第 18 章　集合元素大操练 ......2
18.1　Java 集合的类框架 ......2
18.2　集合的基本用法 ......3
18.2.1　包装类的自动装箱和拆箱 ......4
18.2.2　Set（集）和 List（列表）的各种具体实现类的特点 ......4
18.2.3　集合的静态 of()方法 ......6
18.3　List（队列） ......7
18.4　Map（映射） ......7
18.5　用 Lambda 表达式遍历集合 ......9
18.6　小结 ......10
18.7　编程实战：计算数学表达式 ......10
18.8　编程实战：计算带括号的数学表达式 ......14
18.9　编程实战：用集合工具对数字排序 ......16
18.10　编程实战：按月份先后循序数兔子 ......17
18.11　编程实战：用映射来存放学生信息 ......19
18.12　编程实战：圆桌报数游戏 ......20

# 第 18 章　集合元素大操练

数组的一个特点是一旦创建，它的长度就固定了，无法改变它的长度。假如要扩充数组的容量，唯一的办法就是重新创建一个新的容量更大的数组，把原来数组的内容拷贝到新的数组中。

假如有一批数据，需要经常对这批数据进行添加或删除的操作，也就是说，这批数据的数目是不固定的。把这样的数据存放在长度固定的数组中，操作起来很不方便。在这种情况下，可以使用 Java 集合。

与数组不同的是，Java 集合是通过类来实现的。Java 集合不仅可以方便地存放数据，而且提供了对数据进行添加、读取和删除等方法。

与数组的另一个不同之处是，Java 集合中不能存放基本类型数据，而只能存放引用类型数据。如图 18-1 所示，Java 集合主要分为三种类型：

- Set（集）：集合中的对象不按特定方式排序，并且没有重复对象。它的有些实现类（TreeSet）能对集合中对象按特定方式排序。
- List（列表）：集合中的对象按照索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象。List 与数组有些相似。
- Queue（队列）：集合中的对象按照先进先出的规则来排列。在队列的末尾添加元素，在队列的头部删除元素。可以有重复对象。双向队列则允许在队列的末尾和头部添加和删除元素。

除了以上三种集合，还有一种 Map（映射）。它和集合有点类似，其特殊之处在于集合中的每一个元素包含一对键（Key）对象和值（Value）对象，集合中没有重复的键对象，值对象可以重复。它的有些实现类（TreeMap）能对集合中的键对象进行排序。

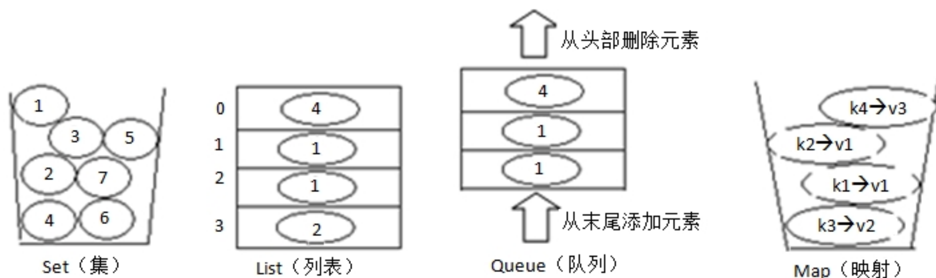


图 18-1　各种 Java 集合

本章内容主要围绕以下问题展开：

- Set 有什么特点？如何对 Set 集进行存取操作？
- List 有什么特点？如何对 List 列表进行读取、在任意位置添加和删除元素的操作？
- Queue 有什么特点？如何对 Queue 队列进行先进先出的存取操作？
- Map 有什么特点？如何对 Map 映射的键对象和值对象进行存取操作？

## 18.1　Java 集合的类框架

Java 集合主要位于 JDK 类库的 java.util 包中，图 18-2 显示了常用的集合接口和类。

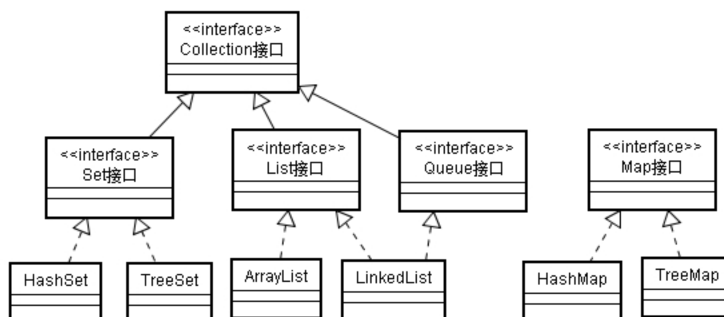


图 18-2　常用的集合接口和类

以上图 18-2 展示了由集合接口和类组成的大家庭，它们一起构成了 Java 集合的基本框架。Set 接口、List 接口和 Queue 接口继承自 Collection 接口。Set 接口有两个实现类：HashSet 和 TreeSet。List 接口有两个实现类：ArrayList 和 LinkedList。Queue 接口有一个实现类：LinkedList。Map 接口有两个实现类：HashMap 和 TreeMap。这些不同的实现类到底如何使用，并且有什么区别呢？在下一节，将通过具体的范例来展示它们的用法。

作为 Java 开发人员，不仅要熟悉 Java 编程的语法，而且要善于运用 JDK 类库以及第三方提供的类库中的现成的接口与类。如何了解这些现成的接口与类的用法呢，可以查阅它们的 JavaDoc 文档。JDK 的类库的 JavaDoc 文档的网址为：http://docs.oracle.com/javase/9/docs/api/ 。此外，在 javathinker.net 网站的主页上，也会提供最新版本的 JDK 类库的 JavaDoc 文档的链接。

## 18.2　集合的基本用法

以下代码创建了一个存放 String 对象的集合，向这个集合中加入了三个元素，然后遍历访问这个集合：

```
Set<String> colors=new HashSet<String>();
colors.add("红色");
colors.add("黄色");
colors.add("白色");

//遍历访问集合中的元素，打印集合中的所有颜色
for(String s:colors)
  System.out.println(s);
```

