☑ get(int index)：获得指定索引位置的元素。

☑ set(int index,Object obj)：将集合中指定索引位置的对象修改为指定的对象。

14.3.2 List 接口的实现类

视频讲解：光盘\TM\lx\14\List 接口的实现类.exe

List 接口的常用实现类有 ArrayList 与 LinkedList。

☑ ArrayList 类实现了可变的数组，允许保存所有元素，包括 null，并可以根据索引位置对集合进行快速的随机访问；缺点是向指定的索引位置插入对象或删除对象的速度较慢。

☑ LinkedList 类采用链表结构保存对象。这种结构的优点是便于向集合中插入和删除对象，需要向集合中插入、删除对象时，使用 LinkedList 类实现的 List 集合的效率较高；但对于随机访问集合中的对象，使用 LinkedList 类实现 List 集合的效率较低。

使用 List 集合时通常声明为 List 类型，可通过不同的实现类来实例化集合。

【例 14.2】 分别通过 ArrayList、LinkedList 类实例化 List 集合，代码如下：

```
List<E> list = new ArrayList<>();
List<E> list2 = new LinkedList<>();
```

在上面的代码中，E 可以是合法的 Java 数据类型。例如，如果集合中的元素为字符串类型，那么 E 可以修改为 String。

【例 14.3】 在项目中创建类 Gather，在主方法中创建集合对象，通过 Math 类的 random()方法随机获取集合中的某个元素，然后移除数组中索引位置是“2”的元素，最后遍历数组。（**实例位置：光盘\TM\sl\14.02**）

```
public class Gather {                                    //创建类 Gather
    public static void main(String[] args) {             //主方法
        List<String> list = new ArrayList<>();           //创建集合对象
        list.add("a");                                   //向集合添加元素
        list.add("b");
        list.add("c");
        int i = (int) (Math.random()*list.size());       //获得 0~2 之间的随机数
        System.out.println("随机获取数组中的元素：" + list.get(i));
        list.remove(2);                                  //将指定索引位置的元素从集合中移除
        System.out.println("将索引是'2'的元素从数组移除后，数组中的元素是：");
        for (int j = 0; j < list.size(); j++) {          //循环遍历集合
            System.out.println(list.get(j));
        }
    }
}
```

运行结果如图 14.3 所示。Math 类的 random()方法可获得一个 0.0~1.0 之间的随机数。

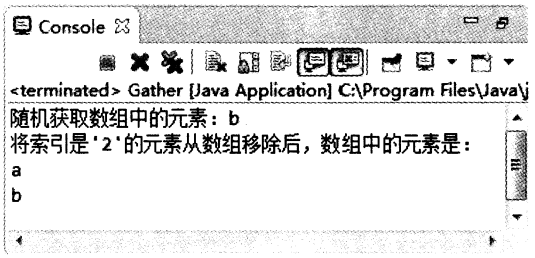


图 14.3　例 14.3 的运行结果

说明

与数组相同，集合的索引也是从 0 开始。

14.4　Set 集合

视频讲解：光盘\TM\lx\14\Set 集合.exe

Set 集合中的对象不按特定的方式排序，只是简单地把对象加入集合中，但 Set 集合中不能包含重复对象。Set 集合由 Set 接口和 Set 接口的实现类组成。Set 接口继承了 Collection 接口，因此包含 Collection 接口的所有方法。

注意

Set 的构造有一个约束条件，传入的 Collection 对象不能有重复值，必须小心操作可变对象（Mutable Object）。如果一个 Set 中的可变元素改变了自身状态导致 Object.equals(Object)=true，则会出现一些问题。

Set 接口常用的实现类有 HashSet 类与 TreeSet 类。

- ☑ HashSet 类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 Set 的迭代顺序，特别是它不保证该顺序恒久不变。此类允许使用 null 元素。
- ☑ TreeSet 类不仅实现了 Set 接口，还实现了 java.util.SortedSet 接口，因此，TreeSet 类实现的 Set 集合在遍历集合时按照自然顺序递增排序，也可以按照指定比较器递增排序，即可以通过比较器对用 TreeSet 类实现的 Set 集合中的对象进行排序。TreeSet 类新增的方法如表 14.2 所示。

表 14.2　TreeSet 类增加的方法

方　法	功 能 描 述
first()	返回此 Set 中当前第一个（最低）元素
last()	返回此 Set 中当前最后一个（最高）元素
comparator()	返回对此 Set 中的元素进行排序的比较器。如果此 Set 使用自然顺序，则返回 null

续表

方　法	功 能 描 述
headSet(E toElement)	返回一个新的 Set 集合，新集合是 toElement（不包含）之前的所有对象
subSet(E fromElement, E fromElement)	返回一个新的 Set 集合，是 fromElement（包含）对象与 fromElement（不包含）对象之间的所有对象
tailSet(E fromElement)	返回一个新的 Set 集合，新集合包含对象 fromElement（包含）之后的所有对象

【例 14.4】 在项目中创建类 UpdateStu，实现 Comparable 接口，重写该接口中的 compareTo()方法。在主方法中创建 UpdateStu 对象，创建集合，并将 UpdateStu 对象添加到集合中。遍历该集合中的全部元素，以及通过 headSet()、subSet()方法获得的部分集合。（**实例位置：光盘\TM\sl\14.03**）

```
import java.util.Iterator;
import java.util.TreeSet;

public class UpdateStu implements Comparable<Object> {          //创建类实现 Comparable 接口
    String name;
    long id;

    public UpdateStu(String name, long id) {                    //构造方法
        this.id = id;
        this.name = name;
    }

    public int compareTo(Object o) {
        UpdateStu upstu = (UpdateStu) o;
        int result = id > upstu.id ? 1 : (id == upstu.id ? 0 : -1);     //参照代码说明
        return result;
    }

    public String getName() {
        return name;
    }

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

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public static void main(String[] args) {
        UpdateStu stu1 = new UpdateStu("李同学", 01011);
        UpdateStu stu2 = new UpdateStu("陈同学", 01021);        //创建 UpdateStu 对象
```

```
        UpdateStu stu3 = new UpdateStu("王同学", 01051);
        UpdateStu stu4 = new UpdateStu("马同学", 01012);
        TreeSet<UpdateStu> tree = new TreeSet<>();
        tree.add(stu1);                                    //向集合添加对象
        tree.add(stu2);
        tree.add(stu3);
        tree.add(stu4);
        Iterator<UpdateStu> it = tree.iterator();     //Set 集合中的所有对象的迭代器
        System.out.println("Set 集合中的所有元素：");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
        it = tree.headSet(stu2).iterator();                //截取排在 stu2 对象之前的对象
        System.out.println("截取前面部分的集合：");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
        it = tree.subSet(stu2, stu3).iterator();           //截取排在 stu2 与 stu3 之间的对象
        System.out.println("截取中间部分的集合");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
    }
}
```

运行结果如图 14.4 所示。

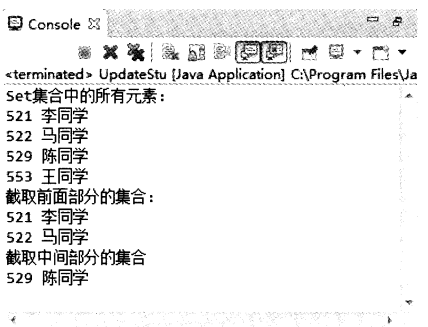


图 14.4　例 14.4 的运行结果

代码说明：存入 TreeSet 类实现的 Set 集合必须实现 Comparable 接口，该接口中的 compareTo(Object o) 方法比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、0 或正整数。

技巧

headSet()、subSet()、tailSet()方法截取对象生成新集合时是否包含指定的参数，可通过如下方法来判别：如果指定参数位于新集合的起始位置，则包含该对象，如 subSet()方法的第一个参数和 tailSet()方法的参数；如果指定参数是新集合的终止位置，则不包含该参数，如 headSet()方法的入口参数和 subSet()方法的第二个入口参数。

14.5 Map 集合

Map 集合没有继承 Collection 接口，其提供的是 key 到 value 的映射。Map 中不能包含相同的 key，每个 key 只能映射一个 value。key 还决定了存储对象在映射中的存储位置，但不是由 key 对象本身决定的，而是通过一种“散列技术”进行处理，产生一个散列码的整数值。散列码通常用作一个偏移量，该偏移量对应分配给映射的内存区域的起始位置，从而确定存储对象在映射中的存储位置。Map 集合包括 Map 接口以及 Map 接口的所有实现类。

14.5.1 Map 接口

视频讲解：光盘\TM\lx\14\Map 接口.exe

Map 接口提供了将 key 映射到值的对象。一个映射不能包含重复的 key，每个 key 最多只能映射到一个值。Map 接口中同样提供了集合的常用方法，除此之外还包括如表 14.3 所示的常用方法。

表 14.3 Map 接口中的常用方法

方　法	功 能 描 述
put(K key, V value)	向集合中添加指定的 key 与 value 的映射关系
containsKey(Object key)	如果此映射包含指定 key 的映射关系，则返回 true
containsValue(Object value)	如果此映射将一个或多个 key 映射到指定值，则返回 true
get(Object key)	如果存在指定的 key 对象，则返回该对象对应的值，否则返回 null
keySet()	返回该集合中的所有 key 对象形成的 Set 集合
values()	返回该集合中所有值对象形成的 Collection 集合

下面通过实例介绍 Map 接口中某些方法的使用。

【例 14.5】 在项目中创建类 UpdateStu，在主方法中创建 Map 集合，并获取 Map 集合中所有 key 对象的集合和所有 values 值的集合，最后遍历集合。（**实例位置：光盘\TM\sl\14.04**）

```
public class UpdateStu {
    public static void main(String[] args) {
        Map<String,String> map = new HashMap<>();          //创建 Map 实例
        map.put("01", "李同学");                            //向集合中添加对象
        map.put("02", "魏同学");
```

```
        Set <String> set = map.keySet();                    //构建 Map 集合中所有 key 对象的集合
        Iterator <String> it = set.iterator();              //创建集合迭代器
        System.out.println("key 集合中的元素：");
        while (it.hasNext()) {                              //遍历集合
            System.out.println(it.next());
        }
        Collection <String> coll = map.values();            //构建 Map 集合中所有 values 值的集合
        it = coll.iterator();
        System.out.println("values 集合中的元素：");
        while (it.hasNext()) {                              //遍历集合
            System.out.println(it.next());
        }
    }
}
```

运行结果如图 14.5 所示。

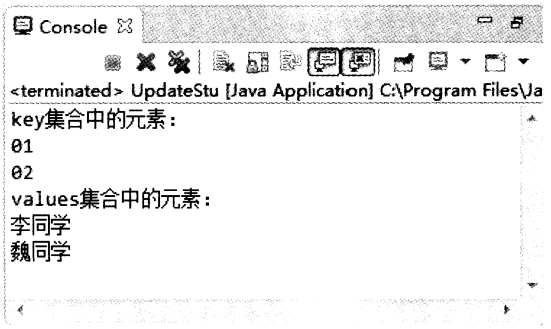


图 14.5　例 14.5 的运行结果

说明

Map 集合中允许值对象是 null，而且没有个数限制。例如，可通过“map.put("05",null);”语句向集合中添加对象。

14.5.2　Map 接口的实现类

视频讲解：光盘\TM\lx\14\Map 接口的实现类.exe

Map 接口常用的实现类有 HashMap 和 TreeMap。建议使用 HashMap 类实现 Map 集合，因为由 HashMap 类实现的 Map 集合添加和删除映射关系效率更高。HashMap 是基于哈希表的 Map 接口的实现，HashMap 通过哈希码对其内部的映射关系进行快速查找；而 TreeMap 中的映射关系存在一定的顺序，如果希望 Map 集合中的对象也存在一定的顺序，应该使用 TreeMap 类实现 Map 集合。

- ☑ HashMap 类是基于哈希表的 Map 接口的实现，此实现提供所有可选的映射操作，并允许使用 null 值和 null 键，但必须保证键的唯一性。HashMap 通过哈希表对其内部的映射关系进行快速查找。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
- ☑ TreeMap 类不仅实现了 Map 接口，还实现了 java.util.SortedMap 接口，因此，集合中的映射关系具有一定的顺序。但在添加、删除和定位映射关系时，TreeMap 类比 HashMap 类性能稍差。由于 TreeMap 类实现的 Map 集合中的映射关系是根据键对象按照一定的顺序排列的，因此不允许键对象是 null。

可以通过 HashMap 类创建 Map 集合，当需要顺序输出时，再创建一个完成相同映射关系的 TreeMap 类实例。

【例 14.6】 通过 HashMap 类实例化 Map 集合，并遍历该 Map 集合，然后创建 TreeMap 实例实现将集合中的元素顺序输出。（**实例位置：光盘\TM\sl\14.05**）

（1）首先创建 Emp 类，代码如下：

```
public class Emp {
    private String e_id;
    private String e_name;
    public Emp( String e_id,String e_name) {
        this.e_id = e_id;
        this.e_name = e_name;
    }
    /*************省略了属性的 setXXX()以及 getXXX()方法******************/
}
```

（2）创建一个用于测试的主类。首先新建一个 Map 集合，并添加集合对象。分别遍历由 HashMap 类与 TreeMap 类实现的 Map 集合，观察两者的不同点。关键代码如下：

```
public class MapText {                                          //创建类 MapText
    public static void main(String[] args) {                   //主方法
        Map<String,String> map = new HashMap<>();              //由 HashMap 实现的 Map 对象

        Emp emp = new Emp("351", "张三");                       //创建 Emp 对象
        Emp emp2 = new Emp("512", "李四");
        Emp emp3 = new Emp("853", "王一");
        Emp emp4 = new Emp("125", "赵六");
        Emp emp5 = new Emp("341", "黄七");

        map.put(emp4.getE_id(), emp4.getE_name());              //将对象添加到集合中
        map.put(emp5.getE_id(), emp5.getE_name());
        map.put(emp.getE_id(), emp.getE_name());
        map.put(emp2.getE_id(), emp2.getE_name());
        map.put(emp3.getE_id(), emp3.getE_name());

                Set <String> set = map.keySet();                //获取 Map 集合中的 key 对象集合
        Iterator <String> it = set.iterator();
        System.out.println("HashMap 类实现的 Map 集合，无序：");
        while (it.hasNext()) {
```

```
            String str = (String) it.next();
            String name = (String) map.get(str);                    //遍历 Map 集合
            System.out.println(str + " " + name);
        }
        TreeMap<String,String> treemap = new TreeMap<>();           //创建 TreeMap 集合对象
        treemap.putAll(map);                                        //向集合添加对象
        Iterator <String> iter = treemap.keySet().iterator();
        System.out.println("TreeMap 类实现的 Map 集合，键对象升序：");
        while (iter.hasNext()) {                                    //遍历 TreeMap 集合对象
            String str = (String) iter.next();                      //获取集合中的所有 key 对象
            String name = (String) treemap.get(str);                //获取集合中的所有 values 值
            System.out.println(str + " " + name);
        }
    }
}
```

运行结果如图 14.6 所示。

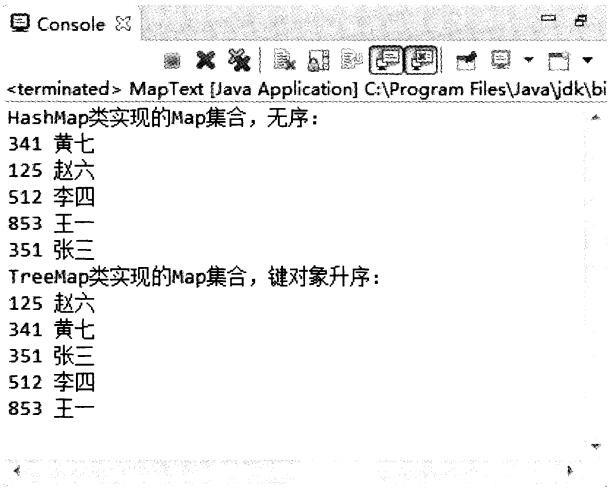


图 14.6　例 14.6 的运行结果

14.6　小　　结

本章介绍了 Java 中常见的集合，包括 List 集合、Set 集合和 Map 集合。对于每种集合的特点应该有所了解，重点掌握集合的遍历、添加对象、删除对象的方法。本章在介绍每种集合时都给出了典型、实用的小例子，以帮助读者掌握集合类的常用方法。集合是 Java 语言中很重要的部分，通过本章的学习，读者应该学会使用集合类。

14.7 实践与练习

1．将 1~100 之间的所有正整数存放在一个 List 集合中，并将集合中索引位置是 10 的对象从集合中移除。（**答案位置：光盘\TM\sl\14.06**）

2．分别向 Set 集合以及 List 集合中添加“A”“a”“c”“C”“a”5 个元素，观察重复值“a”能否重复地在 List 集合以及 Set 集合中添加。（**答案位置：光盘\TM\sl\14.07**）

3．创建 Map 集合，创建 Emp 对象，并将 Emp 对象添加到集合中（Emp 对象的 id 作为 Map 集合的键），并将 id 为“015”的对象从集合中移除。（**答案位置：光盘\TM\sl\14.08**）

