

# 第13章 集 合

集合接口  
具体的集合  
集合框架

算法  
遗留的集合

在实现方法时，选择不同的数据结构会导致其实现风格以及性能存在着很大差异。需要快速地搜索成千上万个（甚至上百万）有序的数据项吗？需要快速地在有序的序列中间插入元素或删除元素吗？需要建立键与值之间的关联吗？

本章将讲述如何利用Java类库帮助我们在程序设计中实现传统的数据结构。在大学的计算机科学课程中，有一门叫做数据结构（Data Structures）的课程，通常要讲授一个学期，因此，有许许多多专门探讨这个重要主题的书籍。与大学课程所讲述的内容不同，这里，将跳过理论部分，仅介绍如何使用标准库中的集合类。

## 13.1 集合接口

Java最初版本只为最常用的数据结构提供了很少的一组类：Vector、Stack、Hashtable、BitSet、与Enumeration接口，其中的Enumeration接口提供了一种用于访问任意容器中各个元素的抽象机制。这是一种很明智的选择，但要想建立一个全面的集合类库还需要大量的时间和高超的技能。

随着Java SE 1.2的问世，设计人员感到是推出一组功能完善的数据结构的时机了。面对一大堆相互矛盾的设计策略，他们希望让类库规模小且易于学习，而不希望像C++的“标准模版库”（即STL）那样复杂，但却又希望能够得到STL率先推出的“泛型算法”所具有的优点。他们希望将传统的类融入新的框架中。与所有的集合类库设计者一样，他们必须做出一些艰难的选择，于是，在整个设计过程中，他们做出了一些独具特色的设计决定。本节将介绍Java集合框架的基本设计，展示使用它们的方法，并解释一些颇具争议的特性背后的考虑。

### 13.1.1 将集合的接口与实现分离

与现代的数据结构类库的常见情况一样，Java集合类库也将接口（interfaces）与实现（implementations）分离。首先，看一下人们熟悉的数据结构——队列（queue）是如何分离的。

队列接口指出可以在队列的尾部添加元素，在队列的头部删除元素，并且可以查找队列中元素的个数。当需要收集对象，并按照“先进先出”的规则检索对象时就应该使用队列（见图13-1）。

一个队列接口的最小形式可能类似下面这样：

```
interface Queue<E> // a simplified form of the interface in the standard library
{
    void add(E element);
    E remove();
    int size();
}
```

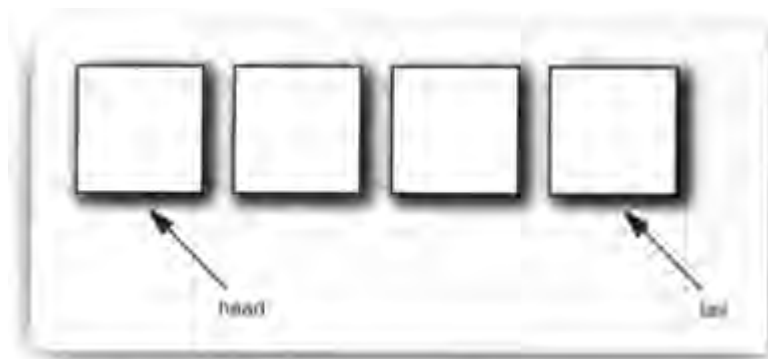


图13-1 队列

这个接口并没有说明队列是如何实现的。队列通常有两种实现方式：一种是使用循环数组；另一种是使用链表（见图13-2）。

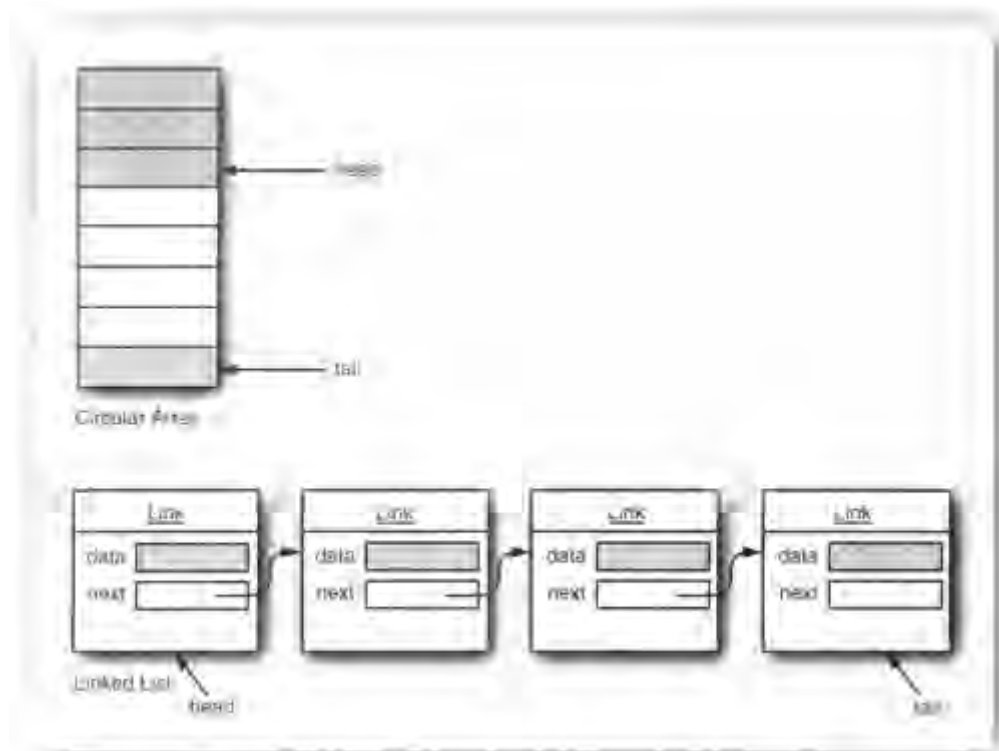


图13-2 队列的实现



注释：从Java SE 5.0开始，集合类是带有类型参数的泛型类。有关泛型类的更多信息，请参看第12章。

每一个实现都可以通过一个实现了Queue接口的类表示。

```
class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private E[] elements;
    private int head;
    private int tail;
}

class LinkedListQueue<E> implements Queue<E> // not an actual library class
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private Link head;
    private Link tail;
}
```



注释：实际上，Java类库没有名为CircularArrayQueue和LinkedListQueue的类。这里，只是以这些类作为示例，解释一下集合接口与实现在概念上的不同。如果需要一个循环数组队列，就可以使用Java SE 6中引入的ArrayDeque类。如果需要一个链表队列，就直接使用LinkedList类，这个类实现了Queue接口。

当在程序中使用队列时，一旦构建了集合就不需要知道究竟使用了哪种实现。因此，只有在构建集合对象时，使用具体的类才有意义。可以使用接口类型存放集合的引用。

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

利用这种方式，一旦改变了想法，可以轻松地使用另外一种不同的实现。只需要对程序的一个地方做出修改，即调用构造器的地方。如果最终觉得LinkedListQueue是个更好的选择，就将代码修改为：

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

为什么选择这种实现，而不选择那种实现呢？接口本身并不能说明哪种实现的效率究竟如何。循环数组要比链表更高效，因此多数人优先选择循环数组。然而，通常这样做也需要付出一定的代价。

循环数组是一个有界集合，即容量有限。如果程序中要收集的对象数量没有上限，就最好使用链表来实现。

在研究API文档时，会发现另外一组名字以Abstract开头的类，例如，AbstractQueue。这些类是为类库实现者而设计的。如果想要实现自己的队列类（也许不太可能），会发现扩展AbstractQueue类要比实现Queue接口中的所有方法轻松得多。

### 13.1.2 Java类库中的集合接口和迭代器接口

在Java类库中，集合类的基本接口是Collection接口。

这个接口有两个基本方法：

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    . . .
}
```

除了这两个方法之外，还有几个方法，将在稍后介绍。

add方法用于向集合中添加元素。如果添加元素确实改变了集合就返回true，如果集合没有发生变化就返回false。例如，如果试图向集中添加一个对象，而这个对象在集中已经存在，这个添加请求就没有实效，因为集中不允许有重复的对象。

iterator方法用于返回一个实现了Iterator接口的对象。可以使用这个迭代器对象依次访问集合中的元素。

#### 1. 迭代器

Iterator接口包含3个方法：

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

通过反复调用next方法，可以逐个访问集合中的每个元素。但是，如果到达了集合的末尾，next方法将抛出一个NoSuchElementException。因此，需要在调用next之前调用hasNext方法。如果迭代器对象还有多个供访问的元素，这个方法就返回true。如果想要查看集合中的所有元素，就请求一个迭代器，并在hasNext返回true时反复地调用next方法。例如：

```
Collection<String> c = . . . ;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    do something with element
}
```

从Java SE 5.0起，这个循环可以采用一种更优雅的缩写方式。用“for each”循环可以更加简练地表示同样的循环操作：

```
for (String element : c)
{
    do something with element
}
```

编译器简单地将“for each”循环翻译为带有迭代器的循环。

“for each”循环可以与任何实现了Iterable接口的对象一起工作，这个接口只包含一个方法：

```
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```

Collection接口扩展了Iterable接口。因此，对于标准类库中的任何集合都可以使用“for each”循环。

元素被访问的顺序取决于集合类型。如果对ArrayList进行迭代，迭代器将从索引0开始，每迭代一次，索引值加1。然而，如果访问HashSet中的元素，每个元素将会按照某种随机的次序出现。虽然可以确定在迭代过程中能够遍历到集合中的所有元素，但却无法预知元素被访问的次序。这对于计算总和或统计符合某个条件的元素个数这类与顺序无关的操作来说，并不是什么问题。



注释：编程老手会注意到：Iterator接口的next和hasNext方法与Enumeration接口的nextElement和hasMoreElements方法的作用一样。Java集合类库的设计者可以选择使用Enumeration接口。但是，他们不喜欢这个接口累赘的方法名，于是引入了具有较短方法名的新接口。

Java集合类库中的迭代器与其他类库中的迭代器在概念上有着重要的区别。在传统的集合类库中，例如，C++的标准模版库，迭代器是根据数组索引建模的。如果给定这样一个迭代器，就可以查看指定位置上的元素，就像知道数组索引i就可以查看数组元素a[i]一样。不需要查找元素，就可以将迭代器向前移动一个位置。这与不需要执行查找操作就可以通过i++将数组索引向前移动一样。但是，Java迭代器并不是这样操作的。查找操作与位置变更是紧密相连的。查找一个元素的惟一方法是调用next，而在执行查找操作的同时，迭代器的位置随之向前移动。

因此，应该将Java迭代器认为是位于两个元素之间。当调用next时，迭代器就越过下一个元素，并返回刚刚越过的那个元素的引用（见图13-3）。



注释：这里还有一个有用的类推。可以将Iterator.next与InputStream.read看作为等效的。从数据流中读取一个字节，就会自动地“消耗掉”这个字节。下一次调用read将会消耗并返回输入的下一个字节。用同样的方式，反复地调用next就可以读取集合中所有元素。

## 2. 删除元素

Iterator接口的remove方法将会删除上次调用next方法时返回的元素。在大多数情况下，在决定删除某个元素之前应该先看一下这个元素是很具有实际意义的。然而，如果想要删除指定位置上的元素，仍然需要越过这个元素。下面是如何删除字符串集合中第一个元素的方法：

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

更重要的是，对next方法和remove方法的调用具有互相依赖性。如果调用remove之前没有调用next将是不合法的。如果这样做，将会抛出一个IllegalStateException异常。

如果想删除两个相邻的元素，不能直接地这样调用：

```
it.remove();  
it.remove(); // Error!
```

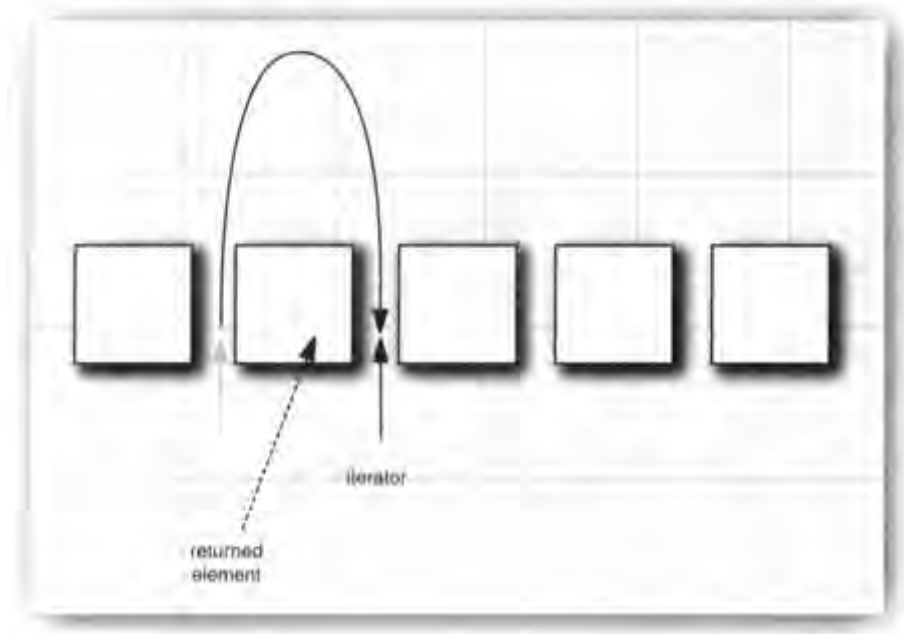


图13-3 向前移动迭代器

相反地，必须先调用next越过将要删除的元素。

```
it.remove();  
it.next();  
it.remove(); // Ok
```

### 3. 泛型实用方法

由于Collection与Iterator都是泛型接口，可以编写操作任何集合类型的实用方法。例如，下面是一个检测任意集合是否包含指定元素的泛型方法：

```
public static <E> boolean contains(Collection<E> c, Object obj)  
{  
    for (E element : c)  
        if (element.equals(obj))  
            return true;  
    return false;  
}
```

Java类库的设计者认为：这些实用方法中的某些方法非常有用，应该将它们提供给用户使用。这样，类库的使用者就不必自己重新构建这些方法了。contains就是这样一个实用方法。

事实上，Collection接口声明了很多有用的方法，所有的实现类都必须提供这些方法。下面列举了其中的一部分：

```
int size()  
boolean isEmpty()  
boolean contains(Object obj)  
boolean containsAll(Collection<?> c)  
boolean equals(Object other)
```

```
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

在这些方法中，有许多方法的功能非常明确，不需要过多的解释。在本节尾部的API注释中可以找到有关它们的完整文档说明。

当然，如果实现Collection接口的每一个类都要提供如此多的例行方法将是一件很烦人的事情。为了能够让实现者更容易地实现这个接口，Java类库提供了一个类AbstractCollection，它将基础方法size和iterator抽象化了，但是在此提供了例行方法。例如：