以上 for(String　s:colors)语句的作用是依次从 colors 集合中取出每个 String 类型的元素，把它赋值給 s 变量，再打印这个 s 变量。

在声明集合类型时，允许把变量声明为接口类型，而实际上引用一个具体实现类的实例，例如：

```
Set<String> colors=new HashSet<String>();
或者:
Collection<String> colors=new ArrayList<String>();
```

以上“<String>”标记指明集合中存放的元素类型为 String 类型。

### 18.2.1 包装类的自动装箱和拆箱

以下代码创建了一个存放 Integer 对象的集合，向这个集合中加入了三个元素，然后遍历访问这个集合：

```
List<Integer> scores=new ArrayList<Integer>();
//向集合中加入元素，int 类型的元素会自动转变为 Integer 对象，自动装箱
scores.add(78);
scores.add(89);
scores.add(93);

//获得列表中索引为 0 的元素，score=78
int score=scores.get(0); //Integer 对象自动转换为 int 类型，自动拆箱
```

集合中只能存放对象，可是以上代码会直接向 scores 集合中加入 int 基本类型的数据，这是怎么回事呢？原来，为了方便开发人员编程，JDK 会自动进行基本类型与相应的包装类型之间的转换。

对于以上代码，scores.add(78)方法会自动把 int 类型的直接数 78 转换为表示 78 的 Integer 对象，再把它加入到集合中。scores.get(0)方法返回的是列表中索引为 0 的 Integer 对象，JDK 会自动把 Integer 对象转换为 int 基本类型数据，再把它赋值給变量 score。

把 int 基本类型自动转换为 Integer 对象的过程叫做装箱，把 Integer 对象自动转换为 int 基本类型的过程叫做拆箱。除了 Integer 类型，其他的 Long、Double 和 Float 等类型也支持自动装箱和拆箱。

### 18.2.2 Set（集）和 List（列表）的各种具体实现类的特点

以下例程 18-1 的 Comp 类演示了集合的基本用法，并且比较了 HashSet、TreeSet 和 ArrayList 这些集合具体实现类的特点。

**例程 18-1** Comp.java

```
import java.util.*;
public class Comp{
 private int[] data={33,11,55,33};

 /* 对各种类型的集合进行添加、删除和遍历操作 */
 public void test(Collection<Integer> col){
  for(int i=0;i<data.length;i++) //把数组中的元素依次加入到集合中
   col.add(data[i]);

  System.out.println("集合中的初始元素如下：");
  print(col); //打印集合中的内容

  col.remove(11);//删除集合中的元素 11

  System.out.println("删除 11 后集合中的元素如下：");
  print(col); //打印集合中的内容

  col.add(44); //向集合中加入元素 44

  System.out.println("集合中加入 44 后的元素如下：");
  print(col); //打印集合中的内容
 }
```

```
    public static void print(Collection<Integer> col){
      for(Integer d : col)  //遍历访问集合
        System.out.print(d+" ");
      System.out.println(); //打印换行符
    }

    public static void main(String args[]){
      Comp comp=new Comp();
      System.out.println("---测试HashSet---");
      comp.test(new HashSet<Integer>());

      System.out.println("---测试TreeSet---");
      comp.test(new TreeSet<Integer>());

      System.out.println("---测试ArrayList---");
      comp.test(new ArrayList<Integer>());
    }
  }
```

Comp 类的 print()方法中的“for(Integer d : col){…}”语句依次从 col 集合中读取一个元素，把它赋值給变量 d，然后在循环体中打印变量 d。

在 Comp 类的 main()方法中，调用 test()方法，依次对 HashSet、TreeSet 和 ArrayList 进行测试，分别对这些集合进行添加元素和删除元素的操作，然后再遍历访问集合，观察集合中元素的变化。

运行“java Comp”命令，得到以下打印结果：

```
---测试HashSet---
集合中的初始元素如下：
33 55 11
删除11后集合中的元素如下：
33 55
集合中加入44后的元素如下：
33 55 44

---测试TreeSet---
集合中的初始元素如下：
11 33 55
删除11后集合中的元素如下：
33 55
集合中加入44后的元素如下：
33 44 55

---测试ArrayList---
集合中的初始元素如下：
33 11 55 33
删除11后集合中的元素如下：
33 55 33
集合中加入44后的元素如下：
33 55 33 44
```

从以上打印结果可以发现被测试集合的以下特点：

- HashSet 和 TreeSet 集合中都不允许存放重复的元素，而 ArrayList 列表中可以存放重复的元素。

- 存放到 TreeSet 集合中的元素会按照数字由小到大自动排序，而 HashSet 和 ArrayList 中的元素不会按照数字由小到大自动排序。
- 存放在 HashSet 集合中的元素的存放没有固定的循序，遍历 HashSet 集合时读取元素的循序和向集合中添加元素的循序不一致。
- 向 ArrayList 列表中添加元素时，这些元素按照索引排序，第一个元素的索引为 0。遍历访问 ArrayList 时，按照索引从小到大依次读取列表中的元素。所以列表的特性和数组有相似之处。

对于列表，可以通过 get(int index)方法来获取索引 index 所对应的元素，列表中第一个元素的索引为 0。例如：

```
List<Integer> list=new ArrayList<Integer>();
list.add(3);  //自动把 3 转换为相应的 Integer 对象，再把它加入到 List 中
list.add(4);
list.add(5);
list.add(2);

int a=list.get(1); //获取索引为 1 的元素。a=4
int b=list.get(3); //获取索引为 3 的元素。b=2
int c=list.get(0); //获取索引为 0 的元素。c=3
```

### 18.2.3 集合的静态 of()方法

当程序向一个集合中添加元素时，如果集合中包含很多元素，需要多次调用集合的 add()方法，例如：

```
Set<String> colors=new HashSet<String>();
colors.add("红色");
colors.add("橙色");
……
colors.add("紫色");
```