第15章

I/O（输入/输出）

（视频讲解：22分钟）

在变量、数组和对象中存储的数据是暂时存在的，程序结束后它们就会丢失。为了能够永久地保存程序创建的数据，需要将其保存在磁盘文件中，这样就可以在其他程序中使用它们。Java 的 I/O 技术可以将数据保存到文本文件、二进制文件甚至是 ZIP 压缩文件中，以达到永久性保存数据的要求。掌握 I/O 处理技术能够提高对数据的处理能力。本章将向读者介绍 Java 的 I/O（输入/输出）技术。

通过阅读本章，您可以：

- 了解流的概念
- 了解输入/输出流的分类
- 掌握文件输入/输出流的使用方法
- 掌握带缓存的输入/输出流的使用方法
- 理解 ZIP 压缩输入/输出流的应用

15.1 流 概 述

视频讲解：光盘\TM\lx\15\流概述.exe

流是一组有序的数据序列，根据操作的类型，可分为输入流和输出流两种。I/O（Input/Output，输入/输出）流提供了一条通道程序，可以使用这条通道把源中的字节序列送到目的地。虽然 I/O 流通常与磁盘文件存取有关，但是程序的源和目的地也可以是键盘、鼠标、内存或显示器窗口等。

Java 由数据流处理输入/输出模式，程序从指向源的输入流中读取源中的数据，如图 15.1 所示。源可以是文件、网络、压缩包或其他数据源。

输出流的指向是数据要到达的目的地，程序通过向输出流中写入数据把信息传递到目的地，如图 15.2 所示。输出流的目标可以是文件、网络、压缩包、控制台和其他数据输出目标。

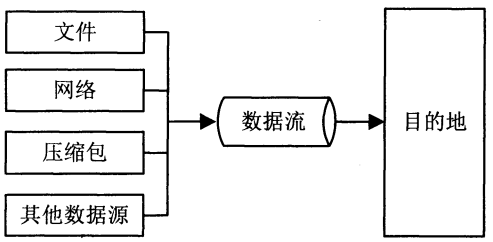


图 15.1 输入模式

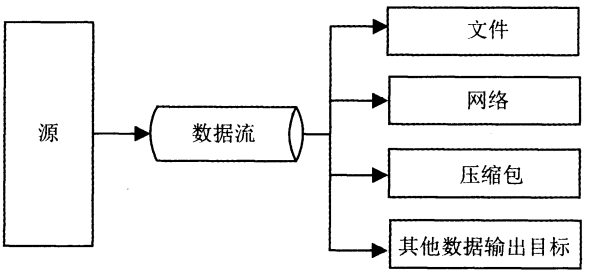


图 15.2 输出模式

15.2 输入/输出流

Java 语言定义了许多类专门负责各种方式的输入/输出，这些类都被放在 java.io 包中。其中，所有输入流类都是抽象类 InputStream（字节输入流）或抽象类 Reader（字符输入流）的子类；而所有输出流都是抽象类 OutputStream（字节输出流）或抽象类 Writer（字符输出流）的子类。

15.2.1 输入流

视频讲解：光盘\TM\lx\15\输入流.exe

InputStream 类是字节输入流的抽象类，是所有字节输入流的父类。InputStream 类的具体层次结构如图 15.3 所示。

该类中所有方法遇到错误时都会引发 IOException 异常。下面是对该类中的一些方法的简要说明。

☑ read()方法：从输入流中读取数据的下一个字节。返回 0~255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值为-1。

- ☑ read(byte[] b)：从输入流中读入一定长度的字节，并以整数的形式返回字节数。
- ☑ mark(int readlimit)方法：在输入流的当前位置放置一个标记，readlimit 参数告知此输入流在标记位置失效之前允许读取的字节数。
- ☑ reset()方法：将输入指针返回到当前所做的标记处。
- ☑ skip(long n)方法：跳过输入流上的 n 个字节并返回实际跳过的字节数。
- ☑ markSupported()方法：如果当前流支持 mark()/reset()操作就返回 true。
- ☑ close 方法：关闭此输入流并释放与该流关联的所有系统资源。

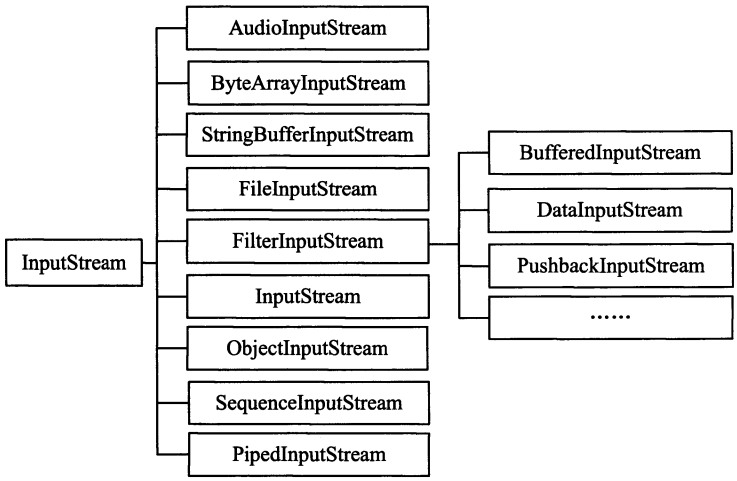


图 15.3　InputStream 类的层次结构

说明

并不是所有的 InputStream 类的子类都支持 InputStream 中定义的所有方法，如 skip()、mark()、reset()等方法只对某些子类有用。

Java 中的字符是 Unicode 编码，是双字节的。InputStream 是用来处理字节的，并不适合处理字符文本。Java 为字符文本的输入专门提供了一套单独的类 Reader，但 Reader 类并不是 InputStream 类的替换者，只是在处理字符串时简化了编程。Reader 类是字符输入流的抽象类，所有字符输入流的实现都是它的子类。Reader 类的具体层次结构如图 15.4 所示。

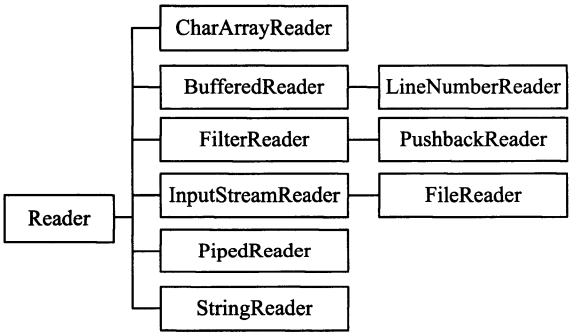


图 15.4　Reader 类的层次结构

Reader 类中的方法与 InputStream 类中的方法类似，读者在需要时可查看 JDK 文档。

15.2.2 输出流

视频讲解：光盘\TM\lx\15\输出流.exe

OutputStream 类是字节输出流的抽象类，此抽象类是表示输出字节流的所有类的超类。OutputStream 类的具体层次如图 15.5 所示。

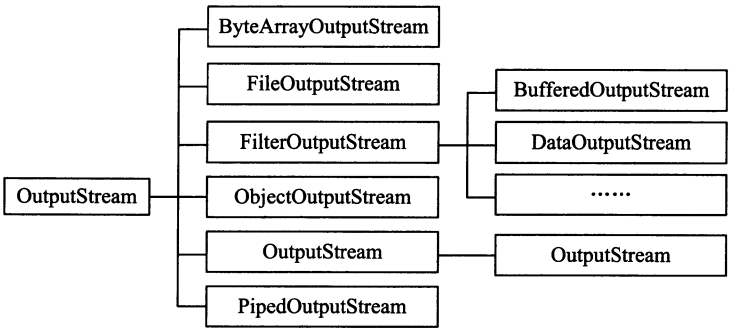


图 15.5 OutputStream 类的层次结构

OutputStream 类中的所有方法均返回 void，在遇到错误时会引发 IOException 异常。下面对 OutputStream 类中的方法作简单的介绍。

- ☑ write(int b)方法：将指定的字节写入此输出流。
- ☑ write(byte[] b)方法：将 b 个字节从指定的 byte 数组写入此输出流。
- ☑ write(byte[] b,int off,int len)方法：将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此输出流。
- ☑ flush()方法：彻底完成输出并清空缓存区。
- ☑ close()方法：关闭输出流。

Writer 类是字符输出流的抽象类，所有字符输出类的实现都是它的子类。Writer 类的层次结构如图 15.6 所示。

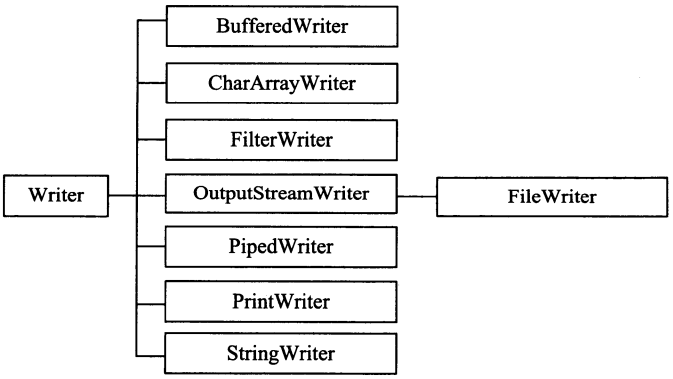


图 15.6 Writer 类的层次结构

15.3　File 类

视频讲解：光盘\TM\lx\15\File 类.exe

File 类是 java.io 包中唯一代表磁盘文件本身的对象。File 类定义了一些与平台无关的方法来操作文件，可以通过调用 File 类中的方法，实现创建、删除、重命名文件等操作。File 类的对象主要用来获取文件本身的一些信息，如文件所在的目录、文件的长度、文件读写权限等。数据流可以将数据写入到文件中，文件也是数据流最常用的数据媒体。

15.3.1　文件的创建与删除

视频讲解：光盘\TM\lx\15\文件的创建与删除.exe

可以使用 File 类创建一个文件对象。通常使用以下 3 种构造方法来创建文件对象。

（1）File(String pathname)

该构造方法通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。

语法如下：

```
new File(String pathname)
```

其中，pathname 指路径名称（包含文件名）。例如：

```
File file = new File("d:/1.txt");
```

（2）File(String parent,String child)

该构造方法根据定义的父路径和子路径字符串（包含文件名）创建一个新的 File 对象。

语法如下：

```
new  File(String parent,String child)
```

☑ parent：父路径字符串。例如，D:/或 D:/doc。

☑ child：子路径字符串。例如，letter.txt。

（3）File(File f , String child)

该构造方法根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

语法如下：

```
new File(File f,String child)
```

☑ f：父路径对象。例如，D:/doc/。

☑ child：子路径字符串。例如，letter.txt。

说明

对于 Microsoft Windows 平台，包含盘符的路径名前缀由驱动器号和一个“:”组成。如果路径名是绝对路径名，还可能后跟“\\”。

当使用 File 类创建一个文件对象后，例如：

```
File file = new File("word.txt");
```

如果当前目录中不存在名称为 word 的文件，File 类对象可通过调用 createNewFile()方法创建一个名称为 word.txt 的文件；如果存在 word.txt 文件，可以通过文件对象的 delete()方法将其删除，如例 15.1 所示。

【例 15.1】 在项目中创建类 FileTest，在主方法中判断 D 盘的 myword 文件夹是否存在 word.txt 文件，如果该文件存在则将其删除，不存在则创建该文件。（**实例位置：光盘\TM\sl\15.01**）

```
public class FileTest {                                    //创建类 FileTest
    public static void main(String[] args) {               //主方法
        File file = new File("word.txt");                  //创建文件对象
        if (file.exists()) {                               //如果该文件存在
            file.delete();                                 //将文件删除
            System.out.println("文件已删除");               //输出的提示信息
        } else {                                           //如果文件不存在
            try {                                          //try 语句块捕捉可能出现的异常
                file.createNewFile();                      //创建该文件
                System.out.println("文件已创建");           //输出的提示信息
            } catch (Exception e) {
                e.printStackTrace();                               }
        }
    }
}
```

运行结果如图 15.7 所示。

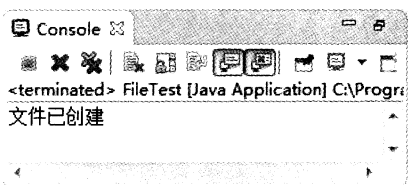


图 15.7 例 15.1 的运行结果

说明

由于 D:/myword 目录下并没有 word.txt 文件，因此运行程序会创建 word.txt 文件。

15.3.2 获取文件信息

视频讲解：光盘\TM\lx\15\获取文件信息.exe

File 类提供了很多方法用于获取一些文件本身信息，其中常用的方法如表 15.1 所示。

表 15.1　File 类的常用方法

方　法	返 回 值	说　明
getName()	String	获取文件的名称
canRead()	boolean	判断文件是否为可读的
canWrite()	boolean	判断文件是否可被写入
exits()	boolean	判断文件是否存在
length()	long	获取文件的长度（以字节为单位）
getAbsolutePath()	String	获取文件的绝对路径
getParent()	String	获取文件的父路径
isFile()	boolean	判断文件是否存在
isDirectory()	boolean	判断文件是否为一个目录
isHidden()	boolean	判断文件是否为隐藏文件
lastModified()	long	获取文件最后修改时间

下面通过实例来介绍如何使用上述的某些方法来获取文件的信息。

【例 15.2】获取当前文件夹下的 word.txt 文件的文件名、文件长度并判断该文件是否为隐藏文件。（实例位置：光盘\TM\sl\15.02）

```
public class FileTest {                                    //创建类
    public static void main(String[] args) {
        File file = new File("word.txt");                  //创建文件对象
        if (file.exists()) {                               //如果文件存在
            String name = file.getName();                  //获取文件名称
            long length = file.length();                   //获取文件长度
            boolean hidden = file.isHidden();              //判断文件是否为隐藏文件
            System.out.println("文件名称：" + name);         //输出信息
            System.out.println("文件长度是：" + length);
            System.out.println("该文件是隐藏文件吗？" + hidden);
        } else {                                           //如果文件不存在
            System.out.println("该文件不存在");              //输出信息
        }
    }
}
```

运行结果如图 15.8 所示。

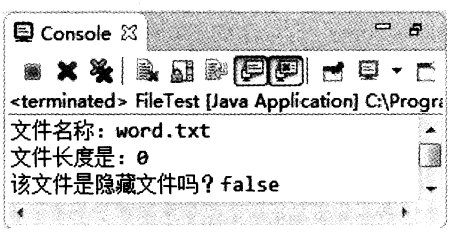


图 15.8　例 15.2 的运行结果

15.4 文件输入/输出流

程序运行期间，大部分数据都在内存中进行操作，当程序结束或关闭时，这些数据将消失。如果需要将数据永久保存，可使用文件输入/输出流与指定的文件建立连接，将需要的数据永久保存到文件中。本节将介绍文件输入/输出流。

15.4.1 FileInputStream 与 FileOutputStream 类

视频讲解：光盘\TM\lx\15\FileInputStream 与 FileOutputStream 类.exe

FileInputStream 类与 FileOutputStream 类都用来操作磁盘文件。如果用户的文件读取需求比较简单，则可以使用 FileInputStream 类，该类继承自 InputStream 类。FileOutputStream 类与 FileInputStream 类对应，提供了基本的文件写入能力。FileOutputStream 类是 OutputStream 类的子类。

FileInputStream 类常用的构造方法如下：

☑ FileInputStream(String name)。

☑ FileInputStream(File file)。

第一个构造方法使用给定的文件名 name 创建一个 FileInputStream 对象，第二个构造方法使用 File 对象创建 FileInputStream 对象。第一个构造方法比较简单，但第二个构造方法允许在把文件连接输入流之前对文件作进一步分析。

FileOutputStream 类有与 FileInputStream 类相同的参数构造方法，创建一个 FileOutputStream 对象时，可以指定不存在的文件名，但此文件不能是一个已被其他程序打开的文件。下面的实例就是使用 FileInputStream 与 FileOutputStream 类实现文件的读取与写入功能的。