```
public abstract class AbstractCollection<E>
    implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();

    public boolean contains(Object obj)
    {
        for (E element : c) // calls iterator()
            if (element.equals(obj))
                return true;
        return false;
    }
    . . .
}
```

此时，一个具体的集合类可以扩展AbstractCollection类了。现在要由具体的集合类提供iterator方法，而contains方法已由AbstractCollection超类提供了。然而，如果子类有更加有效的方式实现contains方法，也可以由子类提供，就这点而言，没有什么限制。

对于类框架来说，这是一个很好的设计。集合类的用户可以使用泛型接口中一组更加丰富的方法，而实际的数据结构实现者并没有需要实现所有例行方法的负担。

#### java.util.Collection<E> 1.2

- Iterator<E> iterator()  
返回一个用于访问集合中每个元素的迭代器。
- int size()  
返回当前存储在集合中的元素个数。
- boolean isEmpty()  
如果集合中没有元素，返回true。
- boolean contains(Object obj)  
如果集合中包含了一个与obj相等的对象，返回true。
- boolean containsAll(Collection<?> other)

如果这个集合包含other集合中的所有元素，返回true。

- `boolean add(Object element)`

将一个元素添加到集合中。如果由于这个调用改变了集合，返回true。

- `boolean addAll(Collection<? extends E> other)`

将other集合中的所有元素添加到这个集合。如果由于这个调用改变了集合，返回true。

- `boolean remove(Object obj)`

从这个集合中删除等于obj的对象。如果有匹配的对象被删除，返回true。

- `boolean removeAll(Collection<?> other)`

从这个集合中删除other集合中存在的所有元素。如果由于这个调用改变了集合，返回true。

- `void clear()`

从这个集合中删除所有的元素。

- `boolean retainAll(Collection<?> other)`

从这个集合中删除所有与other集合中的元素不同的元素。如果由于这个调用改变了集合，返回true。

- `Object[] toArray()`

返回这个集合的对象数组。

- `<T> T[] toArray(T[] arrayToFill)`

返回这个集合的对象数组。如果arrayToFill足够大，就将集合中的元素填入这个数组中。剩余空间填补null；否则，分配一个新数组，其成员类型与arrayToFill的成员类型相同，其长度等于集合的大小，并添入集合元素。

#### **API** `java.util.Iterator<E> 1.2`

- `boolean hasNext()`

如果存在可访问的元素，返回true。

- `E next()`

返回将要访问的下一个对象。如果已经到达了集合的尾部，将抛出一个NoSuchElementException。

- `void remove()`

删除上次访问的对象。这个方法必须紧跟在访问一个元素之后执行。如果上次访问之后，集合已经发生了变化，这个方法将抛出一个IllegalStateException。

## 13.2 具体的集合

这里并不打算更加详细地介绍所有的接口，但我们认为先介绍一下Java类库提供的具体数据结构还是很有用途的。透彻地介绍了人们想使用的类之后，再回过头研究一些抽象的概念，看一看集合框架组织这些类的方式。表13-1展示了Java类库中的集合，并简要描述了每个集合类的用途（鉴于简单起见，省略了将在第14章中介绍的线程安全集合）。在表13-1中，除了以Map结尾的类之外，其他类都实现了Collection接口。而以Map结尾的类实现了Map接口。有关



Map接口的内容将在稍后介绍。

表13-1 Java库中的具体集合

集合类型	描述
ArrayList	一种可以动态增长和缩减的索引序列
LinkedList	一种可以在任何位置进行高效地插入和删除操作的有序序列
ArrayDeque	一种用循环数组实现的双端队列
HashSet	一种没有重复元素的无序集合
TreeSet	一种有序集
EnumSet	一种包含枚举类型值的集
LinkedHashSet	一种可以记住元素插入次序的集
PriorityQueue	一种允许高效删除最小元素的集合
HashMap	一种存储键/值关联的数据结构
TreeMap	一种键值有序排列的映射表
EnumMap	一种键值属于枚举类型的映射表
LinkedHashMap	一种可以记住键/值项添加次序的映射表
WeakHashMap	一种其值无用武之地后可以被垃圾回收器回收的映射表
IdentityHashMap	一种用==，而不是用equals比较键值的映射表

### 13.2.1 链表

在本书中，有很多示例已经使用了数组以及动态的ArrayList类。然而，数组和数组列表都有一个重大的缺陷。这就是从数组的中间位置删除一个元素要付出很大的代价，其原因是数组中处于被删除元素之后的所有元素都要向数组的前端移动（见图13-4）。在数组中间的位置上插入一个元素也是如此。

另外一个大家非常熟悉的数据结构——链表（linked list）解决了这个问题。尽管数组在连续的存储位置上存放对象引用，但链表却将每个对象存放在独立的结点中。每个结点还存放着序列中下一个结点的引用。在Java程序设计语言中，所有链表实际上都是双向链接的（doubly linked）——即每个结点还存放着指向前驱结点的引用（见图13-5）。

从链表中间删除一个元素是一个很轻松的操作，即需要对被删除元素附近的结点更新一下即可（见图13-6）。

你也许曾经在数据结构课程中学习过如何实现链表的操作。在链表中添加或删除元素时，绕来绕去的指针可能已经给人们留下了极坏的印象。如果真是如此的话，就会为Java集合类库提供一个类LinkedList而感到拍手称快。

在下面的代码示例中，先添加3个元素，然后再将第2个元素删除：

```
List<String> staff = new LinkedList<String>(); // LinkedList implements List
```



图13-4 从数组中删除一个元素

```
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
Iterator iter = staff.iterator();  
String first = iter.next(); // visit first element  
String second = iter.next(); // visit second element  
iter.remove(); // remove last visited element
```

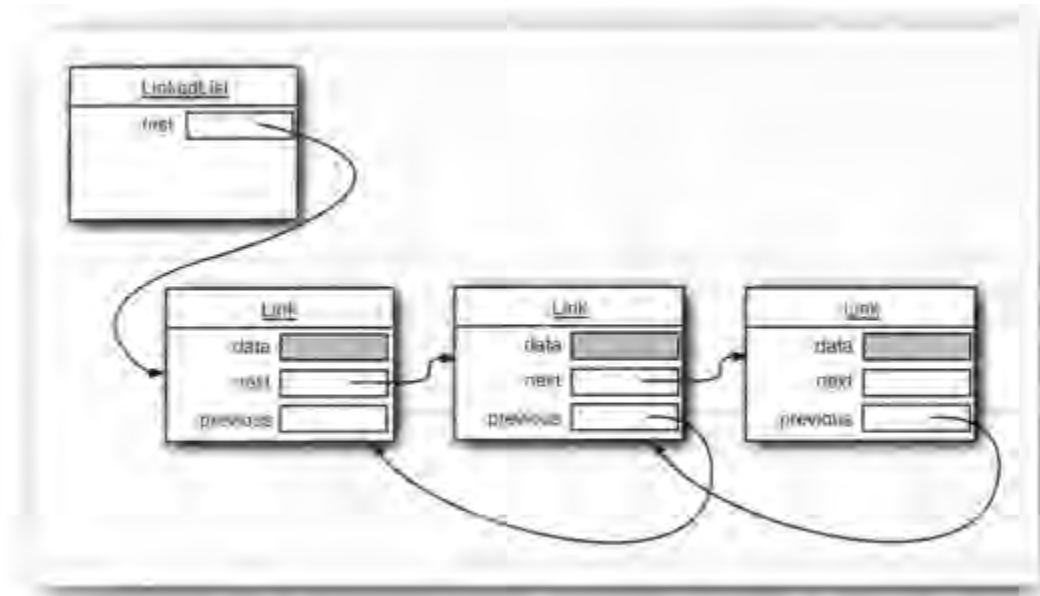


图13-5 双向链表

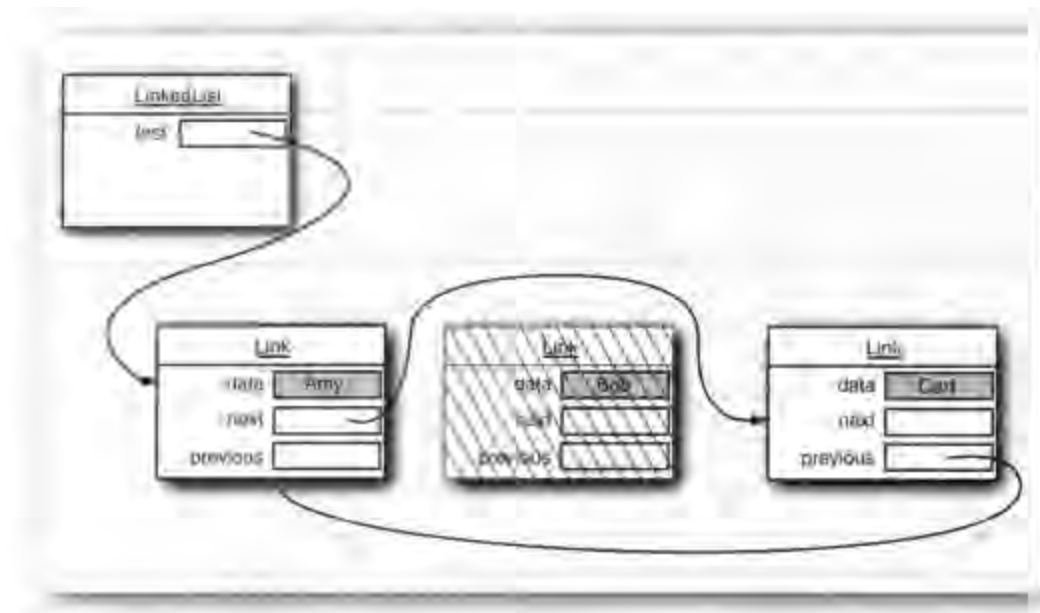


图13-6 从链表中删除一个元素

但是，链表与泛型集合之间有一个重要的区别。链表是一个有序集合（ordered collection），

每个对象的位置十分重要。LinkedList.add 方法将对象添加到链表的尾部。但是，常常需要将元素添加到链表的中间。由于迭代器是描述集合中位置的，所以这种依赖于位置的add方法将由迭代器负责。只有对自然有序的集合使用迭代器添加元素才有实际意义。例如，下一节将要讨论的集（set）类型，其中的元素完全无序。因此，在Iterator 接口中就没有add方法。相反地，集合类库提供了子接口ListIterator，其中包含add方法：

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    ...
}
```

与Collection.add不同，这个方法不返回boolean类型的值，它假定添加操作总会改变链表。另外，ListIterator接口有两个方法，可以用来反向遍历链表。

```
E previous()
boolean hasPrevious()
```

与next方法一样，previous方法返回越过的对象。

LinkedList类的listIterator方法返回一个实现了ListIterator接口的迭代器对象。

```
ListIterator<String> iter = staff.listIterator();
```

Add方法在迭代器位置之前添加一个新对象。例如，下面的代码将越过链表中的第一个元素，并在第二个元素之前添加“Juliet”（见图13-7）：

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```

如果多次调用add方法，将按照提供的次序把元素添加到链表中。它们被依次添加到迭代器当前位置之前。

当用一个刚刚由Iterator方法返回，并且指向链表表头的迭代器调用add操作时，新添加的元素将变成列表的新表头。当迭代器越过链表的最后一个元素时（即hasNext返回false），添加的元素将变成列表的新表尾。如果链表有n个元素，有n+1个位置可以添加新元素。这些位置与迭代器的n+1个可能的位置相对应。例如，如果链表包含3个元素，A、B、C，就有4个位置（标有|）可以插入新元素：

```
|ABC
A|BC
AB|C
ABC|
```



注释：在用“光标”类比时要格外小心。remove操作与BACKSPACE键的工作方式不太一样。在调用next之后，remove方法确实与BACKSPACE键一样删除了迭代器左侧的元素。但是，如果调用previous就会将右侧的元素删除掉，并且不能在同一行中调用两次remove。

add方法只依赖于迭代器的位置，而remove方法依赖于迭代器的状态。

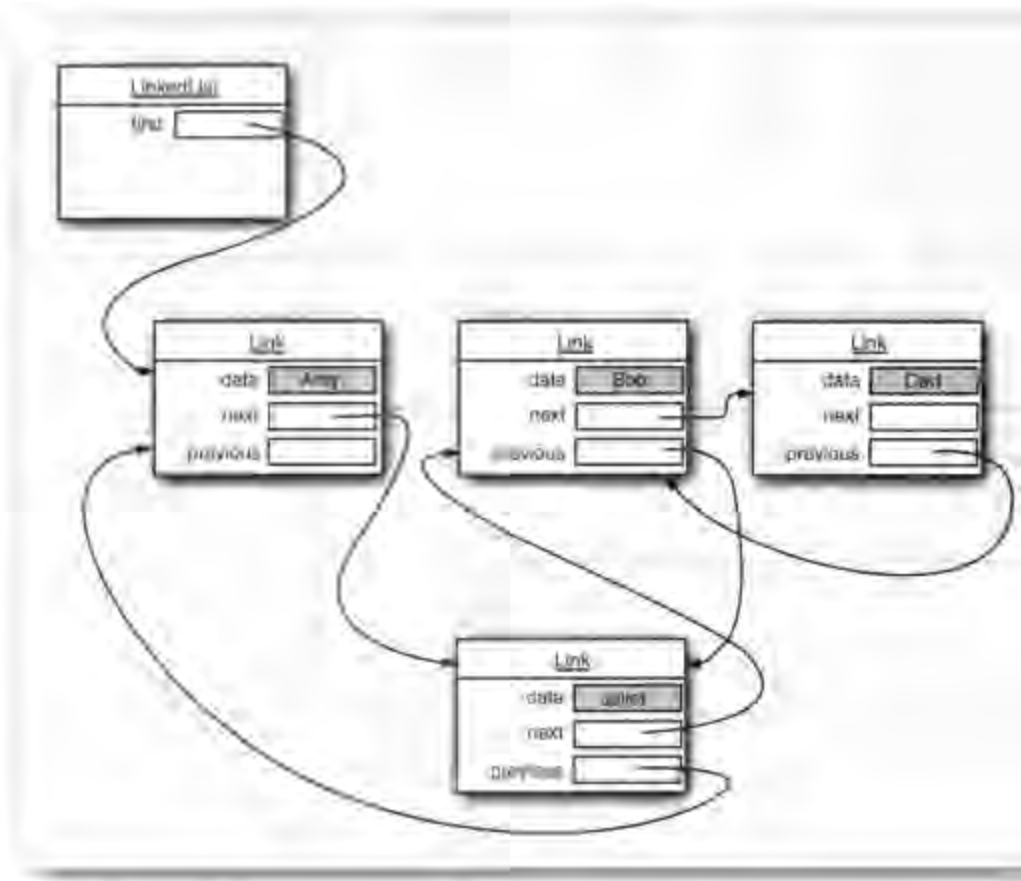


图13-7 将一个元素添加到链表中

最后需要说明，set方法用一个新元素取代调用next或previous方法返回的上一个元素。例如，下面的代码将用一个新值取代链表的第一个元素：

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // returns first element
iter.set(newValue); // sets first element to newValue
```

可以想像，如果在某个迭代器修改集合时，另一个迭代器对其进行遍历，一定会出现混乱的状况。例如，一个迭代器指向另一个迭代器刚刚删除的元素前面，现在这个迭代器就是无效的，并且不应该再使用。链表迭代器的设计使它能够检测到这种修改。如果迭代器发现它的集合被另一个迭代器修改了，或是被该集合自身的方法修改了，就会抛出一个ConcurrentModificationException异常。例如，看一看下面这段代码：

