

有序集合，其中每个元素的位置很重要。有两种访问元素的协议：一种是通过迭代器，另一种是通过 `get` 和 `set` 方法随机访问。后者不适用于链表，但当然 `get` 和 `set` 方法对数组很有用。集合类库提供了我们熟悉的 `ArrayList` 类，这个类也实现了 `List` 接口。`ArrayList` 封装了一个动态再分配的对象数组。

注释：对于一个经验丰富的 Java 程序员来说，需要一个动态数组时，可能会使用 `Vector` 类。为什么要用 `ArrayList` 而不是 `Vector` 呢？原因很简单：`Vector` 类的所有方法都是同步的。可以安全地从两个线程访问一个 `Vector` 对象。但是，如果只从一个线程访问 `Vector`（这种情况更为常见），代码就会在同步操作上白白浪费大量的时间。而与之不同，`ArrayList` 方法不是同步的，因此，不需要同步时建议使用 `ArrayList`，而不要使用 `Vector`。

9.3.3 散列集

链表和数组允许你根据意愿指定元素的次序。但是，如果想要查找某个特定的元素，却又不记得它的位置，就需要访问所有元素，直到找到匹配的元素为止。如果集合中包含的元素很多，这就会耗费很长时间。如果不注意元素的顺序，还有几种数据结构允许你更快速地查找元素。缺点是，这些数据结构不允许你控制元素出现的次序，它们会按照对自己最方便的方式组织元素。

有一种众所周知的数据结构，可以用于快速地查找对象，这就是散列表（hash table）。散列表为每个对象计算一个整数，称为散列码（hash code）。散列码是以某种方式由对象的实例字段得出的一个整数，这种方式可以尽可能保证有不同数据的对象将生成不同的散列码。表 9-2 列出了几个散列码的示例，它们是由 `String` 类的 `hashCode` 方法得到的。

如果定义你自己的类，你就要负责实现自己的 `hashCode` 方法。有关 `hashCode` 方法的详细内容请参见第 5 章。注意，你的实现应该与 `equals` 方法兼容，即如果 `a.equals(b)` 为 `true`，那么 `a` 与 `b` 必须有相同的散列码。

现在，重要的是要能够快速地计算出散列码，并且这个计算只与要计算散列的那个对象的状态有关，与散列表中的其他对象无关。

在 Java 中，散列表实现为链表数组。每个列表被称为桶（bucket，参见图 9-10）。要想查找一个对象在表中的位置，就要先计算它的散列码，然后与桶的总数取余，所得到的数就是保存这个元素的那个桶的索引。例如，如果某个对象的散列码为 76268，总共有 128 个桶，那么这个对象应该保存在第 108 号桶中（因为 $76\,268 \% 128$ 的余数是 108）。或许很幸运，这个桶中没有其他元素，此时将元素直接插入这个桶中就可以了。当然，有时候会遇到桶已经填充了元素的情况。这种现象被称为散列冲突（hash collision）。这时，需要将新对象与那个

表 9-2 `hashCode` 方法得到的散列码

字符串	散列码
"Lee"	76268
"lee"	107020
"eel"	100300

桶中的所有对象进行比较，查看这个对象是否已经存在。如果散列码合理地随机分布，而且桶的数目足够大，需要比较的次数就会很少。

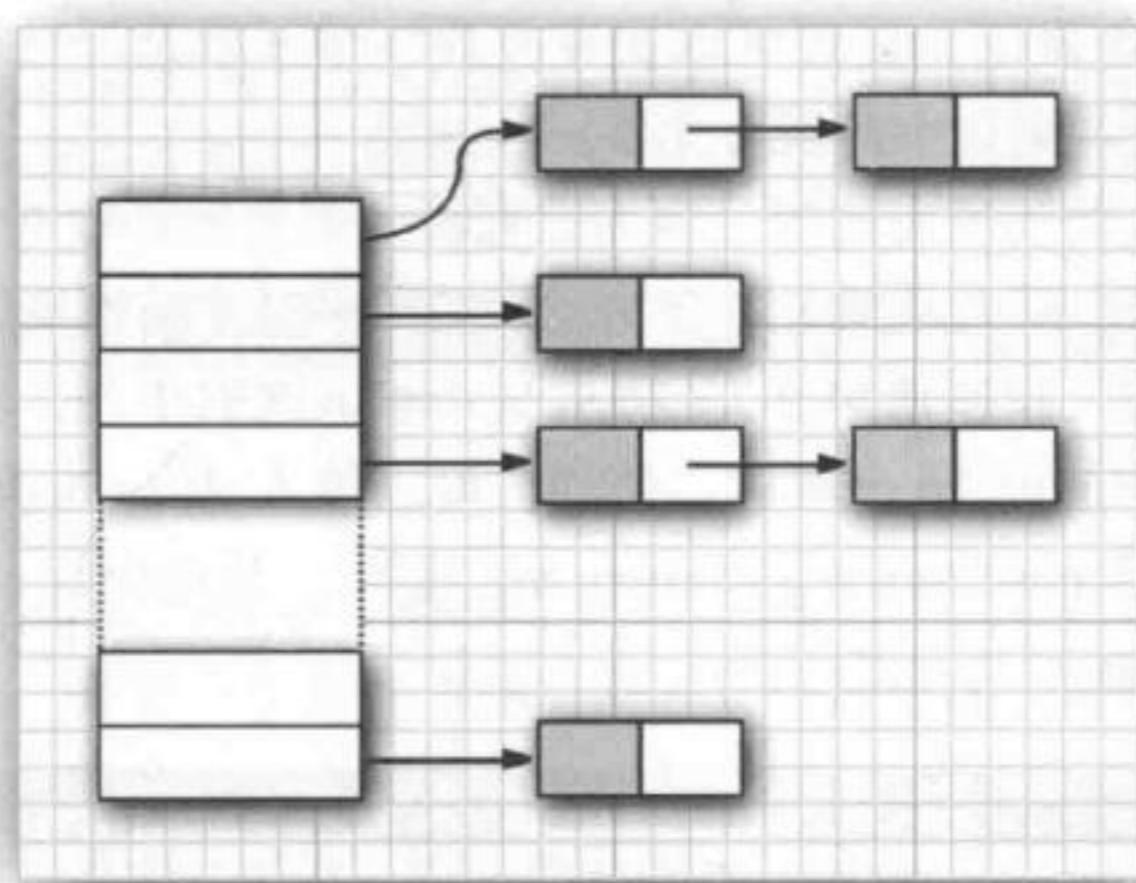


图 9-10 散列表

注释：在 Java 8 中，桶满时会从链表变为平衡二叉树。如果选择的散列函数不好，会产生很多冲突，或者如果有恶意代码试图在散列表中填充多个有相同散列码的值，改为平衡二叉树能提高性能。

