

第9章 集合

▲ Java 集合框架

▲ 集合框架中的接口

▲ 具体集合

▲ 映射

▲ 副本与视图

▲ 算法

▲ 遗留的集合

以自然的方式实现方法或者非常关注性能时，你选择的不同数据结构会带来很大差异。是否需要快速地搜索成千上万（甚至上百万）个有序的数据项？是否需要快速地在有序序列中间插入元素或删除元素？是否需要在键与值之间建立关联？

本章将介绍如何利用 Java 类库帮助我们实现程序设计所需的传统数据结构。在大学的计算机科学课程中，有一门数据结构（Data Structure）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。

9.1 Java 集合框架

Java 最初的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 Enumeration 接口提供了一种抽象机制，用于访问任意容器中的元素。这是一个很明智的选择，要想建立一个全面的集合类库，这需要大量的时间和高超的技能。

随着 Java 1.2 的问世，设计人员感到是时候推出一组功能完备的数据结构了。面对一大堆相互冲突的设计问题，他们希望让类库规模很小而且要易于学习，不希望像 C++ 的“标准模板库”（即 STL）那样复杂，但又希望能够得到 STL 率先提出的“泛型算法”所具有的优点。他们还希望遗留的类能融入这个新框架。与集合类库的所有设计者一样，他们必须做出一些艰难的选择，于是，在这个过程中，他们做出了一些独具特色的设计决定。这一节将介绍 Java 集合框架的基本设计，展示如何具体使用，并解释一些颇具争议的特性背后的考虑。

9.1.1 集合接口与实现分离

与现代的数据结构类库的常见做法一样，Java 集合类库也将接口（interface）与实现（implementation）分离。下面利用我们熟悉的一个数据结构——队列（queue）来说明接口与实现如何分离。

队列接口（queue interface）指出可以在队尾添加元素，在队头删除元素，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”方式获取对象时，就应该使用队列（见图 9-1）。

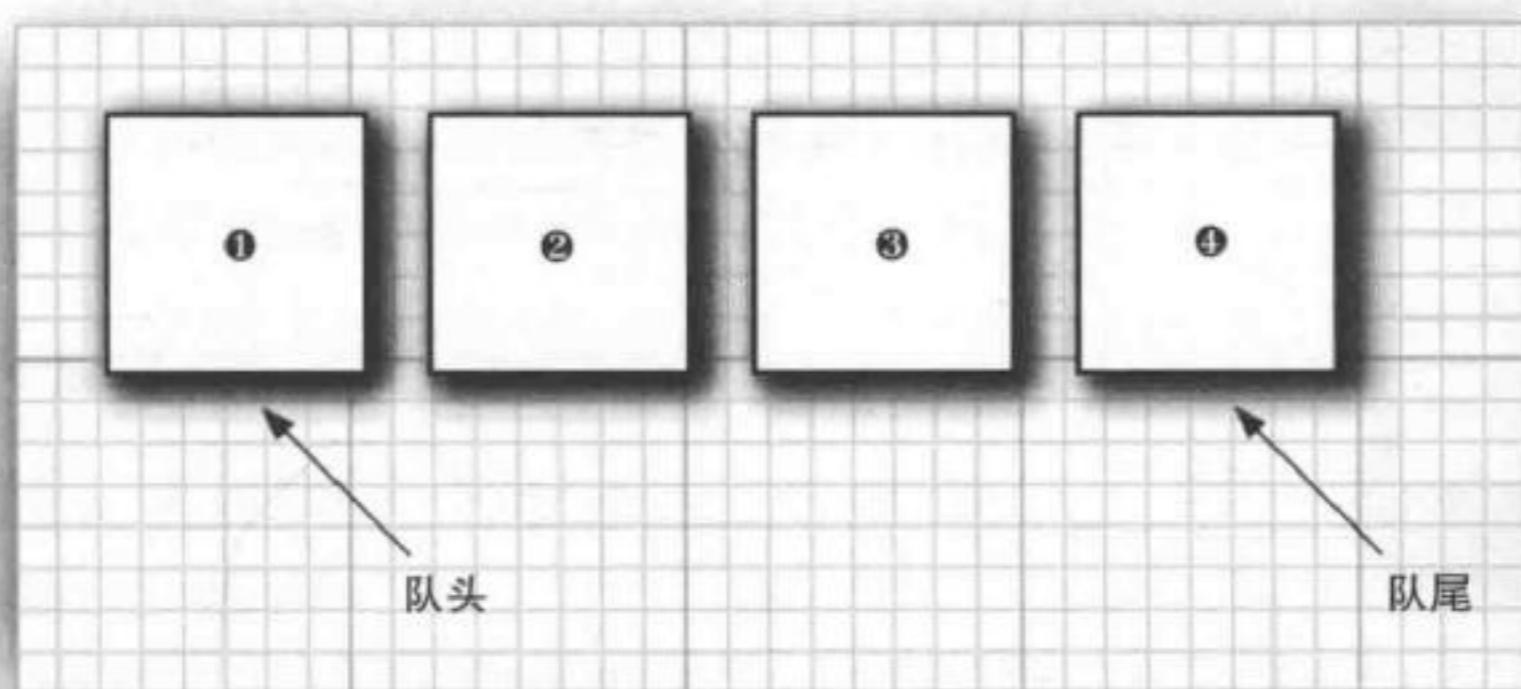


图 9-1 队列

队列接口的最简形式可能如下所示：

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

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 9-2）。

每个实现都可以用一个实现了 Queue 接口的类表示。

```
public class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
    private int head;
    private int tail;

    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private E[] elements;
}

public class LinkedListQueue<E> implements Queue<E> // not an actual library class
{
    private Link head;
    private Link tail;

    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
}
```

