1 Java集合框架

1.1 集合接口与实现分离

Java集合类库将接口与实现分离，下面用队列来说明是如何分离的。

队列接口的最简形式类似下面这样：

```
public interface Queue<E>
{
    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。每一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>
{
    private int head;
    private int tail;
    private E[] elements;

    CircularArrayQueue(int capacity) { ... }
    public void add(E element) { ... }
    public E remove() { ... }
    public int size() { ... }
}

public class LinkedListQueue<E> implements Queue<E>
{
    private Link head;
    private Link tail;

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

在程序中使用队列时，可以使用接口变量存放集合引用。构造集合时，需要哪种实现，就使用相应的类来构造集合。之后的使用过程中就不需要考虑具体实现了，只需要调用方法即可。例如：

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

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：

```
/* java.util.Collection<E> */
int size()
    // 返回当前存储在集合中的元素个数
Iterator<E> iterator()
    // 返回一个用于访问集合中各个元素的迭代器
```

除此之外，`Collection`接口还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：

```
/* java.util.Collection<E> */
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(E element)
    // 向集合中添加元素。如果由于这个调用改变了集合，返回 true
```

```
boolean addAll(Collection<?> 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()
    // 删除集合中的所有元素
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(T[] arrayToFill)
    // 返回这个集合中的对象的数组。
    // 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
    // 如果 arrayToFill 不够大，就分配一个新数组，其成员类型与 arrayToFill 相同，其长度等
于集合的大小，并填充集合元素
```

还有一个有用的默认方法：

```
default boolean removeIf(Predicate<? super E> filter)
    // 从这个集合删除 filter 返回 true 的元素。如果由于这个调用改变了集合，返回 true
```

如果实现`Collection`接口的每个类都要提供这么多抽象方法，这将是一件很繁琐的事情。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`，它保持基础方法`size`和`iterator`仍为抽象方法，而为实现者实现了其他的抽象方法。这样，具体集合类可以扩展`AbstractCollection`类。

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`方法会抛出`NoSuchElementException`异常。因此，需要在调用`next`之前调用`hasNext`方法，如果迭代器对象还有可以访问的元素，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`时反复调用`next`方法。例如：

```
Collection<String> c = ...;
Iterator<String> iter = e.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    // 对 element 做操作
}
```

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)
{
    // 对 element 做操作
}
```

编译器简单地将for each循环转换为带有迭代器的循环。for each循环可以处理任何实现了`Iterable`接口的对象，这个接口只包含一个抽象方法：

```
public interface Iterable<E>
{
    Iterable<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，将对迭代器的每一个元素调用这个lambda表达式，直到没有元素为止。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。如果访问`HashSet`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。

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。Java迭代器的查找操作与位置变更紧密耦合，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。

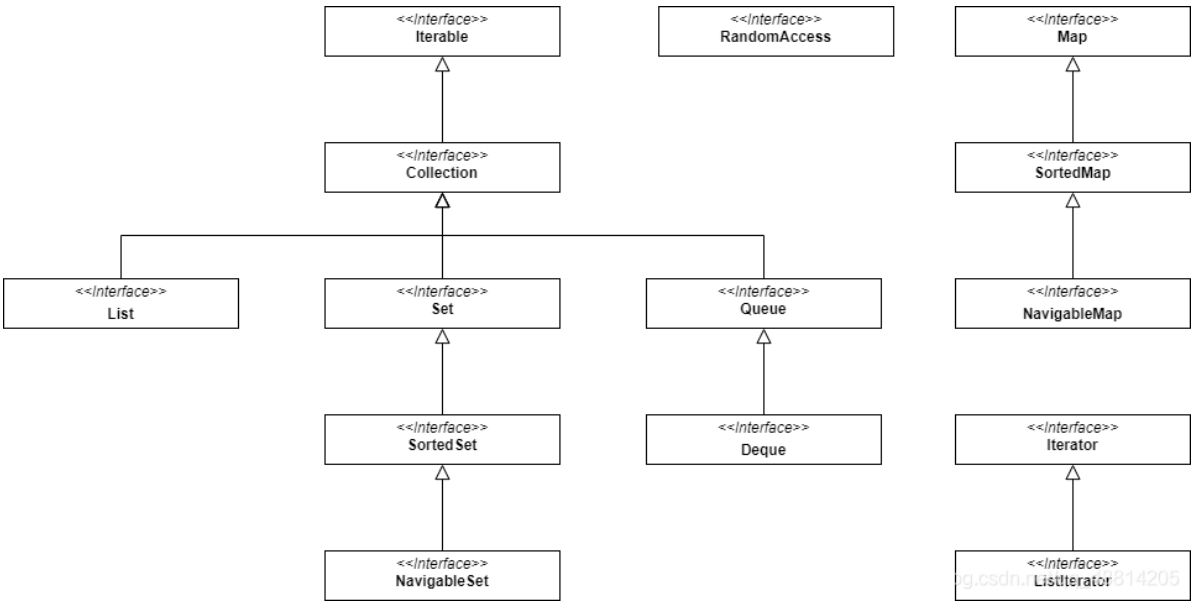
`Iterator`接口的`remove`方法将会删除上次调用`next`方法时返回的元素。例如，删除一个字符串集合中的第一个元素可以如下操作：