为了简化编程，从 JDK9 开始，Set 和 List 接口都增加了静态 of()方法，它能够创建并返回一个不可改变的 Set 或 List 对象，并且添加到集合中的元素可以直接在 of()方法的参数中指定。of()方法用多种重载形式，例如 Set 接口包含以下多个重载的 of()方法：

```
of(E... elements)
of(E e1, E e2)
of(E e1, E e2, E e3)
of(E e1, E e2, E e3, E e4)
of(E e1, E e2, E e3, E e4, E e5)
of(E e1, E e2, E e3, E e4, E e5, E e6)
of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
```

of()方法的参数用来设定集合中存放的元素。以下 OfTester 类演示了 of()方法的用法：

```
import java.util.*;
public class OfTester{
  public static void main(String[] args){
    Set<String> colors=Set.of("红色","橙色","黄色",
                 "绿色","蓝色","靛色","紫色");
```

```
        for(String s:colors)
          System.out.println(s); //打印集合中的颜色

        colors.add("白色"); //运行时抛出异常
      }
    }
```

OfTester 类的 main()方法首先调用 Set.of("红色"……)静态方法，获得一个 Set 对象，它包含了参数中指定的 7 个 String 类型的元素。由此可见，调用 Set.of()方法可以简化向集合添加元素的编码。

值得注意的是，使用集合的 of()方法必须遵守一个重要约束，即返回的集合对象是不可改变的，这意味着不允许对这样的集合对象进行添加或删除元素的操作。

运行 OfTester 类，当 Java 虚拟机执行 main()方法中的“colors.add("白色");”语句时，会抛出 UnsupportedOperationException 异常，原因就在于不允许对不可改变的 colors 集合进行添加元素的操作。

# 18.3 List（队列）

多数人都有过在火车站售票大厅排队等待购票的经历。后加入的人排在队列的末尾，排在队列头部的人优先购票后离开队列。从 JDK5 开始，用 java.util.Queue 接口来表示队列。队列的特点是向末尾添加元素，从队列头部删除元素，队列中允许有重复元素。

LinkedList 类不仅实现了 List 接口，还实现了 Queue 接口，因此，可以对 LinkedList 进行先进先出的队列操作。

假定“Tom”、“Mike”、“Linda”和“Mary”依次排队去买火车票。以下程序代码就演示了这个四个人进入队列以及离开队列的过程：

```
Queue<String> queue=new LinkedList<String>();

//从尾部进入队列
queue.add("Tom");
queue.add("Mike");
queue.add("Linda");
queue.add("Mary");

//从头部离开队列
System.out.println(queue.remove()); //打印Tom
System.out.println(queue.remove()); //打印Mike
System.out.println(queue.remove()); //打印Linda
System.out.println(queue.remove()); //打印Mary
```

# 18.4 Map（映射）

Map（映射）是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键对象和值对象。向 Map 集合中加入元素时，必须提供一对键对象和值对象，从 Map 集合中检索元素时，只要给出键对象，就会返回对应的值对象。以下程序通过 Map 的 put(Object key,Object value)方法向集合中加入元素，通过 Map 的 get(Object key)方法来检索与键对象对应的值对象：

```
Map<String,String> map=new HashMap<String,String>();
map.put("1","Monday");
map.put("2","Tuesday");
map.put("3","Wendsday");
map.put("4","Thursday");

String day=map.get("2");  //day的值为“Tuesday”
```

以上“Map<String,String>”中的“<String,String>”用来指定键对象和值对象的类型。Map 集合中的键对象不允许重复，对于值对象则没有唯一性的要求，可以将任意多个键对象映射到同一个值对象上。例如以下 Map 集合中的键对象“1”和“one”都和同一个值对象“Monday”对应：

```
Map<String,String> map=new HashMap<String,String>();
map.put("1","Mon.");
map.put("1","Monday");
map.put("one","Monday");

Set<Map.Entry<String,String>> set=map.entrySet();
for(Map.Entry entry : set)  //entry表示Map中的一对键与值
  System.out.println(entry.getKey()+":"+entry.getValue());
```

由于第一次和第二次加入 Map 中的键对象都为“1”，因此第一次加入的值对象将被覆盖，Map 集合中最后只有两个元素，分别为：

```
“1”对应“Monday”
“one”对应“Monday”
```

Map 的 entrySet()方法返回一个 Set 集合，在这个集合中存放了 Map.Entry 类型的元素，每个 Map.Entry 对象代表 Map 中的一对键与值。Map.Entry 对象的 getKey()方法返回键，getValue()方法返回值。

从 JDK9 开始，Map 接口也和 Set 以及 List 接口一样，增加了静态的 of()方法。Map 接口的 of()方法用来创建并返回一个不可改变的 Map 对象，并且添加到 Map 中的元素由 of()方法的参数指定，例如：

```
Map<String,String> map=
    Map.of("1","Monday","2","Tuesday","3","Wendsday","4","Thursday");
```

Map 有两种比较常用的实现：HashMap 和 TreeMap。HashMap 按照哈希算法来存取键对象，有很好的存取性能。TreeMap 能对键对象进行排序。以下程序中的 TreeMap 会对四个 String 类型的键对象“1”、“3”、“4”和“2”自动进行排序：

```
Map<String,String> map=new TreeMap<String,String>();
map.put("1","Monday");
map.put("3","Wednesday");
map.put("4","Thursday");
map.put("2","Tuesday");

Set<String> keys=map.keySet();
for(String key:keys){
  String value=map.get(key);
  System.out.println(key+" "+value);
}
```

TreeMap 以及本章前文提到的 TreeSet 都有排序功能，默认情况下，它们会进行自然排

序。例如对于数字，就从小到大排序；对于字符串，就按照字典循序来排序。Map 的 keySet()方法返回集合中所有键对象的集合，以上程序的打印结果为：

```
1 Monday
2 Tuesday
3 Wednesday
4 Thursday
```