```
List<String> list = . . . ;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

由于iter2检测出这个链表被从外部修改了，所以对iter2.next的调用抛出了一个 ConcurrentModificationException异常。

为了避免发生并发修改的异常，请遵循下述简单规则：可以根据需要给容器附加许多的迭代器，但是这些迭代器只能读取列表。另外，再单独附加一个既能读又能写的迭代器。

有一种简单的方法可以检测到并发修改的问题。集合可以跟踪改写操作（诸如添加或删除元素）的次数。每个迭代器都维护一个独立的计数值。在每个迭代器方法的开始处检查自己改写操作的计数值是否与集合的改写操作计数值一致。如果不一致，抛出一个ConcurrentModificationException异常。



注释：对于并发修改列表的检测有一个奇怪的例外。链表只负责跟踪对列表的结构修改，例如，添加元素、删除元素。set操作不被视为结构性修改。可以将多个迭代器附加给一个链表，所有的迭代器都调用set方法对现有结点的内容进行修改。在本章后面所介绍的Collections类的许多算法都需要使用这个功能。

现在已经介绍了LinkedList类的各种基本方法。可以使用ListIterator类从前后两个方向遍历链表中的元素，并可以添加、删除元素。

在上一节已经看到，Collection接口中声明了许多用于对链表操作的有用方法。其中大部分方法都是在LinkedList类的超类AbstractCollection中实现的。例如，toString方法调用了所有元素的toString，并产生了一个很长的格式为 [A,B, C]的字符串。这为调试工作提供了便利。可以使用contains方法检测某个元素是否出现在链表中。例如，如果链表中包含一个等于“Harry”的字符串，调用staff.contains(“Harry”)后将会返回true。

在Java类库中，还提供了许多在理论上存在一定争议的方法。链表不支持快速地随机访问。如果要查看链表中第n个元素，就必须从头开始，越过n - 1个元素。没有捷径可走。鉴于这个原因，在程序需要采用整数索引访问元素时，程序员通常不选用链表。

尽管如此，LinkedList类还是提供了一个用来访问某个特定元素的get方法：

```
LinkedList<String> list = . . . ;  
String obj = list.get(n);
```

当然，这个方法的效率并不高。如果发现自己正在使用这个方法，说明有可能对于所要解决的问题使用了错误的数据结构。

绝对不应该使用这种让人误解的随机访问方法来遍历链表。下面这段代码的效率极低：

```
for (int i = 0; i < list.size(); i++)  
    do something with list.get(i);
```

每次查找一个元素都要从列表的头部重新开始搜索。LinkedList对象根本不做任何缓存位置信息的操作。



注释：get方法做了微小的优化：如果索引大于size() / 2就从列表尾端开始搜索元素。

列表迭代器接口还有一个方法，可以告之当前位置的索引。实际上，从概念上讲，由于Java迭代器指向两个元素之间的位置，所以可以同时产生两个索引：nextIndex方法返回下一次

调用next方法时返回元素的整数索引；previousIndex方法返回下一次调用previous方法时返回元素的整数索引。当然，这个索引只比nextIndex返回的索引值小1。这两个方法的效率非常高，这是因为迭代器保持着当前位置的计数值。最后需要说一下，如果有一个整数索引n，list.listIterator(n) 将返回一个迭代器，这个迭代器指向索引为n的元素前面的位置。也就是说，调用next与调用list.get(n)会产生同一个元素，只是获得这个迭代器的效率比较低。

如果链表中只有很少几个元素，就完全没有必要为get方法和set方法的开销而烦恼。但是，为什么要优先使用链表呢？使用链表的惟一理由是尽可能地减少在列表中间插入或删除元素所付出的代价。如果列表只有少数几个元素，就完全可以使用ArrayList。

我们建议避免使用以整数索引表示链表中位置的所有方法。如果需要对集合进行随机访问，就使用数组或ArrayList，而不要使用链表。

例13-1中的程序使用的就是链表。它简单地创建了两个链表，将它们合并在一起，然后从第二个链表中每间隔一个元素删除一个元素，最后测试removeAll方法。建议跟踪一下程序流程，并要特别注意迭代器。从这里会发现绘制一个下面这样的迭代器位置示意图是非常有用的：

```
|ACE  |BDFG
A|CE  |BDFG
AB|CE B|DFG
...
```

注意调用

```
System.out.println(a);
```

通过调用AbstractCollection类中的toString方法打印出链表a中的所有元素。

#### 例13-1 LinkedListTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates operations on linked lists.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class LinkedListTest
9. {
10.     public static void main(String[] args)
11.     {
12.         List<String> a = new LinkedList<String>();
13.         a.add("Amy");
14.         a.add("Carl");
15.         a.add("Erica");
16.
17.         List<String> b = new LinkedList<String>();
18.         b.add("Bob");
19.         b.add("Doug");
20.         b.add("Frances");
21.         b.add("Gloria");
22.
23.         // merge the words from b into a
24.
```

```
25.     ListIterator<String> aIter = a.listIterator();
26.     Iterator<String> bIter = b.iterator();
27.
28.     while (bIter.hasNext())
29.     {
30.         if (aIter.hasNext()) aIter.next();
31.         aIter.add(bIter.next());
32.     }
33.
34.     System.out.println(a);
35.
36.     // remove every second word from b
37.
38.     bIter = b.iterator();
39.     while (bIter.hasNext())
40.     {
41.         bIter.next(); // skip one element
42.         if (bIter.hasNext())
43.         {
44.             bIter.next(); // skip next element
45.             bIter.remove(); // remove that element
46.         }
47.     }
48.
49.     System.out.println(b);
50.
51.     // bulk operation: remove all words in b from a
52.
53.     a.removeAll(b);
54.
55.     System.out.println(a);
56. }
57. }
```

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

- `ListIterator<E> listIterator()`  
返回一个列表迭代器，以使用来访问列表中的元素。
- `ListIterator<E> listIterator(int index)`  
返回一个列表迭代器，以使用来访问列表中的元素，这个元素是第一次调用`next`返回的给定索引的元素。
- `void add(int i, E element)`  
在给定位置添加一个元素。
- `void addAll(int i, Collection<? extends E> elements)`  
将某个集合中的所有元素添加到给定位置。
- `E remove(int i)`  
删除给定位置的元素并返回这个元素。
- `E get(int i)`  
获取给定位置的元素。

- `E set(int i, E element)`  
用新元素取代给定位置的元素，并返回原来那个元素。
- `int indexOf(Object element)`  
返回与指定元素相等的元素在列表中第一次出现的位置，如果没有这样的元素将返回 - 1。
- `int lastIndexOf(Object element)`  
返回与指定元素相等的元素在列表中最后一次出现的位置，如果没有这样的元素将返回 - 1。

**API** `java.util.ListIterator<E> 1.2`

- `void add(E newElement)`  
在当前位置前添加一个元素。
- `void set(E newElement)`  
用新元素取代`next`或`previous`上次访问的元素。如果在`next`或`previous`上次调用之后列表结构被修改了，将抛出一个`IllegalStateException` 异常。
- `boolean hasPrevious()`  
当反向迭代列表时，还有可供访问的元素，返回`true`。
- `E previous()`  
返回前一个对象。如果已经到达了列表的头部，就抛出一个 `NoSuchElementException`异常。
- `int nextIndex()`  
返回下一次调用`next`方法时将返回的元素索引。
- `int previousIndex()`  
返回下一次调用`previous` 方法时将返回的元素索引。

**API** `java.util.LinkedList<E> 1.2`

- `LinkedList()`  
构造一个空链表。
- `LinkedList(Collection<? extends E> elements)`  
构造一个链表，并将集合中所有的元素添加到这个链表中。
- `void addFirst(E element)`
- `void addLast(E element)`  
将某个元素添加到列表的头部或尾部。
- `E getFirst()`
- `E getLast()`  
返回列表头部或尾部的元素。
- `E removeFirst()`
- `E removeLast()`  
删除并返回列表头部或尾部的元素。



### 13.2.2 数组列表

在上一节中，介绍了List接口和实现了这个接口的LinkedList类。List接口用于描述一个有序集合，并且集合中每个元素的位置十分重要。有两种访问元素的协议：一种是用迭代器，另一种是用get和set方法随机地访问每个元素。后者不适用于链表，但对数组却很有用。集合类库提供了一种大家熟悉的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，在需要动态数组时，可能会使用Vector类。为什么要用ArrayList取代Vector呢？原因很简单：Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象。但是，如果由一个线程访问Vector，代码要在同步操作上耗费大量的时间。这种情况还是很常见的。而ArrayList方法不是同步的，因此，建议在不需要同步时使用ArrayList，而不要使用Vector。

### 13.2.3 散列集

链表和数组可以按照人们的意愿排列元素的次序。但是，如果想要查看某个指定的元素，却又忘记了它的位置，就需要访问所有元素，直到找到为止。如果集合中包含的元素很多，将会消耗很多时间。如果不在意元素的顺序，可以有几种能够快速查找元素的数据结构。其缺点是无法控制元素出现的次序。它们将按照有利于其操作目的的原则组织数据。

有一种众所周知的数据结构，可以快速地查找所需要的对象，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）。散列码是由对象的实例域产生的一个整数。更准确地说，具有不同数据域的对象将产生不同的散列码。表13-2列出了几个散列码的示例，它们是由String类的hashCode方法产生的。

表13-2 由hashCode函数导出的散列码

串	散 列 码
" Lee "	76268
" lee "	107020
" eel "	100300

如果自定义类，就要负责实现这个类的hashCode方法。有关hashCode方法的详细内容请参看第5章。注意，自己实现的hashCode方法应该与equals方法兼容，即如果a.equals(b)为true，a与b必须具有相同的散列码。

现在，最重要的问题是散列码要能够快速地计算出来，并且这个计算只与要散列的对象状态有关，与散列表中的其他对象无关。

在Java中，散列表用链表数组实现。每个列表被称为桶（bucket）（参看图13-8）。要想查找表中对象的位置，就要先计算它的散列码，然后与桶的总数取余，所得到的结果就是保存这个元素的桶的索引。例如，如果某个对象的散列码为76268，并且有128个桶，对象应该保存在第108号桶中（76268除以128余108）。或许会很幸运，在这个桶中没有其他元素，此时将元素直接插入到桶中就可以了。当然，有时候会遇到桶被占满的情况，这也是不可避免的。这种现象被称为散列冲突（hash collision）。这时，需要用新对象与桶中的所有对象进行比较，查看这个对象是否已经存在。如果散列码是合理且随机分布的，桶的数目也足够大，需要比较的次数

就会很少。

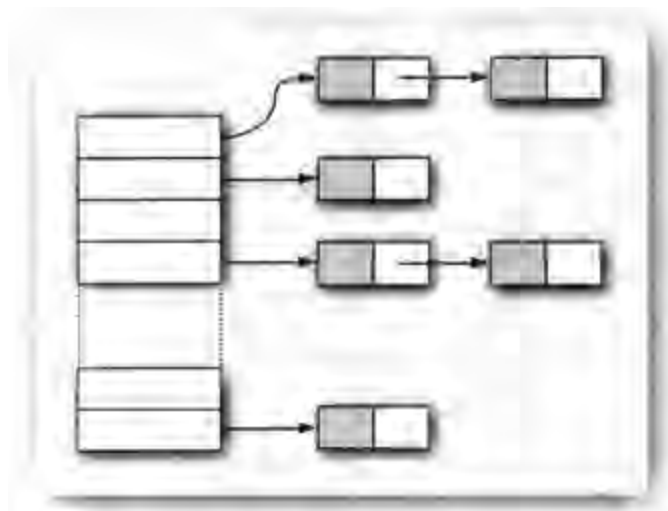


图13-8 散列表

如果想更多地控制散列表的运行性能，就要指定一个初始的桶数。桶数是指用于收集具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，就会增加冲突的可能性，降低运行性能。

如果大致知道最终会有多少个元素要插入到散列表中，就可以设置桶数。通常，将桶数设置为预计元素个数的75%~150%。有些研究人员认为：尽管还没有确凿的证据，但最好将桶数设置为一个素数，以防键的集聚。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被自动地转换为2的下一个幂）。

当然，并不是总能够知道需要存储多少个元素的，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，并将所有元素插入到这个新表中，然后丢弃原来的表。装填因子（load factor）决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），而表中超过75%的位置已经填入元素，这个表就会用双倍的桶数自动地进行再散列。对于大多数应用程序来说，装填因子为75%是比较合理的。

散列表可以用于实现几个重要的数据结构。其中最简单的是set类型。set是没有重复元素的元素集合。set的add方法首先在集中查找要添加的对象，如果不存在，就将这个对象添加进去。

Java集合类库提供了一个HashSet类，它实现了基于散列表的集。可以用add方法添加元素。contains方法已经被重新定义，用来快速地查看是否某个元素已经出现在集中。它只在某个桶中查找元素，而不必查看集中的所有元素。

散列集迭代器将依次访问所有的桶。由于散列将元素分散在表的各个位置上，所以访问它们的顺序几乎是随机的。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（例13-2）将从System.in读取单词，然后将它们添加到集中，最后，再打印出集中的所有单词。例如，可以将Alice in Wonderland（可以从<http://www.gutenberg.net>找到）的文本输入到这个程序中，并通过下列命令运行：

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

这个程序将读取输入的所有单词，并且将它们添加到散列集中。然后遍历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，元素在数据结构中的位置也会发生变化。

#### 例13-2 SetTest.java

