第5章 Java集合框架


本章习题

【本章知识点】

- Collection 接口与 Map 接口。
- List 接口及其实现类：ArrayList 与 LinkedList。
- 泛型。
- Set 接口及其实现类：HashSet 与 TreeSet。
- Iterator(迭代器)。
- Map 接口的实现类：HashMap 与 TreeMap。
- Collections 工具类。

在编写程序过程中，经常需要存储结构相同但数量不定的对象，不同的应用场景对顺序、速度和查找的需求各有侧重。因此，Java 集合框架(Java Collections Framework，JCF)应运而生，以满足这些多样化的需求。本章首先介绍 Collection(集合)和 Map(映射)这两大核心接口及其子接口，如 List(列表)、Set(集)等；然后深入剖析其常用的实现类，包括 ArrayList/LinkedList、HashSet/TreeSet、HashMap/TreeMap 的实现机制及性能特点；最后探讨 Comparable/Comparator 排序、Iterator 遍历以及 Collections 工具类中的算法。通过这些内容的学习，程序员可以根据具体场景灵活选择顺序表、链表、哈希表或哈希树等数据结构，从而实现高效的数据存取、排序和批量处理。

5.1 容器演进与集合框架

Java 1 时代，开发者只能依赖原生数组和少量工具类。数组长度固定、类型单一，一旦容量不足就必须手动扩容、复制；Vector(向量)和 Hashtable(哈希表)虽能动态增长，却把所有方法都声明为 synchronized(同步)，线程安全换来高昂性能损耗。当时，“容器”这一概念尚未明确，代码中遍布着烦琐的类型转换和显式的同步操作，使得项目规模稍大便难以维护，如同陷入了一片泥潭。

1998 年发布的 Java 2 彻底改写了编码历史。Sun 引入了一整套接口体系：Collection、Set、List、Map 4 大概念被抽象为独立接口，并提供了 ArrayList、LinkedList、HashSet、HashMap 等高效实现。统一的 size()、add()、remove()方法让容器操作第一次拥有了“面向接口”的优雅；同时，Iterator(迭代器)模式取代了 Enumeration，fail-fast 机制(快速失败机制，Java 集合框架中用于检测并发修改的机制)让并发修改异常可被及早发现。Java 集合框架由此正式诞生。

Java 5.0 将 Queue(队列)及其子接口 Deque 纳入标准，为生产者-消费者场景提供了 LinkedList、PriorityQueue 等实现。Java 5.0 的泛型消除了强制类型转换；并发包 java.util.concurrent 则带来了线程安全的 ConcurrentHashMap、CopyOnWriteArrayList 等，让“读多写少”与“高并发”成为性能调优的常态。

Java 8.0 引入的 Stream API 与 Lambda 表达式，使过滤、映射、归约可在一条链式调用中完成；同时 Collections.sort 被 List.sort＋Comparator.comparing 取代，代码更简洁。Java 9.0 提供的 List.of、Set.of、Map.of 等静态工厂，则让不可变容器成为默认首选，既节省内存又避免并发隐患。

Java 集合框架主要包括接口、实现类与工具类的三层结构，它们之间的关系如图 5-1 所示。

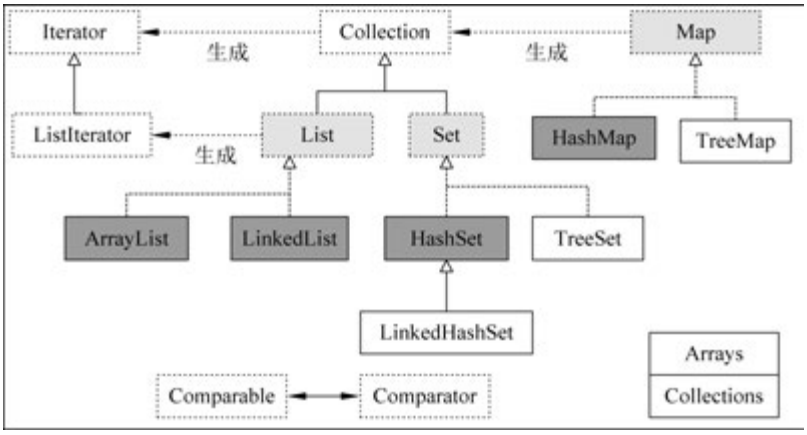

图 5-1　Java 集合框架中接口、实现类和工具类之间的关系

（1）接口层：Collection 与 Map 两大父接口；Collection 接口派生出三个主要子接口，分别为 List（有序且元素可重复）、Set（元素不可重复）、Queue（先进先出或优先级队列）。

（2）实现类层：ArrayList（线性表/动态数组）、LinkedList（双向链表）、HashMap（数组＋链表＋红黑树）、TreeMap（红黑树）、HashSet（HashMap 包壳）、TreeSet（TreeMap 包壳）等。

（3）工具类层：在 Java 集合框架中，Iterator（迭代器）用于统一遍历集合；List 接口通过实现 Comparable 接口或使用 Comparator（比较器）来定义排序规则；而 Collections 和 Arrays 类提供了批量操作、同步包装、空安全处理、二分查找等多种实用算法。

至此，Java 容器完成了从“零散工具”到“统一框架”的华丽转身，为开发者提供了丰富、高效且易维护的数据结构选择。

图 5-1 中主要包括三个接口：Map、List 和 Set，而且每个接口都有两三个常用的实现类。虚线框代表“接口”，实线框代表普通（实现）类。实线箭头代表继承，点线箭头代表实现接口。实心箭头表示一个类可生成箭头指向的那个类的对象。例如，任何集合（Collection）都可以生成一个迭代器（Iterator），而一个列表（List）可以生成一个 ListIterator（以及原始的 Iterator，因为 List 是从 Collection 继承的）。

本章内容基于数据结构的理论展开，感兴趣的读者可以查阅相关论文与书籍以获得更深入的理解。

5.2　Collection 接口

下面是 Collection 接口的声明：

```
public interface Collection<E> extends Iterable<E>
```

Collection 接口从 Iterable 接口继承，代表所有实现 Collection 接口以及 Collection 子接

口的类都可以使用 Iterator，Iterator 会在后文讲解。先来看看 Collection 接口的常用方法，如表 5-1 所示。

表 5-1 Collection 接口的常用方法

方 法	描 述
add()	向集合中添加一个元素
addAll()	将另一个集合的所有元素添加到本集合
clear()	清空本集合
contains()	判断集合中是否存在某元素
containsAll()	判断一个集合是否为本集合的子集
equals()	判断两个集合是否相等
isEmpty()	判断集合是否为空
remove()	从集合中移除一个元素
size()	查看集合中元素的数量
toArray()	将本集合转换为一个同类型数组并返回数组引用
iterator()	生成一个本集合的迭代器

可以看出，Collection 接口提供了对一组元素集合的添加、删除以及一些常用操作，这些功能都是元素集合必须的。

Collection 接口的两个最常见子接口是 List 与 Set。

▶ 5.2.1 List 接口及其实现类

List 接口是 Collection 的一个有序子接口。元素的顺序性意味着每个元素都占据一个独特的位置，该位置表示元素间的先后关系。可以根据元素位置访问元素，能按固定顺序遍历集合，并确保元素按特定顺序（如插入顺序）排列。因此，能精确控制每个元素的插入位置。用户可以使用“位置”（类似于数组下标）访问 List 中的元素，这点与 Java 数组类似。顺序（Sequence）是 List 区别于其他集合的核心特征，也是其最重要的属性。

List 在 Collection 基础上新增了许多与“位置”相关的方法，位置也被称为元素的“索引”（Index），便于在 List 中插入或删除元素，如表 5-2 所示。

表 5-2 List 接口新增方法

方 法	描 述
add()	重载的 add()方法，可在指定位置处插入元素
remove()	重载的 remove()方法，可在指定位置处删除元素
get(int)	获取指定位置的元素
set()	重设指定位置的元素的值
indexOf()	获取指定元素的位置
lastIndexOf()	获取指定元素最后一次出现的位置
subList()	获取列表的子列表，需给出起始位置和终止位置
listIterator()	生成一个列表迭代器，此迭代器功能更强大

实现 List 接口的常用类有 ArrayList（线性表）和 LinkedList（链表）等，下面分别加以介绍。

1. ArrayList

2.2.3 节介绍过 Java 数组，它既能保存基本类型数据，又能保存对象，并能快速存取元素。然而数组存在局限性：创建并初始化后，其容量即固定不变。当空间不足时，只能创建新数组

并将所有元素复制过去，这种方式效率低下。在实际编程中，由于无法预先知道所需对象的数量，因此需要一种能够动态调整容量的数据结构，如动态数组，以适应不同情况下的需求。Java 的 ArrayList 就是用来解决此问题，但 ArrayList 仅能保存对象，不能保存基本类型。

ArrayList 实现了容量可变的数组，兼具数组快速存取的优点和容量动态变化的特性，故也被称为动态数组。在 ArrayList 中间位置插入或删除元素时，效率不高，原因在于其需要维护连续的存储空间，因此，任何对中部元素的增、删操作，都会引发后续所有元素的位置移动。ArrayList 实现了“线性表”数据结构，因此其优缺点正是线性表的优缺点。

使用 ArrayList 十分便捷，以下示例展示了其基本用法。

【例 5.1】 ArrayList 的使用。

```
 1. import java.util.ArrayList;
 2. public class TestArrayList {
 3.     public static void main(String args[]) {
 4.         ArrayList a = new ArrayList();     //建立一个空的线性表
 5.         a.add( "cat" );                    //添加字符串对象
 6.         a.add( "dog" );
 7.         a.add( "cat" );
 8.         a.remove(0);
 9.         for(int i = 0; i < a.size(); i++) { // a.size() = 2
10.             System.out.print(a.get(i) + ",");
11.         }
12.     }
13. }
```

输出结果：

```
dog,cat,
```

程序说明：第 4 行代码创建空线性表 a；第 5～7 行代码向 a 添加三个字符串对象；第 8 行删除最先插入的字符串"cat"。最后输出剩余的两个字符串，其中 a.size()方法记录表内元素个数。可见 ArrayList 的使用十分简便且无须维护。