【例 15.3】 使用 FileOutputStream 类向文件 word.txt 写入信息，然后通过 FileInputStream 类将文件中的数据读取到控制台上。（**实例位置：光盘\TM\sl\15.03**）

```
public class FileTest {                          //创建类
    public static void main(String[] args) {    //主方法
        File file = new File("word.txt");        //创建文件对象
        try {                                    //捕捉异常
            //创建 FileOutputStream 对象
            FileOutputStream out = new FileOutputStream(file);
            //创建 byte 型数组
            byte buy[] = "我有一只小毛驴，我从来也不骑。".getBytes();
            out.write(buy);                      //将数组中的信息写入到文件中
            out.close();                         //将流关闭
        } catch (Exception e) {                  //catch 语句处理异常信息
            e.printStackTrace();                 //输出异常信息
        }
        try {
            //创建 FileInputStream 类对象
```

```
            FileInputStream in = new FileInputStream(file);
            byte byt[] = new byte[1024];    //创建 byte 数组
            int len = in.read(byt);         //从文件中读取信息
            //将文件中的信息输出
            System.out.println("文件中的信息是：" + new String(byt, 0, len));
            in.close();                     //关闭流
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如图 15.9 所示。

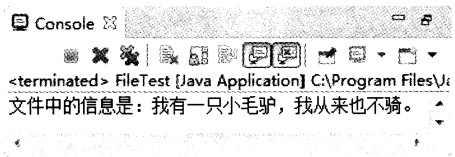


图 15.9　例 15.3 的运行结果

说明

虽然 Java 在程序结束时自动关闭所有打开的流，但是当使用完流后，显式地关闭所有打开的流仍是一个好习惯。一个被打开的流有可能会用尽系统资源，这取决于平台和实现。如果没有将打开的流关闭，当另一个程序试图打开另一个流时，可能会得不到需要的资源。

15.4.2　FileReader 和 FileWriter 类

视频讲解：光盘\TM\lx\15\FileReader 和 FileWriter 类.exe

使用 FileOutputStream 类向文件中写入数据与使用 FileInputStream 类从文件中将内容读出来，都存在一点不足，即这两个类都只提供了对字节或字节数组的读取方法。由于汉字在文件中占用两个字节，如果使用字节流，读取不好可能会出现乱码现象，此时采用字符流 Reader 或 Writer 类即可避免这种现象。

FileReader 和 FileWriter 字符流对应了 FileInputStream 和 FileOutputStream 类。FileReader 流顺序地读取文件，只要不关闭流，每次调用 read()方法就顺序地读取源中其余的内容，直到源的末尾或流被关闭。

下面通过一个应用程序介绍 FileReader 与 FileWriter 类的用法。

【例 15.4】 本实例创建 Swing 窗体，单击窗体中的“写入文件”按钮实现将文本框中的数据写入到磁盘文件中，单击“读取文件”按钮，系统将磁盘文件中的信息显示在文本框中。（**实例位置：光盘\TM\sl\15.04**）

```
/********************* 省略了导入相应的包 *************************/
public class Ftest extends JFrame {                    //创建类，继承 JFrame 类
    private static final long serialVersionUID = 1L;
    private JPanel jContentPane = null;                //创建面板对象
```

```
private JTextArea jTextArea = null;                    //创建文本域对象
private JPanel controlPanel = null;                    //创建面板对象
private JButton openButton = null;                     //创建按钮对象
private JButton closeButton = null;                    //创建按钮对象
/********************* 省略了对窗体进行布局代码 *****************/
private JButton getOpenButton() {
    if (openButton == null) {
        openButton = new JButton();
        openButton.setText("写入文件");                 //修改按钮的提示信息
        openButton.addActionListener(new ActionListener() {
            //按钮的单击事件
            public void actionPerformed(ActionEvent e) {
                //创建文件对象
                File file = new File("word.txt");
                try {
                    //创建 FileWriter 对象
                    FileWriter out = new FileWriter(file);
                    //获取文本域中文本
                    String s = jTextArea.getText();
                    out.write(s);                      //将信息写入磁盘文件
                    out.close();                       //将流关闭
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });
    }
    return openButton;
}
private JButton getCloseButton() {
    if (closeButton == null) {
        closeButton = new JButton();
        closeButton.setText("读取文件");                //修改按钮的提示信息
        closeButton.addActionListener(new ActionListener(){
            //按钮的单击事件
            public void actionPerformed(ActionEvent e) {
                File file = new File("word.txt");      //创建文件对象
                try {
                    //创建 FileReader 对象
                    FileReader in = new FileReader(file);
                    char byt[] = new char[1024];        //创建 char 型数组
                    int len = in.read(byt);            //将字节读入数组
                    //设置文本域的显示信息
                    jTextArea.setText(new String(byt, 0, len));
                    in.close();                        //关闭流
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });
```

```
        }
        return closeButton;
    }
    public Ftest() {
        super();
        initialize();
    }
    private void initialize() {
        this.setSize(300, 200);
        this.setContentPane(getJContentPane());
        this.setTitle("JFrame");
    }
    private JPanel getJContentPane() {
        if (jContentPane == null) {
            jContentPane = new JPanel();
            jContentPane.setLayout(new BorderLayout());
            jContentPane.add(getJTextArea(), BorderLayout.CENTER);
            jContentPane.add(getControlPanel(), BorderLayout.SOUTH);
        }
        return jContentPane;
    }
    public static void main(String[] args) {    //主方法
        Ftest thisClass = new Ftest();          //创建本类对象
        thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        thisClass.setVisible(true);             //设置该窗体为显示状态
    }
}
```

运行结果如图 15.10 所示。

图 15.10　例 15.4 的运行结果

该程序设计了一个文本域和两个按钮，当单击“读取文件”按钮时，会将磁盘文件中的数据信息显示到文本域中；当单击“写入文件”按钮时，会将用户在文本域中的输入信息写入到磁盘文件中。

15.5 带缓存的输入/输出流

缓存是 I/O 的一种性能优化。缓存流为 I/O 流增加了内存缓存区。有了缓存区，使得在流上执行 skip()、mark()和 reset()方法都成为可能。

15.5.1 BufferedInputStream 与 BufferedOutputStream 类

视频讲解：光盘\TM\lx\15\BufferedInputStream 与 BufferedOutputStream 类.exe

BufferedInputStream 类可以对所有 InputStream 类进行带缓存区的包装以达到性能的优化。BufferedInputStream 类有两个构造方法：

☑ BufferedInputStream(InputStream in)。

☑ BufferedInputStream(InputStream in,int size)。

第一种形式的构造方法创建了一个带有 32 个字节的缓存流；第二种形式的构造方法按指定的大小来创建缓存区。一个最优的缓存区的大小，取决于它所在的操作系统、可用的内存空间以及机器配置。从构造方法可以看出，BufferedInputStream 对象位于 InputStream 类对象之前。图 15.11 描述了字节数据读取文件的过程。

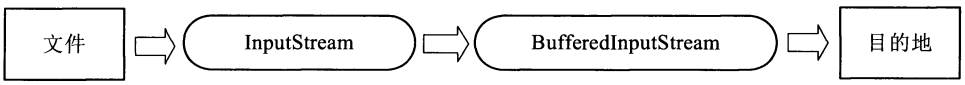


图 15.11 BufferedInputStream 读取文件过程

使用 BufferedOutputStream 输出信息和用 OutputStream 输出信息完全一样，只不过 BufferedOutputStream 有一个 flush()方法用来将缓存区的数据强制输出完。BufferedOutputStream 类也有两个构造方法：

☑ BufferedOutputStream(OutputStream in)。

☑ BufferedOutputStream(OutputStream in,int size)。

第一种构造方法创建一个有 32 个字节的缓存区，第二种构造方法以指定的大小来创建缓存区。

注意

flush()方法就是用于即使在缓存区没有满的情况下，也将缓存区的内容强制写入到外设，习惯上称这个过程为刷新。flush()方法只对使用缓存区的 OutputStream 类的子类有效。当调用 close()方法时，系统在关闭流之前，也会将缓存区中的信息刷新到磁盘文件中。

15.5.2 BufferedReader 与 BufferedWriter 类

视频讲解：光盘\TM\lx\15\BufferedReader 与 BufferedWriter 类.exe

BufferedReader 类与 BufferedWriter 类分别继承 Reader 类与 Writer 类。这两个类同样具有内部缓存

机制，并可以以行为单位进行输入/输出。

根据 BufferedReader 类的特点，总结出如图 15.12 所示的字符数据读取文件的过程。

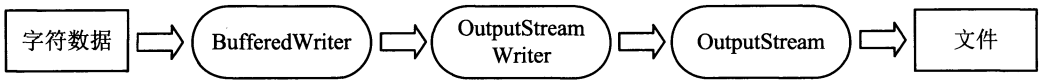


图 15.12　BufferedReader 类读取文件的过程

BufferedReader 类常用的方法如下。

☑ read()方法：读取单个字符。

☑ readLine()方法：读取一个文本行，并将其返回为字符串。若无数据可读，则返回 null。

BufferedWriter 类中的方法都返回 void。常用的方法如下。

☑ write(String s, int off, int len)方法：写入字符串的某一部分。

☑ flush()方法：刷新该流的缓存。

☑ newLine()方法：写入一个行分隔符。

在使用 BufferedWriter 类的 Write()方法时，数据并没有立刻被写入输出流，而是首先进入缓存区中。如果想立刻将缓存区中的数据写入输出流，一定要调用 flush()方法。

【例 15.5】 本实例向指定的磁盘文件中写入数据，并通过 BufferedReader 类将文件中的信息分行显示。代码如下：（**实例位置：光盘\TM\sl\15.05**）

```
public class Student {                                          //创建类
    public static void main(String args[]) {                   //主方法
        //定义字符串数组
        String content[] = { "好久不见", "最近好吗", "常联系" };
        File file = new File("word.txt");                       //创建文件对象
        try {
            FileWriter fw = new FileWriter(file);               //创建 FileWriter 类对象
            //创建 BufferedWriter 类对象
            BufferedWriter bufw = new BufferedWriter(fw);
            for (int k = 0; k < content.length; k++) {          //循环遍历数组
                bufw.write(content[k]);                         //将字符串数组中的元素写入到磁盘文件中
                bufw.newLine();                                 //将数组中的单个元素以单行的形式写入文件
            }
            bufw.close();                                       //将 BufferedWriter 流关闭
            fw.close();                                         //将 FileWriter 流关闭
        } catch (Exception e) {                                 //处理异常
            e.printStackTrace();
        }
        try {
            FileReader fr = new FileReader(file);               //创建 FileReader 类对象
            //创建 BufferedReader 类对象
            BufferedReader bufr = new BufferedReader(fr);
            String s = null;                                    //创建字符串对象
            int i = 0;                                          //声明 int 型变量
            //如果文件的文本行数不为 null，则进入循环
            while ((s = bufr.readLine()) != null) {
                i++;                                            //将变量做自增运算
```

```
                System.out.println("第" + i + "行:" + s);     //输出文件数据
            }
            bufr.close();                                    //将 FileReader 流关闭
            fr.close();                                      //将 FileReader 流关闭
        } catch (Exception e) {                              //处理异常
            e.printStackTrace();
        }
    }
}
```

运行结果如图 15.13 所示。

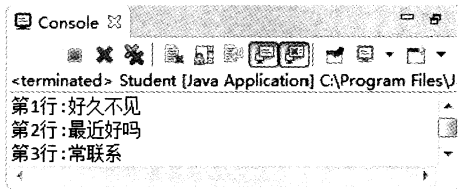


图 15.13　例 15.5 的运行结果

15.6　数据输入/输出流

视频讲解：光盘\TM\lx\15\数据输入输出流.exe

数据输入/输出流（DataInputStream 类与 DataOutputStream 类）允许应用程序以与机器无关的方式从底层输入流中读取基本 Java 数据类型。也就是说，当读取一个数据时，不必再关心这个数值应当是哪种字节。

DataInputStream 类与 DataOutputStream 类的构造方法如下。

☑ DataInputStream(InputStream in)：使用指定的基础 InputStream 创建一个 DataInputStream。

☑ DataOutputStream(OutputStream out)：创建一个新的数据输出流，将数据写入指定基础输出流。

DataOutputStream 类提供了如下 3 种写入字符串的方法。

☑ writeBytes(String s)。

☑ writeChars(String s)。

☑ writeUTF(String s)。

由于 Java 中的字符是 Unicode 编码，是双字节的，writeBytes 只是将字符串中的每一个字符的低字节内容写入目标设备中；而 writeChars 将字符串中的每一个字符的两个字节的内容都写到目标设备中；writeUTF 将字符串按照 UTF 编码后的字节长度写入目标设备，然后才是每一个字节的 UTF 编码。

DataInputStream 类只提供了一个 readUTF()方法返回字符串。这是因为要在一个连续的字节流读取一个字符串，如果没有特殊的标记作为一个字符串的结尾，并且不知道这个字符串的长度，就无法知道读取到什么位置才是这个字符串的结束。DataOutputStream 类中只有 writeUTF()方法向目标设备中写入字符串的长度，所以也能准确地读回写入字符串。

【例 15.6】 分别通过 DataOutputStream 类的 writeUTF()、writeChars()和 writeBytes()方法向指定的

磁盘文件中写入数据，并通过 DataInputStream 类的 readUTF()方法将写入的数据输出到控制台上。（**实例位置：光盘\TM\sl\15.06**）

```
public class Example_01 {                              //创建类
    public static void main(String[] args) {           //主方法
        try {
            //创建 FileOutputStream 对象
            FileOutputStream fs = new FileOutputStream("word.txt");
            //创建 DataOutputStream 对象
            DataOutputStream ds = new DataOutputStream(fs);
            ds.writeUTF("使用 writeUFT()方法写入数据;");  //写入磁盘文件数据
            ds.writeChars("使用 writeChars()方法写入数据;");
            ds.writeBytes("使用 writeBytes()方法写入数据.");
            ds.close();                                    //将流关闭
            //创建 FileInputStream 对象
            FileInputStream fis = new FileInputStream("word.txt");
            //创建 DataInputStream 对象
            DataInputStream dis = new DataInputStream(fis);
            System.out.print(dis.readUTF());               //将文件数据输出
        } catch (Exception e) {
            e.printStackTrace();                           //输出异常信息
        }
    }
}
```

运行结果如图 15.14 所示。

使用记事本程序将 word.txt 打开，如图 15.15 所示。尽管在记事本程序中看不出 writeUTF()写入的字符串是“使用 writeUFT()方法写入数据”，但程序通过 readUTF()读回后显示在屏幕上的仍是“使用 writeUFT()方法写入数据”。但如果使用 writeChars()和 writeBytes()方法写入字符串后，再读取回来就不容易了，读者不妨编写程序尝试一下。

图 15.14　例 15.6 的运行结果

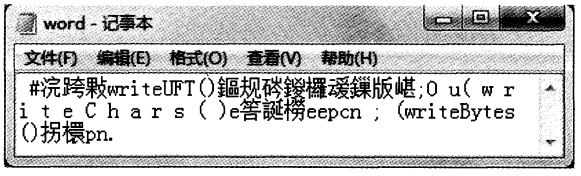


图 15.15　word.txt 文件内容

15.7　ZIP 压缩输入/输出流

ZIP 压缩管理文件（ZIP archive）是一种十分典型的文件压缩形式，使用它可以节省存储空间。关于 ZIP 压缩的 I/O 实现，在 Java 的内置类中提供了非常好用的相关类，所以其实现方式非常简单。本节将介绍使用 java.util.zip 包中的 ZipOutputStream 与 ZipInputStream 类来实现文件的压缩/解压缩。如要从 ZIP 压缩管理文件内读取某个文件，要先找到对应该文件的“目录进入点”（从它可知该文件在 ZIP 文件内的

位置），才能读取这个文件的内容。如果要将文件内容写入 ZIP 文件内，必须先写入对应于该文件的“目录进入点”，并且把要写入文件内容的位置移到此进入点所指的位置，然后再写入文件内容。

Java 实现了 I/O 数据流与网络数据流的单一接口，因此数据的压缩、网络传输和解压缩的实现比较容易。ZipEntry 类产生的对象，是用来代表一个 ZIP 压缩文件内的进入点（entry）。ZipInputStream 类用来读取 ZIP 压缩格式的文件，所支持的包括已压缩及未压缩的进入点（entry）。ZipOutputStream 类用来写出 ZIP 压缩格式的文件，而且所支持的包括已压缩及未压缩的进入点（entry）。下面介绍利用 ZipEntry、ZipInputStream 和 ZipOutputStream 3 个 Java 类实现 ZIP 数据压缩方式的编程方法。

15.7.1 压缩文件

视频讲解：光盘\TM\lx\15\压缩文件.exe

利用 ZipOutputStream 类对象，可将文件压缩为.zip 文件。ZipOutputStream 类的构造方法如下：

```
ZipOutputStream(OutputStream out);
```

ZipOutputStream 类的常用方法如表 15.2 所示。

表 15.2 ZipOutputStream 类的常用方法

方　法	返 回 值	说　明
putNextEntry(ZipEntry e)	void	开始写一个新的 ZipEntry，并将流内的位置移至此 entry 所指数据的开头
write(byte[] b , int off , int len)	void	将字节数组写入当前 ZIP 条目数据
finish()	void	完成写入 ZIP 输出流的内容，无须关闭它所配合的 OutputStream
setComment(String comment)	void	可设置此 ZIP 文件的注释文字

下面的实例为压缩 E 盘的 hello 文件夹，在该文件夹中有 hello1.txt 和 hello2.txt 文件，并将压缩后的 hello.zip 文件夹保存在 E 盘根目录下。

【例 15.7】 本实例创建类 MyZip，在 zip()方法中实现使用 ZipOutputStream 类对文件进行压缩，在主方法中调用该方法。（**实例位置：光盘\TM\sl\15.07**）

```
public class MyZip {                                          //创建类
    private void zip(String zipFileName, File inputFile) throws Exception {
        ZipOutputStream out = new ZipOutputStream(new FileOutputStream(
                zipFileName));                                //创建 ZipOutputStream 类对象
        zip(out, inputFile, "");                              //调用方法
        System.out.println("压缩中...");                       //输出信息
        out.close();                                          //将流关闭
    }
    private void zip(ZipOutputStream out, File f, String base)
            throws Exception {                                //方法重载
        if (f.isDirectory()) {                                //测试此抽象路径名表示的文件是否为一个目录
            File[] fl = f.listFiles();                        //获取路径数组
            if (base.length() != 0) {
                out.putNextEntry(new ZipEntry(base + "/")); //写入此目录的 entry
            }
```