提示：散列表的键要尽可能属于一个实现了 Comparable 接口的类。这样一来，就能保证不会由于散列码分布不均匀而导致性能低下。

如果想更多地控制散列表的性能，可以指定一个初始的桶数。桶数是指用于收集有相同散列值的桶的数目。如果要插入到散列表中的元素太多，冲突数就会增加，这会降低检索性能。

如果大致知道最终会有多少个元素要插入散列表中，就可以设置桶数。通常，要将桶数设置为预计元素个数的 75% ~ 150%。有些研究人员认为：将桶数设置为一个素数是一个好主意，以防止键的聚集。不过，对此并没有确凿的证据。标准类库使用的桶数是 2 的幂，默认值为 16（为表大小提供的任何值都将自动调整为 2 的下一个幂值）。

当然，并不总是能够知道需要存储多少个元素，也有可能最初的估计过低。如果散列表太满，就需要再散列（rehashed）。如果要对散列表再散列，就需要创建一个桶数更多的表，并将所有元素插入这个新表中，然后丢弃原来的表。装填因子（load factor）可以确定何时对散列表进行再散列。例如，如果装填因子为 0.75（默认值），而表中已经填满了 75% 以上，就会自动再散列，新表的桶数是原来的两倍。对于大多数应用来说，装填因子为 0.75 是合理的。

散列表可以用于实现很多重要的数据结构。其中最简单的是集类型。集是没有重复元素

的元素集合。集的 add 方法首先尝试在这个集中查找要添加的对象，只有这个元素不存在时才会添加这个对象。

Java 集合类库提供了一个 HashSet 类，它基于散列表实现了一个集。可以用 add 方法添加元素。contains 方法被重新定义，以便可以快速查找一个元素是否已经在集中。它只查看一个桶中的元素，而不必查看集合中的所有元素。

散列集迭代器将依次访问所有的桶。因为散列将元素分散存放在表中，所以会以一种看起来随机的顺序访问元素。只有不关心集合中元素的顺序时才应该使用 HashSet。

本节末尾的示例程序（程序清单 9-2）将从 System.in 读取单词，然后将它们添加到一个集中，最后再打印出集中的前 20 个单词。例如，可以输入 *Alice in Wonderland*（《爱丽丝漫游仙境》）的文本（可以从 <http://www.gutenberg.org> 找到），从命令行 shell 运行这个程序：

```
java SetTest < alice30.txt
```

程序清单 9-2 set/SetTest.java

```

1 package set;
2
3 import java.util.*;
4
5 /**
6  * This program uses a set to print all unique words in System.in.
7  * @version 1.12 2015-06-21
8  * @author Cay Horstmann
9 */
10 public class SetTest
11 {
12     public static void main(String[] args)
13     {
14         var words = new HashSet<String>();
15         long totalTime = 0;
16
17         try (var in = new Scanner(System.in))
18         {
19             while (in.hasNext())
20             {
21                 String word = in.next();
22                 long callTime = System.currentTimeMillis();
23                 words.add(word);
24                 callTime = System.currentTimeMillis() - callTime;
25                 totalTime += callTime;
26             }
27         }
28
29         Iterator<String> iter = words.iterator();
30         for (int i = 1; i <= 20 && iter.hasNext(); i++)
31             System.out.println(iter.next());
32         System.out.println("... ");
33         System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
34     }
35 }
```

这个程序将读取输入的所有单词，将它们添加到散列集中。然后迭代处理散列集中的不同单词，最后打印出单词的数量（*Alice in Wonderland* 共有 5909 个不同的单词，包括开头的版权声明）。单词以随机的顺序出现。

! 警告：在更改集中的元素时要格外小心。如果元素的散列码发生了改变，这个元素在数据结构中的位置也会变化。

API java.util.HashSet<E> 1.2

- `HashSet()`
构造一个空散列集。
- `HashSet(Collection<? extends E> elements)`
构造一个散列集，并将一个集合中的所有元素添加到这个散列集中。
- `HashSet(int initialCapacity)`
构造一个具有指定容量（桶数）的空散列集。
- `HashSet(int initialCapacity, float loadFactor)`
构造一个有指定的容量和装填因子的空散列集。如果大小 / 容量比大于这个装填因子，散列表会再散列为一个更大的散列表。

API java.lang.Object 1.0

- `int hashCode()`
返回这个对象的散列码。散列码可以是任何整数，包括正数或负数。`equals` 和 `hashCode` 的定义必须兼容，即如果 `x.equals(y)` 为 `true`, `x.hashCode()` 必须等于 `y.hashCode()`。

9.3.4 树集

`TreeSet` 类与散列集十分类似，不过，它比散列集有所改进。树集是一个有序集合（sorted collection）。可以以任意顺序将元素插入集合中。在对集合进行遍历时，值将自动地按照排序后的顺序出现。例如，假设插入 3 个字符串，然后访问添加的所有元素。

```
var sorter = new TreeSet<String>();
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.out.println(s);
```

这时，值将按照有序顺序打印：Amy Bob Carl。正如 `TreeSet` 类名所示，排序是用一个树数据结构完成的（当前实现使用的是红黑树（red-black tree）。有关红黑树的详细介绍请参见 *Introduction to Algorithms*[⊖]，作者是 Thomas Cormen、Charles Leiserson、Ronald Rivest 和 Clifford Stein [The MIT Press, 2009]）。每次将一个元素添加到树中时，都会将其放置在正确的排序位置上。因此，迭代器总是以有序的顺序访问每个元素。