需注意，所有集合均不能保存基本类型。若需保存基本类型，应先将其转换为封装类再存入集合。例 5.1 中 String 非基本类型，故无须封装。

ArrayList 提供三种构造方法，如表 5-3 所示。

表 5-3 ArrayList 的构造方法

方　法	描　述
ArrayList()	构造一个空的线性表
ArrayList(Collection)	根据已有集合构造线性表
ArrayList(int)	构造指定初始容量大小的线性表

新建一个 ArrayList 很简单，如下使用构造方法构造了三个线性表：

```
ArrayList a = new ArrayList();       //a.size() = 0
ArrayList b = new ArrayList(20);     //b.size() = 0
ArrayList c = new ArrayList(b);      //c.size() = 0
```

为什么构造线性表需要指定容量？因为 ArrayList 内部维护了一个 Object 类型数组，所有增、删、改、查操作都作用于这个数组。创建 ArrayList 时，必须同时创建并初始化该数组，因此需要确定其容量。如例 5.1 中 b 的容量为 20，意味着其内部数组容量为 20。无参构造方法（如 a）默认容量为 10，而 c 的容量则与 b 相同。

这种情况可能导致问题：持续向 ArrayList 添加元素时，容量终将被耗尽。此时

ArrayList 会创建容量翻倍的新数组，并将原数组数据全量复制到新数组——这种扩容操作代价高昂。例如向容量为 10 的 ArrayList 连续添加 200 个元素，将触发扩容。根据 ArrayList 的扩容机制，每次扩容大约为原容量的 1.5 倍，因此，从容量 10 开始，添加 200 个元素将触发多次扩容，直到能够容纳所有元素。若初始创建时指定容量 200，则可完全避免此问题。因此，准确预估容量是提升 ArrayList 性能的关键。

值得注意的是，ArrayList 常用方法多定义于 List 或 Collection 接口，故可通过接口引用 ArrayList：

```
List a = new ArrayList();
a.add(…);
a.remove(…);
```

上述代码中，第 1 行代码将生成的线性表对象实例赋给接口引用变量 a，这是向上转型的过程。其优势在于，初始采用 ArrayList 结构时，若后续需改用 LinkedList，则仅需将第 1 行改为：

```
List a = new LinkedList();
```

后续代码无须修改，显著提升代码可维护性。这种用法十分普遍，是面向对象编程中接口与实现分离的典型应用。后文中，只要不涉及某些类的特殊方法，都优先使用接口定义类。

ArrayList 不擅长查找操作，频繁调用 indexOf()、contains()等方法会导致效率低下。同样，ArrayList 也不适用于频繁增、删中间数据，反复执行 remove(i)、add(i,e)这类操作同样会降低效率。

2. LinkedList

LinkedList 实现了“链表”数据结构，因此其优缺点直接体现了链表的特性。LinkedList 针对顺序存取进行了优化，在列表中间执行插入或删除操作时效率较高。但由于未采用 ArrayList 的顺序存储机制，因此 LinkedList 无法通过索引实现快速随机存取。

LinkedList 内部采用了带头节点的双向链表结构，从而支持高效的双向访问，在这种结构下，插入和删除操作均能在常数时间内高效完成。该双向链表的具体结构如图 5-2 所示。

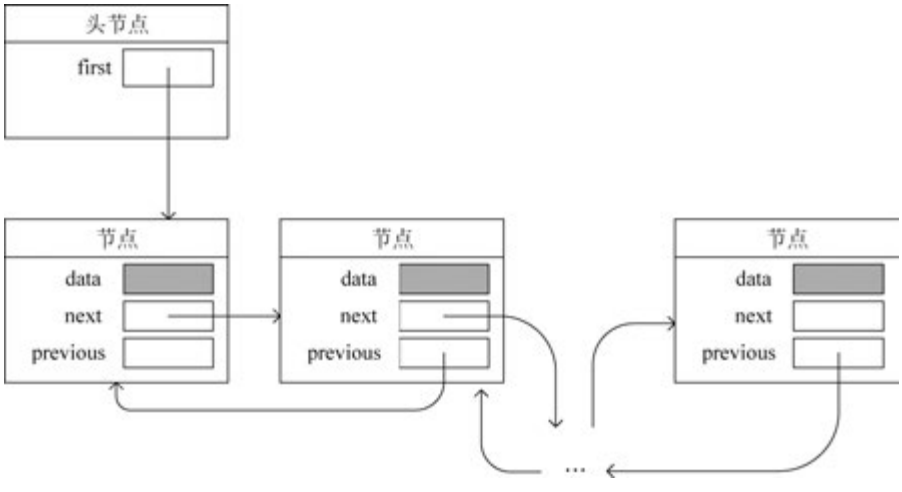

图 5-2 带头节点的双向链表的具体结构

线性表在连续存储空间中顺序存放对象引用，而链表将每个对象存储于独立节点内。每个节点保存有序列中后继节点的引用；由于链表是双向结构，节点还保存指向前驱节点的引用。在链表中删除(或添加)元素的代价极小——仅需更新被删除(或添加)元素前驱与后继节

点的引用即可。删除操作如图 5-3 所示。

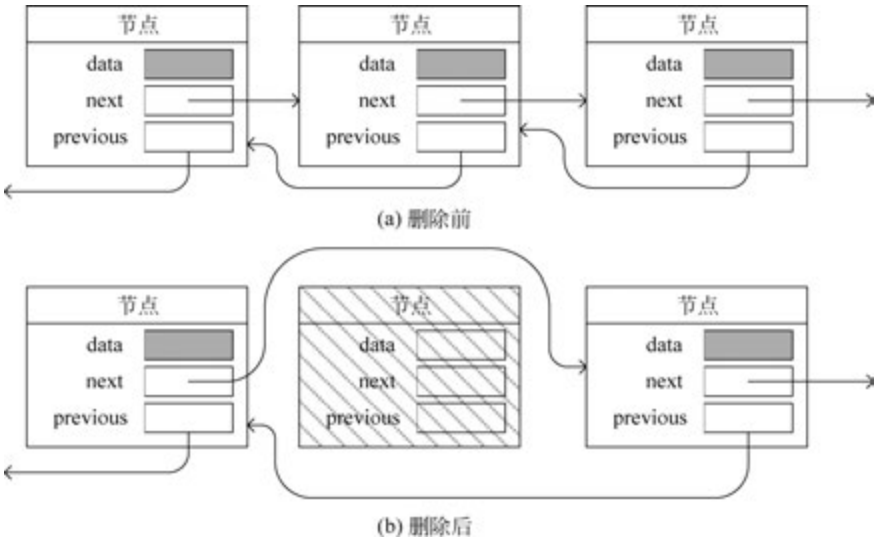

图 5-3　双向链表删除操作

LinkedList 在 List 接口的基础上增加了一些特殊的方法，如表 5-4 所示。

表 5-4　LinkedList 新增方法

方　法	描　述
addFirst()	向链表头插入元素
addLast()	向链表尾插入元素
getFirst()	获取链表头元素
getLast()	获取链表尾元素
removeFirst()	移除链表头元素
removeLast()	移除链表尾元素

这些方法含义非常好理解。LinkedList 还有许多未列举的方法，例如 element()方法获取链表头元素、offer()按队列规则向链表尾添加元素、peek()获取链表头元素、poll()获取并移除链表头元素等。这些方法本质上是表 5-4 中方法的组合调用，部分甚至与其中方法功能完全相同仅名称不同。例如，element()方法与表 5-4 中的 getFirst()方法完全一致，之所以为相同功能设置不同名称，是为了符合通用命名规范或兼容其他编程语言，从而增强代码的通用性和可读性。

使用 LinkedList 可轻松构建 Stack（栈）、Queue（队列）等数据结构。一起来看看如何用 LinkedList 实现栈结构。栈具备先进后出（FILO）特性，其核心方法包括元素入栈方法 push()、元素出栈方法 pop()以及查看栈顶元素方法 top()。

【例 5.2】 利用 LinkedList 类构造“栈”结构。

```
1. import java.util.LinkedList;
2. class StackL {                          //构造栈结构
3.     private LinkedList list = new LinkedList();
4.     public void push(Object o) {        //进栈
5.         list.addFirst(o);
6.     }
7.     public Object top() {               //查看栈顶元素
8.         return list.getFirst();
```

```
 9.     }
10.     public Object pop() {              //出栈
11.         return list.removeFirst();
12.     }
13. }
14. public class ConstructStack {
15.     public static void main(String[] args) {
16.         StackL s = new StackL();
17.         s.push("cat");
18.         s.push("dog");
19.         s.push("monkey");
20.         s.pop();
21.         System.out.println(s.top());
22.     }
23. }
```

输出结果：

```
dog
```

程序说明：构建栈的过程十分简单，构建队列也与之类似。此例中 monkey 最后入栈，调用 pop()方法后最先出栈，符合人们对栈的预期。

在 ArrayList 中进行随机访问(即 get())以及循环访问性能最佳，但这些操作对于 LinkedList 却代价高昂。另外，在列表中部进行插入和删除操作，LinkedList 的效率则远高于 ArrayList。实践经验是初始选择使用 ArrayList，若发现因大量插入和删除导致性能下降，再考虑切换为 LinkedList。

▶ 5.2.2　对象的比较

数组和集合中经常需要使用对象的比较功能。例如，在 2.2.3 节中提到的数组工具类方法 Arrays.sort()可以对数组进行排序。那么，当数组元素为对象类型时，依据什么标准来判断数组中两个对象之间的大小，进而对其进行排序呢？Java 为此提供了两个用于比较大小的接口，任何实现了这些接口之一的对象，都将具备比较大小的基础能力。