```
            for (int i = 0; i < fl.length; i++) {                    //循环遍历数组中的文件
                zip(out, fl[i], base + fl[i]);
            }
        } else {
            out.putNextEntry(new ZipEntry(base));    //创建新的进入点
            //创建 FileInputStream 对象
            FileInputStream in = new FileInputStream(f);
            int b;                                                   //定义 int 型变量
            System.out.println(base);
            while ((b = in.read()) != -1) {                          //如果没有到达流的尾部
                out.write(b);                                        //将字节写入当前 ZIP 条目
            }
            in.close();                                              //关闭流
        }
    }
    public static void main(String[] temp) {                         //主方法
        MyZip book = new MyZip();                                    //创建本例对象
        try {
            //调用方法，参数为压缩后的文件与要压缩的文件
            book.zip("E:/hello.zip", new File("E:/hello"));
            System.out.println("压缩完成");                          //输出信息
        } catch (Exception ex) {
        }
    }
}
```

运行程序，可发现控制台的变化如图 15.16 所示。

从这个实例中可以看出，每一个 ZIP 文件中可能包含多个文件（本实例中包含了实例的源代码文件）。使用 ZipOutputStream 类将文件写入目标 ZIP 文件时，必须先使用 ZipOutputStream 对象的 putNextEntry()方法，写入个别文件的 entry，将流内目前指到的位置移到该 entry 所指的开头位置。

执行完之后，在当前项目目录下会产生 hello.zip 压缩文件，将其打开后如图 15.17 所示。

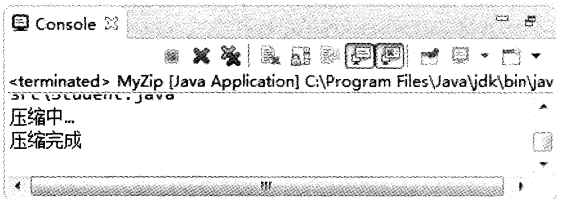


图 15.16　例 15.7 的运行结果

图 15.17　hello.zip 文件

15.7.2　解压缩 ZIP 文件

视频讲解：光盘\TM\lx\15\解压缩 ZIP 文件.exe

ZipInputStream 类可读取 ZIP 压缩格式的文件，包括已压缩和未压缩的条目（entry）。ZipInputStream

类的构造方法如下：

```
ZipInputStream(InputStream in)
```

ZipInputStream 类的常用方法如表 15.3 所示。

表 15.3　ZipInputStream 类的常用方法

方　法	返 回 值	说　明
read(byte[] b , int off , int len)	int	读取目标 b 数组内 off 偏移量的位置，长度是 len 字节
available()	int	判断是否已读完目前 entry 所指定的数据。已读完返回 0，否则返回 1
closeEntry()	void	关闭当前 ZIP 条目并定位流以读取下一个条目
skip(long n)	long	跳过当前 ZIP 条目中指定的字节数
getNextEntry()	ZipEntry	读取下一个 ZipEntry，并将流内的位置移至该 entry 所指数据的开头
createZipEntry(String name)	ZipEntry	以指定的 name 参数新建一个 ZipEntry 对象

下面的实例是将执行例 15.7 后生成的压缩文件 hello.zip 进行解压，解压后的文件存储在 ZipEntry 类的 getName()方法获取的目录下。

【例 15.8】 创建类 Decompressing，通过 ZipInputStream 类将例 15.7 生成的压缩文件解压到指定文件夹中。（**实例位置：光盘\TM\sl\15.08**）

```
public class Decompressing {                                    //创建文件
    public static void main(String[] temp) {
        File file = new File("hello.zip");                      //当前压缩文件
        ZipInputStream zin;                                     //创建 ZipInputStream 对象
        try {                                                   //try 语句捕获可能发生的异常
            ZipFile zipFile = new ZipFile(file);                //创建压缩文件对象
            zin = new ZipInputStream(new FileInputStream(file)); //实例化对象，指明要进行解压的文件
            ZipEntry entry = zin.getNextEntry();                //跳过根目录，获取下一个 ZipEntry
            while (((entry = zin.getNextEntry()) != null)
                    && !entry.isDirectory()) {                  //如果 entry 不为空，并不在同一目录下
                File tmp = new File("C:\\" + entry.getName());  //解压出的文件路径
                if (!tmp.exists()) {                            //如果该文件不存在
                    tmp.getParentFile().mkdirs();               //创建文件父类文件夹路径
                    OutputStream os = new FileOutputStream(tmp); //将文件目录中的文件放入输出流
                    //用输入流读取压缩文件中制定目录中的文件
                    InputStream in = zipFile.getInputStream(entry);
                    int count = 0;                              //创建临时变量
                    while ((count = in.read()) != -1) {         //如有输入流可以读取到数值
                        os.write(count);                        //输出流写入
                    }
                    os.close();                                 //关闭输出流
                    in.close();                                 //关闭输入流
                }
                zin.closeEntry();                               //关闭当前 entry
                System.out.println(entry.getName() + "解压成功");
            }
            zin.close();                                        //关闭流
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

运行结果如图 15.18 所示。

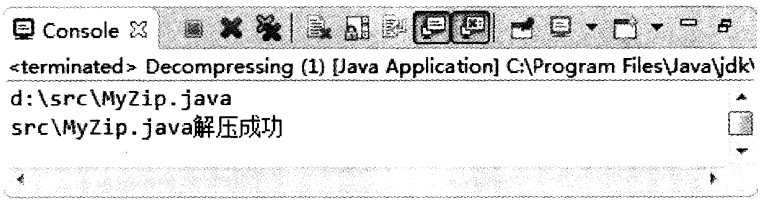


图 15.18　例 15.8 的运行结果

程序执行完毕之后，E盘的hello文件夹中将包含hello1.txt与hello2.txt文件。本实例是通过ZipEntry类的getName()方法得知此文件的名称（含path），借此来决定压缩之后的目录和文件名。使用ZipInputStream类来解压文件，必须先使用ZipInputStream类的getNextEntry()方法来取得其内的第一个ZipEntry。

15.8　小　　结

本章介绍了 Java 输入/输出流，Java I/O（输入/输出）机制提供了一套简单的标准化 API，以方便从不同的数据源读取和写入字符或字节数据。学习 Java 的 I/O 处理技术，必须了解 Java 的字节流和字符流。对于字节流和字符流扩展的子类也应熟练掌握。这些子类所实现的数据流可以把数据输出到指定的设备终端，或者从指定的设备终端输入数据。另外，使用数据流来读取磁盘文件信息以及使用数据流向磁盘文件写入信息，也是本章的重点，读者应该熟练掌握。

15.9　实践与练习

1．编写程序，实现读取文件时出现一个表示读取进度的进度条。可使用 javax.swing 包提供的输入流类 ProgressMonitorInputStream。（**答案位置：光盘\TM\sl\15.09**）

2．编写程序，使用字符输入、输出流读取文件，将一段文字加密后存入文件，然后再读取，并将加密前与加密后的文件输出。（**答案位置：光盘\TM\sl\15.10**）

3．编写程序，实现当用户输入姓名和密码时，将每一个姓名和密码加在文件中，如果用户输入 done，就结束程序。（**答案位置：光盘\TM\sl\15.11**）

第16章

反射

（视频讲解：22分钟）

通过 Java 的反射机制，程序员可以更深入地控制程序的运行过程，如在程序运行时对用户输入的信息进行验证，还可以逆向控制程序的执行过程。

另外，Java 还提供了 Annotation 功能，该功能建立在反射机制的基础上，本章对此也作了讲解，包括定义 Annotation 类型的方法和在程序运行时访问 Annotation 信息的方法。为了便于读者理解，在讲解过程中还结合了大量的实例。

通过阅读本章，您可以：

- 学会通过反射访问构造方法的方法
- 学会通过反射访问成员变量的方法
- 学会通过反射访问方法的方法
- 学会定义 Annotation 类型的方法
- 学会访问 Annotation 信息的方法

16.1　Class 类与 Java 反射

通过 Java 反射机制，可以在程序中访问已经装载到 JVM 中的 Java 对象的描述，实现访问、检测和修改描述 Java 对象本身信息的功能。Java 反射机制的功能十分强大，在 java.lang.reflect 包中提供了对该功能的支持。

众所周知，所有 Java 类均继承了 Object 类，在 Object 类中定义了一个 getClass()方法，该方法返回一个类型为 Class 的对象。例如下面的代码：

```
Class textFieldC = textField.getClass();          //textField 为 JTextField 类的对象
```

利用 Class 类的对象 textFieldC，可以访问用来返回该对象的 textField 对象的描述信息。可以访问的主要描述信息如表 16.1 所示。

表 16.1　通过反射可访问的主要描述信息

组 成 部 分	访 问 方 法	返回值类型	说　明
包路径	getPackage()	Package 对象	获得该类的存放路径
类名称	getName()	String 对象	获得该类的名称
继承类	getSuperclass()	Class 对象	获得该类继承的类
实现接口	getInterfaces()	Class 型数组	获得该类实现的所有接口
构造方法	getConstructors()	Constructor 型数组	获得所有权限为 public 的构造方法
	getConstructor(Class<?>…parameterTypes)	Constructor 对象	获得权限为 public 的指定构造方法
	getDeclaredConstructors()	Constructor 型数组	获得所有构造方法，按声明顺序返回
	getDeclaredConstructor(Class<?>… parameterTypes)	Constructor 对象	获得指定构造方法
方法	getMethods()	Method 型数组	获得所有权限为 public 的方法
	getMethod(String name, Class<?>… parameterTypes)	Method 对象	获得权限为 public 的指定方法
	getDeclaredMethods()	Method 型数组	获得所有方法，按声明顺序返回
	getDeclaredMethod(String name, Class<?>… parameterTypes)	Method 对象	获得指定方法
成员变量	getFields()	Field 型数组	获得所有权限为 public 的成员变量
	getField(String name)	Field 对象	获得权限为 public 的指定成员变量
	getDeclaredFields()	Field 型数组	获得所有成员变量，按声明顺序返回
	getDeclaredField(String name)	Field 对象	获得指定成员变量
内部类	getClasses()	Class 型数组	获得所有权限为 public 的内部类
	getDeclaredClasses()	Class 型数组	获得所有内部类
内部类的声明类	getDeclaringClass()	Class 对象	如果该类为内部类，则返回它的成员类，否则返回 null

说明

在通过getFields()和getMethods()方法依次获得权限为public的成员变量和方法时，将包含从超类中继承到的成员变量和方法；而通过方法getDeclaredFields()和getDeclaredMethods()只是获得在本类中定义的所有成员变量和方法。

16.1.1 访问构造方法

视频讲解：光盘\TM\lx\16\访问构造方法.exe

在通过下列一组方法访问构造方法时，将返回 Constructor 类型的对象或数组。每个 Constructor 对象代表一个构造方法，利用 Constructor 对象可以操纵相应的构造方法。

- ☑ getConstructors()。
- ☑ getConstructor(Class<?>…parameterTypes)。
- ☑ getDeclaredConstructors()。
- ☑ getDeclaredConstructor(Class<?>…parameterTypes)。

如果是访问指定的构造方法，需要根据该构造方法的入口参数的类型来访问。例如，访问一个入口参数类型依次为 String 和 int 型的构造方法，通过下面两种方式均可实现。

```
objectClass.getDeclaredConstructor(String.class, int.class);
objectClass.getDeclaredConstructor(new Class[] { String.class, int.class });
```

Constructor 类中提供的常用方法如表 16.2 所示。

表 16.2 Constructor 类的常用方法

方　法	说　明
isVarArgs()	查看该构造方法是否允许带有可变数量的参数，如果允许则返回 true，否则返回 false
getParameterTypes()	按照声明顺序以 Class 数组的形式获得该构造方法的各个参数的类型
getExceptionTypes()	以 Class 数组的形式获得该构造方法可能抛出的异常类型
newInstance(Object…initargs)	通过该构造方法利用指定参数创建一个该类的对象，如果未设置参数则表示采用默认无参数的构造方法
setAccessible(boolean flag)	如果该构造方法的权限为 private，默认为不允许通过反射利用 newInstance(Object…initargs)方法创建对象。如果先执行该方法，并将入口参数设为 true，则允许创建
getModifiers()	获得可以解析出该构造方法所采用修饰符的整数

通过 java.lang.reflect.Modifier 类可以解析出 getModifiers()方法的返回值所表示的修饰符信息，在该类中提供了一系列用来解析的静态方法，既可以查看是否被指定的修饰符修饰，还可以以字符串的形式获得所有修饰符。该类常用静态方法如表 16.3 所示。

表 16.3 Modifier 类中的常用解析方法

静 态 方 法	说　明
isPublic(int mod)	查看是否被 public 修饰符修饰，如果是则返回 true，否则返回 false
isProtected(int mod)	查看是否被 protected 修饰符修饰，如果是则返回 true，否则返回 false
isPrivate(int mod)	查看是否被 private 修饰符修饰，如果是则返回 true，否则返回 false

续表

静 态 方 法	说　明
isStatic(int mod)	查看是否被 static 修饰符修饰，如果是则返回 true，否则返回 false
isFinal(int mod)	查看是否被 final 修饰符修饰，如果是则返回 true，否则返回 false
toString(int mod)	以字符串的形式返回所有修饰符

例如，判断对象 constructor 所代表的构造方法是否被 private 修饰，以及以字符串形式获得该构造方法的所有修饰符的典型代码如下：

```
int modifiers = constructor.getModifiers();
boolean isEmbellishByPrivate = Modifier.isPrivate(modifiers);
String embellishment = Modifier.toString(modifiers);
```

【例 16.1】 访问构造方法。（实例位置：光盘\TM\sl\16.01）

首先创建一个 Example_01 类，在该类中声明一个 String 型成员变量和 3 个 int 型成员变量，并提供 3 个构造方法。具体代码如下：

```
public class Example_01 {
    String s;
    int i, i2, i3;
    private Example_01() {
    }
    protected Example_01(String s, int i) {
        this.s = s;
        this.i = i;
    }
    public Example_01(String... strings) throws NumberFormatException {
        if (0 < strings.length)
            i = Integer.valueOf(strings[0]);
        if (1 < strings.length)
            i2 = Integer.valueOf(strings[1]);
        if (2 < strings.length)
            i3 = Integer.valueOf(strings[2]);
    }
    public void print() {
        System.out.println("s=" + s);
        System.out.println("i=" + i);
        System.out.println("i2=" + i2);
        System.out.println("i3=" + i3);
    }
}
```

然后编写测试类 Main_01，在该类中通过反射访问 Example_01 类中的所有构造方法，并将该构造方法是否允许带有可变数量的参数、入口参数类型和可能抛出的异常类型信息输出到控制台。关键代码如下：

```
public class Main_01 {
    public static void main(String[] args) {
```

```
        Example_01 example = new Example_01("10", "20", "30");
        Class<? extends Example_01> exampleC = example.getClass();
        //获得所有构造方法
        Constructor[] declaredConstructors = exampleC.getDeclaredConstructors();
        for (int i = 0; i < declaredConstructors.length; i++) {                    //遍历构造方法
            Constructor<?> constructor = declaredConstructors[i];
            System.out.println("查看是否允许带有可变数量的参数：" + constructor.isVarArgs());
            System.out.println("该构造方法的入口参数类型依次为：");
            Class[] parameterTypes = constructor.getParameterTypes();    //获取所有参数类型
            for (int j = 0; j < parameterTypes.length; j++) {
                System.out.println(" " + parameterTypes[j]);
            }
            System.out.println("该构造方法可能抛出的异常类型为：");
            //获得所有可能抛出的异常信息类型
            Class[] exceptionTypes = constructor.getExceptionTypes();
            for (int j = 0; j < exceptionTypes.length; j++) {
                System.out.println(" " + exceptionTypes[j]);
            }
            Example_01 example2 = null;
            while (example2 == null) {
                try {//如果该成员变量的访问权限为 private，则抛出异常，即不允许访问
                    if (i == 2)//通过执行默认没有参数的构造方法创建对象
                        example2 = (Example_01) constructor.newInstance();
                    else if (i == 1)
                        //通过执行具有两个参数的构造方法创建对象
                        example2 = (Example_01) constructor.newInstance("7", 5);
                    else {//通过执行具有可变数量参数的构造方法创建对象
                        Object[] parameters = new Object[] { new String[] { "100", "200", "300" } };
                        example2 = (Example_01) constructor.newInstance(parameters);
                    }
                } catch (Exception e) {
                    System.out.println("在创建对象时抛出异常，下面执行 setAccessible()方法");
                    constructor.setAccessible(true);// 设置为允许访问
                }
            }
            if (example2 != null) {
                example2.print();
                System.out.println();
            }
        }

    }
}
```

运行本实例，当通过反射访问构造方法 Example_01()时，将输出如图 16.1 所示的信息；当通过反射访问构造方法 Example_01(String s, int i)时，将输出如图 16.2 所示的信息；当通过反射访问构造方法 Example_01(String…strings)时，将输出如图 16.3 所示的信息。