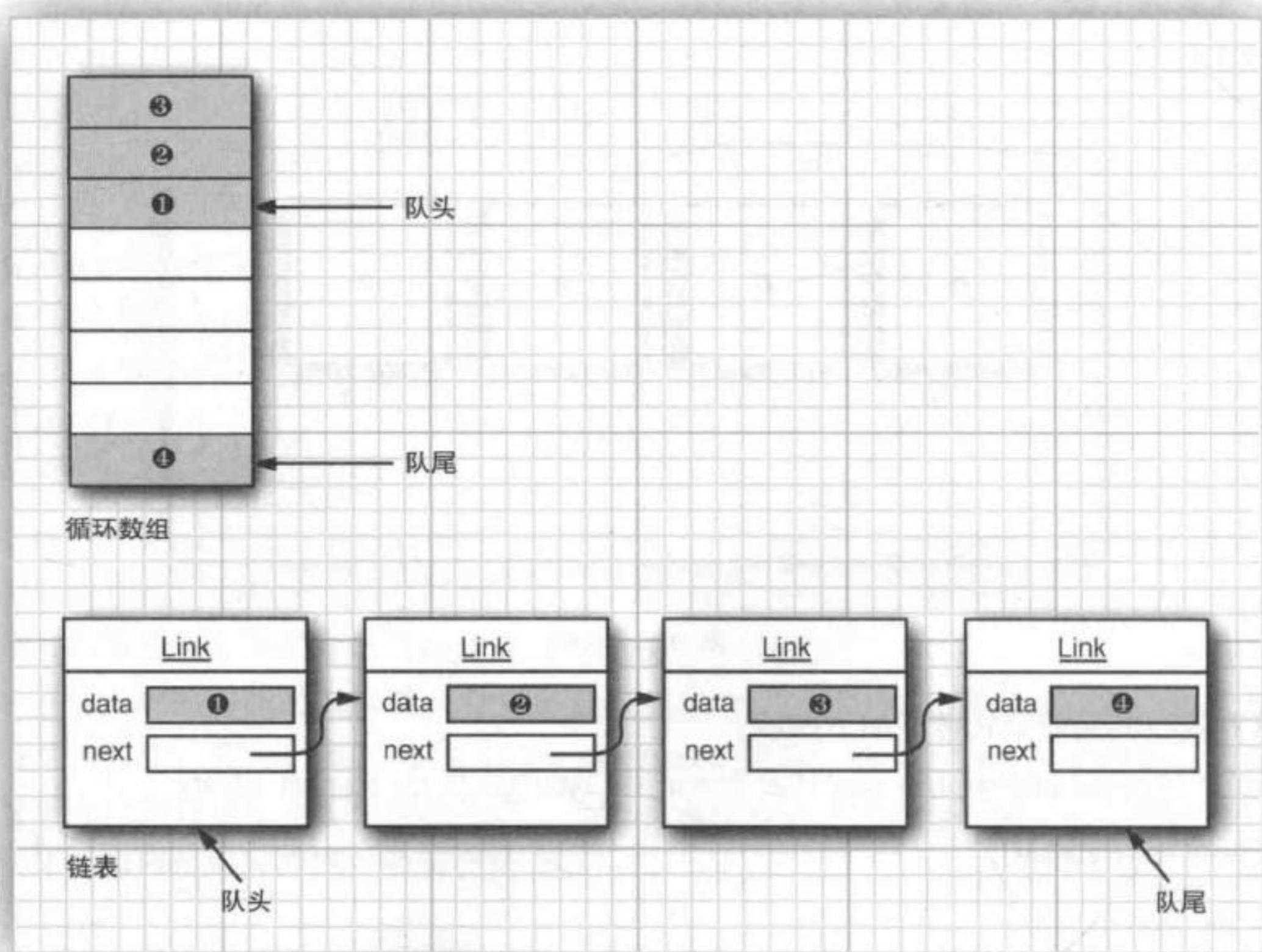


图 9-2 队列的实现

注释：实际上，Java 类库没有名为 `CircularArrayQueue` 和 `LinkedListQueue` 的类。这里只是以这些类为例来解释集合接口与实现在概念上的区分。如果需要一个循环数组队列，可以使用 `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。

在程序中使用队列时，一旦已经构造了集合，你不需要知道究竟使用了哪种实现。因此，只是在构造集合对象时，才会使用具体的类。可以使用接口类型（interface type）存放集合引用。

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

利用这种方法，一旦改变了想法，你可以很轻松地使用另外一种不同的实现。只需要修改程序中的一个地方（即调用构造器的语句）。如果觉得 `LinkedListQueue` 是个更好的选择，就将代码修改为：

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

为什么选择这种实现，而不选择其他实现呢？接口本身并不能说明一种实现的效率如何。某种程度上讲，循环数组要比链表更高效，因此多数人优先选择循环数组。不过，通常

来讲，这样做也需要付出一定的代价。

循环数组是一个有界 (bounded) 集合，它的容量有限。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 接口中的所有方法更容易。

9.1.2 Collection 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，稍后将会介绍。

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 (set) 中添加一个对象，而这个对象在集中已经存在，这个 add 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。下一节讨论迭代器。

9.1.3 迭代器

Iterator 接口包含 4 个方法：

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

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，在调用 next 之前需要调用 hasNext 方法。如果迭代器对象还有更多可以访问的元素，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，当 hasNext 返回 true 时就反复地调用 next 方法。例如：

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

```

    do something with element
}

```

可以更加简洁地将这个循环写为“for each”循环：

```

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

```

编译器会把“for each”循环转换为一个带迭代器的循环。

“for each”循环可以处理任何实现了 Iterable 接口的对象，这个接口只有一个抽象方法：

```

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。

也可以不写循环，而是调用 forEachRemaining 方法并提供一个 lambda 表达式（它会处理一个元素）。将对迭代器的每一个元素调用这个 lambda 表达式，直到再没有元素为止。