1. Comparable 接口

java.lang.Comparable 接口中只有一个方法：

```
int compareTo(Object o)
```

其中，对象 o 是被比较对象。大于、等于或小于指定对象 o，分别返回正整数、0 或负整数。

【例 5.3】 一个实现了 Comparable 接口的可比较类。

```
1. import java.lang.Comparable;
2. class Employee implements Comparable {
3.     int id;
4.     String name;
5.     public int compareTo(Object o) {
6.         Employee e = (Employee) o;
7.         return this.id - e.id;
8.     }
9.  }
```

程序说明：在公司中，通常员工号靠前的员工被视为公司的老员工，其各方面福利待遇应优于员工号较小的新员工。例子中的员工类 Employee 通过员工号 id 来比较它们的大小，id 值相同即代表为同一员工。在 Java 的视角中，这个 Employee 类是具备大小比较能力的，因此

当调用 Arrays.sort()方法对 Employee 类型数组进行排序时，系统将自动调用 Employee 类中的 compareTo()方法进行比较，并按照 id 值从小到大的顺序进行排列。然而，对于未实现 compareTo()方法的类，Java 无法进行大小比较，此时调用 Arrays.sort()方法将会引发异常。

2. Comparator 接口

同样是比较大小，Comparator 和 Comparable 接口在使用方法和应用领域上存在显著差异。java.util.Comparator 接口定义了一个 compare()方法：

```
int compare((Object o1, Object o2)
```

compare()方法的返回值与 Comparable 接口的 compareTo()方法类似。当对象 o1 大于、等于或小于对象 o2 时，分别返回正整数、0 或负整数。任何实现了 Comparator 接口的类都将成为一个“比较器”，具备比较两个对象大小的能力。

尽管 Comparator 和 Comparable 两个接口都能实现对象的大小比较，但 Comparator 具有更优的模块性。Comparator 可以被视为一种算法的实现，实现了算法与数据的分离。Comparator 适用于以下场景。

（1）当类的设计者未考虑比较问题、未实现 Comparable 接口时，可以通过 Comparator 实现排序，而无须修改对象本身。

（2）通过构建不同的比较器，可以实现多种排序标准，例如升序、降序等。

如例 5.4 所示，进一步完善了例 5.3 中 Employee 类的比较功能。

【例 5.4】 实现了 Comparable 接口的降序比较器。

```
1. class DescComparator implements Comparator { //降序比较器
2.     public int compare(Object o1, Object o2) {
3.         Comparable c1 = (Comparable)o1;
4.         return 0 - c1.compareTo(o2);
5.     }
6. }
```

程序说明：当需要将例 5.3 中的 Employee 类按 id 降序排列时，可以使用 DescComparator 类作为比较器。从第 4 行代码可以看出，该类通过将 Comparable 接口方法的结果取负，使得排序结果与 Comparable 的排序结果相反。此时，可以通过以下语句实现降序排列：

```
Arrays.sort((employees, new DescComparator());   //此处 employees 是一个 Employee 集合
```

Arrays.sort()方法在使用比较器时需要两个参数：第一个参数为被比较的对象集合；第二个参数为比较器的引用。DescComparator 比较器不仅适用于 Employee 类，还能为所有实现了 Comparable 接口的类进行各自的降序排列。这充分体现了 Comparator 模块化的优势，既增强了通用性，又简化了代码。

▶ 5.2.3 泛型

泛型(Generics)是 Java 集合框架中很重要的一个概念，介绍泛型之前，先来看一段程序。

【例 5.5】 无泛型程序示例。

```
1. import java.util.*;
2. class Cat{}
3. class Dog{}
4. class NoGenerics {
5.     public static void main(String[] args) {
6.         List a = new ArrayList();
```

```
 7.          a.add( "haha" );                    //插入一个 String 类对象
 8.          a.add( new Cat() );                 //插入一个 Cat 类对象
 9.          a.add( new Dog() );                 //插入一个 Dog 类对象
10.          String s = (String)a.get(0);        //取出第一个元素
11.      }
12. }
```

程序说明：这段程序可以正常执行。在程序中，由于所有集合类型都将插入的元素视为Object类处理，因此可以向同一个ArrayList中插入Cat类、Dog类、String类或其他任意类的对象。因为在插入集合前，所有元素都被强制转换成了Object类型。但这带来了如下问题。

(1) 当从列表a中取出元素并作为String使用时(例如将其赋值给String类型变量s)，将产生编译异常，因为系统认为取出的元素属于Object类型。此时必须先将get()方法的结果显式地强制转换为String类型才能继续使用，这无疑增加了代码复杂度及出错的可能性。

(2) 后续使用列表a时，可能误将其他类对象(如Cat)插入其中，而系统不会提示错误。直到强制转换时，代码才会抛出ClassCastException。

为此，有时需要定义仅保存单一类型的集合，例如限定某列表仅包含String类型。于是Java引入了泛型机制。Java中许多重要类(如本章的集合类)均已实现泛型化。泛型的核心目标是提升Java程序的类型安全性。借助泛型为变量添加显式类型约束，编译器得以在语义层面验证先前仅存于开发者思维中的类型假设。若缺乏泛型，此类假设仅为隐式约定，运行时方能暴露可能的违背之处；而泛型通过在变量声明时捕获并固化类型信息，使编译器能在编译阶段强制执行类型契约，从而将原本潜伏至运行时、以ClassCastException形式显现的错误前移至编译期加以拦截。此机制不仅显著缩短缺陷定位周期，也整体提升了软件系统的可靠性。与此同时，泛型消除了冗余的强制类型转换，既增强了代码的简洁性与可读性，又进一步降低了因显式转型而引入错误的概率。

泛型是Java 5引入的语言特性，旨在为类、接口或方法提供编译阶段的参数化类型机制。简言之，它将“具体操作的类型”抽象为形式参数，由调用者在编译时明确指定。这不仅消除了强制类型转换的需求，还引入了额外的类型检查机制，有效防止错误类型的对象被存入集合。

1. 泛型变量

在声明泛型类变量时，需使用尖括号“< >”来指定形式类型参数。形式类型参数与实际类型参数的关系，类似于方法中的形式参数与实际参数，不同之处在于，这里的类型参数代表的是数据类型。需要注意的是，Java泛型不支持基本数据类型，这一点与C++不同。

泛型可用于类和接口。例如：

```
List < String >  a = new ArrayList < String >();
```

这句代表的是列表a能且只能保存String类或String子类类型的元素，也可以说只能保存可以强制类型转换为String类型的元素。此时例5.5可改为例5.6所示的代码。

【例5.6】 带泛型程序示例。

```
1. class Cat{}
2. class Dog{}
3. class WithGenerics {
4.     public static void main(String[] args) {
5.         List < String > a = new ArrayList < String >(); //使用泛型
6.         a.add("haha");
7.         //a.add(new Cat());  //报错,无法插入
8.         //a.add(new Dog());  //报错,无法插入
```

```
 9.          String s = a.get(0);   //无须强制转换,取出的元素类型就是 String 类型
10.     }
11. }
```

程序说明：定义泛型列表 a 时必须显式指定两次类型参数。首先在声明变量 a 的类型 List＜T＞时指定，其次在创建 ArrayList 实例时需再次声明具体类型（第 5 行）。当编译器识别 List＜String＞类型变量时，即可推断泛型 T 已绑定为 String 类型，因此明确调用 a.get()将返回 String 类型对象（第 10 行）。

除异常、枚举及匿名内部类外，所有类均可使用泛型机制。

2. 泛型方法

方法本身也可以声明为泛型，无论它所属的类是否为泛型类。这么做的主要目的在于借助类型参数在方法的多个形式参数之间建立编译期类型约束。

例如，下面代码中的 testGen()方法，根据它的第一个参数的布尔值，它将返回第二个或第三个参数：

```
public < T > T testGen (boolean b, T first, T second) {
    return b ? first : second;
}
```

可以调用 testGen()方法，而不用显式地告诉编译器 T 代表什么类型；编译器只知道这些值类型都必须相同。

编译器允许调用下面的代码，因为编译器可以使用类型推理来推断出替代 T 的 String 或 Integer 满足所有的类型约束。

```
String s = testGen(true, "a", "b");
Integer i = testGen(false, new Integer(1), new Integer(2));
```

但是，编译器不允许调用下面的代码，因为没有类型会满足所需的类型约束。

```
String s = testGen(true, "pi", new Float(3.14));
```

3. 泛型集合

所有的标准集合框架的接口都是泛型化的，如表 5-5 所示。

表 5-5　标准集合框架接口泛型定义

接　　口	声　　明
Collection	interface Collection＜E＞ extends Iterable＜E＞
Map	interface Map＜K,V＞
List	interface List＜E＞ extends Collection＜E＞
Set	interface Set＜E＞ extends Collection＜E＞

Collection 接口中的 E 代表集合中元素的类型，Map 接口中的 K、V 代表映射中键与值的类型。这些字母只是代表某个类型的符号，可以任意指定。

类似地，集合接口的实现都是用相同类型参数泛型化的，例如 ArrayList＜E＞实现 List＜E＞。同样，集合类也使用泛型化的方法，例如接口 Collection＜E＞中的 add()方法声明如下：

```
interface Collection < E > {
    boolean add(E e);
}
```

5.2.2 节介绍的 Comparable 接口也是泛型化的，所以实现 Comparable 的对象能声明它可以与什么类型进行比较。通常是对象本身的类型，但有时也可能是其父类。

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

声明实现 Comparable 接口的类(如 String)时,不仅需要声明该类支持比较,还需要说明其可比较的对象类型(通常是其自身类型)。

```
public class String implements Comparable<String> {
    …
}
```

4. 泛型的使用

在例 5.2 中使用 LinkedList 类构造了一个栈结构,但是这个栈结构没有使用泛型约束其元素的类型,现将其改造成一个带泛型的类,使其通用性更强。

【例 5.7】 利用 LinkedList 类构造泛型化的栈结构。

```
 1. import java.util.LinkedList;
 2. class StackL<T> {
 3.     private LinkedList<T> list = new LinkedList<>();
 4.     public void push(T t) {
 5.         list.addFirst(t);
 6.     }
 7.     public T top() {
 8.         return list.getFirst();
 9.     }
10.     public T pop() {
11.         return list.removeFirst();
12.     }
13. }
14. public class StackGen {
15.     public static void main(String[] args) {
16.         StackL<String> s = new StackL<>();
17.         s.push("cat");
18.         s.push("dog");
19.         s.push("monkey");
20.         s.pop();
21.         System.out.println(s.top());
22.     }
23. }
```