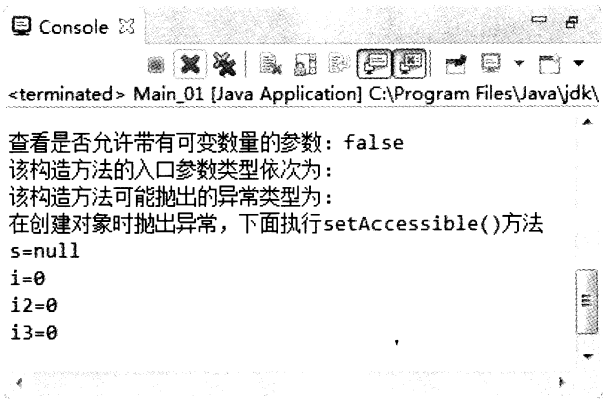


图 16.1　访问 Example_01()输出的信息

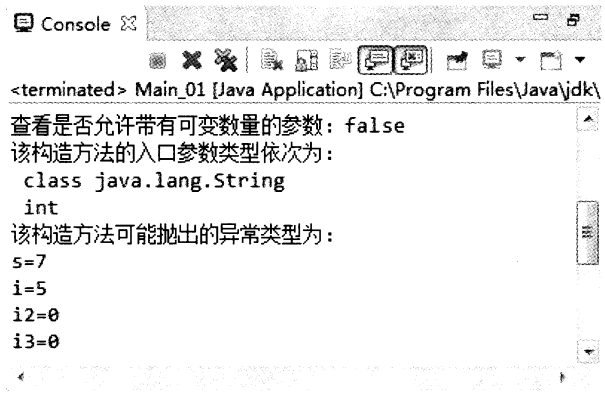


图 16.2　访问 Example_01(String s, int i)输出的信息

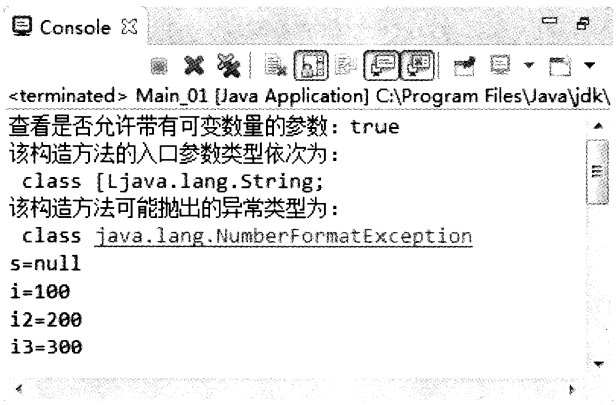


图 16.3　访问 Example_01(String…strings)输出的信息

16.1.2　访问成员变量

视频讲解：光盘\TM\lx\16\访问成员变量.exe

在通过下列一组方法访问成员变量时，将返回 Field 类型的对象或数组。每个 Field 对象代表一个

成员变量，利用 Field 对象可以操纵相应的成员变量。

- ☑ getFields()。
- ☑ getField(String name)。
- ☑ getDeclaredFields()。
- ☑ getDeclaredField(String name)。

如果是访问指定的成员变量，可以通过该成员变量的名称来访问。例如，访问一个名称为 birthday 的成员变量，访问方法如下：

```
object. getDeclaredField("birthday");
```

Field 类中提供的常用方法如表 16.4 所示。

表 16.4 Field 类的常用方法

方　法	说　明
getName()	获得该成员变量的名称
getType()	获得表示该成员变量类型的 Class 对象
get(Object obj)	获得指定对象 obj 中成员变量的值，返回值为 Object 型
set(Object obj, Object value)	将指定对象 obj 中成员变量的值设置为 value
getInt(Object obj)	获得指定对象 obj 中类型为 int 的成员变量的值
setInt(Object obj, int i)	将指定对象 obj 中类型为 int 的成员变量的值设置为 i
getFloat(Object obj)	获得指定对象 obj 中类型为 float 的成员变量的值
setFloat(Object obj, float f)	将指定对象 obj 中类型为 float 的成员变量的值设置为 f
getBoolean(Object obj)	获得指定对象 obj 中类型为 boolean 的成员变量的值
setBoolean(Object obj, boolean z)	将指定对象 obj 中类型为 boolean 的成员变量的值设置为 z
setAccessible(boolean flag)	此方法可以设置是否忽略权限限制直接访问 private 等私有权限的成员变量
getModifiers()	获得可以解析出该成员变量所采用修饰符的整数

【例 16.2】 访问成员变量。（**实例位置：光盘\TM\sl\16.02**）

首先创建一个 Example_02 类，在该类中依次声明一个 int、float、boolean 和 String 型的成员变量，并将它们设置为不同的访问权限。具体代码如下：

```
public class Example_02 {
    int i;
    public float f;
    protected boolean b;
    private String s;
}
```

然后通过反射访问 Example_02 类中的所有成员变量，将成员变量的名称和类型信息输出到控制台，并分别将各个成员变量在修改前后的值输出到控制台。关键代码如下：

```
import java.lang.reflect.Field;
public class Main_02 {
    public static void main(String[] args) {
        Example_02 example = new Example_02();
        Class exampleC = example.getClass();
```

```
        //获得所有成员变量
        Field[] declaredFields = exampleC.getDeclaredFields();
        for (int i = 0; i < declaredFields.length; i++) {
            Field field = declaredFields[i];                    //遍历成员变量
            //获得成员变量名称
            System.out.println("名称为：" + field.getName());
            Class fieldType = field.getType();                  //获得成员变量类型
            System.out.println("类型为：" + fieldType);
            boolean isTurn = true;
            while (isTurn) {
                //如果该成员变量的访问权限为 private，则抛出异常，即不允许访问
                try {
                    isTurn = false;
                    //获得成员变量值
                    System.out.println("修改前的值为：" + field.get(example));
                    //判断成员变量的类型是否为 int 型
                    if (fieldType.equals(int.class)) {
                        System.out.println("利用方法 setInt()修改成员变量的值");
                        field.setInt(example, 168);       //为 int 型成员变量赋值
                        //判断成员变量的类型是否为 float 型
                    } else if (fieldType.equals(float.class)) {
                      System.out.println("利用方法 setFloat()修改成员变量的值");
                        //为 float 型成员变量赋值
                        field.setFloat(example, 99.9F);
                        //判断成员变量的类型是否为 boolean 型
                    } else if (fieldType.equals(boolean.class)) {
                     System.out.println("利用方法 setBoolean()修改成员变量的值");
                        //为 boolean 型成员变量赋值
                        field.setBoolean(example, true);
                    } else {
                        System.out.println("利用方法 set()修改成员变量的值");
                        //可以为各种类型的成员变量赋值
                        field.set(example, "MWQ");
                    }
                    //获得成员变量值
                    System.out.println("修改后的值为：" + field.get(example));
                } catch (Exception e) {
                    System.out.println("在设置成员变量值时抛出异常，"
                            + "下面执行 setAccessible()方法！");
                    field.setAccessible(true); //设置为允许访问
                    isTurn = true;
                }
            }
            System.out.println();
        }
    }
}
```

运行本例，在控制台将输出如图 16.4 所示的信息，会发现在访问权限为 private 的成员变量 s 时，需要执行 setAccessible()方法，并将入口参数设为 true，否则不允许访问。

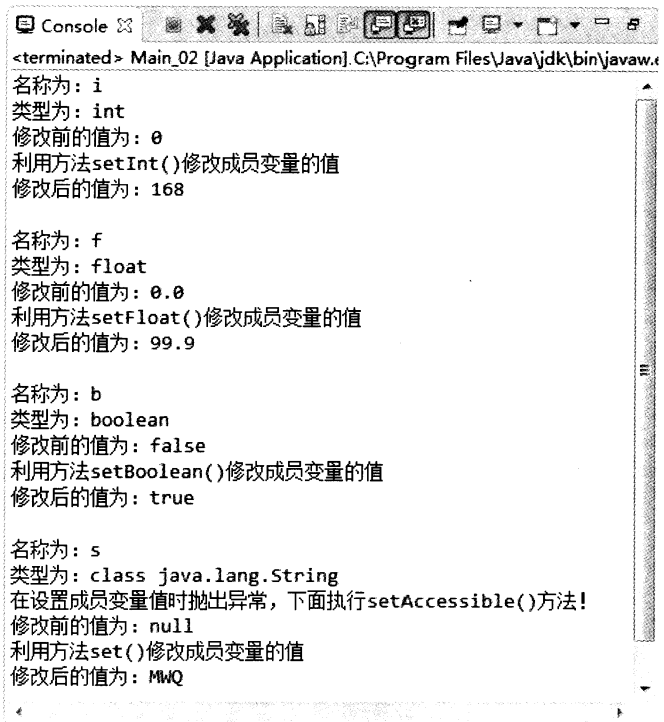


图 16.4　通过反射访问成员变量

16.1.3　访问方法

视频讲解：光盘\TM\lx\16\访问方法.exe

在通过下列一组方法访问方法时，将返回 Method 类型的对象或数组。每个 Method 对象代表一个方法，利用 Method 对象可以操纵相应的方法。

- ☑ getMethods()。
- ☑ getMethod(String name, Class<?>…parameterTypes)。
- ☑ getDeclaredMethods()。
- ☑ getDeclaredMethod(String name, Class<?>…parameterTypes)。

如果是访问指定的方法，需要根据该方法的名称和入口参数的类型来访问。例如，访问一个名称为 print、入口参数类型依次为 String 和 int 型的方法，通过下面两种方式均可实现：

- ☑ objectClass.getDeclaredMethod("print", String.class, int.class);
- ☑ objectClass.getDeclaredMethod("print", new Class[] {String.class, int.class });

Method 类中提供的常用方法如表 16.5 所示。

表 16.5　Method 类的常用方法

方　法	说　明
getName()	获得该方法的名称
getParameterTypes()	按照声明顺序以 Class 数组的形式获得该方法的各个参数的类型
getReturnType()	以 Class 对象的形式获得该方法的返回值的类型
getExceptionTypes()	以 Class 数组的形式获得该方法可能抛出的异常类型
invoke(Object obj, Object…args)	利用指定参数 args 执行指定对象 obj 中的该方法，返回值为 Object 型
isVarArgs()	查看该构造方法是否允许带有可变数量的参数，如果允许则返回 true，否则返回 false
getModifiers()	获得可以解析出该方法所采用修饰符的整数

【例 16.3】 访问方法。（实例位置：光盘\TM\sl\16.03）

首先创建一个 Example_03 类，并编写 4 个典型的方法。具体代码如下：

```
public class Example_03 {
    static void staticMethod() {
        System.out.println("执行 staticMethod()方法");
    }
    public int publicMethod(int i) {
        System.out.println("执行 publicMethod()方法");
        return i * 100;
    }
    protected int protectedMethod(String s, int i)
            throws NumberFormatException {
        System.out.println("执行 protectedMethod()方法");
        return Integer.valueOf(s) + i;
    }
    private String privateMethod(String... strings) {
        System.out.println("执行 privateMethod()方法");
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < strings.length; i++) {
            stringBuffer.append(strings[i]);
        }
        return stringBuffer.toString();
    }
}
```

然后通过反射访问 Example_03 类中的所有方法，将各个方法的名称、入口参数类型、返回值类型等信息输出到控制台，并执行部分方法。关键代码如下：

```
//获得所有方法
Method[] declaredMethods = exampleC.getDeclaredMethods();
for (int i = 0; i < declaredMethods.length; i++) {
    Method method = declaredMethods[i];                    //遍历方法
    System.out.println("名称为：" + method.getName());      //获得方法名称
    System.out.println("是否允许带有可变数量的参数：" + method.isVarArgs());
    System.out.println("入口参数类型依次为：");
    //获得所有参数类型
    Class[] parameterTypes = method.getParameterTypes();
    for (int j = 0; j < parameterTypes.length; j++) {
```

```
            System.out.println(" " + parameterTypes[j]);
        }
        //获得方法返回值类型
        System.out.println("返回值类型为：" + method.getReturnType());
        System.out.println("可能抛出的异常类型有：");
        //获得方法可能抛出的所有异常类型
        Class[] exceptionTypes = method.getExceptionTypes();
        for (int j = 0; j < exceptionTypes.length; j++) {
            System.out.println(" " + exceptionTypes[j]);
        }
        boolean isTurn = true;
        while (isTurn) {
            //如果该方法的访问权限为 private，则抛出异常，即不允许访问
            try {
                isTurn = false;
                if("staticMethod".equals(method.getName()))
                  method.invoke(example);                          //执行没有入口参数的方法
                else if("publicMethod".equals(method.getName()))
                  System.out.println("返回值为："
                          + method.invoke(example, 168));     //执行方法
                else if("protectedMethod".equals(method.getName()))
                  System.out.println("返回值为："
                          + method.invoke(example, "7", 5));  //执行方法
                else if("privateMethod".equals(method.getName())) {
                    Object[] parameters = new Object[] { new String[] {
                            "M", "W", "Q" } };                     //定义二维数组
                    System.out.println("返回值为："
                            + method.invoke(example, parameters));
                }
            } catch (Exception e) {
                System.out.println("在执行方法时抛出异常，"
                        + "下面执行 setAccessible()方法！");
                method.setAccessible(true);                        //设置为允许访问
                isTurn = true;
            }
        }
        System.out.println();
    }
```

注意

在反射中执行具有可变数量的参数的构造方法时，需要将入口参数定义成二维数组。

运行本实例，将依次访问方法 staticMethod()、publicMethod()、protectedMethod()和 privateMethod()，输出到控制台的信息依次如图 16.5、图 16.6、图 16.7 和图 16.8 所示。

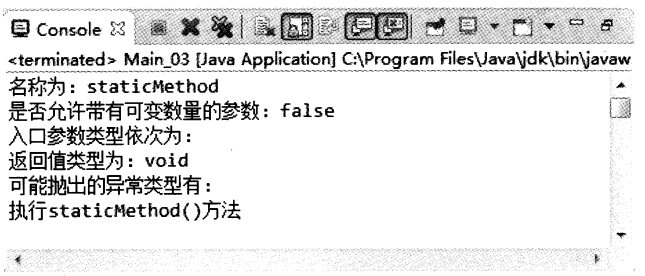


图 16.5　访问 staticMethod()方法输出的信息

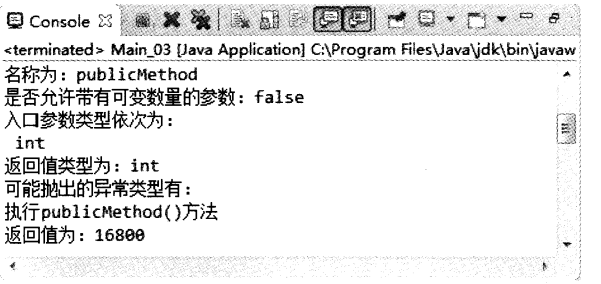


图 16.6　访问 publicMethod()方法输出的信息

图 16.7　访问 protectedMethod()方法输出的信息

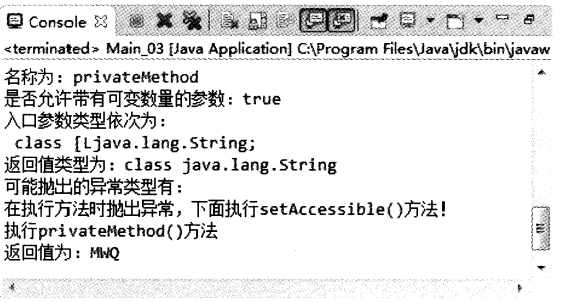


图 16.8　访问 privateMethod()方法输出的信息

16.2　使用 Annotation 功能

Java 中提供了 Annotation 功能，该功能可用于类、构造方法、成员变量、方法、参数等的声明中。该功能并不影响程序的运行，但是会对编译器警告等辅助工具产生影响。本节将介绍 Annotation 功能的使用方法。

16.2.1　定义 Annotation 类型

视频讲解：光盘\TM\lx\16\定义 Annotation 类型.exe

在定义 Annotation 类型时，也需要用到用来定义接口的 interface 关键字，但需要在 interface 关键字前加一个“@”符号，即定义 Annotation 类型的关键字为@interface，这个关键字的隐含意思是继承了 java.lang.annotation.Annotation 接口。例如，下面的代码就定义了一个 Annotation 类型。

```
public @interface NoMemberAnnotation {
}
```

上面定义的 Annotation 类型@NoMemberAnnotation 未包含任何成员，这样的 Annotation 类型被称为 marker annotation。下面的代码定义了一个只包含一个成员的 Annotation 类型。

```
public @interface OneMemberAnnotation {
```

```
    String value();
}
```

☑ String：成员类型。可用的成员类型有 String、Class、primitive、enumerated 和 annotation，以及所列类型的数组。

☑ value：成员名称。如果在所定义的 Annotation 类型中只包含一个成员，通常将成员名称命名为 value。

下面的代码定义了一个包含多个成员的 Annotation 类型。

```
public @interface MoreMemberAnnotation {
    String describe();
    Class type();
}
```

在为 Annotation 类型定义成员时，也可以为成员设置默认值。例如，下面的代码在定义 Annotation 类型时就为成员设置了默认值。

```
public @interface DefaultValueAnnotation {
    String describe() default "<默认值>";
    Class type() default void.class;
}
```

在定义 Annotation 类型时，还可以通过 Annotation 类型@Target 来设置 Annotation 类型适用的程序元素种类。如果未设置@Target，则表示适用于所有程序元素。枚举类 ElementType 中的枚举常量用来设置@Targer，如表 16.6 所示。