```
iterator.forEachRemaining(element -> do something with element);
```

访问元素的顺序取决于集合类型。如果迭代处理一个 ArrayList，迭代器将从索引 0 开始，每迭代一次，索引值加 1。不过，如果访问 HashSet 中的元素，会按照一种基本上随机的顺序获得元素。虽然可以确保在迭代过程中能够遍历到集合中的所有元素，但是无法预知访问各元素的顺序。这通常并不是什么问题，因为对于计算总和或统计匹配之类的计算，顺序并不重要。

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 集合类库的设计者本来可以选择使用 Enumeration 接口，但是他们不喜欢这个接口累赘的方法名，于是引入了有较短方法名的一个新接口。

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一个重要的区别。在传统的集合类库中，例如，C++ 的标准模板库，迭代器是根据数组索引创建的。如果给定这样一个迭代器，可以查找存储在指定位置上的元素，就像如果知道数组索引 *i*，就可以查找数组元素 *a[i]*。不需要查找元素，也可以将迭代器向前移动一个位置。这与不执行查找而通过调用 *i++* 向前移动数组索引的操作一样。但是，Java 迭代器并不是这样处理的。查找操作与位置变化紧密耦合。查找一个元素的唯一方法是调用 *next*，而在执行查找操作的同时，迭代器的位置就会随之向前移动。

因此，可以认为 Java 迭代器位于两个元素之间。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 9-3）。

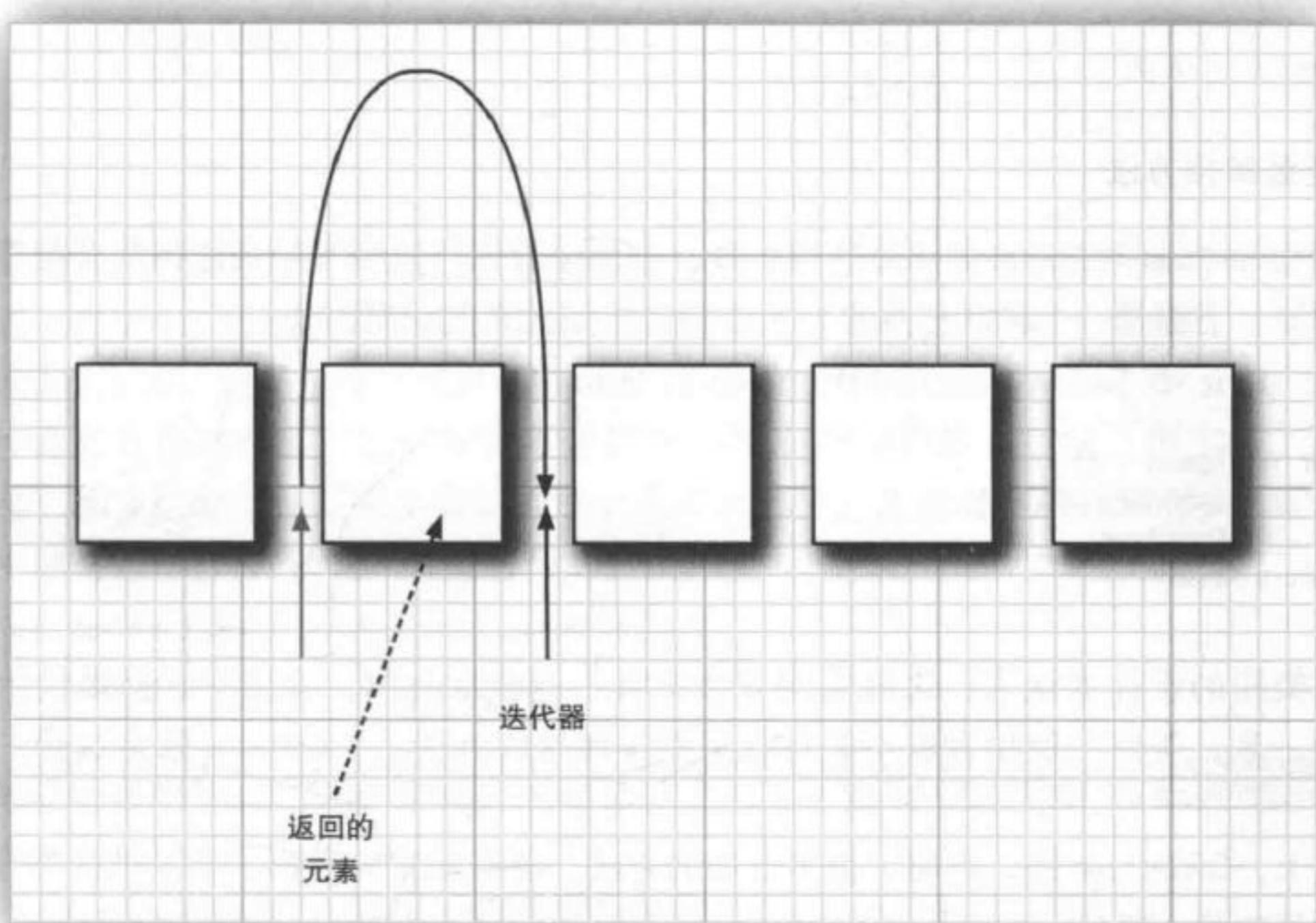


图 9-3 向前移动迭代器

注释：这里还可以做一个有用的对比。可以认为 `Iterator.next` 等价于 `InputStream.read`。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中的下一个字节。类似地，反复地调用 `next` 就可以读取集合中的所有元素。

`Iterator` 接口的 `remove` 方法会删除上次调用 `next` 方法返回的元素。在大多数情况下，这是有道理的，在决定某个元素确实是要删除的元素之前，应该先看一下这个元素。不过，如果想要删除指定位置上的元素，仍然需要越过这个元素。例如，可以如下删除一个字符串集合中的第一个元素：

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

更重要的是，`next` 方法和 `remove` 方法调用之间存在依赖性。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，不能直接这样调用：

```
it.remove();
it.remove(); // ERROR
```

实际上，必须先调用 `next` 越过将要删除的元素。

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

9.1.4 泛型实用方法

由于 Collection 与 Iterator 都是泛型接口，这意味着你可以编写处理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：这些实用方法中有一些非常有用，应该将它们提供给用户使用。这样一来，类库的使用者就不必自己重新实现这些方法了。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()
```

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。在本节末尾的 API 注释中可以找到有关它们的完整说明。

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，这将是一件很烦人的事情。为了能够让实现者更轻松一些，Java 类库提供了一个类 AbstractCollection，其中保持基础方法 size 和 iterator 仍为抽象方法，但是为实现者实现了其他例行方法。例如：

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

    public boolean contains(Object obj)
    {
        for (E element : this) // calls iterator()
```

```

        if (element.equals(obj))
            return true;
        return false;
    }
    ...
}

```

这样一来，具体集合类可以扩展 `AbstractCollection` 类。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。不过，如果子类有更加高效的方式实现 `contains` 方法，也完全可以提供 `contains` 方法。

这种做法有些过时了。这些方法最好是 `Collection` 接口的默认方法。但实际上并不是这样。不过，确实已经增加了很多默认方法。其中大部分方法都与流的处理有关（有关内容将在卷Ⅱ中讨论）。另外，还有一个很有用的方法：