```
Iterator<String> it = c.iterator();
it.next(); // 跳过第一个元素
it.remove(); // 删除这个元素
```

`next`方法和`remove`方法调用之间存在依赖性。如果调用`remove`之前没有调用`next`，将会抛出`IllegalStateException`异常。例如，删除两个相邻的元素应该如下调用：

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

2 集合框架中的接口



集合有两个基本接口：`Collection`和`Map`。`Map`接口描述键/值对的对应关系，将键（key）映射到值（value），每个键最多映射到一个值。

`List`接口是一个有序集合，元素会添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，或者使用整数索引来访问。使用迭代器访问时必须顺序地访问元素，而使用整数索引可以按任意顺序访问元素，因此使用整数索引访问也称为随机访问。

`ListIterator`接口是`Iterator`的一个子接口，它是一种用于列表的迭代器，允许程序员向前或向后遍历列表。

Java 1.4引入了一个标记接口`RandomAccess`，这个接口不包含任何方法，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。例如，由数组实现的有序集合可以快速地随机访问，适合使用`List`接口；而用链表实现的有序集合不适合随机访问，最好使用迭代器来遍历。

`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。`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。

`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视图的方法。

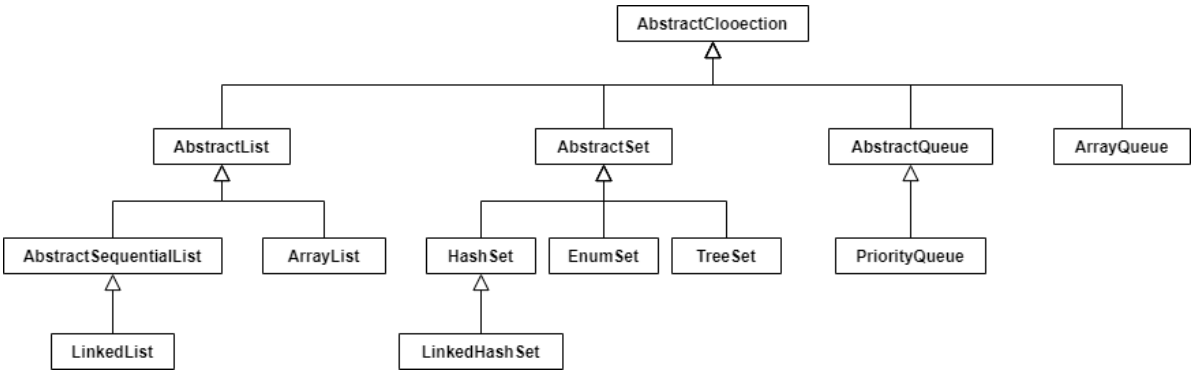
Java 6引入了接口`NavigableSet`和`NavigableMap`，其中包含一些用于搜索和遍历有序集和映射的方法。

3 具体集合

下表展示了Java类库中的集合：

集合类型	描述
`ArrayList`	可以动态增长和缩减的一个索引序列
`LinkedList`	可以在任何位置高效插入和删除的一个有序序列
`ArrayDeque`	实现为循环数组的一个双端队列
`HashSet`	没有重复元素的一个无序集合
`TreeSet`	一个有序集
`EnumSet`	一个包含枚举类型值的集
`LinkedHashSet`	一个可以记住元素插入次序的集
`PriorityQueue`	允许高效删除最小元素的一个集合
`HashMap`	存储键/值关联的一个数据结构
`TreeMap`	键有序的一个映射
`EnumMap`	键属于枚举类型的一个映射
`LinkedHashMap`	可以记住键/值项添加次序的一个映射
`WeakHashMap`	值不会在别处使用时就可以被垃圾回收的一个映射
`IdentityHashMap`	用 `==` 而不是 `equals` 比较键的一个映射

下图展示了这些类之间的关系：



3.1 链表

在Java中，所有链表都是双向链表。Java类库提供了`LinkedList`类实现链表，它实现了`List`接口，`List`接口和`LinkedList`类的方法列举如下：

```
/* java.util.List<E> */
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)
    // 在给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给定位置
```

```
E remove(int i)
    // 删除并返回给定位置的元素
E get(int i)
    // 获取给定位置的元素
E set(int i, E element)
    // 用一个新元素替换给定位置的元素，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

/* java.util.LinkedList<E> */
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)
    // 将指定元素添加到链表的头部
void addLast(E element)
    // 将指定元素添加到链表的尾部
E getFirst()
    // 返回链表头部的元素
E getLast()
    // 返回链表尾部的元素
E removeFirst()
    // 删除并返回链表头部的元素
E removeLast()
    // 删除并返回链表尾部的元素
```

集合类库提供了一个子接口 `ListIterator`，其中的方法如下：

```
/* java.util.ListIterator<E> */
void add(E newElement)
    // 在当前位置前添加一个元素
void set(E newElement)
    // 用新元素替换 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。如果已经到达列表器的头部，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
```

`LinkedList` 类有两个 `add` 方法，一个来自 `Collection` 接口，将对象添加到链表的尾部，有 `boolean` 类型的返回值；另一个来自 `List` 接口，将对象添加到索引值指定的位置，没有返回值。

`ListIterator` 接口也有一个 `add` 方法，将对象添加到迭代器位置之前。当用一个刚由 `listIterator` 方法返回并指向链表表头的迭代器调用 `add` 时，新添加的元素将变成链表的新表头。当迭代器越过链表的最后一个元素时，添加的元素将成为链表的新表尾。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。

如果某个迭代器修改集合时，另一个迭代器却在遍历这个集合，一定会出现混乱。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。为了避免发生并发修改异常，请遵循以下规则：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。

有一种简单的方法检测并发修改。集合可以跟踪更改操作的次数，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次数。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是否与集合的更改操作数相等，如果不一致，就抛出`ConcurrentModificationException`异常。

但是对于链表，只跟踪对链表的结构性修改，例如添加和删除。`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，所有的迭代器都调用`set`方法修改现有链接的内容。

链表不支持快速随机访问。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。例如，`get`方法用于获取给定位置的元素，但是每次调用`get`方法都要从链表的头部重新开始搜索，效率低下，应该避免使用这个方法。

下面的例子具体使用了链表：

```
import java.util.*;

public class LinkedListTest
{
    public static void main(String[] args)
    {
        LinkedList<String> a = new LinkedList<>();
        a.add("Amy");
        a.add("Carl");
        a.add("Erica");

        LinkedList<String> b = new LinkedList<>();
        b.add("Bob");
        b.add("Doug");
        b.add("Frances");
        b.add("Gloria");

        ListIterator<String> aIter = a.listIterator();
        Iterator<String> bIter = b.iterator();

        // 将 b 中的元素合并到 a 中
        while (bIter.hasNext())
        {
            if (aIter.hasNext()) aIter.next();
            aIter.add(bIter.next());
        }
        System.out.println(a);

        // 从 b 中每间隔一个元素删除一个元素
        bIter = b.iterator();
        while (bIter.hasNext())
        {
            bIter.next();
            if (bIter.hasNext())
            {
                bIter.next();
                bIter.remove();
            }
        }
```

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

        a.removeAll(b);
        System.out.println(a);
    }
}
```

3.2 散列集

散列表（hash table）是一种数据结构，可以用于快速查找对象。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字段得出的一个整数，有不同数据的对象将产生不同的散列码。

在Java中，散列表用链表数组实现。每个链表被称为**桶**（bucket）。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。如果桶中没有其他元素，此时将元素直接插入到桶中就行了。有时会遇到桶已经被填充的情况，这种现象被称为**散列冲突**（hash collision），这时需要将新对象与桶中的所有对象进行比较，查看这个对象是否已经存在。如果散列码合理地随机分布，桶的数目也足够大，需要比较的次数就会很少。

