：

```
// 父类 Farm<T>( 农场类 )
public class Farm<T>{
   protected int plantNum=0;                    // 农作物种植数量
   public void plantCrop(T crop){               // 种植农作物的方法
      plantNum ++;
   }
}
// 子类果园类继承泛型类 Farm<T>
public class FruitFarm<T> extends Farm<T>{
   public void plantCrop(List<T> list){         // 重写种植农作物的方法
      plantNum+=list.size();
   }
}
```

至此，任务 2 已经全部完成。

本章总结

本章学习了以下知识点：

- 集合弥补了数组的缺陷，它比数组更灵活实用，而且不同的集合适用于不同场合。
- Java 集合框架共有三大类接口，即 Map 接口、List 接口和 Set 接口。

- ArrayList 和数组采用相同的存储方式，它的特点是长度可变且可以存储任何类型的数据，它的优点在于遍历元素和随机访问元素的效率较高。
- LinkedList 采用链表存储方式，优点在于插入、删除元素时效率较高。
- Iterator 为集合而生，专门实现集合的遍历，它隐藏了各种集合实现类的内部细节，提供了遍历集合的统一编程接口。
- HashMap 是最常用的 Map 实现类，它的特点是存储键值对数据，优点是查询指定元素效率高。
- 泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数，使代码可以应用于多种数据类型。
- 使用泛型集合在创建集合对象时指定集合中元素的类型，从集合中取出元素时无需进行强制类型转换。
- 在集合中使用泛型只是泛型多种应用的一种，在接口、类、方法等方面也有着泛型的广泛应用。
- 如果数据类型不确定，可以通过泛型方法的方式，达到简化代码、提高代码重用性的目的。

本章练习

1．编写 Java 程序，创建一个 HashMap 对象，并在其中添加学生的姓名和成绩，键为学生姓名（String 类型），值为学生成绩（Integer 类型）。使用增强 for 循环遍历该 HashMap，并输出学生成绩。程序输出结果如图 1.9 所示。

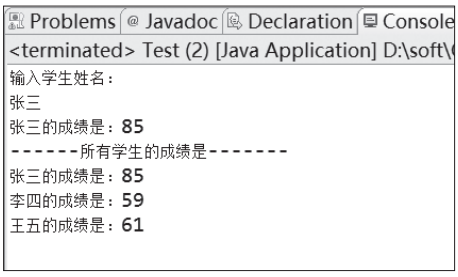

图 1.9　学生成绩查询结果

2．编写 Java 程序，创建 3 个 ArrayList 对象，每个对象中添加一些学员的姓名。再创建 HashMap 对象，以年级名称为键，存放学员的 ArrayList 为值。然后从 HashMap 对象中获取某个班级的学员信息并输出。程序输出结果如图 1.10 所示。

3．编写 Java 程序，创建学员类 Student，并添加姓名、年龄、性别等字段，创建 3 个 ArrayList<T> 对象，指定 T 为 Student 类，每个 ArrayList<T> 中添加一些学员对象，再创建 HashMap<K,V> 对象，以年级名称为键，指定为 String 类型，指定 value 类型为 ArrayList<Student>，值为存放学员的 ArrayList<T> 对象，然后从 HashMap<K,V>

对象中获取某个班级的学员信息并输出。程序输出结果如图 1.11 所示。

图 1.10　班级学生列表

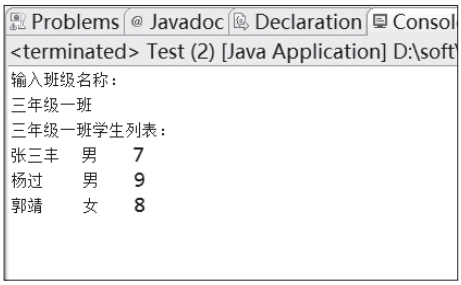

图 1.11　查询班级全部学员信息