## 18.5 用 Lambda 表达式遍历集合

本书第 15 章的 15.5 节（用 Lambda 表达式代替内部类）已经介绍了 Lambda 表达式的一些用法，本节将介绍如何用 Lambda 表达式来遍历集合。从 JDK5 开始，Java 集合都实现了 java.util.Iterable 接口，它的 forEach()方法能够遍历集合中的每个元素。forEach()方法的完整定义如下：

```
default void forEach(Consumer<? super T> action)
```

forEach()方法有一个 Consumer 接口类型的 action 参数，这个参数用来指定对集合中每个元素的具体操作行为。action 参数所引用的 Consumer 实例必须实现 Consumer 接口的 accept(T t)方法，在该方法中指定对参数 t 所执行的具体操作。

例如以下 forEach()方法中的 Lambda 表达式相当于是 Consumer 类型的匿名对象，它指定对每个元素的操作为打印这个元素：

```
Set<String> names=Set.of("Tom","Mike","Mary");
//打印 names 集合中的每个元素
names.forEach((name) -> System.out.println(name + ","));
```

以下 forEach()方法中的 Lambda 表达式指定了更加复杂的遍历 datas 列表中元素的行为，会比较每个元素与 bottom 变量的大小，并且打印比较结果：

```
List<Integer> datas=List.of(67,50,88,34,79);
final int bottom=50;

//打印 datas 集合中的每个元素与 bottom 变量比较的结果
datas.forEach((data) ->{
  if(data>bottom)
    System.out.println(data+">"+bottom);
  else if(data==bottom)
    System.out.println(data+"=="+bottom);
  else
    System.out.println(data+"<"+bottom);
});
```

以上 Lambda 表达式中“->”符号后面的可执行语句有好几行，因此放在大括号内“{}”。运行以程序代码，会得到以下打印结果：

```
67>50
50==50
88>50
34<50
79>50
```

## 18.6　小结

本章涉及的 Java 知识点总结如下：

- 集合和数组的区别

集合的长度可变，而数组的长度固定。集合中只能存放引用类型的数据，而数组可以存放基本类型和引用类型的数据。集合除了可以存放数据，还提供了各种灵活地操纵数据的方法。而数组的操作方式比较单一。

- Set（集）

集是最简单的一种集合，集合中的对象不按特定方式排序，并且没有重复对象。Set 接口主要有两个实现类 HashSet 和 TreeSet。HashSet 类按照哈希算法来存取集合中的对象，存取速度比较快。TreeSet 类具有排序功能。

- List（列表）

列表的主要特征是其元素以线性方式存储，列表中允许存放重复对象。List 接口主要的实现类包括：ArrayList 和 LinkedList。ArrayList 代表长度可变的数组，允许对元素进行快速地随机访问，但是向 ArrayList 中插入与删除元素的速度较慢。LinkedList 在实现中采用链表数据结构。对顺序访问进行了优化，向 List 中插入和删除元素的速度较快，随机访问则相对较慢。随机访问是指检索位于特定索引位置的元素。

- Queue（队列）

队列的特点是向末尾添加元素，从队列头部删除元素，队列中允许有重复元素。LinkedList 实现了 Queue 接口。

- Map（映射）

映射是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键对象和值对象。Map 有两种比较常用的实现：HashMap 和 TreeMap。HashMap 按照哈希算法来存取键对象，有很好的存取性能。TreeMap 能对键对象进行排序。

## 18.7　编程实战：计算数学表达式

假定用户在命令行输入的第一个参数是一个数学表达式（例如“8+2*3+8-3”），这个表达式中的运算符包括：“+”、“-”、“*”或者“/”，表达式中的数字可以是浮点数或整数。请编写一个程序来计算这个数学表达式。

**编程提示**

以下例程 18-2 的 Calculater 类实现了计算数学表达式的功能。它通过 parse()方法解析字符串形式的数学表达式，把其中的数字和运算符都存放到一个列表中，这样可以更方便地存取表达式中的内容。接下来再通过 calculater()方法计算存放在列表中的数学表达式。