输出结果:

```
dog
```

程序说明:此例中的 StackL 类和例 5.6 中的 ArrayList 类一样拥有泛型效果。编写程序时经常用到泛型。

▶ 5.2.4 Set 接口及其实现类

Set 接口的声明如下:

```
public interface Set<E> extends Collection<E>
```

Set 接口实现了数学意义上的"集合"数据结构。与 List 接口不同,Set 接口最显著的特性是不允许存储重复元素;此外,Set 接口不具备 List 接口的顺序概念,这正如数学中集合{1,2}与{2,1}完全等价。在 List 接口中,元素根据索引相互区分;相比之下,Set 接口中的元素则凭借其值确保唯一性。

Set 接口继承自 Collection 接口。不同于 List 接口，Set 接口并未引入新方法，其所有功能均继承自 Collection 接口。因此，Set 接口在行为上完全等同于 Collection 接口，只是额外限定了所有元素值必须互异。

Set 接口的几种常见实现类包括 HashSet、TreeSet 和 LinkedHashSet。其使用方式与 List 接口类似，简单示例如下：

```
1. Set<String> set = new HashSet<>();
2. set.add("cat");
3. set.add("dog");
4. set.add("cat");                    //无法插入
5. set.remove("cat");
6. System.out.println(set);
```

输出结果：

```
[dog]
```

上述代码中，第 4 行无法插入重复元素 cat。HashSet 中只保留了不重复的元素，因为 Set 接口不允许保存相同值的元素。

1. HashSet（哈希集）

Set 接口中不允许保存相同值的元素，那么要向 Set 接口中添加某个元素 a 时就要保证 a 的值和现有 Set 接口中的元素值全部不相等。很自然的一个想法就是在向 Set 接口中添加 a 之前，将 a 的值和每一个 Set 接口中的元素进行比较，如果不相等则和下一个比较直到无元素结束，如果相等则返回 false，无法添加。实际上就是在数据集中查找 a，如果找到了就不添加，如果没找到就正常添加。可是如果 Set 接口中的元素数量巨大，每添加一个元素就要做一次上述这样的循环比较，显然效率十分低下。

有一种可以快速查找特定对象的数据结构，即哈希表。哈希表可以对每一个对象计算一个整数，称为哈希码（Hash Code）。这个整数可以经过一些变换生成对象保存的地址。也就是说给出一个对象，利用哈希算法可以迅速计算出这个对象的位置从而跳过繁冗的循环比较工作，达到高效查找的目的。HashSet 正是实现了这种数据结构的 Set 接口实现类。

HashSet 的构造方法如表 5-6 所示。

表 5-6　HashSet 的构造方法

方　法	描　述
HashSet()	构造空的 HashSet，容量为 16，加载因子为 0.75
HashSet(Collection)	从其他集合构造 HashSet
HashSet(int)	指定初始容量、加载因子为 0.75 的 HashSet
HashSet(int,float)	指定初始容量与加载因子的 HashSet

关于初始容量和加载因子的含义见后文。

HashSet 实现了 Set 接口，是最常用的 Set 接口实现类。因此，HashSet 也不允许保存相同值的元素。具体可参考例 5.8。

【例 5.8】 尝试向 Set 接口中添加重复的 Integer 值。

```
1. import java.util.*;
2. public class HashSetTest {
3.     public static void main(String[] args) {
4.         Set<Integer> set = new HashSet<Integer>();  //保存整数的 HashSet
5.         for(int i = 0; i<100; i++) {
6.             Integer temp = new Integer(i%6);
```

```
            set.add(temp);
        }
        for(Iterator<Integer> it = set.iterator(); it.hasNext(); ) {
            System.out.print(it.next() + ",");
        }
    }
}
```

输出结果：

```
0,1,2,3,4,5,
```

程序说明：该例向 HashSet 中插入 0～100 中所有除以 6 的余数，最后只插入了 6 个数字——0,1,2,3,4,5。其他相同的数字因为 HashSet 中已存在而无法插入。无法插入时不会报错，但此时 add()方法返回值为 false。

在 Java 中，HashSet 使用“链表数组”实现，如图 5-4 所示。

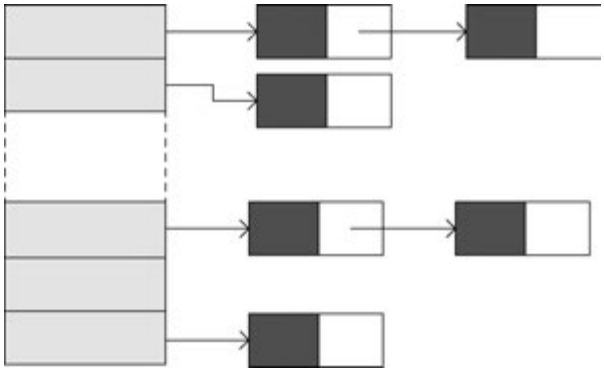

图 5-4　HashSet 中的链表数组

下面通过向 HashSet 中添加元素的过程来介绍其实现机制。当向 HashSet 中添加元素 a 时，首先调用 a.hashCode()方法获取其哈希码，据此计算出元素在底层数组中的位置，即确定其应插入哪一个链表。定位到具体链表后，需调用 a.equals()方法逐一比较该链表中的每个元素。若存在相同的元素，则 a 被视为已添加过，不再重复加入链表；否则，将 a 加入链表头部。

理解了 HashSet 的添加元素机制后，查找原理与之相同。直接根据元素的 hashCode()和 HashSet 链表数组的大小进行取模计算，即可得出元素在数组中的位置，随后只需检查该位置对应的链表中是否存在目标数据即可。查找的代价主要在于链表遍历环节，但由于每条链表实际存储的数据量极少，甚至可能为空链表，此时遍历的代价微乎其微。因此，哈希算法高效查找的基础在于确保每条链表中的数据量尽可能少。

不同的哈希码计算出相同的哈希值是很可能的，这在哈希算法中称为“冲突”。例如，某个数据的哈希码为 69822，而当前 HashSet 链表数组大小为 128，那么该数据将可能被插入数组的第 62 个链表中(69822%128=62)，即所有哈希码符合 128 * n+62 的元素都会存入同一链表。设计哈希算法时应着力减少冲突发生。

Java 中，如 HashSet 和 HashMap 这样的集合默认初始容量为 16，这是因为 16 是 2 的 4 次幂，有助于提高查询效率。即默认数组大小为 16，包含 16 条链表，且所有容量均为 2 的幂。当 16 条链表中 75%存有数据时，即达到默认加载因子 0.75 的阈值。此时 HashSet 将重建数组，即完全抛弃原有结构，新建大小为 32(2 的 5 次幂)的链表数组，并重新计算所有数据的存

储位置。这一过程持续迭代。合理设计容量与加载因子对哈希集的效率影响显著。

在了解了 HashSet 的机制之后，下面来看例 5.9。

【例 5.9】 往 HashSet 中添加自定义类的对象元素。

```
 1. import java.util.Iterator;
 2. import java.util.Set;
 3. import java.util.HashSet;
 4. class Employee {
 5.     int id;
 6.     public Employee(int id) {
 7.         this.id = id;
 8.     }
 9. }
10. public class HashSetNoHashCode {
11.     public static void main(String[] args) {
12.         Set<Employee> set = new HashSet<Employee>();
13.         set.add(new Employee(12));
14.         set.add(new Employee(1));
15.         set.add(new Employee(1));
16.         for(Iterator<Employee> it = set.iterator(); it.hasNext();) {
17.             Employee temp = it.next();
18.             System.out.println("id = " + temp.id + ",hashcode = " + temp.hashCode());
19.         }
20.     }
21. }
```

输出结果：

```
id = 12,hashcode = 33263331
id = 1,hashcode = 6413875
id = 1,hashcode = 21174459
```

程序说明：第 4～9 行代码定义了员工(Employee)类，其中仅包含员工号(id)一个成员变量。在一般公司中，员工号相同的员工被视为同一员工。第 12 行声明了一个员工类的 HashSet。第 13～15 行代码分别向该员工集合中插入了 id 为 12、1、1 的员工对象。第 16～18 行循环显示员工集合中元素的 id 号和哈希码。结果显示全部插入成功，这与 Set 接口规定不能插入同值元素相悖。原因何在？

在 Object 类中，hashCode()方法用于获取哈希码。默认情况下，每个对象拥有一个基于其内存地址的哈希码。因此，尽管后两个员工对象的属性(id)值相同，但它们是两个独立对象，存储位置不同，故拥有不同的哈希码。在它们被加入 HashSet 时，可能会根据哈希码被分配到不同的链表，因此被视为独立的元素。即使因哈希冲突进入同一链表，Java 默认的 equals()方法比较对象地址，结果返回 false，同样会被视为不同元素而成功添加。

那么，为何例 5.8 中的 Integer 类 HashSet 能区分相同值？理论上，即使两个 Integer 类保存相同整数值，也应视为不同对象。原因在于 Java 中封装类(如 Integer、Double)和 String 类重写了 hashCode()与 equals()方法。这些方法的返回值取决于对象的内容。例如，两个字符串字面值相同的 String 对象，其 hashCode()方法返回值必然相等，equals()方法结果必为 true，是因为 String 的 hashCode()方法由其内容计算得出，equals()方法也进行内容比较。

综上分析，添加到 HashSet 的对象所属类需重写 equals()与 hashCode()方法。重写 hashCode()方法旨在确保哈希查找的高效性，重写 equals()方法则保证相同元素无法重复插入。以下示例将改写例 5.9 中的 Employee 类，其余部分保持不变。