[⊖] 此书中文版《算法导论》已由机械工业出版社引进出版，ISBN：978-7-111-40701-0。——编辑注

将一个元素添加到树中要比添加到散列表中慢，参见表 9-3 中的比较，但是，与检查数组或链表中的重复元素相比，使用树还是会快得多。如果树中包含 n 个元素，查找新元素的正确位置平均需要 $\log_2 n$ 次比较。例如，如果一棵树包含了 1000 个元素，添加一个新元素大约需要比较 10 次。

表 9-3 将元素添加到散列集和树集

文 档	单词总数	不同单词个数	HashSet	TreeSet
<i>Alice in Wonderland</i>	28 195	5 909	5 秒	7 秒
<i>The Count of Monte Cristo</i>	466 300	37 545	75 秒	98 秒

注释：要使用树集，必须能够比较元素。这些元素必须实现 Comparable 接口，或者构造集时必须提供一个 Comparator（Comparable 和 Comparator 接口在第 6 章介绍过）。

回头看表 9-3，你可能很想知道是否应该总是使用树集而不是散列集。毕竟，添加元素所花费的时间看上去并没有增加太多，而且元素会自动排序。答案取决于所要收集的数据。如果不需要数据有序，就没有必要付出排序的开销。更重要的是，对于某些数据来说，对其进行排序要比给出一个散列函数更加困难。散列函数只需要将对象适当地打乱存放，而比较函数必须精确地区分各个对象。

为了具体地了解它们之间的差异，可以考虑收集一个矩形集的任务。如果使用 TreeSet，就需要提供 Comparator<Rectangle>。如何比较两个矩形呢？按面积比较吗？这行不通。可能会有两个不同的矩形，它们的坐标不同，但面积却相同。树的排序顺序必须是全序（total ordering）。也就是说，任意两个元素都必须是可比较的，只有在两个元素相等时比较结果才为 0。矩形确实有一种排序方式（按照坐标的词典顺序排序），但这很牵强，而且计算很繁琐。相比之下，已经为 Rectangle 类定义了散列函数，它直接对坐标计算散列。

注释：从 Java 6 起，TreeSet 类实现了 NavigableSet 接口。这个接口增加了几个便利方法，用于查找元素以及反向遍历。详细信息请参见 API 注释。

程序清单 9-3 的程序中创建了 Item 对象的两个树集。第一个按照部件编号排序，这是 Item 对象的默认排序顺序。第二个使用一个定制比较器按照描述信息排序。程序清单 9-4 中是 Item 对象。

程序清单 9-3 treeSet/TreeSetTest.java

```

1 package treeSet;
2
3 import java.util.*;
4
5 /**
6 * This program sorts a set of Item objects by comparing their descriptions.
7 * @version 1.13 2018-04-10
8 * @author Cay Horstmann
9 */

```

```
10 public class TreeSetTest
11 {
12     public static void main(String[] args)
13     {
14         var parts = new TreeSet<Item>();
15         parts.add(new Item("Toaster", 1234));
16         parts.add(new Item("Widget", 4562));
17         parts.add(new Item("Modem", 9912));
18         System.out.println(parts);
19
20         var sortByDescription = new TreeSet<Item>(Comparator.comparing(Item::getDescription));
21
22         sortByDescription.addAll(parts);
23         System.out.println(sortByDescription);
24     }
25 }
```

程序清单 9-4 treeSet/Item.java

```
1 package treeSet;
2
3 import java.util.*;
4
5 /**
6  * An item with a description and a part number.
7  */
8 public class Item implements Comparable<Item>
9 {
10     private String description;
11     private int partNumber;
12
13     /**
14      * Constructs an item.
15      * @param aDescription the item's description
16      * @param aPartNumber the item's part number
17      */
18     public Item(String aDescription, int aPartNumber)
19     {
20         description = aDescription;
21         partNumber = aPartNumber;
22     }
23
24     /**
25      * Gets the description of this item.
26      * @return the description
27      */
28     public String getDescription()
29     {
30         return description;
31     }
32
33     public String toString()
34     {
35         return "[description=" + description + ", partNumber=" + partNumber + "]";
36     }
37 }
```

```

36 }
37
38 public boolean equals(Object otherObject)
39 {
40     if (this == otherObject) return true;
41     if (otherObject == null) return false;
42     if (getClass() != otherObject.getClass()) return false;
43     var other = (Item) otherObject;
44     return Objects.equals(description, other.description) && partNumber == other.partNumber;
45 }
46
47 public int hashCode()
48 {
49     return Objects.hash(description, partNumber);
50 }
51
52 public int compareTo(Item other)
53 {
54     int diff = Integer.compare(partNumber, other.partNumber);
55     return diff != 0 ? diff : description.compareTo(other.description);
56 }
57 }

```

API **java.util.TreeSet<E>** 1.2

- **TreeSet()**
- **TreeSet(Comparator<? super E> comparator)**
构造一个空树集。
- **TreeSet(Collection<? extends E> elements)**
- **TreeSet(SortedSet<E> s)**

构造一个树集，并增加一个集合或有序集中的所有元素（对于后一种情况，要使用同样的顺序）。

API **java.util.SortedSet<E>** 1.2

- **Comparator<? super E> comparator()**

返回用于对元素进行排序的比较器。如果元素用 Comparable 接口的 compareTo 方法进行比较则返回 null。

- **E first()**
- **E last()**

返回有序集中的最小元素或最大元素。

API **java.util.NavigableSet<E>** 6

- **E higher(E value)**
- **E lower(E value)**

返回大于 value 的最小元素或小于 value 的最大元素，如果没有这样的元素则返回 null。

- E ceiling(E value)

- E floor(E value)

返回大于等于 value 的最小元素或小于等于 value 的最大元素，如果没有这样的元素则返回 null。

- E pollFirst()

- E pollLast()

删除并返回这个集中的最大元素或最小元素，这个集为空时返回 null。

- Iterator<E> descendingIterator()