```
default boolean removeIf(Predicate<? super E> filter)
```

这个方法用于删除满足某个条件的元素。

`java.util.Collection<E>` 1.2

- `Iterator<E> iterator()`
返回一个迭代器，可以用于访问集合中的元素。
- `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)`
如果集合中包含一个与 `obj` 相等的对象，返回 `true`。
- `boolean containsAll(Collection<?> other)`
如果这个集合中包含 `other` 集合中的所有元素，返回 `true`。
- `boolean add(E 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` 集合中的所有元素。如果由于这个调用改变了集合，返回 `true`。
- `default boolean removeIf(Predicate<? super E> filter) 8`
从这个集合删除让 `filter` 返回 `true` 的所有元素。如果由于这个调用改变了集合，则返回 `true`。

- `void clear()`
从这个集合中删除所有元素。
- `boolean retainAll(Collection<?> other)`
从这个集合中删除所有与 `other` 集合中元素不同的元素。如果由于这个调用改变了集合，返回 `true`。
- `Object[] toArray()`
返回这个集合中的对象的数组。
- `<T> T[] toArray(IntFunction<T[]> generator) 11`
返回这个集合中的对象的数组。这个数组用 `generator` 构造，这通常是一个构造器表达式 `T[]::new`。

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

- `boolean hasNext()`
如果存在另一个可访问的元素，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末尾，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。如果访问上一个元素之后集合已经发生了变化，这个方法将抛出一个 `IllegalStateException`。
- `default void forEachRemaining(Consumer<? super E> action) 8`
访问元素，并传递到指定的动作，直到再没有更多元素，或者这个动作抛出一个异常。

9.2 集合框架中的接口

Java 集合框架为不同类型的集合定义了大量接口，如图 9-4 所示。

集合有两个基本接口：`Collection` 和 `Map`。我们已经看到，可以用以下方法在集合中插入元素：

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

不过，映射包含键 / 值对，要用 `put` 方法在映射中插入元素：

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

要从集合读取元素，可以用迭代器访问元素。不过，可以使用 `get` 方法从映射中读取值：

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

`List` 是一个有序集合（ordered collection）。元素会增加到容器中的特定位置。可以采用两种方式访问元素：使用迭代器访问，或者使用一个整数索引来访问。后面这种方法称为随机访问（random access），因为这样可以按任意顺序访问元素。与之不同，使用迭代器访问时，必须顺序地访问元素。

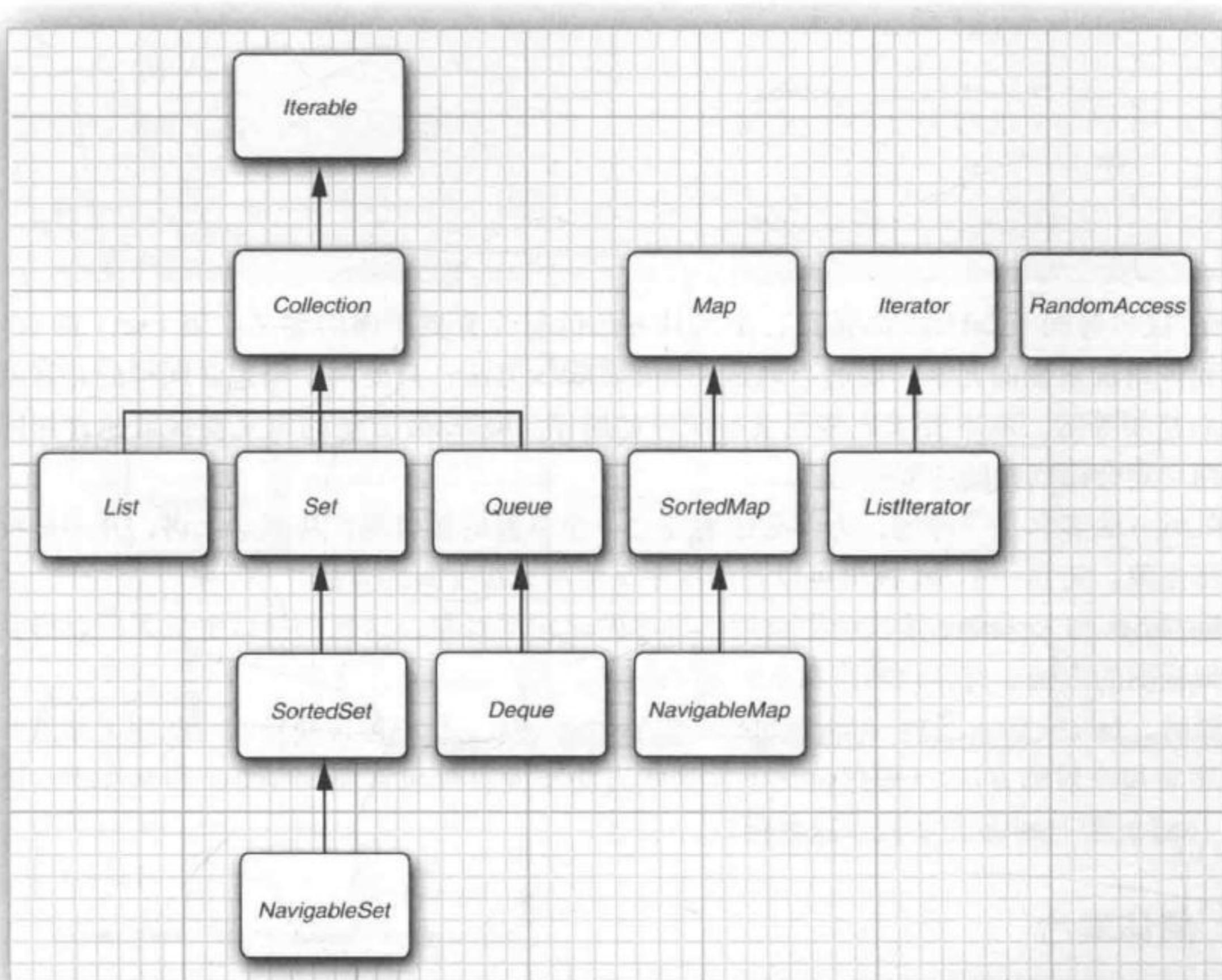


图 9-4 集合框架的接口

List 接口定义了多个用于随机访问的方法：

```

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

ListIterator 接口是 Iterator 的一个子接口。它定义了一个方法用于在迭代器位置前面增加一个元素：

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

坦率地讲，集合框架的这个方面设计得很不好。实际上有两种有序集合，其性能开销有很大差异。由数组支持的有序集合可以快速地随机访问，因此适合使用 List 方法并提供一个整数索引来访问。与之不同，链表尽管也是有序的，但是随机访问很慢，所以最好使用迭代器来遍历。如果原先提供两个接口就会容易一些了。