表 16.6　枚举类 ElementType 中的枚举常量

枚 举 常 量	说　明
ANNOTATION_TYPE	表示用于 Annotation 类型
TYPE	表示用于类、接口和枚举，以及 Annotation 类型
CONSTRUCTOR	表示用于构造方法
FIELD	表示用于成员变量和枚举常量
METHOD	表示用于方法
PARAMETER	表示用于参数
LOCAL_VARIABLE	表示用于局部变量
PACKAGE	表示用于包

通过 Annotation 类型@Retention 可以设置 Annotation 的有效范围。枚举类 RetentionPolicy 中的枚举常量用来设置@Retention，如表 16.7 所示。如果未设置@Retention，Annotation 的有效范围为枚举常量 CLASS 表示的范围。

表 16.7　枚举类 RetentionPolicy 中的枚举常量

枚 举 常 量	说　明
SOURCE	表示不编译 Annotation 到类文件中，有效范围最小
CLASS	表示编译 Annotation 到类文件中，但是在运行时不加载 Annotation 到 JVM 中
RUNTIME	表示在运行时加载 Annotation 到 JVM 中，有效范围最大

【例 16.4】 定义并使用 Annotation 类型。（实例位置：光盘\TM\sl\16.04）

首先定义一个用来注释构造方法的 Annotation 类型@Constructor_Annotation，有效范围为在运行时加载 Annotation 到 JVM 中。完整代码如下：

```
import java.lang.annotation.*;
@Target(ElementType.CONSTRUCTOR)
//用于构造方法
@Retention(RetentionPolicy.RUNTIME)
//在运行时加载 Annotation 到 JVM 中
public @interface Constructor_Annotation {
    String value() default "默认构造方法"; //定义一个具有默认值的 String 型成员
}
```

然后定义一个用来注释字段、方法和参数的 Annotation 类型@Field_Method_Parameter_Annotation，有效范围为在运行时加载 Annotation 到 JVM 中。完整代码如下：

```
import java.lang.annotation.*;
//用于字段、方法和参数
@Target( { ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
//在运行时加载 Annotation 到 JVM 中
public @interface Field_Method_Parameter_Annotation {
    String describe();                    //定义一个没有默认值的 String 型成员
    Class type() default void.class;      //定义一个具有默认值的 Class 型成员
}
```

最后编写一个 Record 类，在该类中运用前面定义的 Annotation 类型@Constructor_Annotation 和@Field_Method_Parameter_Annotation 对构造方法、字段、方法和参数进行注释。完整代码如下：

```
public class Record {
    @Field_Method_Parameter_Annotation(describe = "编号", type = int.class)
    //注释字段
    int id;
    @Field_Method_Parameter_Annotation(describe = "姓名", type = String.class)
    String name;
    @Constructor_Annotation()
    //采用默认值注释构造方法
    public Record() {
    }
    @Constructor_Annotation("立即初始化构造方法")
    public Record(//注释构造方法
        @Field_Method_Parameter_Annotation(describe = "编号",
                type = int.class) int id,
        @Field_Method_Parameter_Annotation(describe = "姓名",
                type = String.class) String name) {
        this.id = id;
        this.name = name;
    }
@Field_Method_Parameter_Annotation(describe = "获得编号",type=int.class)
```

```
    public int getId() {//注释方法
        return id;
    }
    @Field_Method_Parameter_Annotation(describe = "设置编号")
    public void setId(//成员 type 采用默认值注释方法
    //注释方法的参数
            @Field_Method_Parameter_Annotation(describe = "编号",
                    type = int.class) int id) {
        this.id = id;
    }
    @Field_Method_Parameter_Annotation(describe = "获得姓名",
            type = String.class)
    public String getName() {
        return name;
    }
    @Field_Method_Parameter_Annotation(describe = "设置姓名")
    public void setName(
            @Field_Method_Parameter_Annotation(describe = "姓名",
                    type = String.class) String name) {
        this.name = name;
    }
}
```

16.2.2 访问 Annotation 信息

视频讲解：光盘\TM\lx\16\访问 Annotation 信息.exe

如果在定义 Annotation 类型时将@Retention 设置为 RetentionPolicy.RUNTIME，那么在运行程序时通过反射就可以获取到相关的 Annotation 信息，如获取构造方法、字段和方法的 Annotation 信息。

类 Constructor、Field 和 Method 均继承了 AccessibleObject 类，在 AccessibleObject 中定义了 3 个关于 Annotation 的方法，其中方法 isAnnotationPresent(Class<? extends Annotation> annotationClass)用来查看是否添加了指定类型的 Annotation，如果是则返回 true，否则返回 false；方法 getAnnotation(Class<T> annotationClass)用来获得指定类型的 Annotation，如果存在则返回相应的对象，否则返回 null；方法 getAnnotations()用来获得所有的 Annotation，该方法将返回一个 Annotation 数组。

在类 Constructor 和 Method 中还定义了方法 getParameterAnnotations()，用来获得为所有参数添加的 Annotation，将以 Annotation 类型的二维数组返回，在数组中的顺序与声明的顺序相同，如果没有参数则返回一个长度为 0 的数组；如果存在未添加 Annotation 的参数，将用一个长度为 0 的嵌套数组占位。

【例 16.5】 访问 Annotation 信息。（**实例位置：光盘\TM\sl\16.05**）

本例将对 16.2.1 节中的例 16.4 进行扩展，实现在程序运行时通过反射访问 Record 类中的 Annotation 信息。首先编写访问构造方法及其包含参数的 Annotation 信息的代码。完整代码如下：

```
Constructor[] declaredConstructors = recordC
        .getDeclaredConstructors();                    //获得所有构造方法
for (int i = 0; i < declaredConstructors.length; i++) {
    Constructor constructor = declaredConstructors[i];      //遍历构造方法
```

```
    //查看是否具有指定类型的注释
    if (constructor
            .isAnnotationPresent(Constructor_Annotation.class)) {
        //获得指定类型的注释
        Constructor_Annotation ca = (Constructor_Annotation) constructor
                .getAnnotation(Constructor_Annotation.class);
        System.out.println(ca.value());                    //获得注释信息
    }
    Annotation[][] parameterAnnotations = constructor
            .getParameterAnnotations();                    //获得参数的注释
    for (int j = 0; j < parameterAnnotations.length; j++) {
        //获得指定参数注释的长度
        int length = parameterAnnotations[j].length;
        if (length == 0)                                   //如果长度为 0，则表示没有为该参数添加注释
            System.out.println("      未添加 Annotation 的参数");
        else
            for (int k = 0; k < length; k++) {
                //获得参数的注释
                Field_Method_Parameter_Annotation pa =
                    (Field_Method_Parameter_Annotation)
                    parameterAnnotations[j][k];
                System.out.print("      " + pa.describe()); //获得参数描述
                System.out.println("      " + pa.type());   //获得参数类型
            }
    }
    System.out.println();
}
```

然后编写访问字段的 Annotation 信息的代码。完整代码如下：

```
Field[] declaredFields = recordC.getDeclaredFields();      //获得所有字段
for (int i = 0; i < declaredFields.length; i++) {
    Field field = declaredFields[i];                       //遍历字段
    //查看是否具有指定类型的注释
    if (field.isAnnotationPresent(
                    Field_Method_Parameter_Annotation.class)) {
        //获得指定类型的注释
        Field_Method_Parameter_Annotation fa = field
                .getAnnotation(Field_Method_Parameter_Annotation.class);
        System.out.print("      " + fa.describe());      //获得字段的描述
        System.out.println("      " + fa.type());        //获得字段的类型
    }
}
```

最后编写访问方法及其包含参数的 Annotation 信息的代码。完整代码如下：

```
Method[] methods = recordC.getDeclaredMethods();           //获得所有方法
for (int i = 0; i < methods.length; i++) {
    Method method = methods[i];                            //遍历方法
    //查看是否具有指定类型的注释
    if (method
```

```
            .isAnnotationPresent(Field_Method_Parameter_Annotation.class)) {
        //获得指定类型的注释
        Field_Method_Parameter_Annotation ma = method
                .getAnnotation(Field_Method_Parameter_Annotation.class);
        System.out.println(ma.describe());                    //获得方法的描述
        System.out.println(ma.type());                        //获得方法的返回值类型
    }
    Annotation[][] parameterAnnotations = method
            .getParameterAnnotations();                       //获得参数的注释
    for (int j = 0; j < parameterAnnotations.length; j++) {
        int length = parameterAnnotations[j].length;          //获得指定参数注释的长度
        if (length == 0)                                      //如果长度为 0，表示没有为该参数添加注释
            System.out.println("      未添加 Annotation 的参数");
        else
            for (int k = 0; k < length; k++) {
                //获得指定类型的注释
                Field_Method_Parameter_Annotation pa =
                    (Field_Method_Parameter_Annotation)
                    parameterAnnotations[j][k];
                System.out.print("      " + pa.describe());  //获得参数的描述
                System.out.println("      " + pa.type());     //获得参数的类型
            }
    }
    System.out.println();
}
```

运行本实例，当执行第一段测试代码时，控制台将输出如图 16.9 所示的信息；当执行第二段测试代码时，控制台将输出如图 16.10 所示的信息；当执行最后一段测试代码时，控制台将输出如图 16.11 所示的信息。

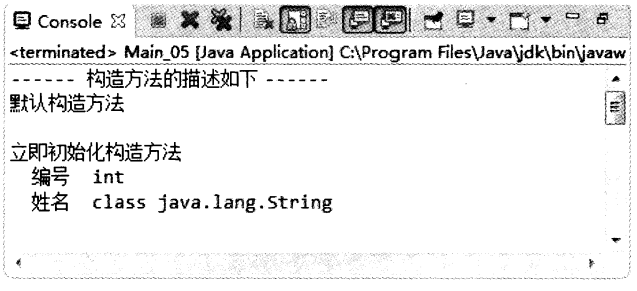


图 16.9　访问构造方法的 Annotation 信息

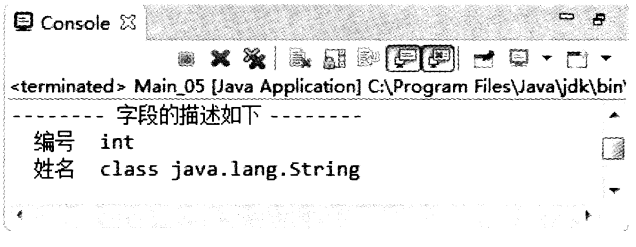


图 16.10　访问字段的 Annotation 信息

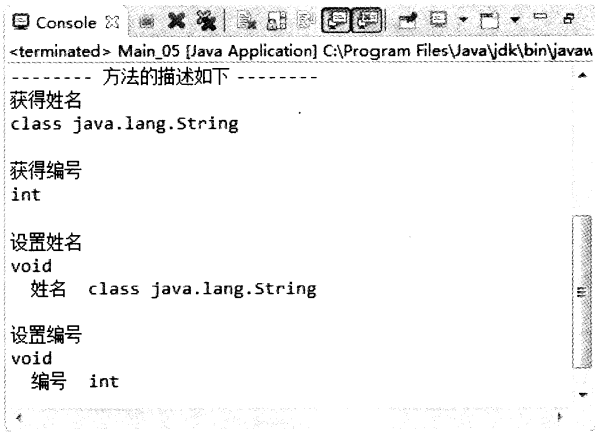


图 16.11　访问方法的 Annotation 信息

16.3　小　　结

通过对本章的学习，相信读者已经掌握了 Java 反射机制的使用方法。利用 Java 反射机制，可以在程序运行时访问类的所有描述信息（经常需要访问的有类的构造方法、成员变量和方法），实现逆向控制程序的执行过程。利用 Annotation 功能，可以对类、构造方法、成员变量、方法、参数等进行注释，在程序运行时通过反射可以读取这些信息，根据读取的信息也可以实现逆向控制程序的执行过程。

16.4　实践与练习

1．利用反射实现通用扩展数组长度的方法。（**答案位置：光盘\TM\sl\16.06**）

2．利用反射初步验证用户输入的信息。（**答案位置：光盘\TM\sl\16.07**）

第17章

枚举类型与泛型

（视频讲解：20分钟）

枚举类型可以取代以往常量的定义方式，即将常量封装在类或接口中，此外，它还提供了安全检查功能。枚举类型本质上还是以类的形式存在。泛型的出现不仅可以让程序员少写某些代码，主要的作用是解决类型安全问题，它提供编译时的安全检查，不会因为将对象置于某个容器中而失去其类型。本章将着重讲解枚举类型与泛型。

通过阅读本章，您可以：

- 掌握枚举类型
- 掌握泛型

17.1　枚 举 类 型

使用枚举类型可以取代以往定义常量的方式，同时枚举类型还赋予程序在编译时进行检查的功能。本节就来详细介绍枚举类型。

17.1.1　使用枚举类型设置常量

视频讲解：光盘\TM\lx\17\使用枚举类型设置常量.exe

以往设置常量，通常将常量放置在接口中，这样在程序中就可以直接使用，并且该常量不能被修改，因为在接口中定义常量时，该常量的修饰符为 final 与 static。常规定义常量的代码如例 17.1 所示。

【例 17.1】 在项目中创建 Constants 接口，在接口中定义常量的常规方式。

```
public interface Constants {
    public static final int Constants_A=1;
    public static final int Constants_B=12;
}
```

枚举类型出现后，逐渐取代了这种常量定义方式。使用枚举类型定义常量的语法如下：

```
public enum Constants{
    Constants_A,
    Constants_B,
    Constants_C
}
```

其中，enum 是定义枚举类型关键字。当需要在程序中使用该常量时，可以使用 Constants.Constants_A 来表示。

下面举例介绍枚举类型定义常量的方式。

【例 17.2】 在项目中创建 Constants 接口，在该接口中定义两个整型变量，其修饰符都是 static 和 final；之后定义名称为 Constants2 的枚举类，将 Constants 接口的常量放置在该枚举类中；最后，创建名称为 Constants 的类文件，在该类中通过 doit()和 doit2()进行不同方式的调用，然后再通过主方法进行调用，体现枚举类型定义常量的方式。（**实例位置：光盘\TM\sl\17.01**）

```
interface Constants {                                    //将常量放置在接口中
   public static final int Constants_A = 1;
   public static final int Constants_B = 12;
}
public class ConstantsTest {
   enum Constants2 {                                     //将常量放置在枚举类型中
        Constants_A, Constants_B
   }
```

```
    //使用接口定义常量
    public static void doit(int c) {                    //定义一个方法，这里的参数为 int 型
        switch (c) {                                    //根据常量的值做不同操作
            case Constants.Constants_A:
                System.out.println("doit() Constants_A");
                break;
            case Constants.Constants_B:
                System.out.println("doit() Constants_B");
                break;
        }
    }
    public static void doit2(Constants2 c) {            //定义一个参数对象是枚举类型的方法
        switch (c) {                                    //根据枚举类型对象做不同操作
            case Constants_A:
                System.out.println("doit2() Constants_A");
                break;
            case Constants_B:
                System.out.println("doit2() Constants_B");
                break;
        }
    }
    public static void main(String[] args) {
        ConstantsTest.doit(Constants.Constants_A);      //使用接口中定义的常量
        ConstantsTest.doit2(Constants2.Constants_A);    //使用枚举类型中的常量
        ConstantsTest.doit2(Constants2.Constants_B);    //使用枚举类型中的常量
        ConstantsTest.doit(3);
        //ConstantsTest.doit2(3);
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 17.1 所示。

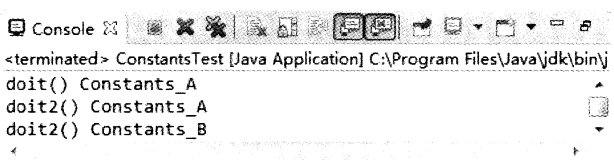


图 17.1　使用枚举类型定义常量

在上述代码中，当用户调用 doit()方法时，即使编译器不接受在接口中定义的常量参数，也不会报错；但调用 doit2()方法，任意传递参数，编译器就会报错，因为这个方法只接受枚举类型的常量作为其参数。

枚举类型也可以在类的内部进行定义，下面将介绍如何在类的内部进行枚举类型的定义。

【例 17.3】 在项目中创建 ConstantsTest 类，该类中以内部类的形式定义枚举类型。

```
public class ConstantsTest {
    enum Constants2{ //将常量放置在枚举类型中
        Constants_A,
        Constants_B
```