```

1. import java.util.*;
2.
3. /**
4.  * This program uses a set to print all unique words in System.in.
5.  * @version 1.10 2003-08-02
6.  * @author Cay Horstmann
7.  */
8. public class SetTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Set<String> words = new HashSet<String>(); // HashSet implements Set
13.         long totalTime = 0;
14.
15.         Scanner in = new Scanner(System.in);
16.         while (in.hasNext())
17.         {
18.             String word = in.next();
19.             long callTime = System.currentTimeMillis();
20.             words.add(word);
21.             callTime = System.currentTimeMillis() - callTime;
22.             totalTime += callTime;
23.         }
24.
25.         Iterator<String> iter = words.iterator();
26.         for (int i = 1; i <= 20; i++)
27.             System.out.println(iter.next());
28.         System.out.println(". . .");
29.         System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
30.     }
31. }
```



#### java.util.HashSet<E> 1.2

- HashSet()
 

构造一个空散列表。
- HashSet(Collection<? extends E> elements)
 

构造一个散列集，并将集合中的所有元素添加到这个散列集中。
- HashSet(int initialCapacity)
 

构造一个空的具有指定容量（桶数）的散列集。

- `HashSet(int initialCapacity, float loadFactor)`  
构造一个具有指定容量和装填因子（一个0.0~1.0之间的数值，确定散列表填充的百分比，当大于这个百分比时，散列表进行再散列）的空散列集。

**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()`。

## 13.2.4 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到集合中。在对集合进行遍历时，每个值将自动地按照排序后的顺序呈现。例如，假设插入3个字符串，然后访问添加的所有元素。

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

这时，每个值将按照顺序打印出来：Amy Bob Carl。正如`TreeSet`类名所示，排序是用树结构完成的（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看《Introduction to Algorithms》，作者是Thomas Cormen、Charles Leiserson、Ronald Rivest和Clifford Stein [The MIT Press, 2001]）<sup>⊖</sup> 每次将一个元素添加到树中时，都被放置在正确的排序位置上。因此，迭代器总是以排好序的顺序访问每个元素。

将一个元素添加到树中要比添加到散列表中慢，但是，与将元素添加到数组或链表的正确位置上相比还是快很多的。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。

因此，将一个元素添加到`TreeSet`中要比添加到`HashSet`中慢。请参看表13-3。不过，`TreeSet`可以自动地对元素进行排序。

表13-3 将元素添加到散列集和树集

文 档	单词总数	不同的单词个数	HashSet	TreeSet
Alice in Wonderland	28195	5909	5秒	7秒
The Count of Monte Cristo	466300	37545	75秒	98秒

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

- `TreeSet()`  
构造一个空树集。

⊖ 本书中文版《算法导论》已由机械工业出版社出版。——编辑注

- `TreeSet(Collection<? extends E> elements)`  
构造一个树集，并将集合中的所有元素添加到树集中。

### 13.2.5 对象的比较

`TreeSet`如何知道希望元素怎样排列呢？在默认情况时，树集假定插入的元素实现了`Comparable`接口。这个接口定义了一个方法：

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

如果`a`与`b`相等，调用`a.compareTo(b)`一定返回0；如果排序后`a`位于`b`之前，则返回负值；如果`a`位于`b`之后，则返回正值。具体返回什么值并不重要，关键是符号（>0、0或<0）。有些标准的Java平台类实现了`Comparable`接口，例如，`String`类。这个类的`compareTo`方法依据字典序（有时称为词典序）对字符串进行比较。

如果要插入自定义的对象，就必须通过实现`Comparable`接口自定义排列顺序。在`Object`类中，没有提供任何`compareTo`接口的默认实现。

例如，下面的代码展示了如何用部件编号对`Item`对象进行排序：

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    ...
}
```

如果对两个正整数进行比较，就像上面示例中的部件编号，就可以直接地返回它们的差。如果第一项<sup>⊖</sup>位于第二项的前面，就返回负值；如果部件编号相同就返回0；否则返回正值。



**警告：**只有整数在一个足够小的范围内，才可以使用这个技巧。如果`x`是一个较大的正整数，`y`是一个较大的负整数，`x - y`有可能会溢出。

然而，使用`Comparable`接口定义排列排序显然有其局限性。对于一个给定的类，只能够实现这个接口一次。如果在一个集合中需要按照部件编号进行排序，在另一个集合中却要按照描述信息进行排序，该怎么办呢？另外，如果需要对一个类的对象进行排序，而这个类的创建者又没有费心实现`Comparable`接口，又该怎么办呢？

在这种情况下，可以通过将`Comparator`对象传递给`TreeSet`构造器来告诉树集使用不同的比较方法。`Comparator`接口声明了一个带有两个显式参数的`compare`方法：

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

---

⊖ 指产品。——译者注

与compareTo方法一样，如果a位于b之前compare方法则返回负值；如果a和b相等则返回0，否则返回正值。

如果按照描述信息进行排序，就直接定义一个实现Comparator接口的类：

```
class ItemComparator implements Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
}
```

然后将这个类的对象传递给树集的构造器：

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

如果构造了一棵带比较器的树，就可以在需要比较两个元素时使用这个对象。

注意，这个比较器没有任何数据。它只是比较方法的持有器。有时将这种对象称为函数对象（function object）。函数对象通常被定义为“瞬时的”，即匿名内部类的实例：

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {
            String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
        }
    });
```



注释：实际上，Comparator<T>接口声明了两个方法：compare和equals。当然，每一个类都有一个 equals 方法；因此，为这个接口声明再添加一个equals方法似乎没有太大好处。API文档解释说，不需要覆盖equals方法，但这样做可能会在某些情况下提高性能。例如，如果从另一个集合添加元素，这个由使用相同比较器的另外一个集添加元素，TreeSet类中的addAll方法的效率会更高。

回头看一看表13-3可能会疑虑：是否总是应该用树集取代散列集。毕竟，添加一个元素所花费的时间看上去并不很长，而且元素是自动排序的。到底应该怎样做将取决于所要收集的数据。如果不需要对数据进行排序，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，对其排序要比散列函数更加困难。散列函数只是将对象适当地打乱存放，而比较却要精确地判别每个对象。

要想具体地了解它们之间的差异，还需要研究一个收集矩形集的任务。如果使用 TreeSet，就需要提供Comparator<Rectangle>。如何比较两个矩形呢？比较面积吗？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。树的排序必须是整体排序。也就是说，任意两个元素必须是可比的，并且只有在两个元素相等时结果才为0。确实，有一种矩形的排序

(按照坐标的词典顺序排列)方式,但它的计算很牵强且很繁琐。相反地,Rectangle类已经定义了散列函数,它直接对坐标进行散列。



注释:从Java SE 6起,TreeSet类实现了NavigableSet接口。这个接口增加了几个便于定位元素以及反向遍历的方法。详细信息请参看API注释。

在例13-3的程序中创建了两个Item对象的树集。第一个按照部件编号排序,这是Item对象的默认顺序。第二个通过使用一个定制的比较器来按照描述信息排序。

### 例13-3 TreeSetTest.java

```
1. /**
2.  @version 1.10 2004-08-02
3.  @author Cay Horstmann
4.  */
5.
6. import java.util.*;
7.
8. /**
9.  This program sorts a set of items by comparing
10. their descriptions.
11. */
12. public class TreeSetTest
13. {
14.     public static void main(String[] args)
15.     {
16.         SortedSet<Item> parts = new TreeSet<Item>();
17.         parts.add(new Item("Toaster", 1234));
18.         parts.add(new Item("Widget", 4562));
19.         parts.add(new Item("Modem", 9912));
20.         System.out.println(parts);
21.
22.         SortedSet<Item> sortByDescription = new TreeSet<Item>(new
23.             Comparator<Item>()
24.             {
25.                 public int compare(Item a, Item b)
26.                 {
27.                     String descrA = a.getDescription();
28.                     String descrB = b.getDescription();
29.                     return descrA.compareTo(descrB);
30.                 }
31.             });
32.
33.         sortByDescription.addAll(parts);
34.         System.out.println(sortByDescription);
35.     }
36. }
37.
38. /**
39.  An item with a description and a part number.
40. */
41. class Item implements Comparable<Item>
42. {
43.     /**
44.     Constructs an item.
```

```
45.     @param aDescription the item's description
46.     @param aPartNumber the item's part number
47.     */
48.     public Item(String aDescription, int aPartNumber)
49.     {
50.         description = aDescription;
51.         partNumber = aPartNumber;
52.     }
53.
54.     /**
55.      * Gets the description of this item.
56.      * @return the description
57.      */
58.     public String getDescription()
59.     {
60.         return description;
61.     }
62.
63.     public String toString()
64.     {
65.         return "[description=" + description
66.             + ", partNumber=" + partNumber + "]";
67.     }
68.
69.     public boolean equals(Object otherObject)
70.     {
71.         if (this == otherObject) return true;
72.         if (otherObject == null) return false;
73.         if (getClass() != otherObject.getClass()) return false;
74.         Item other = (Item) otherObject;
75.         return description.equals(other.description)
76.             && partNumber == other.partNumber;
77.     }
78.
79.     public int hashCode()
80.     {
81.         return 13 * description.hashCode() + 17 * partNumber;
82.     }
83.
84.     public int compareTo(Item other)
85.     {
86.         return partNumber - other.partNumber;
87.     }
88.
89.     private String description;
90.     private int partNumber;
91. }
```

**API** java.lang.Comparable<T> 1.2

## • int compareTo(T other)

将这个对象（this）与另一个对象（other）进行比较，如果this位于other之前则返回负值；如果两个对象在排列顺序中处于相同的位置则返回0；如果this位于other之后则返回正值。



**API** java.util.Comparator<T> 1.2

- `int compare(T a, T b)`

将两个对象进行比较，如果a位于b之前则返回负值；如果两个对象在排列顺序中处于相同的位置则返回0；如果a位于b之后则返回正值。

**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()`

返回一个按照递减顺序遍历集中元素的迭代器。

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

- `TreeSet()`

构造一个用于排列Comparable对象的树集。

- `TreeSet(Comparator<? super E> c)`

构造一个树集，并使用指定的比较器对其中的元素进行排序。

- `TreeSet(SortedSet<? extends E> elements)`

构造一个树集，将有序集中的所有元素添加到这个树集中，并使用与给定集相同的元素比较器。

### 13.2.6 队列与双端队列

前面已经讨论过，队列可以让人们有效地在尾部添加一个元素，在头部删除一个元素。有两个端头的队列，即双端队列，可以让人们有效地在头部和尾部同时添加或删除元素。不支持在队列中间添加元素。在Java SE 6中引入了Deque接口，并由ArrayDeque和LinkedList类实现。这两个类都提供了双端队列，而且在必要时可以增加队列的长度。在第14章将会看到有限队列和有限双端队列。

**API** java.util.Queue<E> 5.0

- 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双端队列的尾部并返回true。如果队列满了，第一个方法将抛出一个IllegalStateException，而第二个方法返回false。

- E remove()
- E poll()

假如队列不空，删除并返回这个队列头部的元素。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()

如果队列不空，返回这个队列头部的元素，但不删除。如果队列空，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头部或尾部。如果队列满了，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()

如果队列不空，删除并返回队列头部的元素。如果队列为空，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()`

如果队列非空，返回队列头部的元素，但不删除。如果队列空，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双端队列。

### 13.2.7 优先级队列

优先级队列（priority queue）中的元素可以按照任意的顺序插入，却总是按照排序的顺序进行检索。也就是说，无论何时调用 `remove` 方法，总会获得当前优先级队列中最小的元素。然而，优先级队列并没有对所有的元素进行排序。如果用迭代的方式处理这些元素，并不需要对它们进行排序。优先级队列使用了一个优雅且高效的数据结构，称为堆（heap）。堆是一个可以自我调整的二叉树，对树执行添加（`add`）和删除（`remove`）操作，可以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，一个优先级队列既可以保存实现了 `Comparable` 接口的类对象，也可以保存在构造器中提供比较器的对象。

使用优先级队列的典型示例是任务调度。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，都将优先级最高的任务从队列中删除（由于习惯上将1设为“最高”优先级，所以会将最小的元素删除）。

例13-4显示了一个正在运行的优先级队列。与 `TreeSet` 中的迭代不同，这里的迭代并不是按照元素的排列顺序访问的。而删除却总是删掉剩余元素中优先级数最小的那个元素。

#### 例13-4 `PriorityQueueTest.java`

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of a priority queue.
5.  * @version 1.00 2004-08-03
6.  * @author Cay Horstmann
7.  */
8. public class PriorityQueueTest
9. {
10.     public static void main(String[] args)
11.     {
12.         PriorityQueue<GregorianCalendar> pq = new PriorityQueue<GregorianCalendar>();
13.         pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
14.         pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
15.         pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
16.         pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse
17.
18.         System.out.println("Iterating over elements...");
19.         for (GregorianCalendar date : pq)
```

```
20.     System.out.println(date.get(Calendar.YEAR));
21.     System.out.println("Removing elements...");
22.     while (!pq.isEmpty())
23.         System.out.println(pq.remove().get(Calendar.YEAR));
24. }
25. }
```

#### java.util.PriorityQueue 5.0

- PriorityQueue()
- PriorityQueue(int initialCapacity)  
构造一个用于存放Comparable对象的优先级队列。
- PriorityQueue(int initialCapacity, Comparator<? super E> c)  
构造一个优先级队列，并用指定的比较器对元素进行排序。

### 13.2.8 映射表

集是一个集合，它可以快速地查找现有的元素。但是，要查看一个元素，需要有要查找元素的精确副本。这不是一种非常通用的查找方式。通常，我们知道某些键的信息，并想要查找与之对应的元素。映射表（map）数据结构就是为此设计的。映射表用来存放键/值对。如果提供了键，就能够查找到值。例如，有一张关于员工信息的记录表，键为员工ID，值为Employee对象。

Java类库为映射表提供了两个通用的实现：HashMap和TreeMap。这两个类都实现了Map接口。

散列映射表对键进行散列，树映射表用键的整体顺序对元素进行排序，并将其组织成搜索树。散列或比较函数只能作用于键。与键关联的值不能进行散列或比较。

应该选择散列映射表还是树映射表呢？与集一样，散列稍微快一些，如果不需要按照排列顺序访问键，就最好选择散列。

下列代码将为存储的员工信息建立一个散列映射表：

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

每当往映射表中添加对象时，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。

要想检索一个对象，必须提供（因而，必须记住）一个键。

```
String s = "987-98-9996";
e = staff.get(s); // gets harry
```

如果在映射表中没有与给定键对应的信息，get将返回null。

键必须是惟一的。不能对同一个键存放两个值。如果对同一个键两次调用put方法，第二个值就会取代第一个值。实际上，put将返回用这个键参数存储的上一个值。

remove方法用于从映射表中删除给定键对应的元素。size方法用于返回映射表中的元素数。

集合框架并没有将映射表本身视为一个集合（其他的数据结构框架则将映射表视为对（pairs）的集合，或者视为用键作为索引的值的集合）。然而，可以获得映射表的视图，这是一组实现了Collection接口对象，或者它的子接口的视图。

有3个视图，它们分别是：键集、值集合（不是集）和键/值对集。键与键/值对形成了一个集，这是因为在映射表中一个键只能有一个副本。下列方法将返回这3个视图（条目集的元素是静态内部类Map.Entry的对象）。

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

注意，keySet既不是HashSet，也不是TreeSet，而是实现了Set接口的某个其他类的对象。Set接口扩展了Collection接口。因此，可以与使用任何集合一样使用keySet。

例如，可以枚举映射表中的所有键：

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



提示：如果想要同时查看键与值，就可以通过枚举各个条目（entries）查看，以避免对值进行查找。可以使用下面这段代码框架：

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

如果调用迭代器的remove方法，实际上就从映射表中删除了键以及对应的值。但是，不能将元素添加到键集的视图中。如果只添加键而不添加值是毫无意义的。如果试图调用add方法，将会抛出一个UnsupportedOperationException异常。条目集视图也有同样的限制，不过，从概念上讲，添加新的键/值对是有意义的。

例13-5显示了映射表的操作过程。首先将键/值对添加到映射表中。然后，从映射表中删除一个键，同时与之对应的值也被删除掉了。接下来，修改与某一个键对应的值，并调用get方法查看这个值。最后，对条目集进行迭代。

#### 例13-5 MapTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of a map with key type String and value type Employee.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
```