注释：为了避免对链表执行随机访问操作，Java 1.4 引入了一个标记接口 RandomAccess。这个接口不包含任何方法，不过可以用它来测试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
}

```

`Set` 接口等同于 `Collection` 接口，不过其方法的行为有更严谨的定义。集（`set`）的 `add` 方法不允许增加重复的元素。要适当地定义集的 `equals` 方法：只要两个集包含同样的元素就认为它们是相等的，而不要求这些元素有同样的顺序。`hashCode` 方法的定义要保证包含相同元素的两个集会得到相同的散列码。

既然方法签名是一样的，为什么还要建立一个单独的接口呢？从概念上讲，并不是所有集合都是集。建立一个 `Set` 接口可以允许程序员编写只接受集的方法。

`SortedSet` 和 `SortedMap` 接口会提供用于排序的比较器对象，这两个接口定义了可以得到集合子集视图的方法。有关内容将在 9.5 节讨论。

最后，Java 6 引入了接口 `NavigableSet` 和 `NavigableMap`，其中包含额外的一些用于搜索和遍历有序集和映射的方法。（理想情况下，这些方法本应直接包含在 `SortedSet` 和 `SortedMap` 接口中。）`TreeSet` 和 `TreeMap` 类实现了这些接口。

9.3 具体集合

表 9-1 展示了 Java 类库中的集合，并简要描述了每个集合类的用途。（为简单起见，这里省略了线程安全集合，那些集合将在第 12 章中介绍。）

表 9-1 Java 类库中的具体集合

集合类型	描 述	参 见
<code>ArrayList</code>	可以动态增长和缩减的一个索引序列	9.3.2 节
<code>LinkedList</code>	可以在任意位置高效插入和删除的一个有序序列	9.3.1 节
<code>ArrayDeque</code>	实现为循环数组的一个双端队列	9.3.5 节
<code>HashSet</code>	没有重复元素的一个无序集合	9.3.2 节
<code>TreeSet</code>	一个有序集	9.3.4 节
<code>EnumSet</code>	一个包含枚举类型值的集	9.4.6 节
<code>LinkedHashSet</code>	一个可以记住元素插入次序的集	9.4.5 节
<code>PriorityQueue</code>	允许高效删除最小元素的一个集合	9.3.6 节
<code>HashMap</code>	存储键 / 值关联的一个数据结构	9.4.4 节
<code>TreeMap</code>	键有序的一个映射	9.4.1 节
<code>EnumMap</code>	键属于枚举类型的一个映射	9.4.6 节
<code>LinkedHashMap</code>	可以记住键 / 值项添加次序的一个映射	9.4.5 节
<code>WeakHashMap</code>	这个映射中的值如果不在别处使用，就会被垃圾回收器回收	9.4.4 节
<code>IdentityHashMap</code>	用 = 而不是用 <code>equals</code> 比较键的一个映射	9.4.7 节