例程 18-2　Calculater.java

```
import java.util.*;
import java.text.DecimalFormat;
public class Calculater{
  //存放表达式支持的所有符号
  protected final Set<String> symbols;
```

```
public Calculater(){
  symbols=new HashSet<String>();
  symbols.add("+");
  symbols.add("-");
  symbols.add("*");
  symbols.add("/");
}

/* 判断表达式中的一个字符是否属于专有符号 */
private boolean isSymbol(String s){
  return symbols.contains(s);  //判断 symbols 集合中是否包含 s
}

/* 计算只包含一个操作符的简单表达式 */
public double calculateSimple(List<String> expr){
  Double data1=Double.parseDouble(expr.get(0));
  Double data2=Double.parseDouble(expr.get(2));
  switch(expr.get(1)){
    case "+": return data1+data2;
    case "-": return data1-data2;
    case "*": return data1*data2;
    case "/": return data1/data2;
    default: return 0;
  }
}

/* 获得子列表*/
public List<String> subExpr(List<String> expr, int startIndex){
  return expr.subList(startIndex,expr.size());
}

/* 在列表开头增加一位数据 */
public List<String> expandExpr(double data,List<String> expr){
  expr.add(0,Double.valueOf(data).toString());
  return expr;
}

/* 把表达式中的“+”改为“-”，把“-”改为“+” */
public  List<String> convert(List<String> expr){
  for(int i=0;i<expr.size();i++){
    if(expr.get(i).equals("+"))
      expr.set(i,"-");
    else if(expr.get(i).equals("-"))
      expr.set(i,"+");
  }

  return expr;
}

/* 计算列表中表达式的值 */
public double calculate(List<String> expr){
  if(expr.size()<3) return Double.parseDouble(expr.get(0));
  if(expr.size()==3)return calculateSimple(expr);
```

```
  //以下代码处理表达式中有两个以上操作符的情况

  Double data1=Double.parseDouble(expr.get(0));
  Double data2=Double.parseDouble(expr.get(2));
  switch(expr.get(1)){
    case "+":  //例如表达式为8+2*3+8-3,那么计算8+(2*3+8-3)
      return data1+calculate(subExpr(expr,2));
    case "-": //例如表达式为8-2*3+8-3,那么计算8-(2*3-8+3),
             //要把子表达式中的“+”和“-”颠倒
      return data1-calculate(convert(subExpr(expr,2)));
    case "*": //例如表达式为2*3+8-3,那么计算6+8-3
       return calculate(expandExpr(data1*data2,subExpr(expr,3)));
    case "/": //例如表达式为6/3+8-3,那么计算2+8-3
       return calculate(expandExpr(data1/data2,subExpr(expr,3)));
    default: return 0;
  }
}

/* 把字符串表达式中的数字和运算符存放到字符串列表中   */
public List<String> parse(String exprStr){
  List<String> expr=new LinkedList<String>();

  //lastIndex变量表示上一个运算符在字符串表达式中的索引
  int lastIndex=-1;

  //nextIndex变量表示下一个运算符在字符串表达式中的索引
  int nextIndex=-1;

  for(int i=0;i<exprStr.length();i++){
    //如果遇到符号,而不是数字
    if(isSymbol(exprStr.substring(i,i+1))){
       nextIndex=i;
       //加入上一个操作元
       expr.add(exprStr.substring(lastIndex+1,nextIndex));
       //加入当前操作符
       expr.add(exprStr.substring(nextIndex,nextIndex+1));

       lastIndex=nextIndex;
    }
  }
  //加入最后一个操作元
  expr.add(exprStr.substring(lastIndex+1,exprStr.length()));

  /* 以下代码展示了用Iterator枚举对象来遍历集合中元素的方式
     用于校正表达式，去除其中多余的空格和空字符 */
  Iterator<String> it = expr.iterator();
  while(it.hasNext()){
    String s = it.next();  //得到集合中的下一个元素
    //在循环中删除表达式中的所有""空字符或者" "空格
    if(s.equals("") || s.equals(" "))
      it.remove();  //删除元素
   }
```

```
    //返回包含表达式的列表
    return expr;
  }

  /*  程序入口main()方法 */
  public static void main(String args[]){
    Calculater calculater=new Calculater();
    double result=calculater.calculate(
                   calculater.parse(args[0]));
    //指定数据显示格式，保留两位小数
    DecimalFormat decimalFormat = new DecimalFormat("#.00");
    System.out.println(decimalFormat.format(result));
  }
}
```

以上 Calculater 类的 parse(String exprStr)方法的参数 exprStr 是待处理的字符串表达式，该方法会把这个字符串形式表达式中的数字和运算符取出来，把它们存放到 expr 列表中，并将它返回。

假定 exprStr 参数的值为“2.5+10*2/2-1.5”，那么 expr 列表的内容为{“2.5”,“+”,“10”,“*”,“2”,“/”,“2”,“-”,“1.5”}。

以上 exprStr 参数为 String 类型，它的 length()方法返回字符串的长度，例如字符串“2.5+10*2/2-1.5”的长度为 14。String 类的 substring(int beginIndex,int endIndex)方法从源字符串中获取子字符串。起始索引是 beginIndex，结束索引是 endIndex-1。字符串中第一个字符的索引是 0。因此 expr.substring(0,1)的返回值是“2”,expr.substring(4,8)的返回值是“10*2”。

Calculater 类的 calculate(List<String> expr)运用了递归算法。下面结合具体的表达式来分析 calculate(List<String> expr)的运算流程。主要分为三种情况：

- expr 列表中只有一个元素，例如 expr 为{“5”}，那么直接返回“5”。
- expr 列表中有三个元素，例如 expr 为{“8”,“+”,“11”}，包含三个元素，那么调用 calculateSimple({“8”,“+”,“11”})方法，得到 19。
- expr 列表中有三个以上元素，这时又分四种情况：
  - expr 列表中第一个运算符是加号“+”，例如 expr 为{“8”,“+”,“2”,“*”,“3”,“+”,“8”,“-”,“3”}，那么用递归算法，继续调用 calculate({“2”,“*”,“3”,“+”,“8”,“-”,“3”})来计算子表达式，得到 11，然后计算 8+11，得到 19。
  - expr 列表中第一个运算符是减号“-”，例如 expr 为{“8”,“-”,“2”,“*”,“3”,“+”,“8”,“-”,“3”}，那么用递归算法，继续调用 calculate({“2”,“*”,“3”,“-”,“8”,“+”,“3”})来计算子表达式，得到 1，然后计算 8-1，得到 7。由于 expr 数组中的第二个元素为“-”，此时递归运算的子表达式是“2*3-8+3”，其中的“+”和“-”跟原先的相应子表达式做了符号颠倒。
  - expr 列表中第一个运算符是乘号“*”，例如 expr 为{“2”,“*”,“3”,“+”,“8”,“-”,“3”}，那么先计算 2*3=6，再用递归算法，继续调用 calculate({“6”,“+”,“8”,“-”,“3”})，得到 11。
  - expr 列表中第一个运算符号是除号“/”，例如 expr 为{“6”,“/”,“3”,“+”,“8”,“-”,“3”}，那么先计算 6/3=2，再用递归算法，继续调用 calculate({“2”,“+”,“8”,“-”,“3”})，得到 7。

Calculater 类的 subExpr(List<String>expr, int startIndex)方法用于从参数 expr 列表中截取一个子列表。它调用了 List 接口的 subList(int fromIndex,int endIndex)方法，该方法返回一个子列表，它包含了原列表中索引从 fromIndex 到 endIndex-1 的元素。例如，假定列表 expr 中的内容为{“8”，“+”，“2”，“*”，“3”，“+”，“8”，“-”，“3”}，那么 expr.subList(2,9)返回的子列表的内容是{“2”，“*”，“3”，“+”，“8”，“-”，“3”}。