```
8. public class MapTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Map<String, Employee> staff = new HashMap<String, Employee>();
13.         staff.put("144-25-5464", new Employee("Amy Lee"));
14.         staff.put("567-24-2546", new Employee("Harry Hacker"));
15.         staff.put("157-62-7935", new Employee("Gary Cooper"));
16.         staff.put("456-62-5527", new Employee("Francesca Cruz"));
17.
18.         // print all entries
19.
20.         System.out.println(staff);
21.
22.         // remove an entry
23.
24.         staff.remove("567-24-2546");
25.
26.         // replace an entry
27.
28.         staff.put("456-62-5527", new Employee("Francesca Miller"));
29.
30.         // look up a value
31.
32.         System.out.println(staff.get("157-62-7935"));
33.
34.         // iterate through all entries
35.
36.         for (Map.Entry<String, Employee> entry : staff.entrySet())
37.         {
38.             String key = entry.getKey();
39.             Employee value = entry.getValue();
40.             System.out.println("key=" + key + ", value=" + value);
41.         }
42.     }
43. }
44.
45. /**
46.  * A minimalist employee class for testing purposes.
47.  */
48. class Employee
49. {
50.     /**
51.      * Constructs an employee with $0 salary.
52.      * @param n the employee name
53.      */
54.     public Employee(String n)
55.     {
56.         name = n;
57.         salary = 0;
58.     }
59.
60.     public String toString()
61.     {
62.         return "[name=" + name + ", salary=" + salary + "]";
63.     }
}
```

```
64.  
65. private String name;  
66. private double salary;  
67. }
```

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

- `V get(K key)`  
获取与键对应的值；返回与键对应的对象，如果在映射表中没有这个对象则返回null。键可以为null。
- `V put(K key, V value)`  
将键与对应的值关系插入到映射表中。如果这个键已经存在，新的对象将取代与这个键对应的旧对象。这个方法将返回键对应的旧值。如果这个键以前没有出现过则返回null。键可以为null，但值不能为null。
- `void putAll(Map<? extends K, ? extends V> entries)`  
将给定映射表中的所有条目添加到这个映射表中。
- `boolean containsKey(Object key)`  
如果在映射表中已经有这个键，返回true。
- `boolean containsValue(Object value)`  
如果映射表中已经有这个值，返回true。
- `Set<Map.Entry<K, V>> entrySet()`  
返回Map.Entry对象的集视图，即映射表中的键/值对。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。
- `Set<K> keySet()`  
返回映射表中所有键的集视图。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。
- `Collection<V> values()`  
返回映射表中所有值的集合视图。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。

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

- `K getKey()`
- `V getValue()`  
返回这个条目的键或值。
- `V setValue(V newValue)`  
设置在映射表中与新值对应的值，并返回旧值。

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

- `HashMap()`

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

用给定的容量和装填因子构造一个空散列映射表（装填因子是一个0.0~1.0之间的数值。这个数值决定散列表填充的百分比。一旦到了这个比例，就要将其再散列到更大的表中）。默认的装填因子是0.75。

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

- `TreeMap(Comparator<? super K> c)`  
构造一个树映射表，并使用一个指定的比较器对键进行排序。
- `TreeMap(Map<? extends K, ? extends V> entries)`  
构造一个树映射表，并将某个映射表中的所有条目添加到树映射表中。
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`  
构造一个树映射表，将某个有序映射表中的所有条目添加到树映射表中，并使用与给定的有序映射表相同的比较器。

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

- `Comparator<? super K> comparator()`  
返回对键进行排序的比较器。如果键是用Comparable接口的compareTo方法进行比较的，返回null。
- `K firstKey()`
- `K lastKey()`  
返回映射表中最小元素和最大元素。

### 13.2.9 专用集与映射表类

在集合类库中，有几个专用的映射表类，本节对它们做一下简要地介绍。

#### 1. 弱散列映射表

设计WeakHashMap类是为了解决一个有趣的问题。如果有一个值，对应的键已经不再使用了，将会出现什么情况呢？假定对某个键的最后一次引用已经消亡，不再有任何途径引用这个值的对象了。但是，由于在程序中的任何部分没有再出现这个键，所以，这个键/值对无法从映射表中删除。为什么垃圾回收器不能够删除它呢？难道删除无用的对象不是垃圾回收器的工作吗？

遗憾的是，事情没有这样简单。垃圾回收器跟踪活动的对象。只要映射表对象是活动的，其中的所有桶也是活动的，它们不能被回收。因此，需要由程序负责从长期存活的映射表中删除那些无用的值。或者使用WeakHashMap完成这件事情。当对键的惟一引用来自散列表条目时，这一数据结构将与垃圾回收器协同工作一起删除键/值对。

下面是这种机制的内部运行情况。WeakHashMap使用弱引用（weak references）保存键。WeakReference对象将引用保存到另外一个对象中，在这里，就是散列表键。对于这种类型的对象，垃圾回收器用一种特有的方式进行处理。通常，如果垃圾回收器发现某个特定的对象已



经没有他人引用了，就将其回收。然而，如果某个对象只能由WeakReference引用，垃圾回收器仍然回收它，但要将引用这个对象的弱引用放入队列中。WeakHashMap将周期性地检查队列，以便找出新添加的弱引用。一个弱引用进入队列意味着这个键不再被他人使用，并且已经被收集起来。于是，WeakHashMap将删除对应的条目。

## 2. 链接散列集和链接映射表

Java SE 1.4增加了两个类：LinkedHashSet和LinkedHashMap，用来记住插入元素项的顺序。这样就可以避免在散列表中的项从表面上看是随机排列的。当条目插入到表中时，就会并入到双向链表中（见图13-9）。

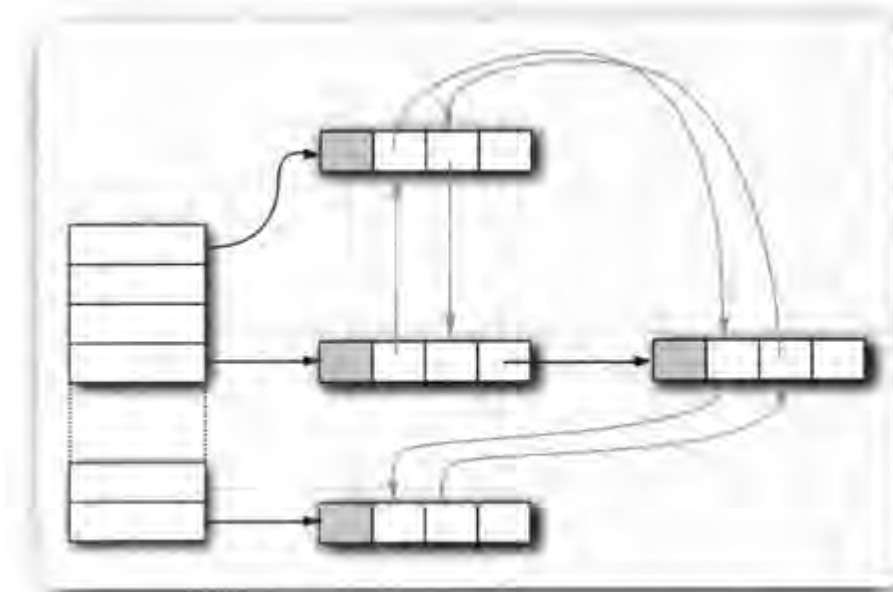


图13-9 链接散列表

例如，在例13-5中包含下列映射表插入的处理：

```
Map staff = new LinkedHashMap();
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()`以下面次序枚举键：

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

并且`staff.values().iterator()`以下列顺序枚举这些值：

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

链接散列映射表将用访问顺序，而不是插入顺序，对映射表条目进行迭代。每次调用get或put，受到影响的条目将从当前的位置删除，并放到条目链表的尾部（只有条目在链表中的位置会受影响，而散列表中的桶不会受影响。一个条目总位于与键散列码对应的桶中）。要项构造这样一个的散列映射表，请调用

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

访问顺序对于实现高速缓存的“最近最少使用”原则十分重要。例如，可能希望将访问频率高的元素放在内存中，而访问频率低的元素则从数据库中读取。当在表中找不到元素项且表又已经满时，可以将迭代器加入到表中，并将枚举的前几个元素删除掉。这些是近期最少使用的几个元素。

甚至可以让这一过程自动化。即构造一个LinkedHashMap的子类，然后覆盖下面这个方法：

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

每当方法返回true时，就添加一个新条目，从而导致删除eldest条目。例如，下面的高速缓存可以存放100个元素：

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

另外，还可以对eldest条目进行评估，以此决定是否应该将它删除。例如，可以检查与这个条目一起存在的时间戳。

### 3. 枚举集与枚举映射表

EnumSet是一个枚举类型元素集的高效实现。由于枚举类型只有有限个实例，所以EnumSet内部用位序列实现。如果对应的值在集中，则相应的位被置为1。

EnumSet类没有公共的构造器。可以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是一个键类型为枚举类型的映射表。它可以直接且高效地用一个值数组实现。在使用时，需要在构造器中指定键类型：

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



注释：在EnumSet的API文档中，将会看到E extends Enum<E>这样奇怪的类型参数。简单地说，它的意思是“E是一个枚举类型。”所有的枚举类型都扩展于泛型Enum类。例如，Weekday扩展于Enum<Weekday>。

#### 4. 标识散列映射表

Java SE 1.4还为另外一个特殊目的增加了另一个类IdentityHashMap。在这个类中，键的散列值不是用hashCode函数计算的，而是用System.identityHashCode方法计算的。这是Object.hashCode方法根据对象的内存地址来计算散列码时所使用的。而且，在对两个对象进行比较时，IdentityHashMap类使用==，而不使用equals。

也就是说，不同的键对象，即使内容相同，也被视为是不同的对象。在实现对象遍历算法（如对象序列化）时，这个类非常有用，可以用来跟踪每个对象的遍历状况。

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

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

用给定的容量和填充因子构造一个空的散列映射表。

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

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

用给定的容量和填充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、填充因子和顺序构造一个空的链接散列映射表。

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

如果想删除eldest元素，并同时返回true，就应该覆盖这个方法。eldest参数是预期要删除的条目。这个方法将在条目添加到映射表之后调用。其默认的实现将返回false。即在默认情况下，旧元素没有被删除。然而，可以重新定义这个方法，以便有选择地返回true。例如，如果最旧的条目符合一个条件，或者映射表超过了一定大小，则返回true。

**API** java.util.EnumSet<E extends Enum<E>> 5.0

- static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)  
返回一个包含给定枚举类型的所有值的集。
- static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)  
返回一个空集，并有足够的空间保存给定的枚举类型所有的值。

- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`  
返回一个包含from~to之间的所有值（包括两个边界元素）的集。
- `static <E extends Enum<E>> EnumSet<E> of(E value)`
- `static <E extends Enum<E>> EnumSet<E> of(E value, E... values)`  
返回包括给定值的集。

**API** `java.util.EnumMap<K extends Enum<K>, V> 5.0`

- `EnumMap(Class<K> keyType)`  
构造一个键为给定类型的空映射集。

**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 $1.5 * \text{expectedMaxSize}$ 的2的最小次幂（`expectedMaxSize`的默认值是21）。

**API** `java.lang.System 1.0`

- `static int identityHashCode(Object obj) 1.1`  
返回`Object.hashCode`计算出来的相同散列码（根据对象的内存地址产生），即使`obj`所属的类已经重新定义了`hashCode`方法也是如此。

### 13.3 集合框架

框架（framework）是一个类的集，它奠定了创建高级功能的基础。框架包含很多超类，这些超类拥有非常有用的功能、策略和机制。框架使用者创建的子类可以扩展超类的功能，而不必重新创建这些基本的机制。例如，Swing就是一种用户界面的机制。

Java集合类库构成了集合类的框架。它为集合的实现者定义了大量的接口和抽象类（见图13-10），并且对其中的某些机制给予了描述，例如，迭代协议。正如前面几节所做的那样，可以使用这些集合类，而不必了解框架。但是，如果想要实现用于多种集合类型的泛型算法，或者是想要增加新的集合类型，了解一些框架的知识是很有帮助的。

集合有两个基本的接口：`Collection`和`Map`。可以使用下列方法向集合中插入元素：

```
boolean add(E element)
```

但是，由于映射表保存的是键/值对，所以可以使用`put`方法进行插入。

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

要想从集合中读取某个元素，就需要使用迭代器访问它们。然而，也可以用`get`方法从映射表读取值：

```
V get(K key)
```

`List`是一个有序集合（ordered collection）。元素可以添加到容器中某个特定的位置。将对

象放置在某个位置上可以采用两种方式：使用整数索引或使用列表迭代器。List接口定义了几个用于随机访问的方法：

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

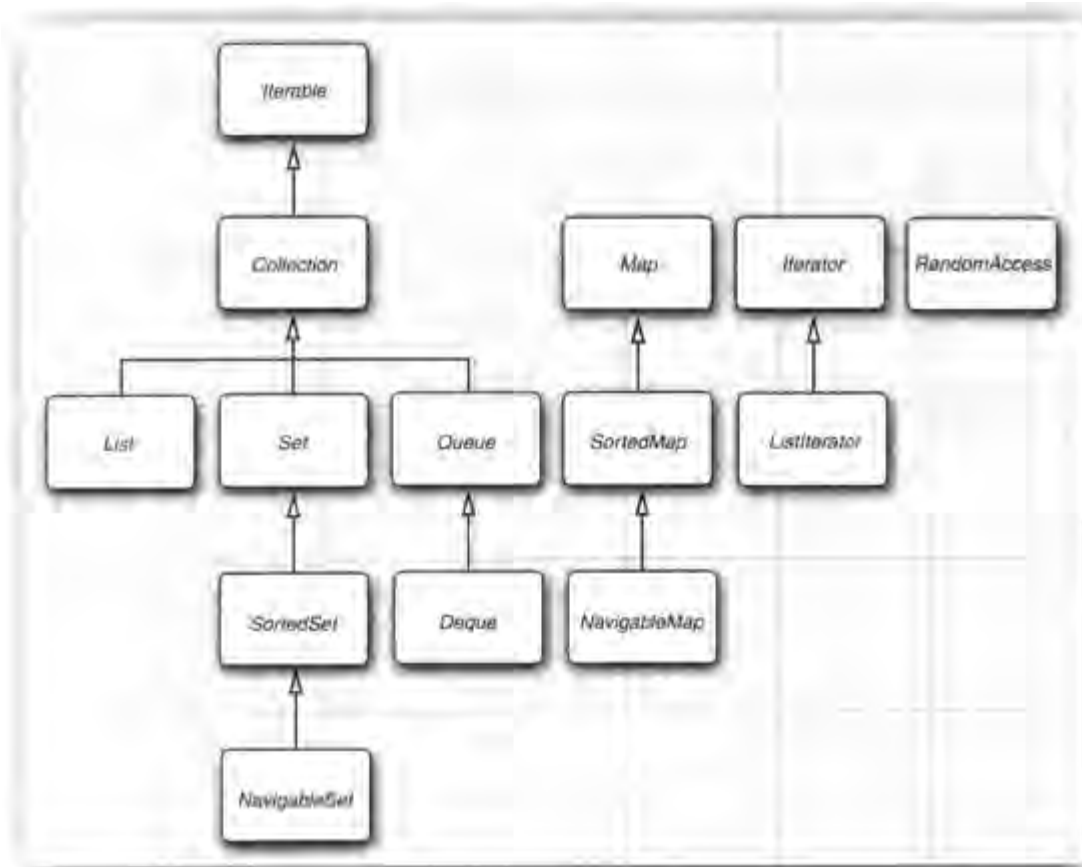


图13-10 集合框架的接口

如同前面所讨论的，List接口在提供这些随机访问方法时，并不管它们对某种特定的实现是否高效。为了避免执行成本较高的随机访问操作，Java SE 1.4引入了一个标记接口RandomAccess。这个接口没有任何方法，但可以用来检测一个特定的集合是否支持高效的随机访问：

```
if (c instanceof RandomAccess)
{
    use random access algorithm
}
else
{
    use sequential access algorithm
}
```