在表 9-1 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。映射的内容将在 9.4 节介绍。

图 9-5 显示了这些类之间的关系。

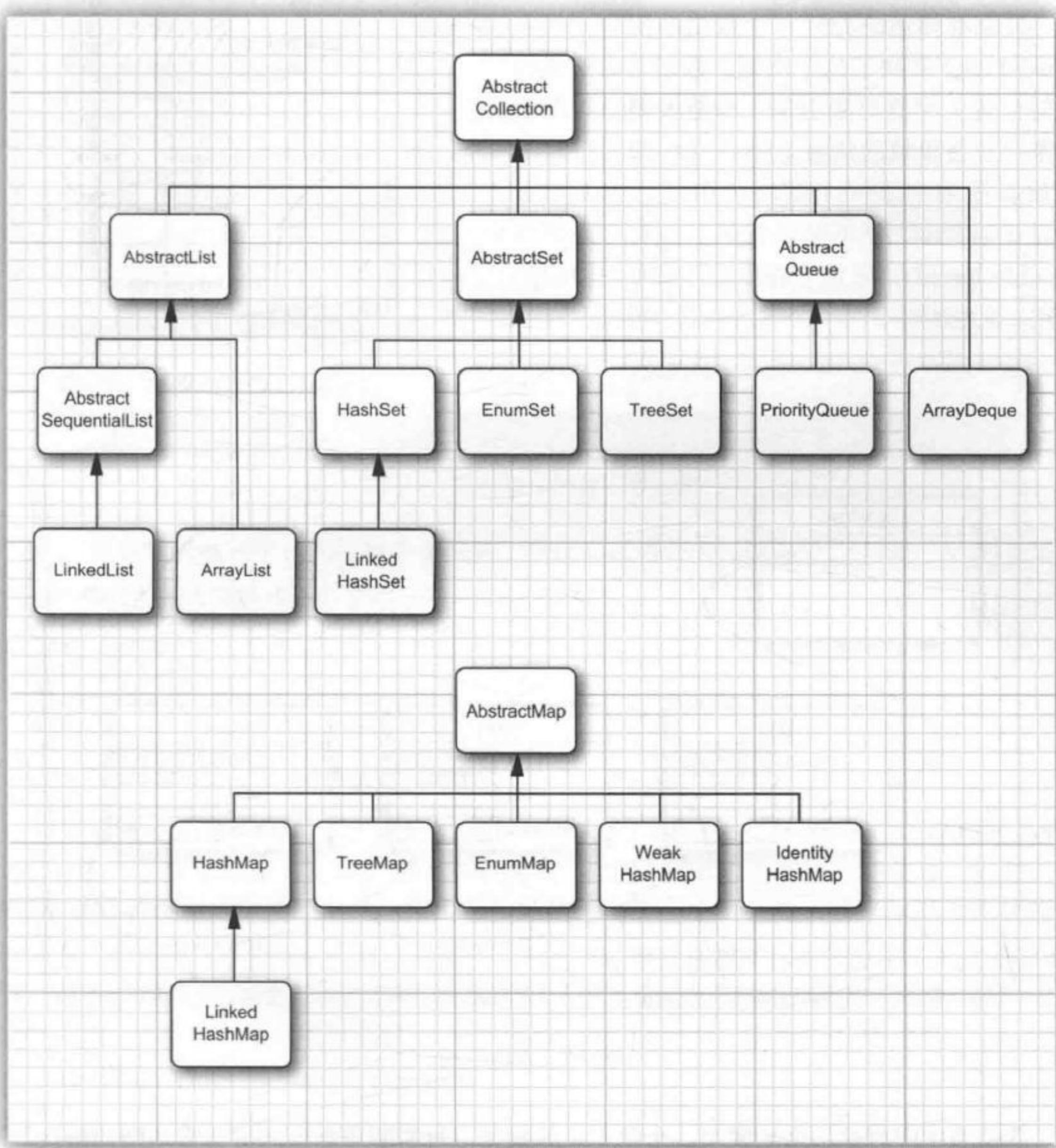


图 9-5 集合框架中的类

9.3.1 链表

本书的很多示例中已经使用了数组和它的动态“兄弟”：`ArrayList` 类。不过，数组和数组

列表都有一个重大的缺陷。这就是从数组中间删除一个元素开销很大，其原因是数组中位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 9-6）。在数组中间插入一个元素也是如此。

大家都知道的另外一个数据结构——链表（linked list）解决了这个问题。数组是在连续的存储位置上存放对象引用，而链表则是将每个对象存放在单独的链接（link）中。每个链接还存放着序列中下一个链接的引用。在 Java 程序设计语言中，所有链表实际上都有双向链接（doubly linked），即每个链接还存储着其前驱的引用（见图 9-7）。

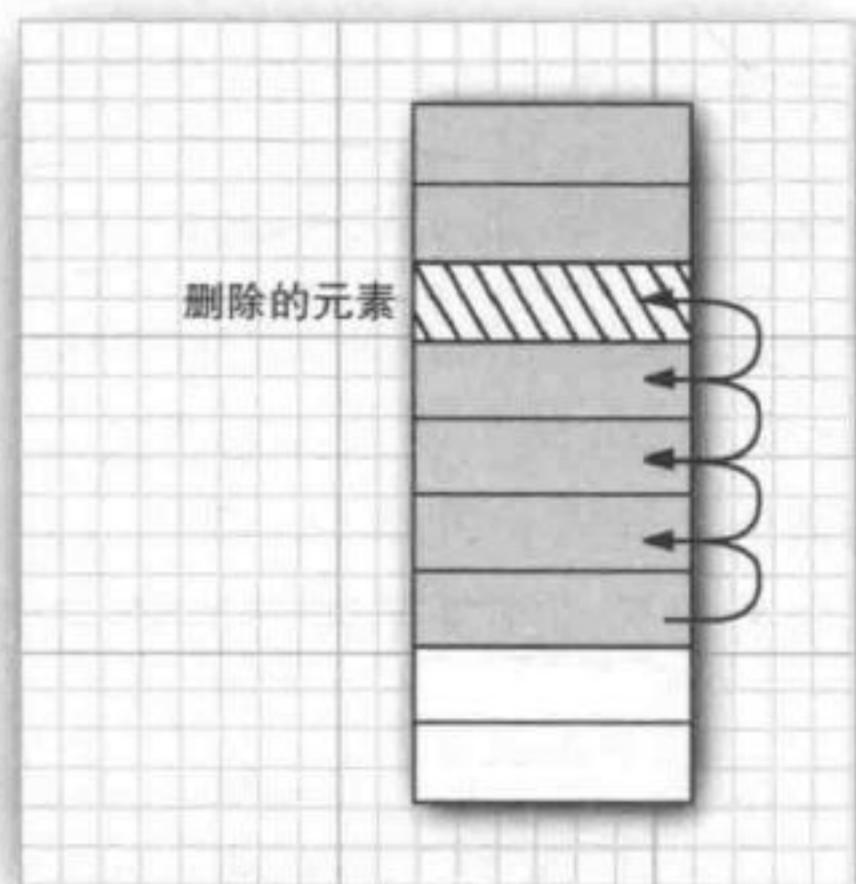


图 9-6 从数组中删除一个元素

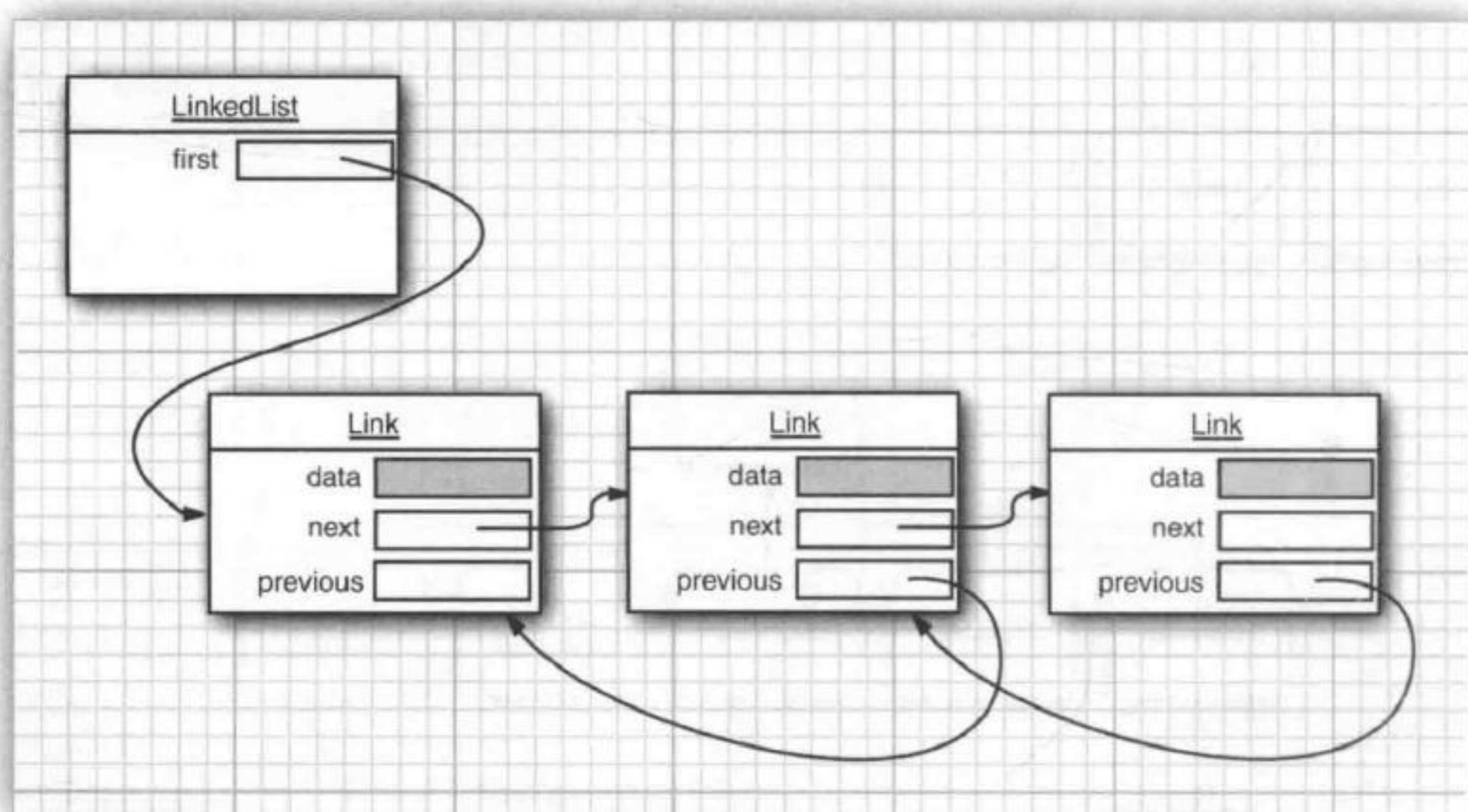


图 9-7 双向链表

从链表中间删除一个元素是一个很轻松的操作，只需要更新所删除元素周围的链接即可（见图 9-8）。

也许你曾经在数据结构课程中学习过如何实现链表。在链表中添加或删除元素时，绕来绕去的链接可能给你留下了糟糕的印象。如果真是如此的话，你肯定会为 Java 集合类库提供了一个可以直接使用的 `LinkedList` 类而感到高兴。

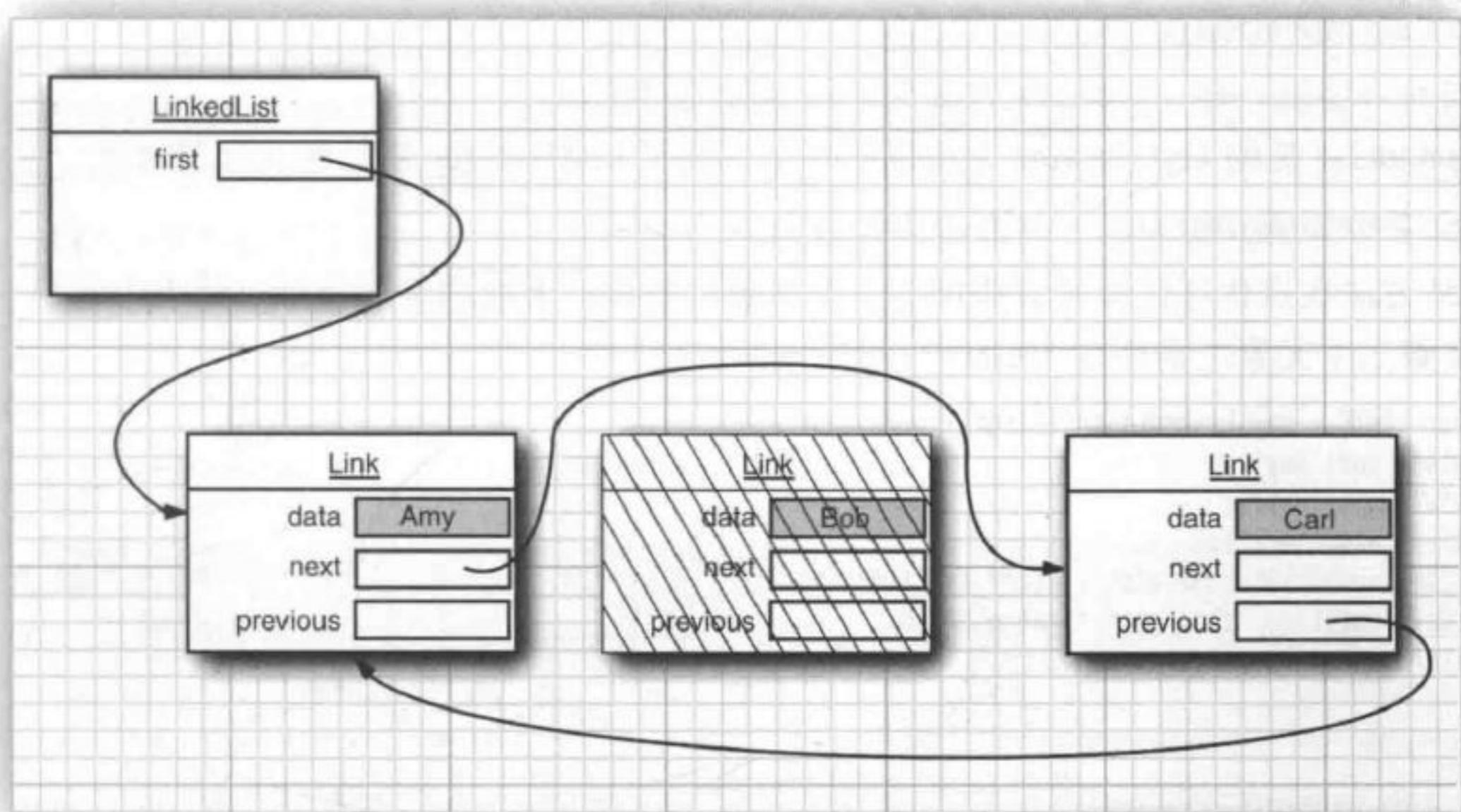


图 9-8 从链表中删除一个元素

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：

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

不过，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。因为迭代器描述了集合中的位置，所以这种依赖于位置的 `add` 方法将由迭代器负责。不过，只有对自然有序的集合使用迭代器来添加元素才有意义。例如，下一节将要讨论的集（set）数据类型中，元素是完全无序的。因此，`Iterator` 接口中没有 `add` 方法。实际上，集合类库提供了一个子接口 `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 `add` 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 方法会返回越过的对象。

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，在第二个元素之前添加 "Juliet" (见图 9-9)：