在Java 8中，桶满时会从链表变为平衡二叉树。

如果想更多地控制散列表的性能，可以指定一个初始的桶数。标准类库使用的桶数是2的幂，默认值为16，为表大小提供的任何值都将自动转换为2的下一个幂值。

如果散列表太满，就需要**再散列**（rehashed），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）可以确定何时对散列表进行再散列，它是一个介于0.0和1.0之间的浮点数，当散列表填充的百分比大于这个装填因子时，散列表进行再散列。

散列表可以用于实现很多重要的数据结构。其中最简单的是集类型。集是没有重复元素的元素集合。Java集合类库提供了一个`HashSet`类，它实现了基于散列表的集。它有4个构造器：

```
/* java.util.HashSet<E> */
HashSet()
    // 构造一个空散列集
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的散列集
HashSet(int initialCapacity, float loadFactor)
    // 构造一个空的具有指定容量和装填因子的散列集
```

`Collecion`接口中的方法被重写。例如，`add`方法首先在这个集中查找要添加的对象，如果不存在，就添加这个对象。`contains`方法用来快速查找某个元素是否已经在集中，它只查看一个桶中的元素，而不必查看集合中的所有元素。

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

下面的示例程序展示了散列集的使用：

```
import java.util.*;

public class SetTest
{
    public static void main(String[] args)
    {
        HashSet<String> words = new HashSet<>();
```

```
        long totalTime = 0;

        try (Scanner in = new Scanner(System.in))
        {
            while (in.hasNext())
            {
                String word = in.next();
                long callTime = System.currentTimeMillis();
                words.add(word);
                callTime = System.currentTimeMillis() - callTime;
                totalTime += callTime;
            }
        }

        Iterator<String> iter = words.iterator();
        for (int i = 1; i <= 20 && iter.hasNext(); i++)
        {
            System.out.println(iter.next());
        }
        System.out.println("...");
        System.out.println(words.size() + " distinct words. " + totalTime + "
milliseconds.");
    }
}
```

3.3 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，每次添加元素时，都会将其放在正确的排序位置上，排序是用红黑树实现的。因此，迭代器总是以有序的顺序访问每个元素。

要使用树集，必须能够比较元素。这些元素必须实现`Comparable`接口，或者构造树集时提供一个`Comparator`。`TreeSet`的构造器如下：

```
TreeSet()
    // 构造一个空树集
TreeSet(Comparator<? extends E> comparator)
    // 构造一个空树集
TreeSet(Collection<? extends E> elements)
    // 构造一个树集，并添加一个集合中的所有元素
TreeSet(SortedSet<E> s)
    // 构造一个树集，并添加一个有序集中的所有元素
```

将一个元素添加到树中要比添加到散列表中慢，但是，与检查数组或链表中的重复元素相比，使用树会快很多。

使用散列集还是树集取决于数据。如果不需要数据是有序的，就没有必要付出排序的开销。对于某些数据来说，对其进行排序要比给出一个散列函数更加困难。

从Java 6起，`TreeSet`类实现了`NavigableSet`接口。这个接口增加了几个查找元素以及反向遍历的便利方法：

```
/* java.util.NavigableSet<E> */
E higher(E value)
    // 返回大于 value 的最小元素，如果没有这样的元素则返回 null
E lower(E value)
```

```
    // 返回小于 value 的最大元素，如果没有这样的元素则返回 null
E ceiling(E value)
    // 返回大于等于 value 的最小元素，如果没有这样的元素则返回 null
E floor(E value)
    // 返回小于等于 value 的最小元素，如果没有这样的元素则返回 null
E pollFirst()
    // 删除并返回这个集中的最大元素，这个集为空时返回 null
E pollLast()
    // 删除并返回这个集中的最小元素，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
```

3.4 队列与双端队列

队列允许在尾部添加元素，在头部删除元素。双端队列（deque）允许在头部和尾部添加或删除元素，不支持在队列中间添加元素。Java 6中引入了`Deque`接口，`ArrayDeque`和`LinkedList`类实现了这个接口，这两个类都可以提供双端队列。相关的API列举如下：

```
/* java.util.Queue<E> */
boolean add(E element)
    // 如果队列没有满，将给定的元素添加到队尾并返回 true。如果队列已满，抛出
IllegalStateException
boolean offer(E element)
    // 如果队列没有满，将给定的元素添加到队尾并返回 true。如果队列已满，返回 false
E remove()
    // 如果队列不为空，删除并返回队头的元素。如果队列为空，抛出 NoSuchElementException
E poll()
    // 如果队列不为空，删除并返回队头的元素。如果队列为空，返回 null
E element()
    // 如果队列不为空，返回队头的元素。如果队列为空，抛出 NoSuchElementException
E peek()
    // 如果队列不为空，返回队头的元素。如果队列为空，返回 null

/* java.util.Deque<E> */
void addFirst(E element)
    // 将给定的对象添加到双端队列的队头。如果双端队列已满，抛出 IllegalStateException
void addLast(E element)
    // 将给定的对象添加到双端队列的队尾。如果双端队列已满，抛出 IllegalStateException
boolean offerFirst(E element)
    // 将给定的对象添加到双端队列的队头。如果双端队列已满，返回 false
boolean offerLast(E element)
    // 将给定的对象添加到双端队列的队尾。如果双端队列已满，返回 false
E removeFirst()
    // 如果双端队列不为空，删除并返回队头的元素。如果双端队列为空，抛出
NoSuchElementException
E removeLast()
    // 如果双端队列不为空，删除并返回队尾的元素。如果双端队列为空，抛出
NoSuchElementException
E pollFirst()
    // 如果双端队列不为空，删除并返回队头的元素。如果双端队列为空，返回 null
E pollLast()
    // 如果双端队列不为空，删除并返回队尾的元素。如果双端队列为空，返回 null
E getFirst()
    // 如果双端队列不为空，返回队头的元素。如果双端队列为空，抛出 NoSuchElementException
E getLast()
    // 如果双端队列不为空，返回队尾的元素。如果双端队列为空，抛出 NoSuchElementException
```

```
E peekFirst()
    // 如果双端队列不为空，返回队头的元素。如果双端队列为空，返回 null
E peekLast()
    // 如果双端队列不为空，返回队尾的元素。如果双端队列为空，返回 null

/* java.util.ArrayDeque<E> */
ArrayDeque()
    // 用初始容量 16 构造一个无限定双端队列
ArrayDeque(int initialCapacity)
    // 用给定的初始容量构造一个无限定双端队列
```

3.5 优先队列

优先队列（priority queue）中的元素可以按照任意的顺序插入，但会按照有序的顺序进行检索。无论何时调用`remove`方法，总会获得当前优先队列中最小的元素。不过，迭代器并不是按照有序顺序来访问元素。