ArrayList类和Vector类都实现了RandomAccess接口。



注释：从理论上讲，有一个独立的Array接口，它扩展于List接口，并声明了随机访问方法是合理的。如果确实有这样一个独立的Array接口，那些需要随机访问的算法就可以使用Array参数，而且也不会无意中将它们应用于随机访问速度很慢的集合了。但是，集合框架的设计者没有选择去定义一个独立的接口，其原因是希望让类库中的接口数量保持在一个较少的水平。此外，不希望对程序员采取一种家长式的作风。这样可以自由地将链表传递给随机访问算法，只是需要清楚这样做会给性能带来什么不良的影响。

ListIterator接口定义了一个方法，用于将一个元素添加到迭代器所处位置的前面：

```
void add(E element)
```

要想获取和删除给定位置的元素，只需要调用Iterator接口中的next方法和remove方法即可。

Set接口与Collection接口是一样的，只是其方法的行为有着更加严谨的定义。集的add方法拒绝添加重复的元素。集的equals方法定义两个集相等的条件是它们包含相同的元素但顺序不必相同。hashCode方法定义应该保证具有相同元素的集将会得到相同的散列码。

既然方法签名是相同的，为什么还要建立一个独立的接口呢？从概念上讲，并不是所有集合都是集。建立Set接口后，可以让程序员编写仅接受集的方法。

SortedSet和SortedMap接口暴露了用于排序的比较器对象，并且定义的方法可以获得集合的子集视图。下一节将讨论这些视图。

最后，Java SE 6引入了接口NavigableSet和NavigableMap，其中包含了几个用于在有序集和映射表中查找和遍历的方法（从理论上讲，这几个方法已经包含在SortedSet和SortedMap的接口中）。TreeSet和TreeMap类实现了这几个接口。

现在，让我们将话题从接口转到实现接口的类上。前面已经讨论过，集合接口有大量的方法，这些方法可以通过更基本的方法加以实现。抽象类提供了许多这样的例行实现：

```
AbstractCollection  
AbstractList  
AbstractSequentialList  
AbstractSet  
AbstractQueue  
AbstractMap
```

如果实现了自己的集合类，就可能要扩展上面某个类，以便可以选择例行操作的实现。

Java类库支持下面几种具体类：

```
LinkedList  
ArrayList  
ArrayDeque  
HashSet  
TreeSet  
PriorityQueue  
HashMap  
TreeMap
```

图13-11展示了这些类之间的关系。

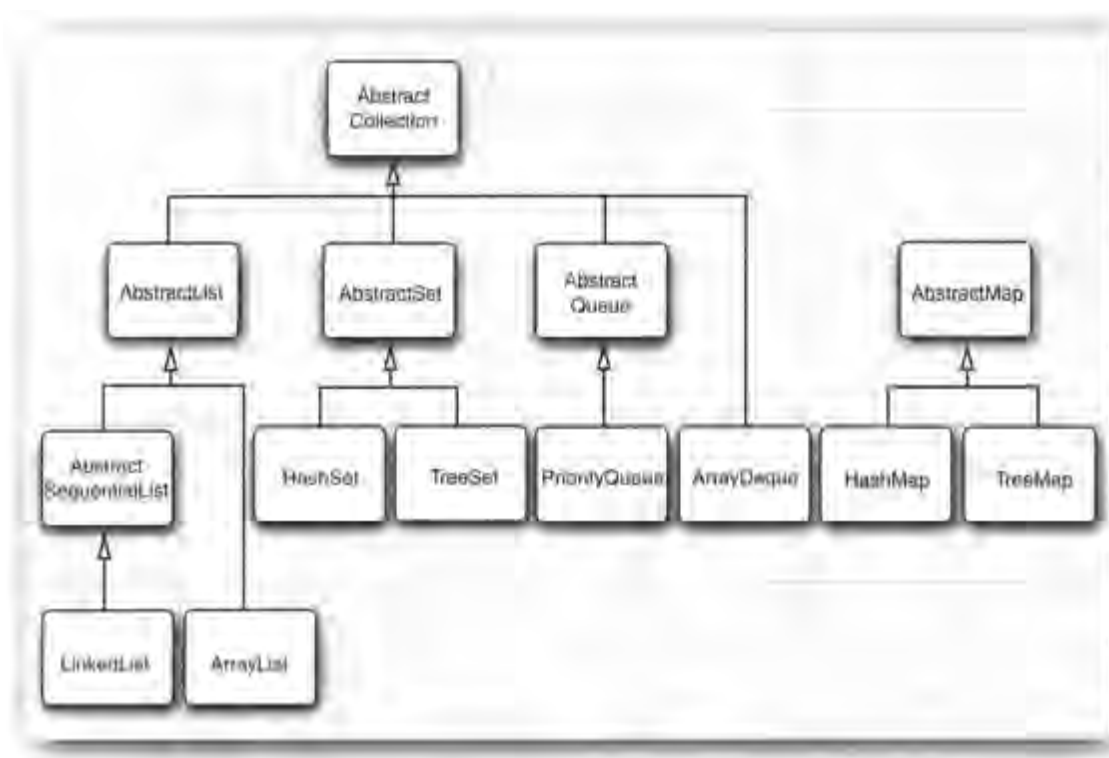


图13-11 集合框架中的类

最后，还有许多Java第一版“遗留”下来的容器类，在集合框架出现就有了，它们是：

Vector  
Stack  
Hashtable  
Properties

这些类已经被集成到集合框架中，见图13-12。本章稍后将会讨论这些类。

### 13.3.1 视图与包装器

看一下图13-10和图13-11可能会感觉：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。然而，这两张图并没有展示出全部的情况。通过使用视图（views）可以获得其他的实现了集合接口和映射表接口的对象。映射表类的keySet方法就是一个这样的示例。初看起来，好像这个方法创建了一个新集，并将映射表中的所有键都填进去，然后返回这个集。但是，情况并非如此。取而代之的是：keySet方法返回一个实现Set接口的类对象，这个类的方法对原映射表进行操作。这种集合称为视图。

视图技术在集框架中有许多非常有用的应用。下面将讨论这些应用。

#### 1. 轻量级集包装器

Arrays类的静态方法asList将返回一个包装了普通Java数组的List包装器。这个方法可以将数组传递给一个期望得到列表或集合变元的方法。例如：

```
Card[] cardDeck = new Card[52];
...
```

```
List<Card> cardList = Arrays.asList(cardDeck);
```

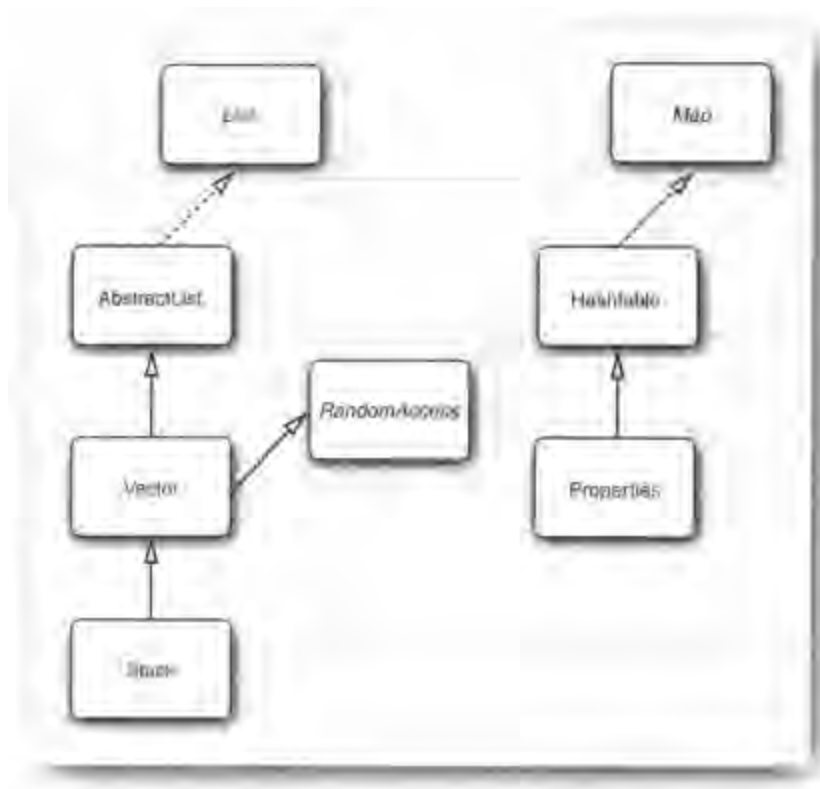


图13-12 集合框架中的遗留类

返回的对象不是`ArrayList`。它是一个视图对象，带有访问底层数组的`get`和`set`方法。改变数组大小的所有方法（例如，与迭代器相关的`add`和`remove`方法）都会抛出一个`UnsupportedOperationException`异常。

从Java SE 5.0开始，`asList`方法声明为一个具有可变数量参数的方法。除了可以传递一个数组之外，还可以将各个元素直接传递给这个方法。例如：

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

这个方法调用

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

将返回一个实现了`List`接口的不可修改的对象，并给人一种包含 $n$ 个元素，每个元素都像是一个`anObject`的错觉。

例如，下面的调用将创建一个包含100个字符串的`List`，每个串都被设置为“`DEFAULT`”：

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

由于字符串对象只存储了一次，所以付出的存储代价很小。这是视图技术的一种巧妙的应用。



注释：`Collections`类包含很多实用方法，这些方法的参数和返回值都是集合。不要将它与`Collection`接口混淆起来。



如果调用下列方法

```
Collections.singleton(anObject)
```

则将返回一个视图对象。这个对象实现了Set接口（与产生List的ncopies方法不同）。返回的对象实现了一个不可修改的单元元素集，而不需要付出建立数据结构的开销。singletonList方法与singletonMap方法类似。

## 2. 子范围

可以为很多集合建立子范围（subrange）视图。例如，假设有一个列表staff，想从中取出第10个～第19个元素。可以使用subList方法来获得一个列表的子范围视图。

```
List group2 = staff.subList(10, 20);
```

第一个索引包含在内，第二个索引则不包含在内。这与String类的substring操作中的参数情况相同。

可以将任何操作应用于子范围，并且能够自动地反映整个列表的情况。例如，可以删除整个子范围：

```
group2.clear(); // staff reduction
```

现在，元素自动地从staff列表中清除了，并且group2为空。

对于有序集和映射表，可以使用排序顺序而不是元素位置建立子范围。SortedSet接口声明了3个方法：

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

这些方法将返回大于等于from且小于to的所有元素子集。有序映射表也有类似的方法：

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

返回映射表视图，该映射表包含键落在指定范围内的所有元素。

Java SE 6引入的NavigableSet接口赋予子范围操作更多的控制能力。可以指定是否包括边界：

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

## 3. 不可修改的视图

Collections还有几个方法，用于产生集合的不可修改视图（unmodifiable views）。这些视图对现有集合增加了一个运行时的检查。如果发现试图对集合进行修改，就抛出一个异常，同时这个集合将保持未修改的状态。

可以使用下面6种方法获得不可修改视图：

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
```

```
Collections.unmodifiableSortedMap
```

每个方法都定义于一个接口。例如，`Collections.unmodifiableList`与`ArrayList`、`LinkedList`或者任何实现了`List`接口的其他类一起协同工作。

例如，假设想要查看某部分代码，但又不触及某个集合的内容，就可以进行下列操作：

```
List<String> staff = new LinkedList<String>();  
...  
lookAt(new Collections.unmodifiableList(staff));
```

`Collections.unmodifiableList`方法将返回一个实现`List`接口的类对象。其访问器方法将从`staff`集合中获取值。当然，`lookAt`方法可以调用`List`接口中的所有方法，而不只是访问器。但是所有的更改器方法（例如，`add`）已经被重新定义为抛出一个`UnsupportedOperationException`异常，而不是将调用传递给底层集合。

不可修改视图并不是集合本身不可修改。仍然可以通过集合的原始引用（在这里是`staff`）对集合进行修改。并且仍然可以让集合的元素调用更改器方法。

由于视图只是包装了接口而不是实际的集合对象，所以只能访问接口中定义的方法。例如，`LinkedList`类有一些非常方便的方法，`addFirst`和`addLast`，它们都不是`List`接口的方法，不能通过不可修改视图进行访问。



**警告：**`unmodifiableCollection`方法（与本节稍后讨论的`synchronizedCollection`和`checkedCollection`方法一样）将返回一个集合，它的`equals`方法不调用底层集合的`equals`方法。相反，它继承了`Object`类的`equals`方法，这个方法只是检测两个对象是否是同一个对象。如果将集或列表转换成集合，就再也无法检测其内容是否相同了。视图就是以这种方式运行的，因为内容是否相等的检测在分层结构的这一层上没有定义妥当。视图将以同样的方式处理`hashCode`方法。

然而，`unmodifiableSet`类和`unmodifiableList`类却使用底层集合的`equals`方法和`hashCode`方法。

#### 4. 同步视图

如果由多个线程访问集合，就必须确保集不会被意外地破坏。例如，如果一个线程试图将元素添加到散列表中，同时另一个线程正在对散列表进行再散列，其结果将是灾难性的。

类库的设计者使用视图机制来确保常规集合的线程安全，而不是实现线程安全的集合类。例如，`Collections`类的静态`synchronizedMap`方法可以将任何一个映射表转换成具有同步访问方法的`Map`：

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

现在，就可以由多线程访问`map`对象了。像`get`和`put`这类方法都是串行操作的，即在另一个线程调用另一个方法之前，刚才的方法调用必须彻底完成。第14章将会详细地讨论数据结构的同步访问。

#### 5. 被检验视图

Java SE 5.0 增加了一组“被检验”视图，用来对泛型类型发生时提供调试支持。如同

第12章中所述，实际上将错误类型的元素私自带到泛型集合中的问题极有可能发生。例如：


```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings; // get warning only, not an error, for compatibility with legacy code
rawList.add(new Date()); // now strings contains a Date object!
```

这个错误的add命令在运行时检测不到。相反，只有在稍后的另一部分代码中调用get方法，并将结果转化为String时，这个类才会抛出异常。被检验视图可以探测到这类问题。下面定义了一个安全列表：

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

视图的add方法将检测插入的对象是否属于给定的类。如果不属于给定的类，就立即抛出一个ClassCastException。这样做的好处是错误可以在正确的位置得以报告：

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Checked list throws a ClassCastException
```