返回一个按照降序遍历集中元素的迭代器。

9.3.5 队列与双端队列

前面已经讨论过，队列允许高效地在队尾添加元素，并在队头删除元素。双端队列 (deque) 在队头和队尾都能高效地添加或删除元素。不支持在队列中间添加元素。Java 6 中引入了 Deque 接口，ArrayDeque 和 LinkedList 类实现了这个接口。这两个类都可以提供双端队列，其大小可以根据需要扩展。第 12 章会介绍限定队列和限定双端队列。

API java.util.Queue<E> 5

- boolean add(E element)

- boolean offer(E element)

如果队列没有满，将给定的元素添加到这个队列的队尾并返回 true。如果队列已满，第一个方法将抛出一个 IllegalStateException，而第二个方法返回 false。

- E remove()

- E poll()

如果队列不为空，删除并返回这个队列队头的元素。如果队列是空的，第一个方法抛出 NoSuchElementException，而第二个方法返回 null。

- E element()

- E peek()

如果队列不为空，返回这个队列队头的元素，但不删除这个元素。如果队列为空，第一个方法将抛出一个 NoSuchElementException，而第二个方法返回 null。

API java.util.Deque<E> 6

- void addFirst(E element)

- void addLast(E element)

- boolean offerFirst(E element)

- boolean offerLast(E element)

将给定的对象添加到双端队列的队头或队尾。如果这个双端队列已满，前面两个方法将抛出一个 IllegalStateException，而后面两个方法返回 false。

- E removeFirst()
- E removeLast()
- E pollFirst()
- E pollLast()

如果这个双端队列不为空，删除并返回双端队列队头的元素。如果双端队列为空，前面两个方法将抛出一个 `NoSuchElementException`，而后面两个方法返回 `null`。

- E getFirst()
- E getLast()
- E peekFirst()
- E peekLast()

如果这个双端队列非空，返回双端队列队头的元素，但不删除这个元素。如果双端队列为空，前面两个方法将抛出一个 `NoSuchElementException`，而后面两个方法返回 `null`。

API `java.util.ArrayDeque<E>` 6

- `ArrayDeque()`
- `ArrayDeque(int initialCapacity)`

用初始容量 16 或给定的初始容量构造一个无限定双端队列。

9.3.6 优先队列

优先队列（priority queue）中的元素可以按照任意的顺序插入，但会按照有序的顺序获取。也就是说，调用 `remove` 方法时，总会获得当前优先队列中最小的元素。不过，优先队列并没有对所有元素进行排序。如果迭代处理这些元素，并不需要对它们进行排序。优先队列使用了一个精巧且高效的数据结构，称为堆（heap）。堆是一个自组织的二叉树，其添加（`add`）和删除（`remove`）操作会让最小的元素移动到根，而不必花费时间对元素进行排序。

与 `TreeSet` 一样，优先队列既可以包含实现了 `Comparable` 接口的类对象，也可以包含构造器中提供的 `Comparator` 对象。

优先队列的典型用法是任务调度。每一个任务有一个优先级，任务以随机顺序添加到队列中。每当启动一个新的任务时，将从队列中删除优先级最高的任务（因为习惯将 1 作为“最高”优先级，所以 `remove` 操作将删除最小的元素）。

程序清单 9-5 显示了一个优先队列的具体使用。与 `TreeSet` 中的迭代不同，这里的迭代并不是按照有序顺序来访问元素。不过，删除操作总是删除剩余元素中最小的那个元素。

程序清单 9-5 `priorityQueue/PriorityQueueTest.java`

```

1 package priorityQueue;
2
3 import java.util.*;
4 import java.time.*;
5
6 /**

```

```

7 * This program demonstrates the use of a priority queue.
8 * @version 1.02 2015-06-20
9 * @author Cay Horstmann
10 */
11 public class PriorityQueueTest
12 {
13     public static void main(String[] args)
14     {
15         var pq = new PriorityQueue<LocalDate>();
16         pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
17         pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
18         pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
19         pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse
20
21         System.out.println("Iterating over elements . . .");
22         for (LocalDate date : pq)
23             System.out.println(date);
24         System.out.println("Removing elements . . .");
25         while (!pq.isEmpty())
26             System.out.println(pq.remove());
27     }
28 }
```

API java.util.PriorityQueue 5

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`
构造一个存放 Comparable 对象的优先队列。
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`
构造一个优先队列，并使用指定的比较器对元素进行排序。

9.4 映射

作为一个集合，集允许你快速地查找现有的元素。但是，要查找一个元素，需要有所查找的那个元素的准确副本。这不是一种常见的查找方式。通常，我们知道某些关键信息，希望查找与之关联的元素。映射（map）数据结构就是为此设计的。映射用来存放键 / 值对。如果提供了键，可以查找一个值。例如，可以存储一个员工记录表，其中键为员工 ID，值为 Employee 对象。在下面的小节中，我们会学习如何使用映射。

9.4.1 基本映射操作

Java 类库为映射提供了两个通用的实现：HashMap 和 TreeMap。这两个类都实现了 Map 接口。散列映射对键进行散列，树映射根据键的顺序将它们组织为一个搜索树。散列或比较函数只应用于键。与键关联的值不进行散列或比较。

应该选择散列映射还是树映射呢？与集一样，散列稍微快一些，如果不需要按照有序的

顺序访问键，最好选择散列映射。

可以如下建立一个散列映射来存储员工信息：

```
var staff = new HashMap<String, Employee>(); // HashMap implements Map
var harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
...
```

每当向映射中添加一个对象时，必须同时提供一个键。在这里，键是一个字符串，对应的值是 Employee 对象。

要获取一个对象，必须使用键（因此必须记住键）。

```
var id = "987-98-9996";
Employee e = staff.get(id); // gets harry
```

如果映射中没有存储与指定键对应的信息，get 将返回 null。

null 返回值可能并不方便。有时对于没有出现在映射中的键，可以有一个合适的默认值。然后使用 getOrDefault 方法。

```
Map<String, Integer> scores = . . .;
int score = scores.getOrDefault(id, 0); // gets 0 if the id is not present
```