优先队列是由堆（heap）实现的。堆是一个可以自组织的二叉树，其添加和删除操作可以让最小的元素移动到根。

与`TreeSet`类一样，`PriorityQueue`类要求要保存的对象实现`Comparable`接口，或者在构造优先队列时提供`Comparator`。`PriorityQueue`类的构造器如下：

```
PriorityQueue()
    // 构造一个优先队列
PriorityQueue(int initialCapacity)
    // 构造一个指定容量的优先队列
PriorityQueue(int initialCapacity, Comparator<? super E> c)
    // 构造一个指定容量的优先队列，并使用指定的比较器对元素进行排序
```

4 映射

4.1 基本映射操作

Java类库为映射提供了两个通用的实现：`HashMap`和`TreeMap`。这两个类都实现了`Map`接口。相关的API列举如下：

```
/* java.util.Map<K, V> */
V get(Object key)
    // 返回与键关联的值。如果映射中没有这个键，则返回 null
default V getOrDefault(Object key, V defaultValue)
    // 返回与键关联的值。如果映射中没有这个键，则返回 defaultValue
V put(K key, V value)
    // 将关联的一对键和值放到映射中。如果这个键已经存在，新的对象将取代与这个键关联的旧对象
    // 这个方法将返回键关联的旧值。如果之前没有这个键，则返回 null
void putAll(Map<? extends K, ? extends V> entries)
    // 将给定映射中的所有映射条目添加到这个映射中
boolean containsKey(Object key)
    // 如果在映射中已经有这个键，返回 true
boolean containsValue(Object value)
    // 如果在映射中已经有这个值，返回 true
default void forEach(BiConsumer<? super K, ? super V> action)
    // 对这个映射中的所有键/值对应用这个动作
V remove(Object key)
    // 从映射中删除给定键对应的元素
```

```
int size()
    // 返回映射中的元素数

/* java.util.HashMap<K, V> */
HashMap()
    // 构造一个空散列映射
HashMap(int initialCapacity)
    // 用给定的容量构造一个空散列映射
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

/* java.util.TreeMap<K, V> */
TreeMap()
    // 构造一个空的树映射
TreeMap(Comparator<? super K> c)
    // 构造一个空的树映射，并使用指定的比较器对键进行排序
TreeMap(Map<? extends K, ? extends V> entries)
    // 构造一个树映射，并将某个映射中的所有映射条目添加到树映射中
TreeMap(SortedMap<? extends K, ? extends V> entries)
    // 构造一个树映射，将某个有序映射中的所有映射条目添加到树映射中，并使用与给定的有序映射相同的比较器
```

散列映射对键进行散列，树映射根据键的顺序将元素组织为一个搜索树。散列或比较函数只应用于键，与键关联的值不进行散列或比较。散列稍快一些，如果不需要按照有序的顺序访问键，最好选择散列映射。

4.2 更新映射条目

下面列举了一些更新映射条目的方法：

```
/* java.util.Map<K, V> */
default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction)
    // 如果 key 与一个非 null 值 v 关联，将函数应用到 v 和 value ，将 key 与结果关联；如果结果为 null，则删除这个键
    // 如果 key 没有关联值或者关联到 null 值，将 key 与 value 关联，返回 get(key)
default V compute(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
    // 将函数应用到 key 和 get(key)，将 key 与结果关联；如果结果为 null，则删除这个键。返回 get(key)
default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
    // 如果 key 与一个非 null 值 v 关联，将函数应用到 key 和 v，将 key 与结果关联；如果结果为 null，则删除这个键
    // 返回 get(key)
default V computeIfAbsent(K key, Function<? super K, ? extends V>
mappingFunction)
    // 如果 key 没有关联值或者关联到 null 值，将函数应用到 key，将 key 与结果关联；如果结果为 null，则删除这个键
    // 返回 get(key)
default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
    // 在所有映射条目上应用这个函数。如果结果非 null，将键与结果关联；如果结果为 null，则将相应的键删除
default V putIfAbscent(K key, V value)
    // 如果 key 不存在或者与 null 关联，则将它与 value 关联，并返回 null。否则返回关联的值
```

处理映射的一个难点就是更新映射条目。正常情况下，可以得到与键关联的值，完成更新，再放回更新后的值。不过，当键第一次出现时，前面的方法就失效了。例如，使用映射统计一个单词在文件中出现的次数，每看到一个单词就将计数器增1：

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

当`word`第一次出现时，调用`get`会返回`null`，因此会出现`NullPointerException`异常。一种简单的补救是使用`getOrDefault`方法：

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

另一种方法是首先调用`putIfAbsent`方法：

```
counts.putIfAbsent(word, 0); // 如果键不存在，则将这个键与 0 关联；如果键存在，则返回关联的值
counts.put(word, counts.get(word) + 1); // 确保键存在，get 方法一定不会为 null
```

`merge`方法可以简化这个常见操作：

```
counts.merge(word, 1, Integer::sum);
    // 如果键不存在，将键与 1 关联
    // 如果键存在，就对原值和 1 使用 Integer::sum 函数，也就是将原值加 1
```

4.3 映射视图

集合框架不认为映射本身是一个集合。不过，可以得到映射的视图（view），这是实现了`Collection`接口或某个子接口的对象。

有3种映射视图：键集、值集合以及键/值对集。键和键/值对可以构成集，因为映射中一个键只能有一个副本；而值不能构成集，只是一个集合。下面的方法会分别返回这3个视图：

```
/* java.util.Map<K, V> */
Set<K> keySet()
    // 返回映射中所有键的一个集视图
    // 可以从这个集中删除元素，所删除的键和相关联的值将从映射中删除，但是不能添加元素
Collection<V> values()
    // 返回映射中所有值的一个集合视图。
    // 可以从这个集合中删除元素，所删除的值和相应的键将从映射中删除，但是不能添加元素
Set<Map.Entry<K, V>> entrySet()
    // 返回 Map.Entry 对象（映射中的键/值对）的一个集视图
    // 可以从这个集中删除元素，它们将从映射中删除，但是不能添加元素
```

`keySet`的返回值不是`HashSet`或`TreeSet`，而是实现了`Set`接口的另外某个类的对象。`Set`接口扩展了`Collection`接口，因此可以像使用任何集合一样使用`keySet`。例如：

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

`Map.Entry<K, V>`接口用于描述键/值对，接口中的重要方法如下：

```
/* java.util.Map.Entry<K, V> */
K getKey() // 返回这个映射条目的键
V getValue() // 返回这个反射条目的值
V setValue(V newValue) // 将相关映射中的值改为新值，并返回原来的值
```