【例 5.10】 改写例 5.9，重写 hashCode()与 equals()方法。

```
 1. import java.util.Iterator;
 2. import java.util.Set;
 3. import java.util.HashSet;
 4. class Employee {
 5.     int id;
 6.     public Employee(int id) {
 7.         this.id = id;
 8.     }
 9.     public int hashCode() {
10.         return this.id;
11.     }
12.     public boolean equals(Object o) {
13.         if (this == o) return true;
14.         if (o == null || getClass() != o.getClass()) return false;
15.         Employee e = (Employee) o;
16.         return this.id == e.id;
17.     }
18. }
19. public class HashSetNoHashCode {
20.     public static void main(String[] args) {
21.         Set<Employee> set = new HashSet<Employee>();
22.         set.add(new Employee(12));
23.         set.add(new Employee(1));
24.         set.add(new Employee(1)); //因为重写了 hashCode()和 equals()方法，此对象无法加入
                                      //Set 接口
25.         for (Iterator<Employee> it = set.iterator(); it.hasNext(); ) {
26.             Employee temp = it.next();
27.             System.out.println("id=" + temp.id + ", hashcode=" + temp.hashCode());
28.         }
29.     }
30. }
31.
```

输出结果：

```
id=1, hashcode=1
id=12, hashcode=12
```

程序说明：由输出结果可以看出，重写 equals()与 hashCode()方法后的 Set 接口可以按照预期将 id 相同的 Employee 归为同一元素，从而不进行添加。

在重写这两个方法时需要注意以下问题。首先，一个合格的 equals()方法必须满足以下条件。

(1) 自反性：对任意 x，x.equals(x)总返回 true。

(2) 对称性：对任意 x 与 y，x.equals(y)总返回 true 当且仅当 y.equals(x)返回 true。

(3) 传递性：对任意 x，y，z，如果 x.equals(y)返回 true 而且 y.equals(z)返回 true，那么 x.equals(z)总返回 true。

(4) 持续性：对任意 x 与 y，多次调用 x.equals(y)总返回相同结果，不会改变。

(5) 对任意非空 x，x.equals(null)总返回 false。

同时，当重写 equals()方法时，还必须确保：如果 a.equals(b)返回 true，那么 a.hashCode()=b.hashCode()。这包含了以下两层含义。

(1) 内容相等的对象，它们的哈希码也一定相等。

(2) 两个对象的哈希码不相等，则两个对象一定不相等。

一般的解决方法是，在 equals()方法中进行比较的属性（如 id），一定也要在 hashCode()方法中进行计算，例 5.10 就符合这个原则。

2. TreeSet（树集）

TreeSet 是基于“红黑树”（一种自平衡二叉搜索树）数据结构实现的顺序集合。在红黑树中，每个节点的值均大于或等于其左子树中所有节点的值，且小于或等于其右子树中所有节点的值，这种结构确保了运行时能够高效地查找和定位目标节点。因此，TreeSet 可从一个 Set 集合生成有序集合。

TreeSet 在添加和取出元素时的性能通常低于 HashSet；因为 TreeSet 在执行查询、添加或删除操作时，需要通过比较确定元素的位置，速度较慢。然而，相较于 HashSet，TreeSet 的核心优势在于其所有元素始终按照指定排序规则保持有序状态。

由于涉及排序，TreeSet 要求元素必须是“可比较的”。这意味着元素所属类需实现 Comparable 接口，或在创建 TreeSet 时为其提供一个 Comparator 比较器。此外，作为 Set 接口的实现，TreeSet 同样要求元素类重写 equals()方法以保证元素唯一性。此时，必须确保当 equals()方法返回 true 时，compareTo()或 compare()方法恰好返回 0，两者通常应保持逻辑一致性。

3. LinkedHashSet（链式哈希集）

LinkedHashSet 是 HashSet 的子类，该类在 HashSet 的基础上通过链表将每个元素关联起来，支持按元素插入顺序进行遍历。LinkedHashSet 能够确保元素顺序的确定性，对于既要求具有常量时间复杂度的存取性能，又需要保持元素顺序的场景尤为适用。

通过一个例子，来了解一些不同的 Set 接口的实现类的特性。

【例 5.11】 HashSet、TreeSet、LinkedHashSet 的比较。

```
 1. import java.util.*;
 2. public class OtherSet {
 3.     public static void main(String args[]) {
 4.         Set<Integer> hashSet = new HashSet<>();
 5.         Set<Integer> linkedHashSet = new LinkedHashSet<>();
 6.         Set<Integer> treeSet = new TreeSet<>();
 7.         System.out.println("--- 添加顺序 ---");
 8.         for(int i=0; i<5; i++){
 9.             int s = (int)(Math.random()*100);
10.             System.out.println("添加: " + s);
11.             hashSet.add(s);
12.             linkedHashSet.add(s);
13.             treeSet.add(s);
14.         }
15.         System.out.println("\n--- 最终存储顺序 ---");
16.         System.out.println("HashSet: " + hashSet);
17.         System.out.println("LinkedHashSet: " + linkedHashSet);
18.         System.out.println("TreeSet: " + treeSet);
19.     }
20. }
```

输出结果：

```
第 0 次随机数产生为: 46
第 1 次随机数产生为: 0
第 2 次随机数产生为: 27
第 3 次随机数产生为: 70
第 4 次随机数产生为: 61
HashSet: [0, 70, 27, 46, 61]
```

```
LinkedHashSet: [46, 0, 27, 70, 61]
TreeSet: [0, 27, 46, 61, 70]
```

程序说明：第 9 行代码生成一个 0～100 的随机整数 s，并将其包装后存入三种 Set 中。输出结果显示：HashSet 的元素存储顺序与添加顺序无关；LinkedHashSet 严格保持元素的添加顺序；TreeSet 则根据元素值大小进行排序存储。由于 Integer 已实现 Comparable 接口，可直接添加到 TreeSet。若添加自定义类对象，需手动实现 Comparable 接口或提供 Comparator 比较器。

通常，TreeSet 在需要有序提取集合元素时体现优势。为保障排序功能，添加至 TreeSet 的元素必须可比较，故要求元素类实现 Comparable 接口或提供 Comparator 比较器。一般情况下，先将元素批量添加至 HashSet，再转换为 TreeSet 进行有序遍历，效率更高。

▶ 5.2.5　Iterator

在介绍 Set 实现类之前，先来讨论一下遍历的问题。在处理数据集时必然涉及集合的遍历问题，例如 List 的遍历方法通常写成：

```
for(int i = 0; i < list.size(); i++) {
    ...
    list.get(i);   //取出元素
    ...
}
```

显然，这里是使用索引 i 以及 get()方法从 List 中提取元素。但是 Set 的元素是没有顺序的，因此也就没有索引这种属性，上面这种遍历方法是无效的。Java 提供了 Iterator(迭代器)来实现集合的遍历。Iterator 的语法如下：

```
public interface Iterator<E>   //E 是 Iterator 遍历的数据类型
```

Iterator 是一种设计模式，用于将集合排列成一个序列，遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。Iterator 通常被看作“轻量级”对象，也就是说，创建它只需付出极少的代价。但也正是由于这个原因，Iterator 存在一些似乎很奇怪的限制。例如，基本 Iterator 只能单向移动。

每一种集合返回的 Iterator 具体类型可能不同，Array 可能返回 ArrayIterator，Set 可能返回 SetIterator，Tree 可能返回 TreeIterator，但是它们都实现了 Iterator 接口，因此，程序不关心到底是哪种 Iterator，它只需要实现这个 Iterator 接口即可，这就是面向对象的接口与实现分离的特点。

Java 中的 Iterator 功能比较简单，并且只能单向移动。Iterator 的常用方法如表 5-7 所示。

表 5-7　Iterator 的常用方法

方　法	描　述
iterator()	要求集合返回一个 Iterator 类型对象
next()	第一次调用时，返回序列的第一个元素。获得序列中的下一个元素，每成功调用一次 Iterator 向后移动一个元素
hasNext()	用于检查序列中是否还有元素
remove()	将 Iterator 新返回的元素删除

1. Iterator 遍历

Iterator 模式总是用同一种逻辑来遍历集合，所有的 Collection 都可以生成自己类型的

Iterator。使用 Iterator 遍历一个 List 的方式如下：

```
List<String> a = new ArrayList<String>();
// … 向列表 a 中添加元素 …
for(Iterator<String> it = a.iterator(); it.hasNext(); ) {
    // …
    String s = it.next(); //取出元素
    // …
}
```

观察上述代码，迭代器操作过程可归纳为：通过 iterator()方法获取指向集合起始位置的 Iterator；以 hasNext()方法作为循环条件，直至所有元素遍历完毕；每次循环中，通过 next()方法获取每个元素。

使用 Iterator 遍历集合的有一个好处，如果希望把上述代码中的 ArrayList 更换为 LinkedList，或者 HashSet，或者其他实现了 Iterable 接口的类，只需改变集合类型声明语句即可（第 1 行代码），而后面的遍历代码无须做任何修改。例如，如果觉得 Set 结构更适合当前的程序，重新定义上例中的 a。

```
Set<String> a = new HashSet<String>();
// … 向集合 a 中添加元素 …
for(Iterator<String> it = a.iterator(); it.hasNext(); ) { //遍历代码不需要改变
    // …
    String s = it.next();                                  //取出元素
    // …
}
```

Iterator 将存取逻辑从不同类型的集合中抽离出来，从而避免向客户端暴露集合的内部实现，有效提升代码复用性。客户端无须直接操作集合类，只需通过 Iterator 发出“向后”或“取当前元素”指令，即可间接遍历整个集合。因此在涉及遍历操作的编码场景中，使用 Iterator 能显著增强代码的健壮性与复用性。

Iterator 是 Java 最基础的迭代器接口，而为列表设计的 ListIterator 则功能更为丰富。ListIterator 继承自 Iterator 接口，支持双向遍历列表，并具备元素插入与删除能力。ListIterator 的新增方法如表 5-8 所示。

表 5-8　ListIterator 的新增方法

方　法	描　述
add()	在即将遍历元素反方向位置插入元素
remove()	删除上一次遍历的元素
hasPrevious()	当前迭代器位置前面是否有元素
previous()	获得前驱元素