键必须是唯一的。不能对同一个键存放两个值。如果用同一个键调用两次 put 方法，第二个值就会取代第一个值。实际上，put 将返回与这个键参数关联的上一个值。

remove 方法从映射中删除给定键对应的元素。size 方法返回映射中的元素数。

要迭代处理映射的键和值，最容易的方法是使用 forEach 方法。可以提供一个接收键和值的 lambda 表达式。映射中的每一项会依序调用这个表达式。

```
scores.forEach((k, v) ->
    System.out.println("key=" + k + ", value=" + v));
```

程序清单 9-6 显示了映射的具体使用。首先将键 / 值对添加到映射中。然后，从映射中删除一个键，同时与之关联的值也会删除。接下来，修改与某一个键关联的值，并调用 get 方法查找一个值。最后，迭代处理元素集。

程序清单 9-6 map/MapTest.java

```
1 package map;
2
3 import java.util.*;
4
5 /**
6  * This program demonstrates the use of a map with key type String and value type Employee.
7  * @version 1.12 2015-06-21
8  * @author Cay Horstmann
9 */
10 public class MapTest
11 {
12     public static void main(String[] args)
13     {
14         var staff = new HashMap<String, Employee>();
15         staff.put("144-25-5464", new Employee("Amy Lee"));
16
17         System.out.println(staff.get("144-25-5464"));
18
19         staff.remove("144-25-5464");
20
21         System.out.println(staff.get("144-25-5464"));
22
23         staff.put("144-25-5464", new Employee("Bob Lee"));
24
25         System.out.println(staff.get("144-25-5464"));
26
27         scores.forEach((k, v) ->
28             System.out.println("key=" + k + ", value=" + v));
29     }
30 }
```

```

16     staff.put("567-24-2546", new Employee("Harry Hacker"));
17     staff.put("157-62-7935", new Employee("Gary Cooper"));
18     staff.put("456-62-5527", new Employee("Francesca Cruz"));
19
20     // print all entries
21
22     System.out.println(staff);
23
24     // remove an entry
25
26     staff.remove("567-24-2546");
27
28     // replace an entry
29
30     staff.put("456-62-5527", new Employee("Francesca Miller"));
31
32     // look up a value
33
34     System.out.println(staff.get("157-62-7935"));
35
36     // iterate through all entries
37
38     staff.forEach((k, v) ->
39         System.out.println("key=" + k + ", value=" + v));
40 }
41 }
```

API `java.util.Map<K, V>` 1.2

- `V get(Object key)`

获得与键关联的值；返回与键关联的对象，或者如果在映射中没有找到这个键，则返回 `null`。实现类可能禁止键为 `null`。

- `default V getOrDefault(Object key, V defaultValue)`

获得与键关联的值；返回与键关联的对象，或者如果在映射中没有找到这个键，则返回 `defaultValue`。

- `V put(K key, V value)`

将关联的一对键和值放到映射中。如果这个键已经存在，新对象将取代之前与这个键关联的对象。这个方法将返回键对应的旧值。如果之前没有这个键，则返回 `null`。实现类可能禁止键或值为 `null`。

- `void putAll(Map<? extends K, ? extends V> entries)`

将指定映射中的所有映射条目添加到这个映射中。

- `boolean containsKey(Object key)`

如果映射中有这个键，返回 `true`。

- `boolean containsValue(Object value)`

如果映射中有这个值，返回 `true`。

- `default void forEach(BiConsumer<? super K, ? super V> action) 8`

对这个映射中的所有键 / 值对应用这个动作。

API `java.util.HashMap<K, V>` 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`

构造一个空散列映射，它具有指定的容量和装填因子（装填因子是一个 $0.0 \sim 1.0$ 之间的数。这个数确定散列表填充到多大比例时就要再散列到一个更大的散列表）。默认的装填因子是 0.75。

API `java.util.TreeMap<K,V>` 1.2

- `TreeMap()`
为实现 `Comparable` 接口的键构造一个空的树映射。
- `TreeMap(Comparator<? super K> c)`
构造一个树映射，并使用一个指定的比较器对键进行排序。
- `TreeMap(Map<? extends K, ? extends V> entries)`
构造一个树映射，并增加一个映射中的所有映射条目。
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`
构造一个树映射，增加一个有序映射中的所有映射条目，并使用与给定有序映射相同的比较器。

API `java.util.SortedMap<K, V>` 1.2

- `Comparator<? super K> comparator()`
返回对键进行排序所用的比较器。如果用 `Comparable` 接口的 `compareTo` 方法对键进行比较，则返回 `null`。
- `K firstKey()`
- `K lastKey()`
返回映射中的最小或最大键。

9.4.2 更新映射条目

处理映射的一个难点是更新映射条目。正常情况下，可以得到与一个键关联的旧值，更新这个值，再放回更新后的值。不过，必须考虑一个特殊情况，即键第一次出现。下面来看一个例子，考虑使用映射来统计一个单词在文件中出现的频度。看到一个单词（`word`）时，我们将计数器增 1，如下所示：

```
counts.put(word, counts.get(word) + 1);
```

这是可以的，不过有一种情况除外：就是第一次看到 `word` 时。在这种情况下，`get` 会返回 `null`，因此会出现一个 `NullPointerException` 异常。

一种简单的补救是使用 `getOrDefault` 方法：

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

另一种方法是首先调用 `putIfAbsent` 方法。只有当键原先不存在（或者映射到 `null`）时才放入一个值。

```
counts.putIfAbsent(word, 0);
counts.put(word, counts.get(word) + 1); // now we know that get will succeed
```

不过还可以做得更好。`merge` 方法可以简化这个常见操作。如果键原先不存在，下面的调用：

```
counts.merge(word, 1, Integer::sum);
```

将把 `word` 与 1 关联，否则使用 `Integer::sum` 函数组合原值和 1（也就是将原值与 1 求和）。

API 注释还描述了另外一些更新映射条目的方法，不过这些方法不太常用。

API `java.util.Map<K, V>` 1.2

- `default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)` 8

如果 `key` 与一个非 `null` 值 `v` 关联，将函数应用到 `v` 和 `value`，将 `key` 与结果关联，或者如果结果为 `null`，则删除这个键。否则，将 `key` 与 `value` 关联，返回 `get(key)`。