如果想同时查看键和值，可以通过枚举映射条目来避免查找值。例如：

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

如果在键集视图上调用迭代器的`remove`方法，会从映射中删除这个键和与它关联的值。不过，不能向键集视图中添加元素，如果试图调用`add`方法，会抛出`UnsupportedOperationException`异常。另外两个视图有同样的限制。

4.4 弱散列映射

`WeakHashMap`可以自动回收那些不在别处使用的映射条目。当对键的唯一引用来自散列表映射条目时，这个数据结构将与垃圾回收器协同工作一起删除键/值对。

下面介绍这种机制的内部工作原理。`WeakHashMap`使用弱引用保存键。`WeakReference`对象将包含另一个对象的引用，在这里就是一个散列表键。对于这种类型的对象，垃圾回收器采用一种特有的方式进行处理。正常情况下，如果垃圾回收器发现某个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，垃圾回收器也会将其回收，但会将引用这个对象的弱引用放入一个队列。`WeakHashMap`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，并且已经回收。于是，`WeakHashMap`将删除相关联的映射条目。

4.5 链接散列集与映射

`LinkedHashSet`和`LinkedHashMap`类会记住插入元素项的顺序，这样就可以避免散列表中的项看起来顺序是随机的。散列集中的所有元素放在散列表中，同时这些元素还组成了一个双向链表，向散列表添加元素的同时，也会将元素按顺序并入双向链表。

链接散列映射也可以使用访问顺序来迭代处理映射条目。每次调用`get`或`put`时，受到影响的项会被移动到链表的尾部（只影响项在链表中的位置，而散列表的桶不受影响）。要构造这样的散列映射，需要使用下面的构造器：

```
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
    // 用给定的初始容量、装填因子和访问顺序构造一个空的链接散列映射
    // 对于第三个参数，true 表示访问顺序，false 表示插入顺序
```

访问顺序对于实现缓存的“最近最少使用”原则十分重要。例如，你可能希望将访问频率高的元素放在内存中，而访问频率低的元素从数据库中读取。当在表中找不到元素相而且表已经相当满时，可以得到表的一个迭代器，并删除它枚举的前几个元素，这些项是近期最少使用的元素。

甚至可以让这个过程自动化，为此需要使用下面的方法：

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
    // 默认实现会返回 false，即在默认情况下，老元素不会被删除
    // eldest 参数是预期可能删除的元素。如果想删除 eldest 元素，就要覆盖为返回 true
    // 可以重新定义这个方法，以便有选择地返回 true。例如，如果 eldest 元素符合一个条件，或者映射超过了一定大小，则返回 true
    // put 和 putAll 方法会调用这个方法，它允许实现者在添加新条目时删除旧条目
    // 如果映射表示缓存，这将非常有用，它允许映射通过删除过时的条目来减少内存消耗
```

可以构造`LinkedHashMap`的一个子类，并覆盖这个方法。例如：

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

4.6 枚举集与映射

`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，相应的位就被置为1。

`EnumSet`类没有公共构造器，要使用静态工厂方法构造这个集：

```
/* java.util.EnumSet<E enteds Enum<E>> */
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可变集
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)
    // 返回一个初始为空的可变集
static <E extends Enum<E>> EnumSet<E> range(E from, E to)
    // 返回一个包含 from~to 之间的所有值（包括两个边界元素）的可变集
static <E extends Enum<E>> EnumSet<E> of(E e)
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2)
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)
    // 返回包括不为 null 的给定元素的可变集
```

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。它可以直接且高效地实现为一个值数组。需要在构造器中指定键类型：

```
/* java.util.EnumMap<K extends Enum<K>, V> */
EnumMap(Class<K> keyType) // 构造一个键为指定类型的空的可变映射
```

4.7 标识散列映射

`IdentityHashMap`类中，键的散列值不是用`hashCode`方法计算的，而是用`System.identityHashCode`方法计算的。这是`Object.hashCode`根据对象的内存地址计算散列码时所使用的方法。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，也被视为不同的对象。在实现对象遍历算法时，这个类非常有用，可

以用来跟踪哪些对象已经被遍历过。

```
/* java.util.IdentityHashMap<K, V> */
IdentityHashMap()
    // 构造一个空的标识散列映射，默认的最大容量为 21
IdentityHashMap(int expectedMaxSize)
    // 构造一个空的标识散列映射，其容量是大于 1.5 * expectedMaxSize 的 2 的最小幂值。

/* java.lang.System */
static int identityHashCode(Object obj)
    // 返回 Object.hashCode 计算的相同散列码（根据对象的内存地址得出）
```

5 视图与包装器

5.1 小集合

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

```
/* java.util.List */
static <E> List<E> of()
static <E> List<E> of(E e1)
static <E> List<E> of(E e1, E e2)
...
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E
e10)
static <E> List<E> of(E... elements)
    // 生成给定元素的一个不可变的列表，元素不能为 null

/* java.util.Set */
static <E> Set<E> of()
static <E> Set<E> of(E e1)
static <E> Set<E> of(E e1, E e2)
...
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E
e10)
static <E> Set<E> of(E... elements)
    // 生成给定元素的一个不可变的集，元素不能为 null

/* java.util.Map */
static <K, V> Map<K, V> of()
static <K, V> Map<K, V> of(K k1, V v1)
...
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5,
V v5,
    K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)
    // 生成给定键和值的一个不可变的映射，键和值不能为 null
static <K, V> Map.Entry<K, V> entry(K k, V v)
    // 生成给定键和值的一个不可变的映射条目，键和值不能为 null
static <K, V> Map<K, V> ofEntries(Map.Entry<? extends K, ? extends V>...
entries)
    // 生成给定映射条目的一个不可变的映射
```

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

对于`Map`接口，则无法提供参数可变的版本，因为参数类型会在键和值类型之间交替。不过它有一个静态方法`ofEntries`，能接受任意多个`Map.Entry<K, V>`对象，可以用静态方法`entry`创建这些对象，例如：

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

用这些方法得到的集合对象是不可修改的，如果试图改变它们的内容，会导致`UnsupportedOperationException`异常。如果需要一个可更改的集合，可以把这个不可更改的集合传递到构造器，例如：

```
ArrayList<String> names = new ArrayList<>(List.of("Peter", "Paul", "Mary"));
```

下面的方法会返回一个实现了`List`接口的不可变的对象：

```
/* java.util.Collections */
static <E> List<E> nCopies(int n, E value)
    // 生成一个不可变的列表，包含 n 个相等的值
```

在Java 9之前，可以使用下面的方法得到列表视图：