```
var 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");
```

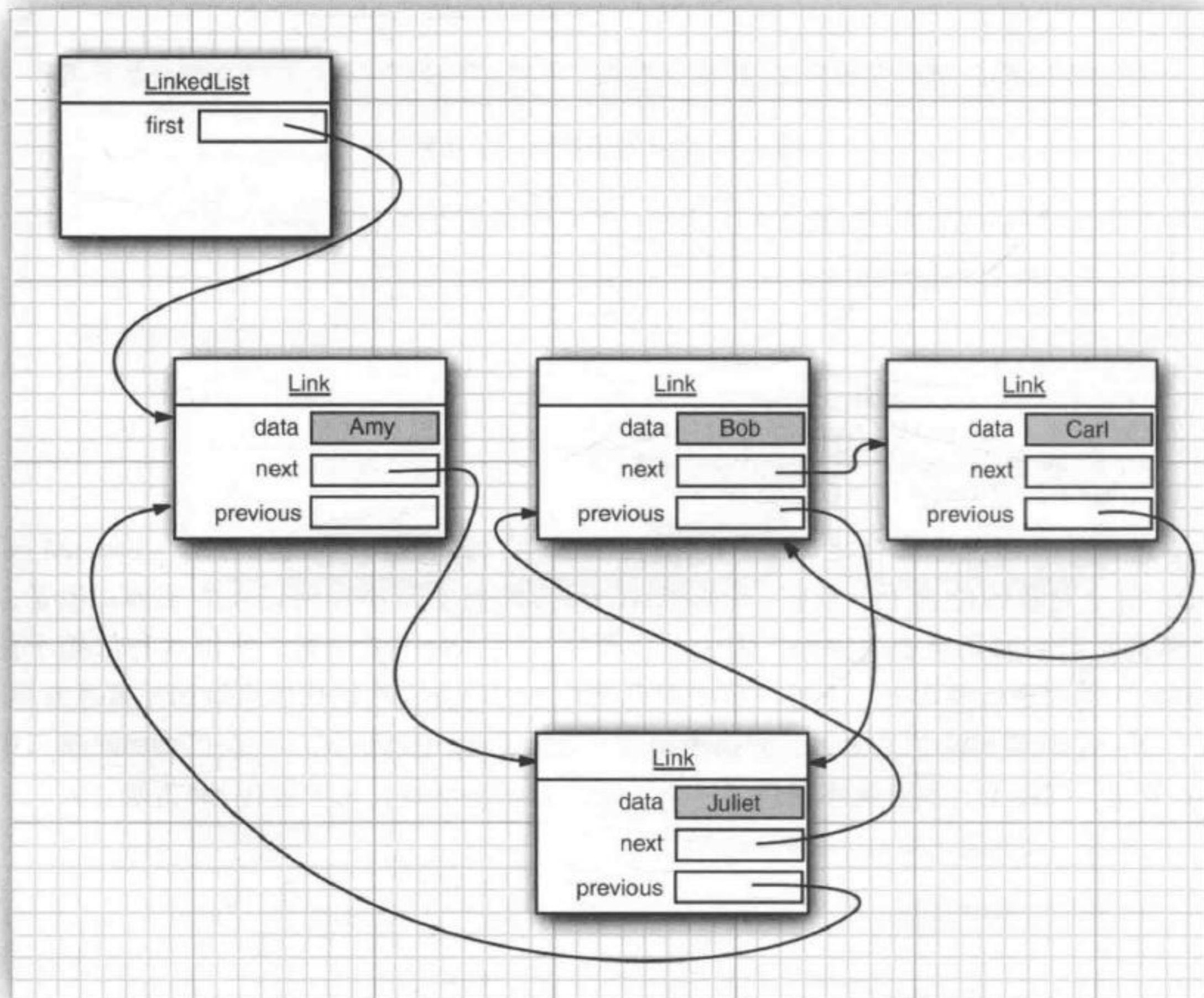


图 9-9 将一个元素添加到链表中

如果多次调用 add 方法，将按照提供元素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。

当用一个刚由 `listIterator` 方法返回并指向链表表头的迭代器来调用 `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` 时），添加的元素将成为列表的新表尾。如果链表有 n 个元素，会有 $n + 1$ 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（标记为 |）可以插入新元素：

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

注释：在用“光标”做类比时要当心。`remove` 操作与退格（Backspace）键的工作方式不太一样。在调用 `next` 之后，`remove` 方法确实会删除迭代器左侧的元素，这与退格键一样。但是，如果调用了 `previous`，则会删除迭代器右侧的元素。而且不能连续调用两次 `remove`。

`add` 方法只依赖于迭代器的位置，而 `remove` 方法不同，它依赖于迭代器的状态。

最后需要说明，`set` 方法会用一个新元素替换调用 `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替换列表的第一个元素：

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

可以想象，如果在某个迭代器修改集合时，另一个迭代器却在遍历这个集合，可能就会出现混乱。例如，假设一个迭代器指向一个元素前面的位置，而另一个迭代器刚刚删除了这个元素，现在前一个迭代器就是无效的，不能再使用。链表迭代器设计为可以检测到这种修改。如果一个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某个方法修改了，就会抛出一个 `ConcurrentModificationException` 异常。例如，考虑下面这段代码：

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

因为 `iter2` 检测出这个列表被外部修改，所以调用 `iter2.next` 会抛出一个 `ConcurrentModificationException` 异常。

为了避免发生并发修改异常，请遵循这样一个简单的规则：可以根据需要为一个集合关联多个迭代器，前提是这些迭代器只能读取集合。或者，可以关联一个能同时读写的迭代器。

检测并发修改的做法很简单。集合会跟踪更改操作（诸如添加或删除元素）的次数。每个迭代器都会为它负责的更改操作维护一个单独的更改操作数。在每个迭代器方法的开始处，迭代器会检查它自己的更改操作数与集合的更改操作数是否相等。如果不一致，就抛出

一个 `ConcurrentModificationException` 异常。

注释：不过，对于并发修改的检测有一个奇怪的例外。链表只跟踪对列表的结构性修改，例如，添加和删除链接。`set` 方法不被视为结构性修改。可以为一个链表关联多个迭代器，所有迭代器都可以调用 `set` 方法改变现有链接的内容。本章后面介绍的 `Collections` 类的许多算法都需要使用这个功能。

现在我们已经了解了 `LinkedList` 类的基本方法。可以使用 `ListIterator` 类从前后两个方向遍历链表中的元素，以及添加和删除元素。

在 9.2 节已经看到，`Collection` 接口还声明了操作链表的很多其他有用的方法。其中大部分方法都是在 `LinkedList` 类的超类 `AbstractCollection` 中实现的。例如，`toString` 方法会调用所有元素的 `toString`，并生成一个格式为 [A, B, C] 的长字符串。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，如果链表中已经包含一个等于 "Harry" 的字符串，调用 `staff.contains("Harry")` 将会返回 `true`。

Java 类库还提供了许多理论上存在一定争议的方法。链表不支持快速随机访问。如果要查看链表中的第 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，需要按整数索引访问元素时，程序员通常不选用链表。

然而，`LinkedList` 类还是提供了一个 `get` 方法，用来访问某个特定元素：

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

当然，这个方法的效率不太高。如果你发现自己正在使用这个方法，说明对于所要解决的问题，你可能使用了错误的数据结构。

绝对不要使用这个“虚假”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开头重新开始搜索。`LinkedList` 对象根本不会缓存位置信息。

注释：`get` 方法做了一个微小的优化：如果索引大于等于 `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，可以告诉你当前位置的索引。实际上，从概念上讲，因为 Java 迭代器指向两个元素之间的位置，所以可以有两个索引：`nextIndex` 方法返回下一次调用 `next` 方法时所返回元素的整数索引；`previousIndex` 方法返回下一次调用 `previous` 方法时所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，因为迭代器会维护当前位置的计数值。最后需要说明一点，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)` 会得到同一个元素，只是获得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，既然如此，最初为什么要使用链表呢？使用链表的唯一理由是尽可能地减少在列表中间

插入或删除元素的开销。如果列表只有很少几个元素，就完全可以使用 `ArrayList`。

建议一定要远离所有使用整数索引表示链表中位置的方法。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。

程序清单 9-1 中的程序具体使用了链表。它创建了两个列表，将它们合并在一起，然后从第二个列表中每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，要特别注意迭代器。可以画出迭代器位置示意图，你会发现这很有帮助，如下所示：

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

注意以下调用：

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

这会调用 `AbstractCollection` 类中的 `toString` 方法打印链表 `a` 中的所有元素。

程序清单 9-1 linkedList/LinkedListTest.java