- `default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)` 8

将函数应用到 `key` 和 `get(key)`。将 `key` 与结果关联，或者如果结果为 `null`，则删除这个键。返回 `get(key)`。

- `default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)` 8

如果 `key` 与一个非 `null` 值 `v` 关联，将函数应用到 `key` 和 `v`，将 `key` 与结果关联，或者如果结果为 `null`，则删除这个键。返回 `get(key)`。

- `default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)` 8

将这个函数应用到 `key`，除非 `key` 与一个非 `null` 值关联。将 `key` 与结果关联，或者如果结果为 `null`，则删除这个键。返回 `get(key)`。

- `default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)` 8

在所有映射条目上调用这个函数。将键与非 `null` 结果关联，如果结果为 `null`，则将相应的键删除。

- `default V putIfAbsent(K key, V value)` 8

如果 `key` 不存在或者与 `null` 关联，则将它与 `value` 关联，并返回 `null`。否则返回关联的值。

9.4.3 映射视图

集合框架不认为映射本身是一个集合。（其他数据结构框架则认为映射是一个键 / 值对集合，或者是按键索引的值集合。）不过，可以得到映射的视图（view）——实现了 `Collection` 接

口或某个子接口的对象。

有 3 种视图：键集、值集合（不是一个集）以及键 / 值对集。键和键 / 值对可以构成一个集，因为映射中一个键只能有一个副本。下面的方法：

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

会分别返回这 3 个视图。（映射条目集的元素是实现了 Map.Entry 接口的类的对象。）

需要说明的是，keySet 不是 HashSet 或 TreeSet，而是实现了 Set 接口的另外某个类的对象。Set 接口扩展了 Collection 接口。因此，可以像使用任何集合一样使用 keySet。

例如，可以枚举一个映射的所有键：

```
Set<String> keys = map.keySet();
for (String key : keys)
{
    do something with key
}
```

如果想同时查看键和值，可以通过枚举映射条目来避免查找值。可以使用以下代码：

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String k = entry.getKey();
    Employee v = entry.getValue();
    do something with k, v
}
```

 提示：通过使用 var 声明可以避免笨拙的 Map.Entry。

```
for (var entry : map.entrySet())
{
    do something with entry.getKey(), entry.getValue()
}
```

或者直接使用 forEach 方法：

```
map.forEach((k, v) ->
{
    do something with k, v
});
```

如果在键集视图上调用迭代器的 remove 方法，实际上会从映射中删除这个键和与它关联的值。不过，不能向键集视图中添加元素。另外，如果只添加一个键而没有同时添加值也是没有意义的。如果试图调用 add 方法，它会抛出一个 UnsupportedOperationException。映射条目集视图有同样的限制，尽管理论上好像可以增加新的键 / 值对，但实际上并不允许。

API `java.util.Map<K, V>` 1.2

- `Set<Map.Entry<K, V>> entrySet()`

返回 Map.Entry 对象（映射中的键 / 值对）的一个集视图。可以从这个集中删除元素，它们也将从映射中删除，但是不能添加任何元素。

- `Set<K> keySet()`

返回映射中所有键的一个集视图。可以从这个集中删除元素，这些键和相关联的值也将从映射中删除，但是不能添加任何元素。

- `Collection<V> values()`

返回映射中所有值的一个集合视图。可以从这个集合中删除元素，所删除的值及相应的键也将从映射中删除，不过不能添加任何元素。

API `java.util.Map.Entry<K, V>` 1.2

- `K getKey()`

- `V getValue()`

返回这个映射条目的键或值。

- `V setValue(V newValue)`

将关联映射中的值改为新值，并返回原来的值。

- `static <K, V> Map.Entry<K, V> copyOf(Map.Entry<? extends K, ? extends V> map) 1.7`

生成给定映射条目的一个副本。不同于映射条目集的元素，这个副本不是“活动的”。调用 `setValue` 不会更新任何映射。

9.4.4 弱散列映射

在集合类库中有几个专用的映射类，我们将在这一节和后面几节中对它们做简要介绍。

设计 `WeakHashMap` 类是为了解决一个有趣的问题。如果有一个值，它对应的键已经不再在程序中的任何地方使用，将会出现什么情况？假设对某个键的最后一个引用已经消失，那么再没有任何途径可以引用这个值对象了。但是，由于程序中的任何部分都不再有这个键，因此无法从映射中删除这个键/值对。为什么垃圾回收器不能删除它呢？删除无用的对象不就是垃圾回收器的工作吗？

遗憾的是，事情没有这么简单。垃圾回收器会跟踪活动的对象。只要映射对象是活动的，其中的所有桶就是活动的，它们就不能被回收。因此，需要由程序负责从长期存活的映射中删除那些无用的值。或者，你可以使用 `WeakHashMap`。当键的唯一引用来自散列表条目时，这个数据结构将与垃圾回收器合作删除键/值对。

下面介绍这种机制的内部工作原理。`WeakHashMap` 使用弱引用（weak reference）保存键。`WeakReference` 对象将包含另一个对象的引用，在这里，就是一个散列表键。对于这种类型的对象，垃圾回收器采用一种特殊的方式进行处理。正常情况下，如果垃圾回收器发现某个特定的对象已经没有引用了，就会将其回收。不过，如果这个对象只能由一个 `WeakReference` 引用，垃圾回收器也会将其回收，但会将引用这个对象的弱引用放入一个队列。`WeakHashMap` 的操作会定期地检查这个队列，查找新加入的弱引用。一个弱引用进入这个队列意味着这个键不再由任何人使用，并且已经回收。于是，`WeakHashMap` 将删除相关联的映射条目。

9.4.5 链接散列集与映射

`LinkedHashSet` 和 `LinkedHashMap` 类会记住插入元素项的顺序。这样可以避免散列表中看起来随机的元素顺序。在散列表中插入元素项时，它们会加入一个双向链表中（见图 9-11）。

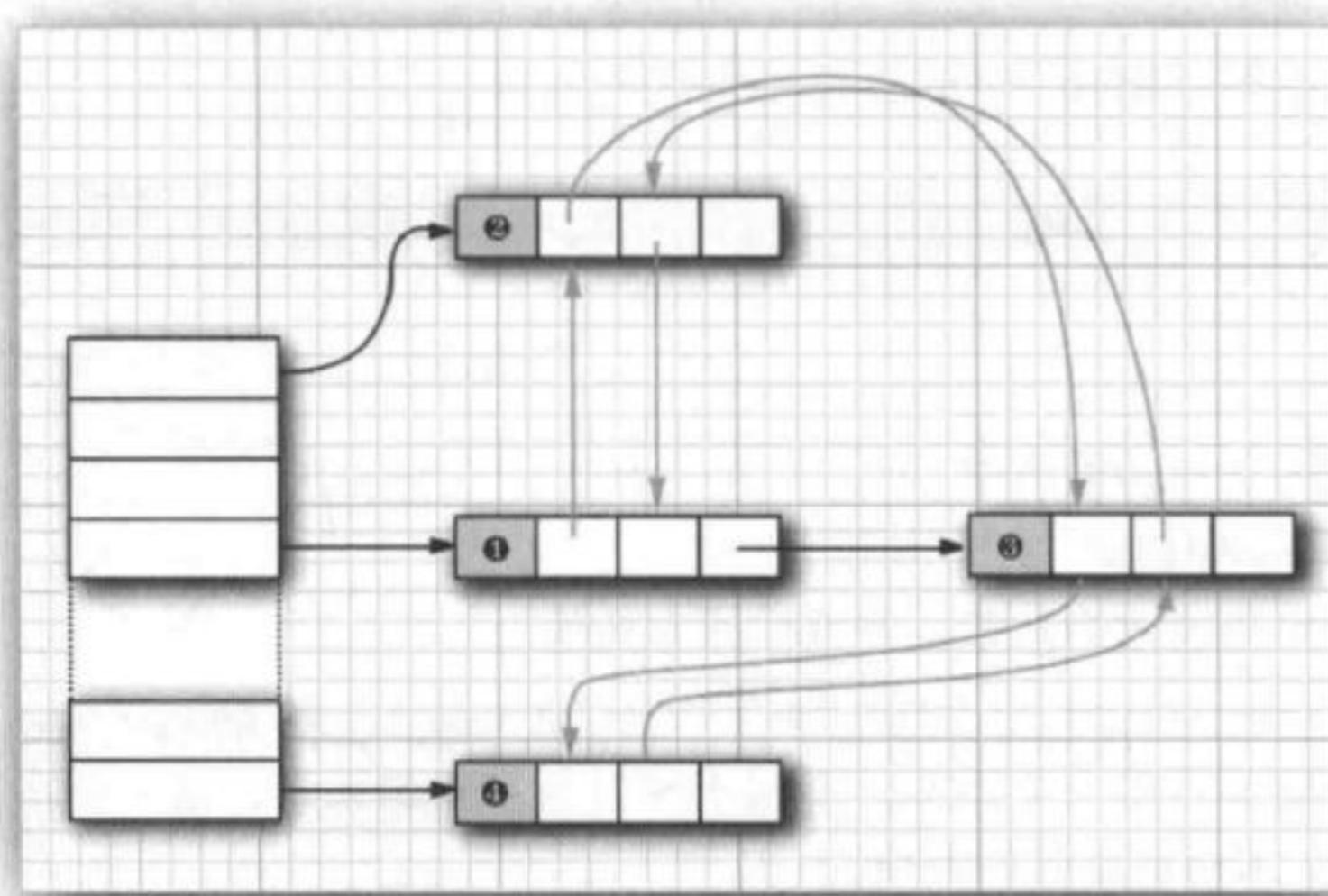


图 9-11 链接散列表

例如，考虑程序清单 9-6 中的以下映射插入操作：