```
/* java.util.Arrays */
static <E> List<E> asList(E... array)
    // 返回一个数组中元素的列表视图
    // 这个列表是可修改的，但其大小不可变。也就是说，可以在这个列表上使用 set，但是不能使用
add 或 remove
```

另外还有一些遗留的方法：

```
/* java.util.Collections */
static <E> List<E> singletonList(E value)
static <E> Set<E> singleton(E value)
static <K, V> Map<K ,V> singletonMap(K key, V value)
    // 生成一个单例列表、集或映射
static <E> List<E> emptyList()
static <T> Set<T> emptySet()
static <E> SortedSet<E> emptySortedSet()
static <E> NavigableSet<E> emptyNavigableSet()
static <K, V> Map<K, V> emptyMap()
static <K, V> SortedMap<K, V> emptySortedMap()
static <K, V> NavigableMap<K, V> emptyNavigableMap()
static <T> Enumeration<T> emptyEnumeration()
static <T> Iterator<T> emptyIterator()
static <T> ListIterator<T> emptyListIterator()
    // 生成空集合、映射或迭代器
```

5.2 子范围

可以为很多集合建立子范围视图。对于列表，可以使用`subList`方法获得列表子范围的视图：

```
/* java.util.List<E> */
List<E> subList(int firstIncluded, int firstExcluded)
    // 返回给定位置范围内的所有元素的列表视图，第一个索引包含在内，第二个索引不包含在内
```

可以对子范围应用任何操作，而且操作会自动反映到整个列表。

对于有序集和映射，可以使用排序顺序建立子范围，相关方法如下：

```
/* java.util.SortedSet<E> */
SortedSet<E> subSet(E firstIncluded, E firstExcluded)
SortedSet<E> headSet(E firstExcluded)
SortedSet<E> tailSet(E firstIncluded)
    // 返回给定范围内元素的视图

/* java.util.SortedMap */
SortedMap<K, V> subMap(K, firstIncluded, K firstExcluded)
SortedMap<K, V> headMap(K firstExcluded)
SortedMap<K, V> tailMap(K, firstIncluded)
    // 返回键在给定范围内的映射条目的映射视图
```

Java 6引入的`NavigableSet`和`NavigableMap`接口允许更多地控制这些子范围操作，可以指定是否包括边界：

```
/* java.util.NavigableSet<E> */
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)
    // 返回给定范围内元素的视图，boolean 标志决定是否包含边界

/* java.util.NavigableMap<K, V> */
NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean
toIncluded)
NavigableMap<K, V> headMap(K to, boolean toIncluded)
NavigableMap<K, V> tailMap(K from, boolean fromIncluded)
    // 返回键在给定范围内的映射条目的映射视图，boolean 标志决定是否包含边界
```

5.3 不可修改的视图

`Collections`类还有几个方法，可以生成集合的不可修改视图。这些视图对现有集合增加了一个运行时检查，如果发现试图对集合进行修改，就抛出`UnsupportedOperationException`异常，集合仍保持不变。

可以使用下面8个方法来获得不可修改视图：

```
/* java.util.Collections */
static <E> Collection<E> unmodifiableCollection(Collection<E> c)
static <E> List<E> unmodifiableList(List<E> c)
static <E> Set<E> unmodifiableSet(Set<E> c)
static <E> SortedSet<E> unmodifiableSortedSet(SortedSet<E> c)
static <E> NavigableSet<E> unmodifiableNavigableSet(NavigableSet<E> c)
static <K, V> Map<K, V> unmodifiableMap(Map<K, V> c)
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> c)
static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, V> c)
```

不可修改的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对集合进行修改，而且仍然可以对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具体的集合对象，所以只能访问接口中定义的方法，而不能使用集合类中定义的便利方法。例如，`LinkedList`类有一些便利方法，如`addFirst`和`addLast`，它们都不是`List`接口的方法，不能通过不可修改的视图访问这些方法。

`unmodifiableCollection`方法将返回一个集合，它的`equals`方法不调用底层集合的`equals`方法。实际上，它继承了`Object`类的`equals`方法，这个方法只是检测两个对象的内存地址是否相同。如果将集或列表转换成集合，就无法检测其内容是否相同了。视图将以同样的方式处理`hashCode`方法。不过，`unmodifiableSet`和`unmodifiableList`方法会使用底层集合的`equals`方法和`hashCode`方法。

5.4 同步视图

如果从多个线程访问集合，必须确保集合不会被意外地破坏。类库使用视图机制来确保常规集合是线程安全的，而没有实现线程安全的集合类。

下面的方法将构造同步视图，确保视图的方法是同步的：

```
/* java.util.Collections */
static <E> Collection<E> synchronizedCollection(Collection<E> c)
static <E> List<E> synchronizedList(List<E> c)
static <E> Set<E> synchronizedSet(Set<E> c)
static <E> SortedSet<E> synchronizedSortedSet(SortedSet<E> c)
static <E> NavigableSet<E> synchronizedNavigableSet(NavigableSet<E> c)
static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> c)
```

5.5 检查型视图

检查型视图用来对泛型类型可能出现的问题提供调试支持，如果插入一个错误类型的元素，视图的方法会抛出一个`ClassCastException`异常。

下面的方法用于构造检查型视图：

```
/* java.util.Collections */
static <E> Collection<E> checkedCollection(Collection<E> c)
static <E> List<E> checkedList(List<E> c)
static <E> Set<E> checkedSet(Set<E> c)
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> c)
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> c)
static <K, V> Map<K, V> checkedMap(Map<K, V> c)
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> c)
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> c)
```

检查型视图受限于虚拟机可以完成的运行时检查。例如，对于`ArrayList<Pair<String>>`由于虚拟机有一个原始`Pair`类，所以无法阻止插入`Pair<Date>`。

6 算法

6.1 排序与混排

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

```
/* java.util.Collections */
static <T extends Comparable<? super T>> void sort(List<T> elements)
    // 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*logn)
```

如果想采用其他方式对列表进行排序，可以使用`List`接口的`sort`方法并传入一个`Comparator`对象：

```
/* java,util.List<E> */
default void sort(Comparator<? super T> comparator)
    // 使用给定的比较器对列表进行排序
```

如果想按照降序对列表进行排序，可以使用静态的便利方法`Comparator.reverseOrder`：

```
/* java.util.Comparator<T> */
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
    // 生成一个比较器，将逆置 Comparable 接口提供的顺序
```

`Comparator`接口的`reversed`方法将逆置比较器提供的顺序：