ListIterator 通常仅适用于 LinkedList 等链表结构，因为 LinkedList 基于双向链表实现，其前驱和后继节点的顺序访问操作已经实现。而 ArrayList 自身具备随机存取能力，可通过索引直接访问元素，因此通常无须使用此 Iterator。

2. for-each 遍历

了解 Iterator 的诸多优势后，使用 Iterator 进行遍历仍略显烦琐。Java 5 提供了一种更简洁的语法结构用于遍历数组和 Collection，其基本语法如下：

```
for(变量类型 变量名:集合){…}
```

下面使用 for-each 遍历一个 Set 接口，代码如下：

```
Set<String> set = new HashSet<String>();
// ... 向集合 set 中添加元素 ...
for(String s : set) {
    System.out.println(s);
}
```

for-each 循环本质上是将迭代器封装为更简洁的语法结构。因此，for-each 仅能遍历数组，以及实现了 java.lang.Iterable 接口的类的实例。

Iterable 接口仅声明了一个 iterator()方法，该方法返回一个实现了 java.util.Iterator 接口的对象。因此，List 与 Set 的实现类均可通过 for-each 进行遍历。

5.3 Map 接口

Map 接口实现了映射功能，是维护键值对关联关系的无序集合。键与值均为对象。同一个 Map 接口中不允许存在重复键，且每个键仅能映射一个值。这与 List 结构高度相似：List 中每个元素拥有唯一索引(键)，每个索引对应特定元素(值)。键值对数据结构在现实中普遍存在，例如“身份证号—居民”“员工号—员工”“MAC 地址—网卡”“邮编号—地区”等。因此，Map 接口的设计具有实际意义。

虽然 Map 接口的大部分方法和 Collection 接口看起来十分相似，但 Map 接口没有继承 Collection 接口。Map 接口的声明如下：

```
public interface Map<K,V>
```

很明显，Map 接口同样使用了泛型，其中 K 代表键的类型，V 代表值的类型，键和值均为对象。例如，构建一个键为 Integer 类型、值为 String 类型的 Map 接口：

```
Map<Integer,String> map = new HashMap<Integer,String>();
```

Map 接口的常用方法如表 5-9 所示。

表 5-9 Map 接口的常用方法

方 法	描 述
put(K,V)	将指定值与指定键相关联，如果此键已存在，则将值替换
putAll(Map)	将指定 Map 接口中的所有键值对复制到当前 Map 接口中
clear()	删除 Map 接口中的所有键值对
remove(Object)	从 Map 接口中删除指定键值对
get(Object)	通过键查找值，如果找不到，则返回 null
size()	返回键值对个数
isEmpty()	判断 Map 接口是否为空
keySet()	返回当前 Map 接口中包含的键的 Set 视图
values()	返回此映射中包含的值的 Collection 视图
entrySet()	返回此映射中包含的键值对的 Set 视图
containsKey(Object)	判断 Map 接口中是否存在指定键
containsValue(Object)	判断 Map 接口中是否存在指定值

下面通过一个程序来展示 Map 接口的使用，这里使用的 HashMap 类是 Map 接口的实现类，相关内容将会在后文讲解。

```
1. Map<String, String> map = new HashMap<>();
2. map.put("dog", "yellow");
```

```
3. map.put("dog", "red");
4. map.put("cat", "red");
5. System.out.println(map);   //此处自动调用 map.toString()方法
6. map.remove("cat");
7. System.out.println(map);
```

输出结果：

```
{cat = red, dog = red}
{dog = red}
```

程序说明：由于在 Map 接口中键是唯一的，当对同一键 dog 调用两次 put()方法，如第 2 行和第 3 行代码，那么第二次调用的值 red 就会替换原来的旧值 yellow，此时 put()方法返回旧值。可以看到键 dog 对应的值为新值 red 。

▶ 5.3.1 Map 接口遍历

Map 接口没有实现 Iterable 接口，因此就无法直接使用迭代器；同时 Map 接口也没有像 List 接口那样的位置概念，因而也无法使用索引。那 Map 接口靠什么来遍历呢？Map 接口提供三种集合的视图，Map 接口的内容可以被当作一组 Key 集合、一组 Value 集合，或者一组 Key-Value 映射集合，分别对应表 5-9 的 keySet()、values()、entrySet()方法的返回值。Map 接口正是靠这三个方法来实现不同视图的遍历。

假设有一个某类型的映射集 map，存有若干键值对，有以下三种遍历方法。

（1）遍历所有的值（Value）。

```
public static void byValue(Map<String, Student> map) {
    Collection<Student> c = map.values();
    for (Iterator<Student> it = c.iterator(); it.hasNext();) {
        Student s = it.next();     //只能访问 Student 对象
    }
}
```

这是最常规的一种遍历方法，使用 Map 接口的 values()方法可以产生值的集合，利用 Collection 的迭代器方法进行遍历，不过这种方法只能遍历值。

（2）遍历所有的键（Key）。

```
public static void byKey (Map<String, Student> map) {
    Set<String> keySet = map.keySet();
    for (Iterator<String> it = keySet.iterator(); it.hasNext();) {
        String s = it.next();
        Student value = map.get(s);     //通过键获取值
    }
}
```

这种方法利用 keySet 进行遍历，它的优点在于可以根据指定的 key 值得到所对应的 value 值，灵活性比第一种方法更好。

（3）遍历所有的键值对（Entry）。

```
public static void byEntry(Map<String, Student> map) {
    Set<Map.Entry<String, Student>> entrySet = map.entrySet();
    for (Iterator<Map.Entry<String, Student>> it = entrySet.iterator(); it.hasNext();) {
        Map.Entry<String, Student> entry = it.next();
        String key = entry.getKey();
        Student value = entry.getValue();
        System.out.println(key + "--->" + value);
    }
```

这种方法比较复杂，但更优。这里使用了 Map.Entry 接口。Map.Entry 是 Map 接口内部定义的一个接口，专门用来保存(K,V)(键值对)的内容。其定义为：

```
public static interface Map.Entry<K,V>
```

Map.Entry 实际上就是把 Map 接口中的每一个键值对整体看成一个对象，那么映射就成为 Map.Entry 类型数据的一个集合，Map.Entry 接口提供了获取键、值还有设置值等方法，此接口定义的方法如表 5-10 所示。

表 5-10 Map.Entry 接口定义的方法

方 法	描 述
getKey()	返回当前键值对的键
getValue()	返回当前键值对的值
setValue(V)	用指定的值替换当前键值对的值

▶ 5.3.2 HashMap

Map 接口必须保持键的唯一性，这与 Set 接口要求元素值唯一的特性一致。实际上，HashSet 的底层实现完全依赖于 HashMap(哈希映射)。HashSet 内部维护着一个 HashMap 实例，其键即为待存储的对象，值则是一个固定的常量 PRESENT。这种设计确保仅需存储键的信息，而 Map 接口中键的不可重复性，则直接保证了 Set 接口内所有元素的唯一性。

HashSet 的所有方法都通过调用其内部 HashMap 的对应方法实现。以 add()方法为例，判断元素是否添加成功，取决于向 Map 接口中存入键值对时是否已存在该键。若返回值为常量 PRESENT(即旧值)，说明键已存在，添加失败；若返回值为 null，则表明首次插入该键，添加成功。

由此可见，HashSet 与 HashMap 本质上是同一数据结构的两种表现形式——HashSet 仅操作 HashMap 的键，而 HashMap 额外提供了对值的操作功能。理解这一点后，后续内容与其类似。

HashMap 的声明如下：

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable,
Serializable
```

HashMap 的构造方法如表 5-11 所示。

表 5-11 HashMap 的构造方法

方 法	描 述
HashMap()	构造空的 HashMap，容量为 16，加载因子为 0.75
HashMap(Collection)	从其他集合构造 HashMap
HashMap(int)	指定初始容量、加载因子为 0.75 的 HashMap
HashMap(int,float)	指定初始容量与加载因子的 HashMap

HashMap 底层与 HashSet 同样，也是一个数组结构，其中每个数组元素对应一条链表，如图 5-4 所示。当向 HashMap 中添加键值对(K,V)时，首先会调用 k.hashCode()方法获取其哈希码，据此计算出该键值对在数组中的位置，即确定其所属的链表。定位到具体链表后，系统将调用 k.equals()方法，将该键与链表中每个元素的键逐一比较。若发现存在相同键的元素，则视为该键已存在，此时会将对应值更新为 V；否则，该键值对会被添加至链表头部。HashMap 的 put()方法完整呈现了这一过程。

【例 5.12】 HashMap 的 put()方法。

```
1. public V put(K key, V value) {
2.     if (key == null)
3.         return putForNullKey(value);
4.     int hash = hash(key.hashCode());
5.     int i = indexFor(hash, table.length);
6.     for (Entry<K,V> e = table[i]; e != null; e = e.next) {
7.         Object k;
8.         if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
9.             V oldValue = e.value;
10.            e.value = value;
11.            e.recordAccess(this);
12.            return oldValue;
13.        }
14.    }
15.    modCount++;
16.    addEntry(hash, key, value, i);
17.    return null;
18. }
```

程序说明：HashMap 将键值对视为一个整体进行处理，即一个 Entry 对象。它采用一个 Entry[]数组来保存所有键值对。存储 Entry 对象时，HashMap 会根据哈希算法确定其在数组中的位置，再通过 equals()方法决定其在对应链表上的具体位置；取出 Entry 对象时，同样先依据哈希算法定位到数组中的位置，再通过 equals()方法从该位置的链表中获取目标 Entry。

与 HashSet 类似，HashMap 键对象所属的类也必须实现 equals()和 hashCode()方法。实现 hashCode()方法是为了确保高效地查找，而实现 equals()方法则是为了保证拥有相同键的元素无法插入 HashMap。

【例 5.13】 未重写 hashCode()与 equals()方法的示例。