Calculater 类的 convert(List<String> expr)方法用于把 expr 列表中的“+”和“-”分别替换为“-”和“+”。它调用了 List 接口的 set(int index, E element)方法，该方法会重新设置 index 索引对应的元素。例如，假定列表 expr 中的内容为{“8”，“-”，“3”}，那么 expr.set(1, "+")会使得列表 expr 的内容变为{“8”，“+”，“3”}。

Calculater 类的 expandExpr(int data,List<String> expr) 方法用于在 expr 列表开头插入一位数据。它调用了 List 接口的 add(int index, E element)方法，该方法会在 index 索引位置插入一个元素，列表中原先从 index 索引开始的元素全部向后移动。例如，假定列表 expr 中的内容为{“+”，“8”，“-”，“3”}，那么 expr.add(0, "2")会使得列表的内容变为{“2”，“+”，“8”，“-”，“3”}。

运行“java Calculater 8+2*3+8-3”，将打印表达式的运算结果为“19.00”。运行“java Calculater 2.5+10*2/2-1.5”，将打印表达式的运算结果为“11.00”。在 Calculater 类的 main()方法中，利用 java.text.DecimalFormat 类来指定浮点数的显示格式为“#.00”，即保留两位小数。

## 18.8 编程实战：计算带括号的数学表达式

扩展本章 18.7 节的 Calculater 类的功能，使表达式可以包含括号。扩展后的计算器能计算类似“(1.1+3.3)*20-(4.8/0.2+5)”这样的表达式。

**编程提示**

表达式中括号的优先级别最高，因此要先计算出一对括号内的子表达式的值。例如对于表达式“(1*(2+3)-(5-4))/2 ”，它的运算步骤如下：

（1）计算(2+3)得 5，再计算(1*5-(5-4))/2

（2）计算(5-4) 得 1，再计算(1*5-1)/2

（3）计算(1*5-1)得 4，再计算 4/2 得 2。

以下例程 18-3 的 Calculater1 类就实现了能对带括号的表达式运算的计算器。

**例程 18-3 Calculater1.java**

```
import java.util.*;
import java.text.DecimalFormat;
public class Calculater1 extends Calculater{  //继承Calculater类

  public Calculater1(){
    //在表达式所支持的符号中增加"("和")"
    symbols.add("(");
    symbols.add(")");
  }

  public double calculate(List<String> expr){
    boolean hasBracket;   //表达式中是否有括号
    int leftIndex=0,rightIndex=0; //这两个变量表示一对左括号和右括号的索引
```

```
    do{
      hasBracket=false;

      //寻找表达式中第一对要处理的括号。
      //例如，对于(1*(2+3)-(5-4))/2，第一对要处理的括号是(2+3)
      for(int i=0;i<expr.size();i++){
        String s=expr.get(i);
        if(s.equals("(")){
          leftIndex=i;
          hasBracket=true;
        }else if(s.equals(")")){
          rightIndex=i;
          break;
        }
      }

      if(hasBracket){
        //获得括号内的子表达式
        List<String> subExpr=expr.subList(leftIndex+1,rightIndex);
        //计算括号内子表达式的值
        double result=super.calculate(subExpr);
        //删除原表达式中的子表达式
        expr.subList(leftIndex,rightIndex+1).clear();
        //把子表达式的值插入到原表达式中
        expr.add(leftIndex,Double.valueOf(result).toString());
      }

    }while(hasBracket);

    return super.calculate(expr);
  }

  /* 程序入口main()方法 */
  public static void main(String args[]){
    Calculater1 calculater=new Calculater1();
    double result=calculater.calculate(
                  calculater.parse(args[0]));
    //指定数据显示格式，保留两位小数
    DecimalFormat decimalFormat = new DecimalFormat("#.00");
    System.out.println(decimalFormat.format(result));
  }
}
```

Calculater1 类继承了 Calculater 父类。在 Calculater 父类中定义了一个 symbols 集合属性，用来存放表达式支持的所有符号。Calculater1 类的构造方法向 symbols 集合属性中增加了“（”和“）”元素，确保 Calculater 父类的 parse()方法能够正确地解析包含括号的字符串表达式。例如对于字符串表达式“(1.1+3.3)*20”，解析后得到的列表中的元素为：{“(”，“1.1”，“+”，“3.3”，“)”，“*”，“20”}。

Calculater1 类覆盖了 Calculater 父类的 calculate()方法。Calculater1 类的 calculate(List<String> expr)方法的运算流程如下：

（1）在一个 do-while 循环中寻找第一对要处理的括号，例如对于表达式“(1*(2+3)-

(5-4))/2”，第一对要处理的括号是“(2+3)”。

（2）接下来调用 Calculater 父类的 calculate(String[]  expr)方法，计算“2+3”得 5。然后把原来表达式中的“(2+3)”替换为“5”。

（3）继续在 do-while 的下一个循环中处理表达式“(1*5-(5-4))/2”。如此循环下去，直到表达式中没有括号，变成“4/2”。

（4）退出 do-while 循环，调用 Calculater 父类的 calculate(String[]    expr)方法，计算“4/2”得 2。

对于以上第二个步骤，把原来表达式中的“(2+3)”替换为“5”，这对应的程序代码为：

```
//删除原表达式中的子表达式
expr.subList(leftIndex,rightIndex+1).clear();
//把子表达式的值插入到原表达式中
expr.add(leftIndex,Double.valueOf(result).toString());
```

List 类的 subList()方法返回当前列表的子列表。例如对于 expr.subList(leftIndex,rightIndex+1)，会返回一个子列表，里面包含了 expr 列表中索引从 leftIndex 到 rightIndex 的元素。对于表达式“(1*(2+3)-(5-4))/2”，expr.subList(3,8)返回的子列表为“(2+3)”。