```
    }
...
}
```

这种形式类似于内部类形式，当编译该类时，除了 ConstantsTest.class 外，还存在 ConstantsTest$1.class 与 ConstantsTest$Constants2.class 文件。

17.1.2　深入了解枚举类型

视频讲解：光盘\TM\lx\17\深入了解枚举类型.exe

1．操作枚举类型成员的方法

枚举类型较传统定义常量的方式，除了具有参数类型检测的优势之外，还具有其他方面的优势。

用户可以将一个枚举类型看作是一个类，它继承于 java.lang.Enum 类，当定义一个枚举类型时，每一个枚举类型成员都可以看作是枚举类型的一个实例，这些枚举类型成员都默认被 final、public、static 修饰，所以当使用枚举类型成员时直接使用枚举类型名称调用枚举类型成员即可。

由于枚举类型对象继承于 java.lang.Enum 类，所以该类中一些操作枚举类型的方法都可以应用到枚举类型中。表 17.1 中列举了枚举类型中的常用方法。

表 17.1　枚举类型的常用方法

方 法 名 称	具 体 含 义	使 用 方 法	举　例
values()	该方法可以将枚举类型成员以数组的形式返回	枚举类型名称.values()	Constants2.values()
valueOf()	该方法可以实现将普通字符串转换为枚举实例	枚举类型名称.valueOf("abc")	Constants2.valueOf("abc")
compareTo()	该方法用于比较两个枚举对象在定义时的顺序	枚举对象.compareTo()	Constants_A.compareTo(Constants_B)
ordinal()	该方法用于得到枚举成员的位置索引	枚举对象.ordinal()	Constants_A.ordinal()

（1）values()

枚举类型实例包含一个 values()方法，该方法将枚举类型的成员变量实例以数组的形式返回，也可以通过该方法获取枚举类型的成员。

【例 17.4】 在项目中创建 ShowEnum 类，在该类中使用枚举类型中的 values()方法获取枚举类型中的成员变量。（**实例位置：光盘\TM\sl\17.02**）

```
import static java.lang.System.out;
public class ShowEnum {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B
    }
    //循环由 values()方法返回的数组
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
```

```
            //将枚举成员变量打印
            out.println("枚举类型成员变量：" + Constants2.values()[i]);
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.2 所示。

在例 17.4 中，由于 values()方法将枚举类型的成员以数组的形式返回，所以根据该数组的长度进行循环操作，然后将该数组中的值返回。

（2）valueOf()与 compareTo()

枚举类型中静态方法 valueOf()可以将普通字符串转换为枚举类型，而 compareTo()方法用于比较两个枚举类型对象定义时的顺序。

【例 17.5】 在项目中创建 EnumMethodTest 类，在该类中使用枚举类型中的 valueOf()与 compareTo()方法。（**实例位置：光盘\TM\sl\17.03**）

```
import static java.lang.System.out;
public class EnumMethodTest {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B
    }
    //定义比较枚举类型方法，参数类型为枚举类型
    public static void compare(Constants2 c) {
        //根据 values()方法返回的数组做循环操作
        for (int i = 0; i < Constants2.values().length; i++) {
            //将比较结果返回
            out.println(c + "与" + Constants2.values()[i] + "的比较结果为："
                    + c.compareTo(Constants2.values()[i]));
        }
    }
    //在主方法中调用 compare()方法
    public static void main(String[] args) {
        compare(Constants2.valueOf("Constants_B"));
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.3 所示。

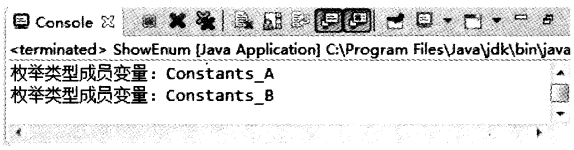


图 17.2　使用枚举类型中的 values()方法获取枚举类型中的成员变量

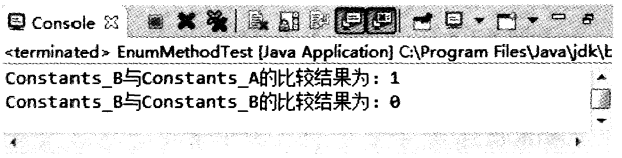


图 17.3　使用 compareTo()方法比较两个枚举类型成员定义的顺序

调用 compareTo()方法返回的结果，正值代表方法中参数在调用该方法的枚举对象位置之前；0 代表两个互相比较的枚举成员的位置相同；负值代表方法中参数在调用该方法的枚举对象位置之后。

（3）ordinal()

枚举类型中的 ordinal()方法用于获取某个枚举对象的位置索引值。

【例 17.6】 在项目中创建 EnumIndexTest 类，在该类中使用枚举类型中的 ordinal()方法获取枚举类型成员的位置索引。（**实例位置：光盘\TM\sl\17.04**）

```
import static java.lang.System.out;
public class EnumIndexTest {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B, Constants_C
    }
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
            //在循环中获取枚举类型成员的索引位置
            out.println(Constants2.values()[i] + "在枚举类型中位置索引值"
                    + Constants2.values()[i].ordinal());
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.4 所示。

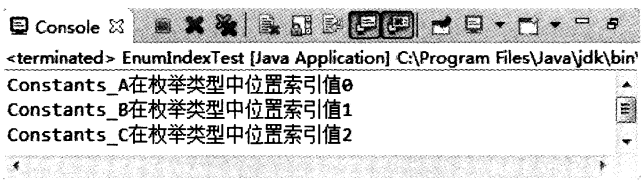


图 17.4　获取枚举类型成员的位置索引

在例 17.6 中，在循环中获取每个枚举对象时，调用 ordinal()方法即可相应获取该枚举类型成员的索引位置。

2．枚举类型中的构造方法

在枚举类型中，可以添加构造方法，但是规定这个构造方法必须为 private 修饰符所修饰。枚举类型定义的构造方法语法如下：

```
enum 枚举类型名称{
    Constants_A("我是枚举成员 A"),
    Constants_B("我是枚举成员 B"),
    Constants_C("我是枚举成员 C"),
    Constants_D(3);
    private String description;
    private Constants2(){            //定义默认构造方法
    }
    //定义带参数的构造方法，参数类型为字符串型
    private Constants2(String description) {
        this.description=description;
    }
    private Constants2(int i){       //定义带参数的构造方法，参数类型为整型
```

```
        this.i=this.i+i;
    }
}
```

从枚举类型构造方法的语法中可以看出，无论是无参构造方法还是有参构造方法，修饰权限都为 private。定义一个有参构造方法后，需要对枚举类型成员相应地使用该构造方法，如 Constants_A("我是枚举成员 A")和 Constants_D(3)语句，相应地使用了参数为 String 型和参数为 int 型的构造方法。然后可以在枚举类型中定义两个成员变量，在构造方法中为这两个成员变量赋值，这样就可以在枚举类型中定义该成员变量的 getXXX()方法了。

下面是在枚举类型中定义构造方法的实例。

【例 17.7】 在项目中创建 EnumIndexTest 类，在该类中定义枚举类型的构造方法。（**实例位置：光盘\TM\sl\17.05**）

```
import static java.lang.System.out;
public class EnumIndexTest {
    enum Constants2 {                                    //将常量放置在枚举类型中
        Constants_A("我是枚举成员 A"),                    //定义带参数的枚举类型成员
        Constants_B("我是枚举成员 B"),
        Constants_C("我是枚举成员 C"),
        Constants_D(3);
        private String description;
        private int i = 4;
        private Constants2() {
        }
         //定义参数为 String 型的构造方法
        private Constants2(String description) {
            this.description = description;
        }
        private Constants2(int i) {                      //定义参数为 int 型的构造方法
            this.i = this.i + i;
        }
        public String getDescription() {                 //获取 description 的值
            return description;
        }
        public int getI() {                              //获取 i 的值
            return i;
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
            out.println(Constants2.values()[i]+"调用 getDescription()方法为："
                    + Constants2.values()[i].getDescription());
        }
        out.println(Constants2.valueOf("Constants_D") + "调用 getI()方法为："
                + Constants2.valueOf("Constants_D").getI());
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.5 所示。

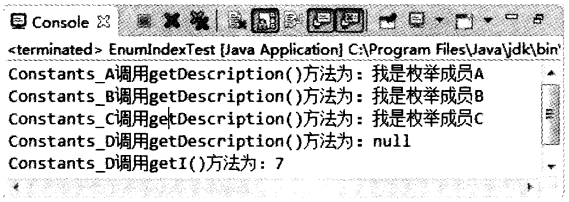


图 17.5　在枚举类型中定义构造方法

在本实例中，调用 getDescription()和 getI()方法，返回在枚举类型定义的构造方法中设置的操作。这里将枚举类型中的构造方法设置为 private 修饰，以防止客户代码实例化一个枚举对象。

除了可以使用例 17.7 中所示的方式定义 getDescription()方法获取枚举类型成员定义时的描述之外，还可以将这个 getDescription()方法放置在接口中，使枚举类型实现该接口，然后使每个枚举类型实现接口中的方法。

【例 17.8】 在项目中创建 d 接口和枚举类型的 AnyEnum 类，在枚举类型 AnyEnum 类中实现带方法的接口，使每个枚举类型成员实现该接口中的方法。（**实例位置：光盘\TM\sl\17.06**）

```
import static java.lang.System.out;
interface d {
    public String getDescription();
    public int getI();
}
public enum AnyEnum implements d {
    Constants_A { //可以在枚举类型成员内部设置方法
        public String getDescription() {
            return ("我是枚举成员 A");
        }
        public int getI() {
            return i;
        }
    },
    Constants_B {
        public String getDescription() {
            return ("我是枚举成员 B");
        }
        public int getI() {
            return i;
        }
    },
    Constants_C {
        public String getDescription() {
            return ("我是枚举成员 C");
        }
        public int getI() {
            return i;
        }
    },
    Constants_D {
        public String getDescription() {
            return ("我是枚举成员 D");
```

```
        }
        public int getI() {
            return i;
        }
    };
    private static int i = 5;
    public static void main(String[] args) {
        for (int i = 0; i < AnyEnum.values().length; i++) {
            out.println(AnyEnum.values()[i] + "调用 getDescription()方法为："
                    + AnyEnum.values()[i].getDescription());
            out.println(AnyEnum.values()[i] + "调用 getI()方法为："
                    + AnyEnum.values()[i].getI());
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.6 所示。

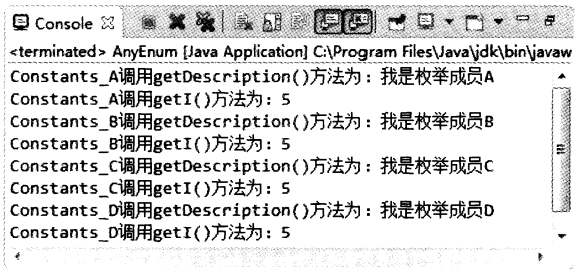


图 17.6　在每个枚举类型成员中实现接口中的方法

17.1.3　使用枚举类型的优势

视频讲解：光盘\TM\lx\17\使用枚举类型的优势.exe

枚举类型声明提供了一种用户友好的变量定义方法，枚举了某种数据类型所有可能出现的值。总结枚举类型，它具有以下特点：

- ☑ 类型安全。
- ☑ 紧凑有效的数据定义。
- ☑ 可以和程序其他部分完美交互。
- ☑ 运行效率高。

17.2　泛　　型

泛型实质上就是使程序员定义安全的类型。在没有出现泛型之前，Java 也提供了对 Object 的引用“任意化”操作，这种“任意化”操作就是对 Object 引用进行向下转型及向上转型操作，但某些强制

类型转换的错误也许不会被编译器捕捉，而在运行后出现异常，可见强制类型转换存在安全隐患，所以在此提供了泛型机制。本节就来探讨泛型机制。

17.2.1　回顾向上转型与向下转型

视频讲解：光盘\TM\lx\17\回顾向上转型与向下转型.exe

在介绍泛型之前，先来看一个例子。

【例 17.9】 在项目中创建 Test 类，在该类中使基本类型向上转型为 Object 类型。（**实例位置：光盘\TM\sl\17.07**）

```
public class Test {
    private Object b;                          //定义 Object 类型成员变量
    public Object getB() {                     //设置相应的 getXXX()方法
        return b;
    }
    public void setB(Object b) {               //设置相应的 setXXX()方法
        this.b = b;
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.setB(new Boolean(true));             //向上转型操作
        System.out.println(t.getB());
        t.setB(new Float(12.3));
        Float f = (Float) (t.getB());          //向下转型操作
        System.out.println(f);
    }
}
```

运行本实例，结果如图 17.7 所示。

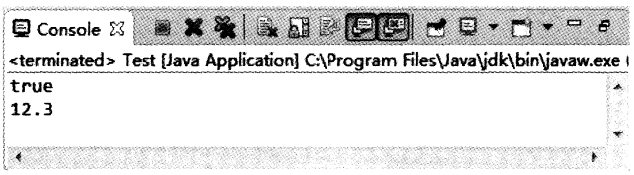


图 17.7　使基本类型向上转型为 Object 类型

在本实例中，Test 类中定义了私有的成员变量 b，它的类型为 Object 类型，同时为其定义了相应的 setXXX()与 getXXX()方法。在类主方法中，将 new Boolean(true)对象作为 setB()方法的参数，由于 setB()方法的参数类型为 Object，这样就实现了向上转型操作。同时在调用 getB()方法时，将 getB()方法返回的 Object 对象以相应的类型返回，这个就是向下转型操作，问题通常就会出现在这里。因为向上转型是安全的，而如果进行向下转型操作时用错了类型，或者并没有执行该操作，就会出现异常,例如以下代码：

```
t.setB(new Float(12.3));
Integer f=(Integer)(t.getB());
System.out.println(f);
```

并不存在语法错误，所以可以被编译器接受，但在执行时会出现 ClassCastException 异常。这样看来，向下转型操作通常会出现问题，而泛型机制有效地解决了这一问题。

17.2.2 定义泛型类

视频讲解：光盘\TM\lx\17\定义泛型类.exe

Object 类为最上层的父类，很多程序员为了使程序更为通用，设计程序时通常使传入的值与返回的值都以 Object 类型为主。当需要使用这些实例时，必须正确地将该实例转换为原来的类型，否则在运行时将会发生 ClassCastException 异常。

在 JDK 1.5 版本以后，提出了泛型机制。其语法如下：

```
类名<T>
```

其中，T 代表一个类型的名称。

如果将例 17.9 改写为定义类时使用泛型的形式，关键代码如例 17.10 所示。

【例 17.10】 在项目中创建 OverClass 类，该类定义了泛型类。

```
public class OverClass<T> {                    //定义泛型类
    private T over;                            //定义泛型成员变量
    public T getOver() {                       //设置 getXXX()方法
        return over;
    }
    public void setOver(T over) {              //设置 setXXX()方法
        this.over = over;
    }
    public static void main(String[] args) {
        //实例化一个 Boolean 型的对象
        OverClass<Boolean> over1 = new OverClass<Boolean>();
        //实例化一个 Float 型的对象
        OverClass<Float> over2 = new OverClass<Float>();
        over1.setOver(true);                   //不需要进行类型转换
        over2.setOver(12.3f);
        Boolean b = over1.getOver();           //不需要进行类型转换
        Float f = over2.getOver();
        System.out.println(b);
        System.out.println(f);
    }
}
```

运行上述代码，结果与图 17.7 所示的结果一致。在例 17.10 中定义类时，在类名后添加了一个<T>语句，这里便使用了泛型机制。可以将 OverClass 类称为泛型类，同时返回和接受的参数使用 T 这个类型。最后在主方法中可以使用 Over<Boolean>形式返回一个 Boolean 型的对象，使用 OverClass<Float>形式返回一个 Float 型的对象，使这两个对象分别调用 setOver()方法不需要进行显式向上转型操作，setOver()方法直接接受相应类型的参数，而调用 getOver()方法时，不需要进行向下转型操作，直接将 getOver()方法返回的值赋予相应的类型变量即可。

从例 17.10 中可以看出，使用泛型定义的类在声明该类对象时可以根据不同的需求指定<T>真正的类型，而在使用类中的方法传递或返回数据类型时将不再需要进行类型转换操作，而是使用在声明泛型类对象时“<>”符号中设置的数据类型。

使用泛型这种形式将不会发生 ClassCastException 异常，因为在编译器中就可以检查类型匹配是否正确。

【例 17.11】 在项目中定义泛型类。

```
OverClass<Float> over2=new OverClass<Float>();
over2.setOver(12.3f);
//Integer i=over2.getOver(); //不能将 boolean 型的值赋予 Integer 变量
```

在例 17.11 中，由于 over2 对象在实例化时已经指定类型为 Float，而最后一条语句却将该对象获取出的 Float 类型值赋予 Integer 类型，所以编译器会报错。而如果使用向下转型操作，就会在运行上述代码时发生异常。

说明

在定义泛型类时，一般类型名称使用 T 来表达，而容器的元素使用 E 来表达，具体的设置读者可以参看 JDK 5.0 以上版本的 API。

17.2.3　泛型的常规用法

视频讲解：光盘\TM\lx\17\泛型的常规用法.exe

1．定义泛型类时声明多个类型

在定义泛型类时，可以声明多个类型。语法如下：

```
MutiOverClass<T1,T2>
MutiOverClass:泛型类名称
```

其中，T1 和 T2 为可能被定义的类型。

这样在实例化指定类型的对象时就可以指定多个类型。例如：

```
MutiOverClass<Boolean,Float>=new MutiOverClass<Boolean,Float>();
```