```
/* java.util.Comparator<T> */
default Comparator<T> reversed()
```

`sort`方法的排序操作是，将所有元素转入一个数组，对数组进行排序，然后将排序后的序列复制回列表。集合类库中使用的排序算法比快速排序慢一些，但它是稳定的。

`sort`方法要求待排序的列表必须是可修改的，但不要求必须可改变大小。可修改（modifiable）是指列表支持`set`方法，可改变大小（resizable）是指列表支持`add`和`remove`方法。

`Collections`类有一个`shuffle`方法，它会随机地混排列表中元素的顺序：

```
/* java.util.Collections */
static void shuffle(List<?> elements)
static void shuffle(List<?> elements, Random r)
    // 随机打乱列表中元素的顺序。这个算法的时间复杂度是 O(n*a(n))，n 是列表的长度，a(n) 是访问元素的平均时间
```

如果提供的列表没有实现`RandomAccess`接口，`shuffle`方法会将元素复制到数组中，打乱数组元素的顺序，再讲打乱顺序后的元素复制回列表。

6.2 二分查找

`Collections`类的`binarySearch`方法实现了二分查找算法：

```
/* java.util.Collections */
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，如果元素类型实现了 RandomAccess 接口，就使用二分查找，其他情况下使用线性查找
    // 这个方法的时间复杂度为 O(a(n)*logn)，n 是列表的长度，a(n) 是访问一个元素的平均时间
```

要求集合必须是有序的，否则算法会返回错误的答案。要想查找某个元素，必须提供集合以及要查找的元素。如果集合没有采用`Comparable`接口的`compareTo`方法进行排序，那么还要提供一个比较器对象。

如果`binarySearch`方法返回一个非负的值，这表示匹配对象的索引；如果返回负值，则表示没有匹配的元素。如果返回负值`i`，说明这个元素不在列表中，可以将这个元素插入索引为`-i-1`的位置，以保持列表的有序性。

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查找链表的中间元素，二分查找就完全失去了优势。因此，如果为`binarySearch`方法提供一个链表，它将自动退化为线性查找。

6.3 简单算法

```
/* java.util.Collections */
static <T extends Object & Comparable<? super T>> T min(Collection<T> elements)
static <T> min(Collection<T> elements, Comparator<? super T> c)
    // 返回集合中最小的元素
static <T extends Object & Comparable<? super T>> T max(Collection<T> elements)
static <T> max(Collection<T> elements, Comparator<? super T> c)
    // 返回集合中最大的元素
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)
    // 将所有的值添加到给定的集合中。如果集合改变了，则返回 true
static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)
    // 用 newValue 替换所有值为 oldValue 的元素
static int indexOfSubList(List<?> l, List<?> s)
    // 返回 l 中第一个等于 s 的子列表的起始索引。如果 l 中不存在等于 s 的子列表，则返回 -1
static int lastIndexOfSubList(List<?> l, List<?> s)
    // 返回 l 中最后一个等于 s 的子列表的起始索引。如果 l 中不存在等于 s 的子列表，则返回 -1
static void swap(List<?> l, int i, int j)
    // 交换给定索引位置的两个元素
static void reverse(List<?> l)
    // 逆置列表中元素的顺序。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)
    // 旋转列表中的元素，将索引 i 的元素移动到位置 (i + d) % l.size()
    // 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)
    // 返回集合中与对象 o 相等的元素的个数
boolean disjoint(Collection<?> c1, Collection<?> c2)
    // 如果两个集合没有共同的元素，则返回 true

/* java.util.List<E> */
default void replaceAll(UnaryOperator<E> op)
    // 对这个列表的所有元素应用这个操作
```

6.4 批操作

`removeAll`和`retainAll`方法可以成批地操作元素，利用批操作可以实现更加复杂的功能。例如，计算两个集的交集可以实现如下：

```
public static <E> Set<E> intersection(Set<E> firstSet, Set<E> secondSet)
{
    Set<E> result = new HashSet<>(firstSet);
    result.retainAll(secondSet);
    return result;
}
```

6.5 集合与数组的转换

如果需要把一个数组转换为集合，`List.of`包装器可以达到这个目的。例如：

```
String[] values = ...;
HashSet<String> staff = new HashSet<>(list.of(values));
```

从集合得到数组可以使用`toArray`方法，例如：

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

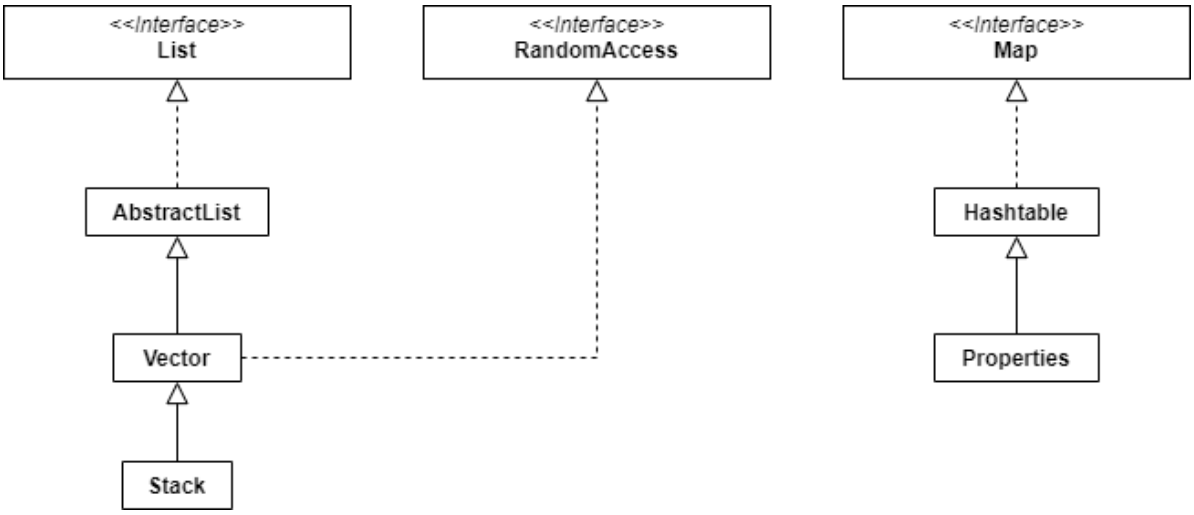
`toArray`方法返回的数组是`Object[]`类型，不能改变它的类型。要想得到正确类型的数组，必须使用`toArray`方法的一个变体，提供一个指定类型且长度为0的数组，这样返回的数组就会创建为相同的数组类型。例如：

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

如果愿意，可以构造一个大小正确的数组，在这种情况下，会直接向构造的这个数组中填充元素，而不会创建新数组。例如：

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

7 遗留的集合



7.1 Hashtable类

`Hashtable`类与`HashMap`类的作用一样，接口也基本相同。`Hashtable`类的方法是同步的。如果对于遗留代码的兼容性没有任何要求，就应该使用`HashMap`。

7.2 枚举

遗留的集合使用`Enumeration`接口遍历元素序列。`Enumeration`接口有两个方法`hasMoreElements`和`nextElement`，这两个方法完全类似于`Iterator`接口的`hasNext`方法和`next`方法。

如果发现遗留的类实现了这个接口，可以使用`Collections.list`将元素收集到一个`ArrayList`中：