“expr.subList(leftIndex,rightIndex+1).clear()”中的 clear()方法清空子列表。由于子列表实际上只是源列表的一个视图，因此当清空子列表时，expr 源列表中的相应内容也被清空。

expr.add(leftIndex,Double.valueOf(result).toString())方法在索引为 leftIndex 的位置插入数据。List 实现类的 add(int index, E element)方法向列表中加入元素时，如果 index 索引位置已经有一个元素，那么这个元素以及后续元素会自动右移。例如假定 List 中的内容本来为{“6”,“+”,“-”,“3”}，执行 add(2,“5”)方法后，List 的内容变为{“6”,“+”,“5”,“-”,“3”}。

运行命令“java Calculater1 (1.1+3.3)*20-(4.8/0.2+5)”，将对命令行提供的表达式进行计算，得到运算结果为“59.00”。

# 18.9　编程实战：用集合工具对数字排序

第 17 章的 17.6 节（数组排序）介绍了用选择算法来对一批数字进行排序。实际上，除了自己辛苦实现复杂算法来对数字排序，还可以利用现成的工具类进行排序。java.util.Collections 类是为 Java 集合提供服务的一个工具类，它的 sort(List  list)方法能够对列表中的元素进行排序。请查阅相关的 API 文档（即 JavaDoc 文档），编写一个程序，利用 Collections 类来为一批数字排序

**编程提示**

以下例程 18-4 的 Sorter 类的 sort()方法能对一个整形数组中的数据进行排序，它先调用 Arrays 集合工具类的 asList(array)静态方法，把 array 数组参数转换成一个 List 对象。Sorder 类接下来再调用 Collections 集合工具类的 sort(list)静态方法对参数 list 列表中的元素排序。最后，Sorter 类把排序后的列表中的元素重新存放到数组 array 中。

例程 18-4　Sorter.java

```
import java.util.*;
```

```
public class Sorter{
  /* 对数组中的数字进行排序 */
  public static void sort(Integer[] array){
    //把数组中的元素存放到一个 List 列表中
    List<Integer> list=Arrays.asList(array);

    Collections.sort(list); //对列表中的元素排序

    //把列表中的元素依次存放到原来的数组中
    for(int i=0;i<list.size();i++)
      array[i]=list.get(i);

    print(array); //打印数组中的内容
  }

  public static void print(Integer[] array){
    for(Integer d : array)
      System.out.print(d+" ");
    System.out.println(); //打印换行符
  }

  public static void main(String args[]){
    //int 自动转换为 Integer 类型
    Integer[] array=new Integer[]{95,77,48,69,82};
    sort(array);
  }
}
```

# 18.10　编程实战：按月份先后循序数兔子

本书第 6 章的 6.8 节（数兔子）已经介绍了一个数兔子的实战编程题：有一对兔子第 1 个月出生，从出生后第 3 个月起每个月都生一对兔子，新生的每对兔子长到第 3 个月后每个月又生一对兔子，假如兔子都不死，请问到了第 n 个月，共有多少对兔子？

第 6 章的 6.8 节的例程 6-3（RabbitCouple.java）运用递归算法来数兔子。这种方法虽然能正确地统计所有兔子对数，但是它不是按照月份的先后循序来统计的。假如要按照月份的先后循环，依次清点每个月出生的兔子对，该如何实现呢？

**编程提示**

可以把已经出生的兔子对放在一个 rabbitCouples 列表中，然后在一个循环中，依次在每个月份都遍历这个列表。在每个月份中，只要列表中的兔子对达到了生兔子的年龄，就会创建一个新的 RabbitCouple 对象，这个对象又被添加到列表中。

以下例程 18-5 的 RabbitCouple 类就按照这种方式来数兔子。它有一个静态成员变量 rabbitCouples 列表，用来存放所有的兔子对。它的 giveBirthAllMonths(int months)方法负责在参数 months 指定的所有月份内统计出生的兔子对总数。

例程 18-5　RabbitCouple.java

```
import java.util.*;
public class RabbitCouple{  /* 表示一个兔子对 */
  //变量 sum 为静态变量,表示所有兔子的对数。初始值为 0
```

```
    private static int sum=0;
    private int bornMonth;  //兔子对出生的月份
    private static List<RabbitCouple> rabbitCouples
              =new ArrayList<RabbitCouple>();  //存放所有的兔子对

    public RabbitCouple(int bornMonth){
      this.bornMonth=bornMonth;
      sum++; //每当有新的一对兔子出生，sum 就增加 1

      //新出生的这对兔子加入到 rabbitCouples 列表中
      rabbitCouples.add(this);
      System.out.println("出生一对新兔子，出生月份："
              +bornMonth+" ，目前共有"+sum+"对兔子");
    }

    public void giveBirth(int month){  /* 在特定的月份生兔子 */
      if(month>=bornMonth+2)
        new RabbitCouple(month);  //生出一对兔子
    }

    public static void giveBirthAllMonths(int months){ /* 每个月依次生兔子 */
      for(int i=1;i<=months;i++){  //依次在每个月份中遍历兔子对列表
        int size=rabbitCouples.size();  //获得列表的当前长度
        //尝试让 rabbitCouples 列表中的每对兔子生新兔子
        for(int j=0;j<size;j++)
          rabbitCouples.get(j).giveBirth(i);
      }
    }

    /*  程序入口 main()方法 */
    public static void main(String args[]){
      int months=8;
      RabbitCouple firstCouple=new RabbitCouple(1); //第一对兔子在第一个月出生
      RabbitCouple.giveBirthAllMonths(months);  //从第一对兔子开始生小兔子
      System.out.println(months+"个月，一共有"+sum+"对兔子");
    }
  }
```