```
var staff = new LinkedHashMap<String, Employee>();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

然后，`staff.keySet().iterator()` 会以下面的顺序枚举键：

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

`staff.values().iterator()` 以下面的顺序枚举值：

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

或者，链接散列映射可以使用访问顺序（access order）而不是插入顺序来迭代处理映射条目。每次调用 `get` 或 `put` 时，受到影响的条目将从当前位置删除，并放在条目链表的末尾（只影响条目链表中的位置，而不影响散列表的桶。一个映射条目总是在键散列码对应的桶中）。要构造这样一个散列映射，需要调用

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

访问顺序对于实现缓存的“最近最少使用”原则十分重要。例如，你可能希望将访问频率高的元素放在内存中，而从数据库读取访问频率低的元素。如果在映射表中没有找到某个元素，而且此时映射表已经很满，可以得到映射表的一个迭代器，并删除它枚举的前几个元素，它们是最近最少使用的几个元素。

甚至可以自动完成这一过程。构造 `LinkedHashMap` 的一个子类，然后覆盖下面这个方法：

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

每当这个方法返回 `true` 时，添加一个新映射条目就会导致删除 `eldest` 映射条目。例如，下面的缓存最多可以存放 100 个元素：

```
var cache = new LinkedHashMap<K, V>(128, 0.75F, true)
{
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
    {
        return size() > 100;
    }
};
```

或者，还可以考虑 `eldest` 映射条目来决定是否将它删除。例如，可以检查随这个映射条目存储的一个时间戳。

9.4.6 枚举集与映射

`EnumSet` 是一个高效的集实现，其元素属于一个枚举类型。因为枚举类型只有有限个实例，所以 `EnumSet` 在内部实现为一个位序列。如果对应的值在集中出现，相应的位则置为 1。