 **警告：**被检测视图受限于虚拟机可以运行的运行时检查。例如，对于Array List <Pair<String>>，由于虚拟机有一个单独的“原始”Pair类，所以，无法阻止插入 Pair <Date>。

#### 6. 关于可选操作的说明

通常，视图有一些局限性，即可能只可以读、无法改变大小、只支持删除而不支持插入，这些与映射表的键视图情况相同。如果试图进行不恰当的操作，受限制的视图就会抛出一个UnsupportedOperationException。

在集合和迭代器接口的API文档中，许多方法描述为“可选操作”。这看起来与接口的概念有所抵触。毕竟，接口的设计目的难道不是负责给出一个类必须实现的方法吗？确实，从理论的角度看，在这里给出的方法很难令人满意。一个更好的解决方案是为每个只读视图和不能改变集合大小的视图建立各自独立的两个接口。不过，这将会使接口的数量成倍增长，这让类库设计者无法接受。

是否应该将“可选”方法这一技术扩展到用户的设计中呢？我们认为不应该。尽管集合被频繁地使用，其实现代码的风格也未必适用于其他问题领域。集合类库的设计者必须解决一组特别严格且又相互冲突的需求。用户希望类库应该易于学习、使用方便，彻底泛型化，面向通用性，同时又与手写算法一样高效。要同时达到所有目标的要求，或者尽量兼顾所有目标完全是不可能的。但是，在自己的编程问题中，很少遇到这样极端的局限性。应该能够找到一种不必依靠极端衡量“可选的”接口操作来解决这类问题的方案。

#### java.util.Collections 1.2

- static <E> Collection unmodifiableCollection(Collection<E> c)
- static <E> List unmodifiableList(List<E> c)
- static <E> Set unmodifiableSet(Set<E> c)
- static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)
- static <K, V> Map unmodifiableMap(Map<K, V> c)

- `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`  
构造一个集合视图，其更改器方法将抛出一个`UnsupportedOperationException`。
- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`  
构造一个集合视图，其方法都是同步的。
- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`  
构造一个集合视图。如果插入一个错误类型的元素，将抛出一个`ClassCastException`。
- `static <E> List<E> nCopies(int n, E value)`
- `static <E> Set<E> singleton(E value)`  
构造一个对象视图，它既可以作为一个拥有 $n$ 个相同元素的不可修改列表，又可以作为一个拥有单个元素的集。

**API** `java.util.Arrays 1.2`

- `static <E> List<E> asList(E... array)`  
返回一个数组元素的列表视图。这个数组是可修改的，但其大小不可变。

**API** `java.util.List<E> 1.2`

- `List<E> subList(int firstIncluded, int firstExcluded)`  
返回给定位位置范围内的所有元素的列表视图。

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

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`  
返回给定范围内的元素视图。

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

- NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)
  - NavigableSet<E> headSet(E to, boolean toIncluded)
  - NavigableSet<E> tailSet(E from, boolean fromIncluded)
- 返回给定范围内的元素视图。boolean 标志决定视图是否包含边界。

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

- SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)
  - SortedMap<K, V> headMap(K firstExcluded)
  - SortedMap<K, V> tailMap(K firstIncluded)
- 返回在给定范围内的键条目的映射表视图。

**API** java.util.NavigableMap<K, V> 6

- NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)
  - NavigableMap<K, V> headMap(K from, boolean fromIncluded)
  - NavigableMap<K, V> tailMap(K to, boolean toIncluded)
- 返回在给定范围内的键条目的映射表视图。boolean 标志决定视图是否包含边界。

### 13.3.2 批操作

到现在为止，列举的绝大多数示例都采用迭代器遍历集合，一次遍历一个元素。然而，可以使用类库中的批操作（bulk operations）避免频繁地使用迭代器。

假设希望找出两个集的交（intersection），即两个集中共有的元素。首先，要建立一个新集，用于存放结果。

```
Set<String> result = new HashSet<String>(a);
```

这里利用了这样一个事实：每一个集合有一个构造器，其参数是保存初始值的另一个集合。接着，调用retainAll方法：

```
result.retainAll(b);
```

result中保存了既在a中出现，也在b中出现的元素。这时已经构成了交集，而且没有使用循环。

可以将这个思路向前推进一步，将批操作应用于视图。例如，假如有一个映射表，将员工的ID映射为员工对象，并且建立了一个将要结束聘用期的所有员工的ID集。

```
Map<String, Employee> staffMap = . . . ;  
Set<String> terminatedIDs = . . . ;
```

直接建立一个键集，并删除终止聘用关系的所有员工的ID即可。

```
staffMap.keySet().removeAll(terminatedIDs);
```

由于键集是映射表的一个视图，所以，键与对应的员工名将会从映射表中自动地删除。

通过使用一个子范围视图，可以将批操作限制于子列表和子集的操作上。例如，假设希望将一个列表的前10个元素添加到另一个容器中，可以建立一个子列表用于选择前10个元素：

```
relocated.addAll(staff.subList(0, 10));
```

这个子范围也可以成为更改操作的对象。

```
staff.subList(0, 10).clear();
```

### 13.3.3 集合与数组之间的转换

由于Java平台API中的大部分内容都是在集合框架创建之前设计的，所以，有时候需要在传统的数组与现代的集合之前进行转换。

如果有一个数组需要转换为集合。Arrays.asList的包装器就可以实现这个目的。例如：

```
String[] values = . . . ;  
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

反过来，将集合转换为数组就有点难了。当然，可以使用toArray方法：

```
Object[] values = staff.toArray();
```

但是，这样做的结果是产生一个对象数组。即使知道集合中包含一个特定类型的对象，也不能使用类型转换：

```
String[] values = (String[]) staff.toArray(); // Error!
```

由toArray方法返回的数组是一个Object[]数组，无法改变其类型。相反，必须使用另外一种toArray方法，并将其设计为所希望的元素类型且长度为0的数组。随后返回的数组将与所创建的数组一样：

```
String[] values = staff.toArray(new String[0]);
```

如果愿意的话，可以构造一个指定大小的数组：

```
staff.toArray(new String[staff.size()]);
```

在这种情况下，没有创建任何一个新数组。



注释：为什么不直接将一个 Class 对象（例如，String.class）传递给toArray方法。其原因是这个方法具有“双重职责”，不仅要填充已有的数组（如果足够长），还要创建一个新数组。

## 13.4 算法

泛型集合接口有一个很大的优点，即算法只需要实现一次。例如，考虑一下计算集合中最大元素这样一个简单的算法。使用传统方式，程序设计人员可能会用循环实现这个算法。下面就是找出数组中最大元素的代码。

```
if (a.length == 0) throw new NoSuchElementException();  
T largest = a[0];  
for (int i = 1; i < a.length; i++)  
    if (largest.compareTo(a[i]) < 0)  
        largest = a[i];
```

当然，为找出数组列表中的最大元素所编写的代码会与此有微小的差别。

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

链表应该怎么做呢？对于链表来说，无法实施高效的随机访问，但却可以使用迭代器。

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

编写这些循环代码有些乏味，并且也很容易出错。是否存在严重错误吗？对于空容器循环能正常工作吗？对于只含有一个元素的容器又会发生什么情况呢？我们不希望每次都测试和调试这些代码，也不想实现下面这一系列的方法：

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

这正是集合接口的用武之地。仔细考虑一下，为了高效地使用这个算法所需要的最小集合接口。采用get和set方法进行随机访问要比直接迭代层次高。在计算链表中最大元素的过程中已经看到，这项任务并不需要进行随机访问。直接用迭代器遍历每个元素就可以计算最大元素。因此，可以将max方法实现为能够接收任何实现了Collection接口的对象。

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

现在就可以使用一个方法计算链表、数组列表或数组中最大元素了。

这是一个非常重要的概念。事实上，标准的C++类库已经有几十种非常有用的算法，每个算法都是在泛型集合上操作的。Java类库中的算法没有如此丰富，但是，也包含了基本的排序、二分查找等实用算法。

### 13.4.1 排序与混排

计算机行业的前辈们有时会回忆起他们当年不得不使用穿孔卡片以及手工地编写排序算法的情形。当然，如今排序算法已经成为大多数编程语言标准库中的一个组成部分，Java程序设计语言也不例外。

Collections类中的sort方法可以对实现了List接口的集合进行排序。

```
List<String> staff = new LinkedList<String>();  
// fill collection . . .;  
Collections.sort(staff);
```

这个方法假定列表元素实现了Comparable接口。如果想采用其他方式对列表进行排序，可以将Comparator对象作为第二个参数传递给sort方法。（已经在前面的章节中介绍过比较器），下面的代码说明了对列表中各项进行排序的基本方法：

```
Comparator<Item> itemComparator = new  
    Comparator<Item>()  
    {  
        public int compare(Item a, Item b)  
        {  
            return a.partNumber - b.partNumber;  
        }  
    };  
Collections.sort(items, itemComparator);
```

如果想按照降序对列表进行排序，可以使用一种非常方便的静态方法Collections.reverseOrder()。这个方法将返回一个比较器，比较器则返回b.compareTo(a)。例如，

```
Collections.sort(staff, Collections.reverseOrder())
```

这个方法将根据元素类型的compareTo方法给定排序顺序，按照逆序对列表staff进行排序。同样，

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

将逆置itemComparator的次序。

人们可能会对sort方法所采用的排序手段感到好奇。通常，在翻阅有关算法书籍中的排序算法时，会发觉介绍的都是有关数组的排序算法，而且使用的是随机访问方式。但是，对列表进行随机访问的效率很低。实际上，可以使用归并排序对列表进行高效的排序（例如，可以参看Addison-Wesley出版社1998年出版的Robert Sedgewick编写的《Algorithms in C++》第366～369页）。然而，Java程序设计语言并不是这样实现的。它直接将所有元素转入一个数组，并使用一种归并排序的变体对数组进行排序，然后，再将排序后的序列复制回列表。

集合类库中使用的归并排序算法比快速排序要慢一些，快速排序是通用排序算法的传统选择。但是，归并排序有一个主要的优点：稳定，即不需要交换相同的元素。为什么要关注相同元素的顺序呢？下面是一种常见的情况。假设有一个已经按照姓名排列的员工列表。现在，要按照工资再进行排序。如果两个雇员的工资相等发生什么情况呢？如果采用稳定的排序算法，将会保留按名字排列的顺序。换句话说，排序的结果将会产生这样一个列表，首先按照工资排序，工资相同者再按照姓名排序。



因为集合不需要实现所有的“可选”方法，因此，所有接受集合参数的方法必须描述什么时候可以安全地将集合传递给算法。例如，显然不能将`unmodifiableList`列表传递给排序算法。可以传递什么类型的列表呢？根据文档说明，列表必须是可修改的，但不必是可以改变大小的。

下面是有关的术语定义：

- 如果列表支持`set`方法，则是可修改的。
- 如果列表支持`add`和`remove`方法，则是可改变大小的。

`Collections`类有一个算法`shuffle`，其功能与排序刚好相反，即随机地混排列表中元素的顺序。例如：

```
ArrayList<Card> cards = . . . ;
Collections.shuffle(cards);
```

如果提供的列表没有实现`RandomAccess`接口，`shuffle`方法将元素复制到数组中，然后打乱数组元素的顺序，最后再将打乱顺序后的元素复制回列表。

例13-6的程序用1~49之间的49个`Integer`对象填充数组。然后，随机地打乱列表，并从打乱后的列表中选前6个值。最后再将选择的数值进行排序和打印。

#### 例13-6 ShuffleTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the random shuffle and sort algorithms.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class ShuffleTest
9. {
10.     public static void main(String[] args)
11.     {
12.         List<Integer> numbers = new ArrayList<Integer>();
13.         for (int i = 1; i <= 49; i++)
14.             numbers.add(i);
15.         Collections.shuffle(numbers);
16.         List<Integer> winningCombination = numbers.subList(0, 6);
17.         Collections.sort(winningCombination);
18.         System.out.println(winningCombination);
19.     }
20. }
```

#### API java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`

使用稳定的排序算法，对列表中的元素进行排序。这个算法的时间复杂度是 $O(n \log n)$ ，其中 $n$ 为列表的长度。

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`

随机地打乱列表中的元素。这个算法的时间复杂度是  $O(n \cdot a(n))$ ， $n$ 是列表的长度， $a(n)$ 是访问元素的平均时间。

- `static <T> Comparator<T> reverseOrder()`  
返回一个比较器，它用与Comparable接口的compareTo方法规定的顺序的逆序对元素进行排序。
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`  
返回一个比较器，它用与comp给定的顺序的逆序对元素进行排序。

#### 13.4.2 二分查找

要想在数组中查找一个对象，通常要依次访问数组中的每个元素，直到找到匹配的元素为止。然而，如果数组是有序的，就可以直接查看位于数组中间的元素，看一看是否大于要查找的元素。如果是，用同样的方法在数组的前半部分继续查找；否则，用同样的方法在数组的后半部分继续查找。这样就可以将查找范围缩减一半。一直用这种方式查找下去。例如，如果数组中有1024个元素，可以在10次比较后定位所匹配的元素（或者可以确认在数组中不存在这样的元素），而使用线性查找，如果元素存在，平均需要512次比较；如果元素不存在，需要1024次比较才可以确认。

Collections类的binarySearch方法实现了这个算法。注意，集合必须是排好序的，否则算法将返回错误的答案。要想查找某个元素，必须提供集合（这个集合要实现List接口，下面还要更加详细地介绍这个问题）以及要查找的元素。如果集合没有采用Comparable接口的compareTo方法进行排序，就还要提供一个比较器对象。

```
i = Collections.binarySearch(c, element);  
i = Collections.binarySearch(c, element, comparator);
```

如果binarySearch方法返回的数值大于等于0，则表示匹配对象的索引。也就是说，`c.get(i)`等于在这个比较顺序下的element。如果返回负值，则表示没有匹配的元素。但是，可以利用返回值计算应该将element插入到集合的哪个位置，以保持集合的有序性。插入的位置是

```
insertionPoint = -i - 1;
```

这并不是简单的 `-i`，因为0值是不确定的。也就是说，下面这个操作：

```
if (i < 0)  
    c.add(-i - 1, element);
```

将把元素插入到正确的位置上。

只有采用随机访问，二分查找才有意义。如果必须利用迭代方式一次次地遍历链表的一半元素来找到中间位置的元素，二分查找就完全失去了优势。因此，如果为binarySearch算法提供一个链表，它将自动地变为线性查找。



注释：在Java SE 1.3中，没有为有序集合提供专门的接口，以进行高效地随机访问，而binarySearch方法使用的是一种拙劣的策略，即检查列表参数是否扩展了Abstract Sequential List类。这个问题在Java SE 1.4中得到了解决。现在binarySearch方法检查列表参数是否实现了RandomAccess接口。如果实现了这个接口，这个方法将采用二分查找；否则，将采用

线性查找。

#### API java.util.Collections 1.2

- static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)
- static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)  
从有序列表中搜索一个键，如果元素扩展了AbstractSequentialList类，则采用线性查找，否则将采用二分查找。这个方法的时间复杂度为 $O(a(n) \log n)$ ， $n$ 是列表的长度， $a(n)$ 是访问一个元素的平均时间。这个方法将返回这个键在列表中的索引，如果在列表中不存在这个键将返回负值 $i$ 。在这种情况下，应该将这个键插入到列表索引 $-i-1$ 的位置上，以保持列表的有序性。