```
1. import java.util.Map;
2. import java.util.HashMap;
3. class MyKey {
4.     int id;
5.     public MyKey(int i) {
6.         id = i;
7.     }
8.     public String toString() {
9.         return "id" + Integer.toString(id);
10.    }
11. }
12. class Counter {
13.    int i = 1;
14.    public String toString() {
15.        return Integer.toString(i);
16.    }
17. }
18. public class MapTest {
19.    public static void main(String[] args) {
20.        Map<MyKey, Counter> map = new HashMap<MyKey, Counter>();
21.        for (int i = 0; i < 10; i++) {
22.            int s = i % 5;
23.            MyKey r = new MyKey(s);
24.            if (map.containsKey(r))
25.               map.get(r).i++;
26.            else
```

```
27.                 map.put(r, new Counter());
28.         }
29.         System.out.println(map);
30.     }
31. }
```

输出结果：

```
{id3 = 1, id2 = 1, id1 = 1, id1 = 1, id3 = 1, id2 = 1, id0 = 1, id0 = 1, id4 = 1, id4 = 1}
```

程序说明：该示例本应计算相同 id 的 MyKey 个数，但从输出结果可见，相同 id 的 MyKey 类并未被归为一类。这与 HashSet 的问题一致：键类(MyKey)未重写 equals()与 hashCode()方法。下面将 MyKey 类稍作修改，重写 hashCode()与 equals()方法后即可看到预期效果。

【例 5.14】 改写例 5.13，重写 hashCode()与 equals()方法。

```
1. class MyKey {
2.     int id;
3.     public MyKey(int i) { id = i; }
4.     public int hashCode() {
5.         return id;
6.     }
7.     public boolean equals(Object o) {
8.         return (o instanceof MyKey) && (id == ((MyKey)o).id);
9.     }
10.    public String toString() {
11.        return "id" + Integer.toString(id);
12.    }
13. }
```

输出结果：

```
{id0 = 2, id1 = 2, id2 = 2, id3 = 2, id4 = 2}
```

程序说明：现在已统计出相同 id 的 MyKey 数量。另外注意，MyKey 类重写了 toString()方法，原因在于 Map 接口的 toString()方法会默认调用每个键值对的键对象与值对象的 toString()方法。

▶ 5.3.3 其他 Map

1. TreeMap(树映射)

和 HashSet 与 HashMap 的关系类似，TreeSet 也是利用 TreeMap 实现的。TreeMap 的特性和 TreeSet 基本相同，也是基于“红黑树”数据结构的实现。查看“键”或“键值对”时，它们会被排序(其次序由 Comparable 或 Comparator 决定)。TreeMap 的特点在于，得到的结果是经过排序的。TreeMap 是唯一带有 subMap()方法的 Map，它可以返回一个子树。

2. LinkedHashMap(链式哈希映射)

作为 HashMap 的有序子类，它在哈希表结构之外额外维护了一条双向链表，用于追踪键值对的逻辑顺序。默认情况下，迭代操作严格遵循元素的插入顺序返回；若构造时启用 accessOrder=true，则会切换为最近最少使用(LRU)顺序。LinkedHashMap 的增、删、改、查操作开销略高于 HashMap，但差异通常可忽略不计。在遍历性能上，LinkedHashMap 仅需线性扫描链表，耗时与实际元素数量成正比；而 HashMap 需遍历整个数组及其中的链表，耗时取决于数组容量与元素数量的总和。因此，当 HashMap 容量远大于实际数据量时，

LinkedHashMap 的迭代效率优势尤为显著。

3. WeakHashMap（弱键映射）

WeakHashMap 是一种“弱键”映射。其键不再被外部强引用时，即可被垃圾回收器回收，映射本身也随之自动移除该项，专为需防止内存泄漏的特定场景而设计。

4. IdentifyHashMap

IdentifyHashMap 使用==代替 equals()方法对“键”进行比较的 HashMap，专为解决特殊问题而设计。

5. HashTable（哈希表）

HashTable 继承自 Dictionary 类，与 HashMap 功能相似，但主要区别在于禁止键（Key）或值（Value）为 null。该结构支持线程同步机制，任一时刻只允许单线程执行写入操作，因此其写入性能较低。

通常情况下，HashMap 使用最为广泛。HashMap 存储的键值对在取出时顺序是随机的，它依据键的 HashCode 存储数据，并能根据键直接获取其值，访问速度极快。在 Map 中进行插入、删除和定位元素操作时，HashMap 是最佳选择。TreeMap 则能取出排序后的键值对。如需按自然顺序或自定义顺序遍历键，TreeMap 的表现优于 HashMap。LinkedHashMap 是 HashMap 的子类。若需要元素保持插入顺序，同时具备较快的查找速度，LinkedHashMap 能够满足需求。它适用于一些特殊场景，例如构建“连接池”时要求资源有序排列，同时需快速查询特定资源的可用性，此时 LinkedHashMap 结构较为合适。

简言之，HashMap、TreeMap 和 LinkedHashMap 的区别，类似于 HashSet、TreeSet 和 LinkedHashSet 之间的差异，只不过在 Set 中顺序针对元素本身，而在 Map 中顺序则针对元素的键。

5.4 集合类型的选择

图 5-1 所示的集合框架是不完整的，为了清晰地表述接口及其实现类的关系而省略了一些抽象类。完整的容器类继承关系如图 5-5 所示。

图 5-5 展示了三个主要集合接口：List、Set 和 Map，每个接口均有多种实现类。通常，有多种集合类均可解决相同问题，因此需要考量选用哪种集合能使程序更高效。

1. List 的选择

在 ArrayList 中进行随机访问（即调用 get()和 set()方法）是高效的，但同样的操作对于 LinkedList 却会带来显著的开销，这是由于 LinkedList 仅支持顺序存取，其按索引存取实际上是通过多次顺序遍历实现的。另外，在列表中部进行插入和删除操作时，LinkedList 的效率远高于 ArrayList。通常最佳做法是优先选择 ArrayList 作为默认 List 实现，若后续因频繁插入删除导致性能下降，再考虑切换至 LinkedList。

2. Set 和 Map 的选择

HashSet 在增、删、改、查等数据处理上具有显著优势，其性能几乎不受元素数量影响，因此在多数场景中应优先考虑 HashSet。然而 TreeSet 能维护元素有序排列，在需要顺序访问时依然适用。得益于平衡二叉树结构，其查找速度约为 $O(\lg n)$ 级别——虽优于 List 但仍不及 HashSet。若需使用 TreeSet，建议先构造 HashSet 再转换为 TreeSet 以提升效率。LinkedHashSet 则适用于同时要求高效存取和有序数据的场景。

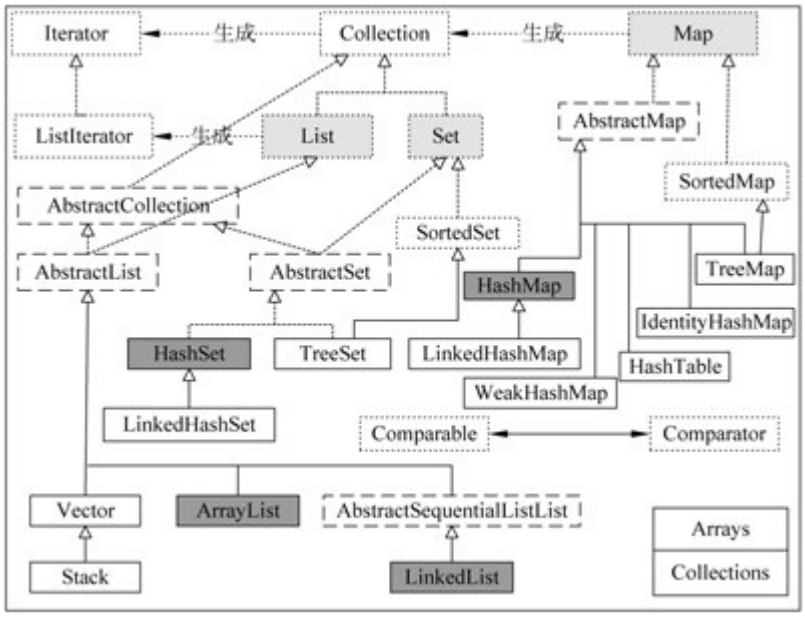

图 5-5　集合框架的继承关系

TreeSet 的元素添加速度优于 List，但存取效率并不突出。可将其视为创建有序列表的途径：红黑树特性天然保证元素有序排列，无须额外排序。构建 TreeSet 后，通过 toArray()方法即可生成有序数组，再配合 Arrays. binarySearch()方法实现快速检索。需要强调的是，此方案仅适用于 HashSet 无法满足需求的场景，毕竟 HashSet 始终保持着最快的操作速度。

Map 的选择逻辑与 Set 完全一致，不再赘述。

5.5　Collections 工具类

如同 Java 为数组提供了 Arrays 工具类，它也为集合类(Collection)提供了功能强大的 Collections 工具类。java. util. Collections 类囊括大量实用方法，能显著简化集合框架相关代码的开发。Javadoc 对该类给出了完整的定义："此类完全由在 Collection 上进行操作或返回 Collection 的静态方法组成。"

Collections 提供了多种操作方法，以下选取一些常用方法，结合示例进行介绍。既然针对 Collection 进行操作，以下便以 ArrayList 类为例，其他 Collection 实现类同理。

【例 5.15】　创建并初始化两个 List。

```
1. double array[] = {12, 1, 1000, 34.7, 12};
2. double arr2[] = {123, 321};
3. List<Double> list = new ArrayList<>();
4. List<Double> list2 = new ArrayList<>();
5. for (double value : array) {
6.     list.add(value);              //Autoboxing: 自动将 double 转换为 Double
7. }
8. for (double value : arr2) {
9.     list2.add(value);
10. }
```

程序说明：下文代码中的 list、list2 专指本例所定义的变量。

1. sort()方法：排序

使用sort()方法可根据元素的自然顺序对指定List进行升序排序。列表中的所有元素都必须实现Comparable接口，也可通过Comparator实现定制排序。例如：