`EnumSet` 类没有公共构造器。要使用静态工厂方法构造这个集：

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

可以使用 `Set` 接口的常用方法来修改 `EnumSet`。

`EnumMap` 是一个映射，它的键属于一个枚举类型。`EnumMap` 可以简单高效地实现为一个值数组。需要在构造器中指定键类型：

```
var personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```

注释：在 `EnumSet` 的 API 文档中，会看到形如 `E extends Enum<E>` 的奇怪的类型参数。简单地说，它的意思就是“`E` 是一个枚举类型。”所有枚举类型都扩展了泛型 `Enum` 类。例如，`Weekday` 扩展了 `Enum<Weekday>`。

9.4.7 标识散列映射

`IdentityHashMap` 有一个很特殊的用途。在这里，键的散列值不是用 `hashCode` 函数计算的，而是用 `System.identityHashCode` 方法计算的。`Object.hashCode` 根据对象的内存地址计算散列码时

就使用了这个方法。另外，对两个对象进行比较时，`IdentityHashMap` 使用了`=`，而不是`equals`。

也就是说，不同的键对象即使内容相同，也被视为不同的对象。在实现对象遍历算法（如对象串行化）时，如果你想跟踪哪些对象已经遍历过，这个类就很有用。

API `java.util.WeakHashMap<K, V>` 1.2

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`

用给定的容量和装填因子构造一个空散列映射。

API `java.util.LinkedHashSet<E>` 1.4

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`

用给定的容量和装填因子构造一个空链接散列集。

API `java.util.LinkedHashMap<K, V>` 1.4

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`

用给定的容量、装填因子和顺序构造一个空链接散列映射。`accessOrder` 参数为`true` 时表示访问顺序，为`false` 时表示插入顺序。

- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`

如果想删除`eldest` 元素，就要覆盖为返回`true`。`eldest` 参数是预期可能要删除的元素。这个方法在向映射中添加一个元素之后调用。默认实现会返回`false`。即在默认情况下，不会删除老元素。不过，可以重新定义这个方法，从而有选择地返回`true`。例如，如果最老的元素符合某个条件，或者如果映射超过了一定大小，则返回`true`。

API `java.util.EnumSet<E extends Enum<E>>` 5

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`
返回一个可变集，包含给定枚举类型的所有值。
- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`
返回一个初始为空的可变集。
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`
返回一个可变集，包含`from ~ to` 之间的所有值（包括`from` 和`to`）。
- `static <E extends Enum<E>> EnumSet<E> of(E e)`
...

- static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
- static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)
返回一个可变集，包括不为 null 的给定元素。
- public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)
- public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
创建一个初始包含给定元素的可变集。在第二个方法中，c 必须是一个 EnumSet 或者非空（以确定元素类型）。

API `java.util.EnumMap<K extends Enum<K>, V>` 5

- `EnumMap(Class<K> keyType)`
构造一个键为给定类型的空的可变映射。

API `java.util.IdentityHashMap<K, V>` 1.4

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`
构造一个空的标识散列映射集，其容量是大于 $1.5 \times \text{expectedMaxSize}$ 的最小的 2 的幂值（`expectedMaxSize` 的默认值是 21）。

API `java.lang.System` 1.0

- static int `identityHashCode(Object obj)` 1.1
返回 `Object.hashCode` 计算的相同散列码（根据对象的内存地址得出），即使 `obj` 所属的类已经重新定义了 `hashCode` 方法。

9.5 副本与视图

如果查看图 9-4 和图 9-5，可能会认为用如此多的接口和抽象类来实现数量并不多的具体集合类似乎没有太大必要。不过，这两个图并没有展示出全部。通过使用视图（view），可以得到其他实现了 `Collection` 接口或 `Map` 接口的对象。你已经见过使用映射类 `keySet` 方法的这样一个例子。初看起来，好像这个方法创建了一个新集，并填入映射中的所有键，然后返回这个集。但是，情况并非如此。实际上，`keySet` 方法返回一个实现了 `Set` 接口的类对象，这个类的方法可以操纵原映射。这种集合称为视图。

视图技术在集合框架中有许多非常有用的应用。下面几节将讨论这些应用。

9.5.1 小集合

Java 9 引入了一些静态方法，可以生成给定元素的集或列表，以及给定键 / 值对的映射。例如，

```
List<String> names = List.of("Peter", "Paul", "Mary");
Set<Integer> numbers = Set.of(2, 3, 5);
```

会分别生成包含 3 个元素的一个列表和一个集。对于映射，需要指定键和值，如下所示：

```
Map<String, Integer> scores = Map.of("Peter", 2, "Paul", 3, "Mary", 5);
```

元素、键或值不能为 null。集和映射键不能重复：

```
numbers = Set.of(13, null); // Error--null element
scores = Map.of("Peter", 4, "Peter", 2); // Error--duplicate key
```

◆ 警告：对于这些集和映射中的迭代顺序，并没有任何保证。实际上，会有意用每次虚拟机启动时随机得到的一个种子搅乱这个顺序。来看看 jshell 的两次运行：

```
$ jshell -q
jshell> Set.of("Peter", "Paul", "Mary")
$1 ==> [Peter, Mary, Paul]
jshell> /exit
$ jshell -q
jshell> Set.of("Peter", "Paul", "Mary")
$1 ==> [Paul, Mary, Peter]
```

有些 Java 程序员编写程序时，其程序的正确性依赖于一个假设，即认为实现细节永远不会改变。这样会让实现类库的程序员很难对实现做有用的修改。在这里，道理很明显，编写程序时不要对元素顺序做任何假设。

List 和 Set 接口有 11 个 of 方法，分别有 0 到 10 个参数，另外还有一个参数个数可变的 of 方法。提供这种特定性是为了提高效率。

对于 Map 接口，则无法提供一个参数可变的版本，因为参数类型会交替为键类型和值类型。不过它有一个静态方法 ofEntries，能接受任意多个 Map.Entry<K, V> 对象（可以用静态方法 entry 创建这些对象）。例如，

```
import static java.util.Map.*;
...
Map<String, Integer> scores = ofEntries(
    entry("Peter", 2),
    entry("Paul", 3),
    entry("Mary", 5));
```

of 和 ofEntries 方法可以生成某些类的对象，这些类对于每个元素会有一个实例变量，或者有一个后备数组提供支持。

这些集合对象是不可修改的 (unmodifiable)。如果试图改变它们的内容，会导致一个 UnsupportedOperationException 异常。

如果需要一个可更改的集合，可以把这个不可修改的集合传递到构造器：

```
var names = new ArrayList<>(List.of("Peter", "Paul", "Mary")); // A mutable list of names
```

以下方法调用

```
Collections.nCopies(n, anObject)
```

会返回一个实现了 List 接口的不可变对象，给人一种错觉：就像有 n 个元素，每个元素看起来是一个 anObject。