#### 13.4.3 简单算法

在Collections类中包含了几个简单且很有用的算法。前面介绍的查找集合中最大元素的示例就在其中。另外还包括：将一个列表中的元素复制到另外一个列表中；用一个常量值填充容器；逆置一个列表的元素顺序。为什么会在标准库中提供这些简单算法呢？大多数程序员肯定可以很容易地采用循环实现这些算法。我们之所以喜欢这些算法是因为：它们可以让程序员阅读算法变成一件轻松的事情。当阅读由别人实现的循环时，必须要揣摩编程者的意图。而在看到诸如Collections.max这样的方法调用时，一定会立刻明白其用途。

下面的API注释描述了Collections类的一些简单算法。

#### API java.util.Collections 1.2

- static <T extends Comparable<? super T>> T min(Collection<T> elements)
- static <T extends Comparable<? super T>> T max(Collection<T> elements)
- static <T> min(Collection<T> elements, Comparator<? super T> c)
- static <T> max(Collection<T> elements, Comparator<? super T> c)  
返回集合中最小的或最大的元素（为清楚起见，参数的边界被简化了）。
- static <T> void copy(List<? super T> to, List<T> from)  
将原列表中的所有元素复制到目标列表的相应位置上。目标列表的长度至少与原列表一样。
- static <T> void fill(List<? super T> l, T value)  
将列表中所有位置设置为相同的值。
- static <T> boolean addAll(Collection<? super T> c, T... values) 5.0  
将所有的值添加到集合中。如果集合改变了，则返回true。
- static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4  
用newValue取代所有值为oldValue的元素。
- static int indexOfSubList(List<?> l, List<?> s) 1.4
- static int lastIndexOfSubList(List<?> l, List<?> s) 1.4

返回 $l$ 中第一个或最后一个等于 $s$ 子列表的索引。如果 $l$ 中不存在等于 $s$ 的子列表,则返回  $-1$ 。

例如,  $l$ 为 $[s, t, a, r]$ ,  $s$ 为 $[t, a, r]$ , 两个方法都将返回索引1。

- `static void swap(List<?> l, int i, int j)` 1.4

交换给定偏移量的两个元素。

- `static void reverse(List<?> l)`

逆置列表中元素的顺序。例如, 逆置列表  $[t, a, r]$  后将得到列表  $[r, a, t]$ 。这个方法的时间复杂度为 $O(n)$ ,  $n$ 为列表的长度。

- `static void rotate(List<?> l, int d)` 1.4

旋转列表中的元素, 将索引 $i$ 的条目移动到位置 $(i + d) \% l.size()$ 。例如, 将列表 $[t, a, r]$  旋转移2个位置后得到 $[a, r, t]$ 。这个方法的时间复杂度为 $O(n)$ ,  $n$ 为列表的长度。

- `static int frequency(Collection<?> c, Object o)` 5.0

返回 $c$ 中与对象 $o$ 相同的元素个数。

- `boolean disjoint(Collection<?> c1, Collection<?> c2)` 5.0

如果两个集合没有共同的元素, 则返回`true`。

#### 13.4.4 编写自己的算法

如果编写自己的算法(实际上, 是以集合作为参数的任何方法), 应该尽可能地使用接口, 而不要使用具体的实现。例如, 假想用一组菜单项填充`JMenu`。传统上, 这种方法可能会按照下列方式实现:

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

但是, 这样会限制方法的调用程序, 即调用程序必须在`ArrayList`中提供选项。如果这些选项需要放在另一个容器中, 首先必须对它们重新包装, 因此, 最好接受一个更加通用的集合。

什么是完成这项工作的最通用的集合接口? 在这里, 只需要访问所有的元素, 这是`Collection`接口的基本功能。下面代码说明了如何重新编写`fillMenu`方法使之接受任意类型的集合。

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

现在, 任何人都可以用`ArrayList`或`LinkedList`, 甚至用`Arrays.asList`包装器包装的数组调用这个方法。



注释: 既然将集合接口作为方法参数是个很好的想法, 为什么Java类库不更多地这样做呢? 例如, `JComboBox`有两个构造器:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

之所以没有这样做，原因很简单：时间问题。Swing类库是在集合类库之前创建的。

如果编写了一个返回集合的方法，可能还想要一个返回接口，而不是返回类的方法，因为这样做可以在日后改变想法，并用另一个集合重新实现这个方法。

例如，编写一个返回所有菜单项的方法getAllItems。

```
List<MenuItem> getAllItems(JMenu menu)
{
    ArrayList<MenuItem> items = new ArrayList<MenuItem>()
    for (int i = 0; i < menu.getItemCount(); i++)
        items.add(menu.getItem(i));
    return items;
}
```

日后，可以做出这样的决定：不复制所有的菜单项，而仅提供这些菜单项的视图。要做到这一点，只需要返回AbstractList的匿名子类。

```
List<MenuItem> getAllItems(final JMenu menu)
{
    return new
        AbstractList<MenuItem>()
        {
            public MenuItem get(int i)
            {
                return item.getItem(i);
            }
            public int size()
            {
                return item.getItemCount();
            }
        };
}
```

当然，这是一项高级技术。如果使用它，就应该将它支持的那些“可选”操作准确地记录在文档中。在这种情况下，必须提醒调用者返回的对象是一个不可修改的列表。

## 13.5 遗留的集合

本节将讨论Java程序设计语言自问世以来就存在的集合类：Hashtable类和非常有用的子类Properties、Vector的子类Stack以及BitSet类。

### 13.5.1 Hashtable类

Hashtable类与HashMap类的作用一样，实际上，它们拥有相同的接口。与Vector类的方法一样。Hashtable的方法也是同步的。如果对同步性或遗留代码的兼容性没有任何要求，就应该使用HashMap。



注释：这个类的名字是Hashtable，带有一个小写的t。在Windows操作系统下，如果使用HashTable会看到一个很奇怪的错误信息，这是因为Windows文件系统对大小写不敏感，而Java编译器却对大小写敏感。

### 13.5.2 枚举

遗留集合使用Enumeration接口对元素序列进行遍历。Enumeration接口有两个方法，即hasMoreElements和nextElement。这两个方法与Iterator接口的hasNext方法和next方法十分类似。

例如，Hashtable类的elements方法将产生一个用于描述表中各个枚举值的对象：

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
    Employee e = e.nextElement();
    . . .
}
```

有时还会遇到遗留的方法，其参数是枚举类型的。静态方法 Collections.enumeration将产生一个枚举对象，枚举集合中的元素。例如：

```
List<InputStream> streams = . . . ;
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
// the SequenceInputStream constructor expects an enumeration
```



注释：在C++中，用迭代器作为参数十分普遍。幸好，在Java的编程平台中，只有极少的程序员沿用这种习惯。传递集合要比传递迭代器更为明智。集合对象的用途更大。当接受方如果需要时，总是可以从集合中获得迭代器，而且，还可以随时地使用集合的所有方法。不过，可能会在某些遗留代码中发现枚举接口，因为这是在Java SE 1.2的集合框架出现之前，它们是泛型集合惟一可以使用的机制。



java.util.Enumeration<E> 1.0

- boolean hasMoreElements()  
如果还有更多的元素可以查看，则返回true。
- E nextElement()  
返回被检测的下一个元素。如果hasMoreElements()返回false，则不要调用这个方法。



java.util.Hashtable<K, V> 1.0

- Enumeration<K> keys()  
返回一个遍历散列表中键的枚举对象。
- Enumeration<V> elements()  
返回一个遍历散列表中元素的枚举对象。



java.util.Vector<E> 1.0

- Enumeration<E> elements()  
返回遍历向量中元素的枚举对象。

### 13.5.3 属性映射表

属性映射表 (property map) 是一个类型非常特殊的映射表结构。它有下面3个特性：

- 键与值都是字符串。
- 表可以保存到一个文件中，也可以从文件中加载。
- 使用一个默认的辅助表。

实现属性映射表的Java平台类称为Properties。

属性映射表通常用于程序的特殊配置选项，参见第10章。

#### java.util.Properties 1.0

- Properties()  
创建一个空的属性映射表。
- Properties(Properties defaults)  
创建一个带有一组默认值的空的属性映射表。
- String getProperty(String key)  
获得属性的对应关系；返回与键对应的字符串。如果在映射表中不存在，返回默认表中与这个键对应的字符串。
- String getProperty(String key, String defaultValue)  
获得在键没有找到时具有的默认值属性；它将返回与键对应的字符串，如果在映射表中不存在，就返回默认的字符串。
- void load(InputStream in)  
从InputStream加载属性映射表。
- void store(OutputStream out, String commentString)  
把属性映射表存储到OutputStream。

### 13.5.4 栈

从1.0版开始，标准类库中就包含了Stack类，其中有大家熟悉的push方法和pop方法。但是，Stack类扩展为Vector类，从理论角度看，Vector类并不太令人满意，它可以让栈使用不属于栈操作的insert和remove方法，即可以在任何地方进行插入或删除操作，而不仅仅是在栈顶。

#### java.util.Stack<E> 1.0

- E push(E item)  
将item压入栈并返回item。
- E pop()  
弹出并返回栈顶的item。如果栈为空，请不要调用这个方法。
- E peek()  
返回栈顶元素，但不弹出。如果栈为空，请不要调用这个方法。

### 13.5.5 位集

Java平台的BitSet类用于存放一个位序列（它不是数学上的集，称为位向量或位数组更为合适）。如果需要高效地存储位序列（例如，标志）就可以使用位集。由于位集将位包装在字节里，所以，使用位集要比使用Boolean对象的ArrayList更加高效。

BitSet类提供了一个便于读取、设置或清除各个位的接口。使用这个接口可以避免屏蔽和其他麻烦的位操作。如果将这些位存储在int或long变量中就必须进行这些繁琐的操作。

例如，对于一个名为bucketOfBits的BitSet，

```
bucketOfBits.get(i)
```

如果第i位处于“开”状态，就返回true；否则返回false。同样地，

```
bucketOfBits.set(i)
```

将第i位置为“开”状态。最后，

```
bucketOfBits.clear(i)
```

将第i位置为“关”状态。



C++注释：C++位集模板与Java平台的BitSet功能一样。



java.util.BitSet 1.0

- `BitSet(int initialCapacity)`  
创建一个位集。
- `int length()`  
返回位集的“逻辑长度”，即1加上位集的最高设置位的索引。
- `boolean get(int bit)`  
获得一个位。
- `void set(int bit)`  
设置一个位。
- `void clear(int bit)`  
清除一个位。
- `void and(BitSet set)`  
这个位集与另一个位集进行逻辑“AND”。
- `void or(BitSet set)`  
这个位集与另一个位集进行逻辑“OR”。
- `void xor(BitSet set)`  
这个位集与另一个位集进行逻辑“XOR”。
- `void andNot(BitSet set)`  
清除这个位集中对应另一个位集中设置的所有位。



### “ Eratosthenes筛子 ” 基准测试

作为位集应用的一个示例，这里给出一个采用“ Eratosthenes筛子 ” 算法查找素数的实现（素数是指只能被1和本身整除的数，例如2、3或5，“ Eratosthenes筛子 ” 算法是最早发现的用来枚举这些基本数字的方法之一）。这并不是一种查找素数的最好方法，但是由于某种原因，它已经成为测试编译程序性能的一种流行的基准。（这也不是一种最好的基准测试方法，它主要用于测试位操作。）

在此，将尊重这个传统，并给出实现。其程序将计算2 ~ 2 000 000之间的所有素数（一共有148 933个素数，或许不打算把它们全部打印出来吧）。

这里并不想深入程序的细节，关键是要遍历一个拥有200万个位的位集。首先将所有的位置为“开”状态，然后，将已知素数的倍数所对应的位都置为“关”状态。经过这个操作保留下来的位对应的就是素数。例13-7是用Java程序设计语言实现的这个算法程序，而例13-8是用C++实现的这个算法程序。



注释：尽管筛选并不是一种好的基准测试方法，这里还是对这个算法的两个算法的运行时间进行了测试。下面是在1.66 GHz双核IBM ThinkPad计算机上运行时间的结果，这台计算机内存为2 GB，操作系统为Ubuntu 7.04。

- C++ (g++ 4.1.2): 360毫秒
- Java (Java SE 6): 105毫秒

我们已经对《Java核心技术》7个版本进行了这项测试，在最后的3个版本中，Java轻松地战胜了C++。公平地说，如果有人改变C++优化器的级别，将可以用60毫秒的时间战胜Java。如果程序运行的时间长到触发Hotspot即时编译器时，Java只与C++打个平手。

#### 例13-7 Sieve.java

```
1. import java.util.*;
2.
3. /**
4.  * This program runs the Sieve of Erathostenes benchmark. It computes all primes up to 2,000,000.
5.  * @version 1.21 2004-08-03
6.  * @author Cay Horstmann
7.  */
8. public class Sieve
9. {
10.     public static void main(String[] s)
11.     {
12.         int n = 2000000;
13.         long start = System.currentTimeMillis();
14.         BitSet b = new BitSet(n + 1);
15.         int count = 0;
16.         int i;
17.         for (i = 2; i <= n; i++)
18.             b.set(i);
19.         i = 2;
20.         while (i * i <= n)
21.         {
22.             if (b.get(i))
```

```
23.     {
24.         count++;
25.         int k = 2 * i;
26.         while (k <= n)
27.         {
28.             b.clear(k);
29.             k += i;
30.         }
31.     }
32.     i++;
33. }
34. while (i <= n)
35. {
36.     if (b.get(i)) count++;
37.     i++;
38. }
39. long end = System.currentTimeMillis();
40. System.out.println(count + " primes");
41. System.out.println((end - start) + " milliseconds");
42. }
43. }
```

**例13-8** Sieve.cpp

```
1. /**
2.  @version 1.21 2004-08-03
3.  @author Cay Horstmann
4. */
5.
6. #include <bitset>
7. #include <iostream>
8. #include <ctime>
9.
10. using namespace std;
11.
12. int main()
13. {
14.     const int N = 2000000;
15.     clock_t cstart = clock();
16.
17.     bitset<N + 1> b;
18.     int count = 0;
19.     int i;
20.     for (i = 2; i <= N; i++)
21.         b.set(i);
22.     i = 2;
23.     while (i * i <= N)
24.     {
25.         if (b.test(i))
26.         {
27.             count++;
28.             int k = 2 * i;
29.             while (k <= N)
30.             {
31.                 b.reset(k);
32.                 k += i;
```

```
33.     }
34.     }
35.     i++;
36. }
37. while (i <= N)
38. {
39.     if (b.test(i))
40.         count++;
41.     i++;
42. }
43.
44. clock_t cend = clock();
45. double millis = 1000.0
46.     * (cend - cstart) / CLOCKS_PER_SEC;
47.
48. cout << count << " primes\n"
49.     << millis << " milliseconds\n";
50.
51. return 0;
52. }
```

到此为止，Java集合框架的旅程就结束了。正如所看到的，Java类库提供了大量的集合类以适应程序设计的需要。在本书的最后一章，将讨论非常重要的并发程序设计。