```
1. Collections.sort(list);
2. for (Iterator<Double> it = list.iterator(); it.hasNext();) {
3.     System.out.print(it.next() + " , ");
4. }
```

输出结果：

```
1.0 , 12.0 , 12.0 , 34.7 , 1000.0 ,
```

Collections中还提供了一个与sort()方法相对应的混排方法——shuffle()。该方法的作用正好与sort()方法相反，它会彻底打乱List中可能存在的任何排序痕迹，即基于随机源对List元素进行重新排列，使每个元素出现在任意位置的概率均等。该方法类似扑克牌中的“洗牌”操作，其调用方式与sort()方法类似。

2. binarySearch()方法：二分法查找

采用二分查找算法可在有序List中快速定位指定元素并返回其索引位置。如前所述，List的线性查找效率较低，而binarySearch()方法能显著提升查找速度。但必须确保List处于有序状态，否则将导致不可预测的结果。例如：

```
1. Collections.sort(list);
2. int i = Collections.binarySearch(list, new Double(34.7));
3. int j = Collections.binarySearch(list, new Double(500));
4. System.out.println("34.7 位置: " + i);
5. System.out.println("500 位置: " + j);
```

程序说明：因List中包含元素34.7，而不包含元素500。第1行代码用sort()方法排好序，第4行显示找到元素34.7，返回其位置（从0开始计位）；第5行因没找到500，所以将显示负数。

输出结果：

```
34.7 位置: 3
500 位置: -5
```

3. reverse()方法：反转

使用reverse()方法可以将List中的元素按逆序排列。例如：

```
1. Collections.reverse(list);
2. for (Iterator<Double> it = list.iterator(); it.hasNext();) {
3.     System.out.print(it.next() + " , ");
4. }
```

输出结果：

```
12.0 , 34.7 , 1000.0 , 1.0 , 12.0 ,
```

4. fill()方法：填充

使用fill()方法可将指定元素填充到List中替换所有现有元素。例如：

```
1. Collections.fill(list, new Double(1234));
2. for (Iterator<Double> it = list.iterator(); it.hasNext();) {
3.     System.out.print(it.next() + " , ");
4. }
```

输出结果：

```
1234.0 , 1234.0 , 1234.0 , 1234.0 , 1234.0 ,
```

5. copy()方法：复制

copy()方法接收两个参数：目标 List 与源 List。该方法将源 List 的元素复制到目标 List 中，完全覆盖目标 List 原有内容。目标 List 的长度必须不小于源 List。若目标 List 长度超出源 List，其超出部分的元素将保持不变。例如：

```
1. Collections.copy(list, list2);
2. for (Iterator<Double> it = list.iterator(); it.hasNext();) {
3.     System.out.print(it.next() + " , ");
4. }
```

输出结果：

```
123.0 , 321.0 , 1000.0 , 34.7 , 12.0 ,
```

6. min()、max()方法：返回最值元素

min()和 max()方法根据指定比较器排序后的顺序，分别返回给定 Collection 中的最小元素与最大元素。这些方法要求 Collection 中的所有元素必须能相互比较。例如：

```
1. System.out.println("最小值: " + Collections.min(list));
2. System.out.println("最大值: " + Collections.max(list));
```

输出结果：

```
最小值: 1.0
最大值: 1000.0
```

Collections 的功能远不止于此，它还提供子列表操作、元素移动、集合同步等强大功能，详细信息可参考 Java API 文档。

小结

本章详细阐述了 Java 集合框架的核心知识体系，涵盖 Collection 与 Map 接口及其核心实现类（如 ArrayList、HashSet、HashMap 等），并深入探讨了泛型、迭代器以及工具类的应用策略，助力开发者依据具体场景甄选最适宜的数据结构。

Java 集合框架由两大族系构成——Collection 和 Map。Collection 描述元素集合，包含子接口 List 和 Set；Map 则描述键值对映射关系。

Collection 接口定义了基础操作方法，如 add()、remove()、contains()等，这些方法普遍适用于所有集合类型。List 接口的主要实现类包括 ArrayList 和 LinkedList。ArrayList 采用动态数组实现，随机访问性能优异，但在集合中部执行插入或删除操作时效率较低。LinkedList 基于双向链表实现，特别适合频繁的插入和删除操作，但其随机访问性能则相对逊色。Set 的实现类包括 HashSet 和 TreeSet。HashSet 依托哈希表实现，严格禁止元素重复，其性能几乎不受元素数量增长的影响。TreeSet 则基于红黑树实现，能够自动维护元素的有序性，查找速度达到 $O(\lg n)$。

Map 接口通过键值对形式存储数据，要求键唯一，而值允许重复。常用方法涵盖 put()、get()、keySet()等。Map 的实现类主要有 HashMap 和 TreeMap。HashMap 同样基于哈希表实现，允许键和值为 null，具备出色的存取速度。TreeMap 基于红黑树实现，能够按键自然顺序或指定比较器排序存储数据，非常适用于需要有序访问键值对的场景。LinkedHashMap 在

HashMap 的基础上额外维护了元素的插入顺序，完美契合那些既需保持元素次序又要求高效存取的场景。

Java 泛型机制将类型检查提前至编译期，有效规避了运行时的 ClassCastException 异常。Iterator 为遍历集合提供了统一协议，成功屏蔽了不同底层集合结构的差异。ListIterator 作为增强版迭代器，额外支持双向遍历及元素替换功能。Collections 工具类集成了诸多实用静态方法，如 sort()、binarySearch()、reverse()、fill()、copy()等。

Java 集合框架凭借其精妙的分层设计与丰富的实现类，高效满足了多样化场景下的数据处理需求。深刻理解接口语义、审慎权衡不同实现类的差异并娴熟运用泛型与工具类，乃是编写高效、健壮集合处理代码的关键所在。

习题

1. Java 数组与线性表在存储元素的数量、类型安全、动态扩容等方面有哪些主要区别？

2. 集合框架的顶层接口是什么接口？其常用的子接口有哪些？

3. 映射接口常用的实现类有哪几个？

4. 比较 List、Map、Set 三个接口，并指出 List、Map、Set 中元素的唯一标识。

5. Java 集合框架中使用泛型的主要目的是什么？它如何在编译期提升代码的类型安全性？

6. 如果需要保证元素唯一性且访问速度最快、按照插入顺序遍历元素、保持元素的自然顺序排序，分别应选择哪种集合实现类？为什么？

7. 简述 ArrayList 和 LinkedList 的不同点，在添加、删除、随机访问等操作上，它们的性能表现有何差异？应如何选择？

8. 简述迭代器的操作过程。Iterator 和 ListIterator 的功能有何不同？请分别给出使用场景和代码示例。

9. 将 1～500 的正整数存放在一个 List 中，并移除 List 中索引位置为 10 的对象。

10. 向 Set 集合以及 List 集合添加"A"、"B"、"c"、"a"、"A" 5 个元素，观察结果。

11. 试说明 Comparable 和 Comparator 在接口设计、实现方式及使用场景上的主要区别，并举例说明在 TreeSet 或 TreeMap 中的应用。

12. 使用 LinkedList 构造一个队列数据结构，需使用泛型，且需要实现下列方法。

get()：出队列，返回出队列的节点。

put()：入队列。

isEmpty()：判断队列是否为空。

实验

1. 编写程序，用 HashMap 模拟一个网上购物车。要求：从键盘输入 5 本书的名称、单价、购买数量，将这些信息存入一个 HashMap，然后将该 HashMap 作为参数调用方法 getSum(HashMap books)，该方法用于计算书的总价并返回。提示：键盘输入可以使用 Scanner 类。

2. 编写程序，设计一个系统来管理和排序学生信息。要求：

(1) 定义 Student 类。创建一个 Student 类，包含 id(学号，int)、name(姓名，String)和

score(成绩,double)三个私有属性。同时提供构造方法和 toString()方法以便打印学生信息。

(2) 实现自然排序。让 Student 类实现 Comparable < Student >接口,重写 compareTo()方法,使得学生的"自然顺序"是按照学号 id 从小到大排序。

(3) 实现自定义排序。另外创建一个 ScoreComparator 类,该类实现 Comparator < Student >接口,用于按学生成绩 score 从高到低进行排序。

3. 编写程序,实现图书借阅系统,用 ArrayList 存储图书对象,体验集合框架的基本用法。

要求:

(1) Book 类。

- 字段: String title; String author; boolean borrowed。
- 方法: borrow(); returnBook(); isBorrowed(); toString()。

(2) 集合框架: 使用 ArrayList < Book >保存所有图书。

(3) 功能菜单(循环显示):

```
  === 图书借阅 ===
1 添加
2 借出
3 归还
4 删除
5 列表
0 退出
```

其中的操作说明如下。

- 添加: 输入书名、作者→add(new Book(...))方法。
- 借出: 先列表→输入索引→调用 borrow()方法。
- 归还: 先列表→输入索引→调用 returnBook()方法。
- 删除: 先列表→输入索引→remove(index)方法。
- 列表: 打印 index 书名 by 作者状态。

(4) 启动行为: 程序启动时自动添加以下两本书。

```
《Java 核心》Cay 可借
《Effective Java》Joshua 可借
```

(5) 输出示例:

```
 === 图书借阅 ===
1 添加
2 借出
3 归还
4 删除
5 列表
0 退出
请选择: 5
0 Java 核心 by Cay 可借
1 Effective Java by Joshua 可借

请选择: 2
请输入要借出的图书编号: 0
《Java 核心》已借出!

请选择: 5
0 Java 核心 by Cay 已借出
1 Effective Java by Joshua 可借
```

```
请选择：3
请输入要归还的图书编号：0
《Java 核心》已归还!

请选择：1
请输入书名：Spring 实战
请输入作者：Craig
添加成功!

请选择：5
0 Java 核心 by Cay 可借
1 Effective Java by Joshua 可借
2 Spring 实战 by Craig 可借

请选择：4
请输入要删除的图书编号：1
已删除《Effective Java》!

请选择：0
感谢使用,再见!
```