```
1 package linkedList;
2
3 import java.util.*;
4
5 /**
6  * This program demonstrates operations on linked lists.
7  * @version 1.12 2018-04-10
8  * @author Cay Horstmann
9  */
10 public class LinkedListTest
11 {
12     public static void main(String[] args)
13     {
14         var a = new LinkedList<String>();
15         a.add("Amy");
16         a.add("Carl");
17         a.add("Erica");
18
19         var b = new LinkedList<String>();
20         b.add("Bob");
21         b.add("Doug");
22         b.add("Frances");
23         b.add("Gloria");
24
25         // merge the words from b into a
26
27         ListIterator<String> aIter = a.listIterator();
28         Iterator<String> bIter = b.iterator();
29
30         while (bIter.hasNext())
31         {
32             if (aIter.hasNext()) aIter.next();
33             aIter.add(bIter.next());
34         }
35     }
36 }
```

```

34     }
35
36     System.out.println(a);
37
38     // remove every second word from b
39
40     bIter = b.iterator();
41     while (bIter.hasNext())
42     {
43         bIter.next(); // skip one element
44         if (bIter.hasNext())
45         {
46             bIter.next(); // skip next element
47             bIter.remove(); // remove that element
48         }
49     }
50
51     System.out.println(b);
52
53     // bulk operation: remove all words in b from a
54
55     a.removeAll(b);
56
57     System.out.println(a);
58 }
59 }
```

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

- **ListIterator<E> listIterator()**
返回一个列表迭代器，用来访问列表中的元素。
- **ListIterator<E> listIterator(int index)**
返回一个列表迭代器，用来访问列表中的元素，第一次调用这个迭代器的 `next` 会返回给定索引的元素。
- **void add(int i, E element)**
在指定位置添加一个元素。
- **void addAll(int i, Collection<? extends E> elements)**
将一个集合中的所有元素添加到指定位置。
- **E remove(int i)**
删除并返回指定位置的元素。
- **E get(int i)**
获取指定位置的元素。
- **E set(int i, E element)**
用一个新元素替换指定位置的元素，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，如果没有匹配的元素将返

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，如果没有匹配的元素将返回 -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)

用一个新元素替换 next 或 previous 访问的上一个元素。如果在上一个 next 或 previous 调用之后列表结构被修改了，将抛出一个 IllegalStateException 异常。

- boolean hasPrevious()

当反向迭代处理列表时，如果还有可以访问的元素，返回 true。

- E previous()

返回前一个对象。如果已经到达列表开头，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的索引。

- int previousIndex()

返回下一次调用 previous 方法时将返回的元素的索引。

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，并将一个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开头或末尾。

- E getFirst()

- E getLast()

返回列表开头或末尾的元素。

- E removeFirst()

- E removeLast()

删除并返回列表开头或末尾的元素。

9.3.2 数组列表

在上一节中，我们了解了 List 接口和实现了这个接口的 LinkedList 类。List 接口描述一个

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