```
/* java.util.Collections */
static <T> ArrayList<T> list(Enumeration<T> e)
    // 返回一个数组列表，其中包含指定枚举按顺序返回的元素
```

或者，在Java 9中，可以把一个枚举转换为一个迭代器：

```
/* java.util.Enumeration */
default Iterator<E> asIterator()
    // 返回一个迭代器，这个迭代器用于遍历此枚举所覆盖的其余元素
```

有时还会遇到遗留的方法希望得到枚举参数。静态方法`Collections.enumeration`将产生一个枚举对象，枚举集合中的元素：

```
/* java.util.Collections */
static <T> Enumeration<T> enumeration(Collection<T> c)
    // 返回一个枚举，可以枚举 c 的元素
```

7.3 属性映射

属性映射（property map）是一个特殊类型的映射结构，它有下面3个特性：

1. 键与值都是字符串。
2. 这个映射可以很容易地保存到文件以及从文件加载。
3. 有一个二级表存放默认值。

实现属性映射的Java平台类名为`Properties`，下面列举这个类的一些重要方法：

```
/* java.util.Properties */
Properties()
    // 创建一个空属性映射
Properties(Properties defaults)
    // 用一组默认值创建一个空属性映射
String getProperty(String key)
    // 返回与键关联的值。如果这个键未在表中出现，则返回默认值表中与这个键关联的值。如果键在默认值表中也未出现，则返回 null
String getProperty(String key, String defaultValue)
    // 返回与键关联的值。如果键在表中未出现，则返回默认字符串
Object setProperty(String key, String value)
    // 设置一个属性，返回给定键之前设置的值
void load(InputStream in) throws IOException
    // 从一个输入流加载一个属性映射
void store(OutputStream out, String header)
    // 将一个属性映射保存到一个输出流，header 是所存储文件的第一行
```

属性映射对于指定程序的配置选项很有用，例如：

```
Properties settings = new Properties();
settings.setProperty("width", "600.0");
settings.setProperty("filename", "/home/cay/books/cj11/code/v1ch11/raven.html");
```

可以使用`store`方法将属性映射列表保存到一个文件中，例如：

```
FileOutputStream out = new FileOutputStream("program.properties");
settings.store(out, "Program Properties");
```

要从文件加载属性，可以使用如下调用：

```
FileOutputStream in = new FileOutputStream("program.properties");
settings.load(in);
```

`System`类提供了获取系统属性的方法：

```
/* java.lang.System */
static Properties getProperties()
    // 获取所有系统属性。应用必须有权限获取所有属性，否则会抛出一个安全异常
static String getProperty(String key)
    // 获取给定键名对应的系统属性。应用必须有权限获取这个属性，否则会抛出一个安全异常
    // 以下属性总是允许获取：
    /*
        java.version
        java.vendor
        java.vendor.url
        java.home
        java.class.path
        java.library.path
        java.class.version
        os.name
        os.version
        os.arch
        file.separator
        path.separator
        line.separator
        java.io.tempdir
        user.name
        user.home
        user.dir
        java.compiler
        java.specification.version
        java.specification.vendor
        java.specification.name
        java.vm.version
        java.vm.vendor
        java.vm.name
    */
```

`Properties`类有两种提供默认值的机制。第一种方法是，通过`getProperty`方法查找一个字符串的值时，可以指定一个默认值，当键不存在时就会自动使用这个默认值。第二种方法是，把所有默认值都放在一个二级属性映射中，并在主属性映射的构造器中使用这个二级映射。例如：

```
Properties defaultSettings= new Properties();
defaultSettings.setProperty("width", "600");
...
Properties settings = new Properties(defaultSettings);
```

7.4 栈

```
/* java.util.Stack<E> */
E push(E item) // 将 item 压入栈并返回 item
E pop() // 弹出并返回栈顶元素。如果栈为空，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，不要调用这个方法
```

`Stack`类扩展了`Vector`类，但是`Vector`类并不令人满意，甚至可以使用并非栈操作的`insert`和`remove`方法在任何地方插入和删除值。

7.5 位集

Java平台的`BitSet`类用于存储一个位序列。由于位集将位包装在字节里，所以使用位集要比使用`ArrayList<Boolean>`高效得多。

`BitSet`类提供了一个便于读取、设置或重置各个位的接口，使用这个接口可以避免掩码和其他调整位的操作。下面列举`BitSet`类的重要方法：

```
/* java.util.BitSet */
BitSet(int initialCapacity)
    // 创建一个位集
int length()
    // 返回位集的逻辑长度，即 1 加上位集的最高位的索引
boolean get(int bit)
    // 获得一个位
void set(int bit)
    // 将指定位设置为 true
void clear(int bit)
    // 将指定位设置为 false
void and(BitSet set)
    // 这个位集与另一个位集进行逻辑与
void or(BitSet set)
    // 这个位集与另一个位集进行逻辑或
void xor(BitSet set)
    // 这个位集与另一个位集进行逻辑异或
void andNot(BitSet set)
    // 对应另一个位集中设置为 1 的所有位，将这个位集中相应的位清除为 0
```

下面的例子是"埃拉托色尼筛选法"的实现，这个算法用来查找素数。首先将位集的所有位置为`true`，然后将已知素数的倍数所对应的位都置为`false`，经过这个操作仍为`true`的位对应的就是素数。

```
import java.util.*;

public class Sieve
{
    public static void main(String[] args)
    {
        int n = 2000000;
        long start = System.currentTimeMillis();
        BitSet bitSet = new BitSet(n + 1);
        int count = 0;
        int i;
        for (i = 2; i <= n; i++)
        {
            bitSet.set(i);
        }
        i = 2;
```

```
        while (i * i <= n)
        {
            if (bitSet.get(i))
            {
                count++;
                int k = 2 * i;
                while (k <= n)
                {
                    bitSet.clear(k);
                    k += i;
                }
            }
            i++;
        }
        while (i <= n)
        {
            if (bitSet.get(i)) count++;
            i++;
        }
        long end = System.currentTimeMillis();
        System.out.println(count + " primes");
        System.out.println((end - start) + " milliseconds");
    }
}
```