运行 RabbitCouple 类，将得到以下打印结果。从这个打印结果可以看出，RabbitCouple 类能按照月份的先后顺序来数兔子对。

```
出生一对新兔子，出生月份：1 ，目前共有 1 对兔子
出生一对新兔子，出生月份：3 ，目前共有 2 对兔子
出生一对新兔子，出生月份：4 ，目前共有 3 对兔子
出生一对新兔子，出生月份：5 ，目前共有 4 对兔子
出生一对新兔子，出生月份：5 ，目前共有 5 对兔子
出生一对新兔子，出生月份：6 ，目前共有 6 对兔子
出生一对新兔子，出生月份：6 ，目前共有 7 对兔子
出生一对新兔子，出生月份：6 ，目前共有 8 对兔子
出生一对新兔子，出生月份：7 ，目前共有 9 对兔子
出生一对新兔子，出生月份：7 ，目前共有 10 对兔子
出生一对新兔子，出生月份：7 ，目前共有 11 对兔子
出生一对新兔子，出生月份：7 ，目前共有 12 对兔子
出生一对新兔子，出生月份：7 ，目前共有 13 对兔子
```

```
出生一对新兔子，出生月份：8 ，目前共有14对兔子
出生一对新兔子，出生月份：8 ，目前共有15对兔子
……
出生一对新兔子，出生月份：8 ，目前共有21对兔子
8个月，一共有21对兔子
```

# 18.11　编程实战：用映射来存放学生信息

有位老师要管理一组学生的信息，每个学生都有唯一的id。要求根据学生的id来检索相应学生的详细信息，或者根据id来删除相应学生的信息，还需要打印所有学生的信息。

**编程提示**

可以用Map映射来建立id与学生信息之间的对应关系。以下例程18-6的Student类，表示学生。

例程18-6　Student.java

```
public class Student{
  private int id;
  private String name;
  private int age;

  public Student(int id,String name,int age){
    this.id=id;
    this.name=name;
    this.age=age;
  }

  public int getId(){return id;}
  public void setId(int id){this.id=id;}

  public String getName(){return name;}
  public void setName(String name){this.name=name;}

  public int getAge(){return age;}
  public void setAge(int age){this.age=age;}
}
```

以下例程18-7的Students类能够管理一组学生信息，它的map属性用来存放所有的id和Student对象之间的对应关系。

例程18-7　Students.java

```
import java.util.*;
public class Students{
  Map<Integer,Student> map=new TreeMap<Integer,Student>();

  /* 加入学生*/
  public void add(Integer id,Student student){
    map.put(id,student);
  }

  /* 根据id检索学生*/
```

```
    public Student get(Integer id){
      return map.get(id);
    }

    /* 根据id删除学生*/
    public void remove(Integer id){
      map.remove(id);
    }

    /* 打印学生信息 */
    public void print(){
      System.out.println("学号  姓名   年龄");

      Set<Integer> ids=map.keySet(); //获得所有学生的id
      for(Integer id:ids){
        Student student=map.get(id);
        System.out.println(id+"     "
          +student.getName()+"   "+student.getAge());
      }
    }

    public static void main(String[] args){
      Students ss=new Students();
      ss.add(2,new Student(2,"Lily",16));
      ss.add(1,new Student(1,"Mike",15));
      ss.add(3,new Student(3,"Mary",17));
      ss.add(4,new Student(4,"Jack",14));

      ss.remove(3); //删除学号为3的学生

      ss.print();
    }
  }
```

Students 类的 add()方法、get()方法和 remove()方法用来添加、读取以及删除学生信息，这些方法都是通过调用 map 映射的相关方法来实现的。

Students 类的 map 属性引用的是 TreeMap 对象。TreeMap 映射的特点是能根据键对象来对元素进行自动排序。运行 Students 类，会得到以下打印结果。从这个打印结果可以看出，map 中的 Student 对象按照 id 学号来排序：

```
学号  姓名   年龄
1     Mike   15
2     Lily   16
4     Jack   14
```

## 18.12　编程实战：圆桌报数游戏

有 n 个客人（以编号 1，2，3...n 分别表示）围坐在一张圆桌周围。从编号为 1 的人开始报数，数到 m 的那个人出列（离开圆桌）；他的下一个人又从 1 开始报数，数到 m 的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列。请编写一个程序来模拟这个报数游戏，依次打印出列的客人的编号。

**编程提示**

可以用列表来存放所有客人的编号。在以下例程 18-8 的 TableGame 类中，count()方法的 point 参数指向当前开始报数的人在列表中的索引。point 的初始值为 0。count()方法的报数流程如下：

（1）如果列表为空，表示所有的人都已经出列，那就直接从方法中返回。

（2）否则，从 point 指针指向的客人开始报数，然后 point 指针递增 1，确保它始终指向当前报数的客人，当 point 指针达到列表末尾，那就把 point 指针重置为 0。

（3）当报数达到 m，就打印当前 point 指针指向的元素，并将该元素从列表中删除。

（4）递归调用 count()方法，继续下一轮报数。

例程 18-8　TableGame.java

```
import java.util.*;
public class TableGame{
  public static void count(List<Integer> list,int m,int point){
    //如果列表为空，就返回
    if(list.size()==0)
      return;

    //模拟一轮报数的过程，确保point 指针始终指向当前报数的人
    for(int i=1;i<m;i++){
      if(point<list.size()-1)
        point++;
      else
        point=0; //当指针到达列表末尾，应该把指针重置为0
    }

    System.out.println("编号"+list.get(point)+"出列");
    list.remove(point); //从队列中删除一人

    count(list,m,point);  //递归调用下一轮报数
  }

  /* 参数n 表示所有客人的人数，参数m 表示报数的最大数 */
  public static void play(int n,int m){
    //列表中存放所有在座的客人的编号
    List<Integer> list=new LinkedList<Integer>();
    for(int i=1;i<=n;i++) //客人从1 开始编号
      list.add(i);

    count(list,m,0);
  }

  public static void main(String[] args){
    play(10,3);
  }
}
```

运行以上 TableGame 类的 main()方法，会模拟 10 个客人从 1 到 3 的报数过程，将得到以下打印结果：

```
编号3出列
编号6出列
```

```
编号 9 出列
编号 2 出列
编号 7 出列
编号 1 出列
编号 8 出列
编号 5 出列
编号 10 出列
编号 4 出列
```