2．定义泛型类时声明数组类型

定义泛型类时也可以声明数组类型，下面的实例中定义泛型时便声明了数组类型。

【例 17.12】 在项目中创建 ArrayClass 类，在该类中定义泛型类声明数组类型。（**实例位置：光盘\TM\sl\17.08**）

```
public class ArrayClass<T> {
    private T[] array;                          //定义泛型数组
    public void SetT(T[] array) {               //设置 SetXXX()方法为成员数组赋值
        this.array = array;
    }
```

```
    public T[] getT() {                          //获取成员数组
        return array;
    }
    public static void main(String[] args) {
        ArrayClass<String> a = new ArrayClass<String>();
        String[] array = { "成员 1", "成员 2", "成员 3", "成员 4", "成员 5" };
        a.SetT(array);                           //调用 SetT()方法
        for (int i = 0; i < a.getT().length; i++) {
            System.out.println(a.getT()[i]);  //调用 getT()方法返回数组中的值
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.8 所示。

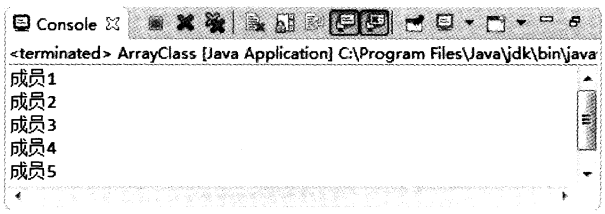


图 17.8　定义泛型类时声明数组类型

本实例在定义泛型类时声明一个成员数组，数组的类型为泛型，然后在泛型类中相应设置 setXXX()与 getXXX()方法。

可见，可以在使用泛型机制时声明一个数组，但是不可以使用泛型来建立数组的实例。例如，下面的代码就是错误的：

```
public class ArrayClass <T>{
    //private T[] array=new T[10]; //不能使用泛型来建立数组的实例
    ...
}
```

3．集合类声明容器的元素

可以使用 K 和 V 两个字符代表容器中的键值和与键值相对应的具体值。

【例 17.13】 在项目中创建 MutiOverClass 类，在该类中使用集合类声明容器的元素。（**实例位置：光盘\TM\sl\17.09**）

```
import java.util.HashMap;
import java.util.Map;
public class MutiOverClass<K, V> {
    public Map<K, V> m = new HashMap<K, V>();        //定义一个集合 HashMap 实例
     //设置 put()方法，将对应的键值与键名存入集合对象中
    public void put(K k, V v) {
        m.put(k, v);
    }
    public V get(K k) {                              //根据键名获取键值
```

```
        return m.get(k);
    }
    public static void main(String[] args) {
        //实例化泛型类对象
        MutiOverClass<Integer, String> mu
        = new MutiOverClass<Integer, String>();
        for (int i = 0; i < 5; i++) {
            //根据集合的长度循环将键名与具体值放入集合中
            mu.put(i, "我是集合成员" + i);
        }
        for (int i = 0; i < mu.m.size(); i++) {
            //调用 get()方法获取集合中的值
            System.out.println(mu.get(i));
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.9 所示。

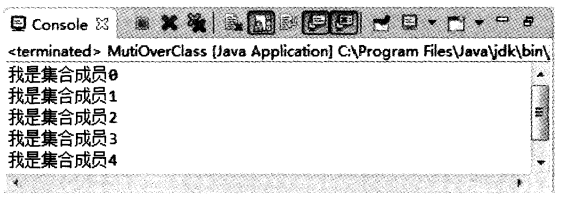


图 17.9　集合类声明容器的元素

其实在例 17.13 中定义的泛型类 MutiOverClass 纯属多余，因为在 Java 中这些集合框架已经都被泛型化了，可以在主方法中直接使用“public Map<K,V> m=new HashMap<K,V>();”语句创建实例，然后相应调用 Map 接口中的 put()与 get()方法完成填充容器或根据键名获取集合中具体值的功能。集合中除了 HashMap 这种集合类型之外，还包括 ArrayList、Vector 等。表 17.2 列举了几个常用的被泛型化的集合类。

表 17.2　常用的被泛型化的集合类

集　合　类	泛 型 定 义
ArrayList	ArrayList<E>
HashMap	HashMap<K,V>
HashSet	HashSet<E>
Vector	Vector<E>

下面的实例演示了这些集合的使用方式。

【例 17.14】 在项目中创建 AnyClass 类，在该类中使用泛型实例化常用集合类。（**实例位置：光盘\TM\sl\17.10**）

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Vector;
public class AnyClass {
```

```
    public static void main(String[] args) {
        //定义 ArrayList 容器，设置容器内的值类型为 Integer
        ArrayList<Integer> a = new ArrayList<Integer>();
        a.add(1);                                    //为容器添加新值
        for (int i = 0; i < a.size(); i++) {
            //根据容器的长度循环显示容器内的值
            System.out.println("获取 ArrayList 容器的值：" + a.get(i));
        }
        //定义 HashMap 容器，设置容器的键名与键值类型分别为 Integer 与 String 型
        Map<Integer, String> m = new HashMap<Integer, String>();
        for (int i = 0; i < 5; i++) {
            m.put(i, "成员" + i);                    //为容器填充键名与键值
        }
        for (int i = 0; i < m.size(); i++) {
            //根据键名获取键值
            System.out.println("获取 Map 容器的值" + m.get(i));
        }
        //定义 Vector 容器，使容器中的内容为 String 型
        Vector<String> v = new Vector<String>();
        for (int i = 0; i < 5; i++) {
            v.addElement("成员" + i);            //为 Vector 容器添加内容
        }
        for (int i = 0; i < v.size(); i++) {
            //显示容器中的内容
            System.out.println("获取 Vector 容器的值" + v.get(i));
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.10 所示。

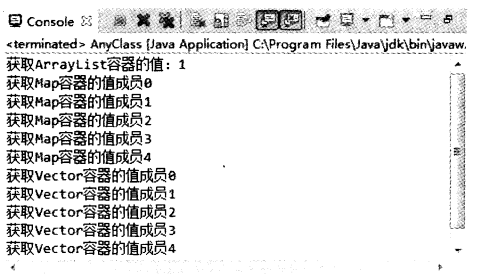


图 17.10　使用泛型实例化常用集合类

17.2.4　泛型的高级用法

视频讲解：光盘\TM\lx\17\泛型的高级用法.exe

泛型的高级用法包括限制泛型可用类型和使用类型通配符等。

1．限制泛型可用类型

默认可以使用任何类型来实例化一个泛型类对象，但 Java 中也对泛型类实例的类型作了限制。语法如下：

```
class 类名称<T extends anyClass>
```

其中，anyClass 指某个接口或类。

使用泛型限制后，泛型类的类型必须实现或继承了 anyClass 这个接口或类。无论 anyClass 是接口还是类，在进行泛型限制时都必须使用 extends 关键字。

【例 17.15】 在项目中创建 LimitClass 类，在该类中限制泛型类型。

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
public class LimitClass<T extends List> { //限制泛型的类型
    public static void main(String[] args) {
        //可以实例化已经实现 List 接口的类
        LimitClass<ArrayList> l1 = new LimitClass<ArrayList>();
        LimitClass<LinkedList> l2 = new LimitClass<LinkedList>();
        //这句是错误的，因为 HashMap 没有实现 List()接口
        //LimitClass<HashMap> l3=new LimitClass<HashMap>();
    }
}
```

在例 17.15 中，将泛型作了限制，设置泛型类型必须实现 List 接口。例如，ArrayList 和 LinkedList 都实现了 List 接口，而 HashMap 没有实现 List 接口，所以在这里不能实例化 HashMap 类型的泛型对象。

当没有使用 extends 关键字限制泛型类型时，默认 Object 类下的所有子类都可以实例化泛型类对象。如图 17.11 所示的两个语句是等价的。

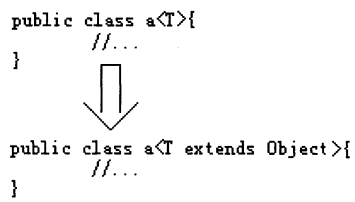


图 17.11　两个等价的泛型类

2．使用类型通配符

在泛型机制中，提供了类型通配符，其主要作用是在创建一个泛型类对象时限制这个泛型类的类型实现或继承某个接口或类的子类。要声明这样一个对象可以使用“?”通配符来表示，同时使用 extends 关键字来对泛型加以限制。

使用泛型类型通配符的语法如下：

```
泛型类名称<? extends List> a=null;
```

其中，<? extends List>表示类型未知，当需要使用该泛型对象时，可以单独实例化。

【例 17.16】 在项目中创建一个类文件，在该类中限制泛型类型。

```
A<? extends List> a=null;
a=new A<ArrayList>();
a=new A<LinkedList>();
```

如果实例化没有实现 List 接口的泛型对象，编译器将会报错。例如，实例化 HashMap 对象时，编

译器将会报错，因为 HashMap 类没有实现 List 接口。

除了可以实例化一个限制泛型类型的实例之外，还可以将该实例放置在方法的参数中。

【例 17.17】 在项目中创建一个类文件，在该类中的方法参数中使用匹配字符串。

```
public void doSomething(A<? extends List> a){
}
```

在上述代码中，定义方式有效地限制了传入 doSomething()方法的参数类型。

如果使用 A<?>这种形式实例化泛型类对象，则默认表示可以将 A 指定为实例化 Object 及以下的子类类型。读者可能对这种编码类型有些疑惑，例 17.18 将直观地介绍 A<?>泛型机制。

【例 17.18】 在泛型中使用通配符形式。

```
List<String> l1=new ArrayList<String>();        //实例化一个 ArrayList 对象
l1.add("成员");                                  //在集合中添加内容
List<?> l2=l1;                                   //使用通配符
List<?> l3=new LinkedList<Integer>();
System.out.println(l2.get(0));                   //获取集合中第一个值
```

在例 17.18 中，List<?>类型的对象可以接受 String 类型的 ArrayList 集合，也可以接受 Integer 类型的 LinkedList 集合。也许有的读者会有疑问，List<?> l2=l1 语句与 List l2=l1 存在何种本质区别？这里需要注意的是，使用通配符声明的名称实例化的对象不能对其加入新的信息，只能获取或删除。例如：

```
l1.set(0, "成员改变");                           //没有使用通配符的对象调用 set()方法
//l2.set(0, "成员改变");                         //使用通配符的对象调用 set()方法，不能被调用
//l3.set(0, 1);
l2.get(0);                                       //可以使用 l2 的实例获取集合中的值
l2.remove(0);                                    //根据键名删除集合中的值
```

从上述代码中可以看出，由于对象 l1 是没有使用 A<?>这种形式初始化出来的对象，所以它可以调用 set()方法改变集合中的值，但 l2 与 l3 则是通过使用通配符的方式创建出来的，所以不能改变集合中的值。

技巧

泛型类型限制除了可以向下限制之外，还可以进行向上限制，只要在定义时使用 super 关键字即可。例如，“A<? super List> a=null;”这样定义后，对象 a 只接受 List 接口或上层父类类型，如“a=new A<Object>();”。

3. 继承泛型类与实现泛型接口

定义为泛型的类和接口也可以被继承与实现。

【例 17.19】 在项目中创建一个类文件，在该类中继承泛型类。

```
public class ExtendClass<T1>{
}
class SubClass<T1,T2,T3> extends ExtendClass<T1>{
}
```

如果在 SubClass 类继承 ExtendClass 类时保留父类的泛型类型，需要在继承时指明，如果没有指明，直接使用 extends ExtendsClass 语句进行继承操作，则 SubClass 类中的 T1、T2 和 T3 都会自动变为 Object，所以在一般情况下都将父类的泛型类型保留。

定义的泛型接口也可以被实现。

【例 17.20】 在项目中创建一个类文件，在该类中实现泛型接口。

```
interface i<T1>{
}
class SubClass2<T1,T2,T3> implements i<T1>{
}
```

17.2.5 泛型总结

视频讲解：光盘\TM\lx\17\泛型总结.exe

下面总结一下泛型的使用方法。

☑ 泛型的类型参数只能是类类型，不可以是简单类型，如 A<int>这种泛型定义就是错误的。

☑ 泛型的类型个数可以是多个。

☑ 可以使用 extends 关键字限制泛型的类型。

☑ 可以使用通配符限制泛型的类型。

17.3 小　　结

本章主要讲述了枚举类型以及泛型的用法。虽然枚举类型与泛型的语法比较简单，但是展开后的写法比较复杂，所以初学者应该仔细揣摩，并且对这两种机制做到简单掌握。此外，读者应该积极了解每个 JDK 版本新增的内容，而查看相应版本的 API 便是一种极为有效的手段。

17.4 实践与练习

1．尝试定义一个枚举类型类，使用 switch 语句获取枚举类型的值。（**答案位置：光盘\TM\sl\17.11**）

2．尝试定义一个泛型类，使用 extends 关键字限制该泛型类的类型为 List 接口，并分别创建两个泛型对象。（**答案位置：光盘\TM\sl\17.12**）

3．尝试定义一个泛型类，并使用通配符。（**答案位置：光盘\TM\sl\17.13**）

第18章

多线程

（视频讲解：21分钟）

如果一次只完成一件事情，会很容易实现，但现实生活中很多事情都是同时进行的，所以在 Java 中为了模拟这种状态，引入了线程机制。简单地说，当程序同时完成多件事情时，就是所谓的多线程程序。多线程应用相当广泛，使用多线程可以创建窗口程序、网络程序等。本章将由浅入深地介绍多线程，除了介绍其概念之外，还结合实例让读者了解如何使程序具有多线程功能。

通过阅读本章，您可以：

- 了解线程
- 掌握实现线程的两种方式
- 掌握线程的生命周期
- 掌握线程的操作方法
- 掌握线程的优先级
- 掌握线程同步机制

18.1　线 程 简 介

视频讲解：光盘\TM\lx\18\线程简介.exe

世间万物都可以同时完成很多工作，例如，人体可以同时进行呼吸、血液循环、思考问题等活动，用户既可以使用计算机听歌，也可以使用它打印文件，而这些活动完全可以同时进行，这种思想放在 Java 中被称为并发，而将并发完成的每一件事情称为线程。

在 Java 中，并发机制非常重要，但并不是所有的程序语言都支持线程。在以往的程序中，多以一个任务完成后再进行下一个项目的模式进行开发，这样下一个任务的开始必须等待前一个任务的结束。Java 语言提供了并发机制，程序员可以在程序中执行多个线程，每一个线程完成一个功能，并与其他线程并发执行，这种机制被称为多线程。

多线程是非常复杂的机制，比如同时阅读 3 本书，首先阅读第 1 本书第 1 章，然后再阅读第 2 本书第 1 章，再阅读第 3 本书第 1 章，回过头再阅读第 1 本书第 2 章，依此类推，就体现了多线程的复杂性。

既然多线程这样复杂，那么它在操作系统中是怎样工作的呢？其实 Java 中的多线程在每个操作系统中的运行方式也存在差异，在此着重说明多线程在 Windows 操作系统中的运行模式。Windows 操作系统是多任务操作系统，它以进程为单位。一个进程是一个包含有自身地址的程序，每个独立执行的程序都称为进程，也就是正在执行的程序。系统可以分配给每个进程一段有限的使用 CPU 的时间（也可以称为 CPU 时间片），CPU 在这段时间中执行某个进程，然后下一个时间片又跳至另一个进程中去执行。由于 CPU 转换较快，所以使得每个进程好像是同时执行一样。

图 18.1 表明了 Windows 操作系统的执行模式。

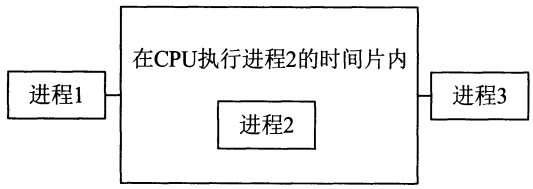


图 18.1　Windows 操作系统的执行模式

一个线程则是进程中的执行流程，一个进程中可以同时包括多个线程，每个线程也可以得到一小段程序的执行时间，这样一个进程就可以具有多个并发执行的线程。在单线程中，程序代码按调用顺序依次往下执行，如果需要一个进程同时完成多段代码的操作，就需要产生多线程。

18.2　实现线程的两种方式

在 Java 中主要提供两种方式实现线程，分别为继承 java.lang.Thread 类与实现 java.lang.Runnable 接口。本节将着重讲解这两种实现线程的方式。

18.2.1 继承 Thread 类

视频讲解：光盘\TM\lx\18\继承 Thread 类.exe

Thread 类是 java.lang 包中的一个类，从这个类中实例化的对象代表线程，程序员启动一个新线程需要建立 Thread 实例。Thread 类中常用的两个构造方法如下：

☑ public Thread()：创建一个新的线程对象。

☑ public Thread(String threadName)：创建一个名称为 threadName 的线程对象。

继承 Thread 类创建一个新的线程的语法如下：

```
public class ThreadTest extends Thread{
}
```

完成线程真正功能的代码放在类的 run()方法中，当一个类继承 Thread 类后，就可以在该类中覆盖 run()方法，将实现该线程功能的代码写入 run()方法中，然后同时调用 Thread 类中的 start()方法执行线程，也就是调用 run()方法。

Thread 对象需要一个任务来执行，任务是指线程在启动时执行的工作，该工作的功能代码被写在 run()方法中。run()方法必须使用以下语法格式：

```
public void run(){
}
```

注意

如果 start()方法调用一个已经启动的线程，系统将抛出 IllegalThreadStateException 异常。

当执行一个线程程序时，就自动产生一个线程，主方法正是在这个线程上运行的。当不再启动其他线程时，该程序就为单线程程序，如在本章以前的程序都是单线程程序。主方法线程启动由 Java 虚拟机负责，程序员负责启动自己的线程。

代码如下：

```
public static void main(String[] args) {
    new ThreadTest().start();
}
```

下面看一个继承 Thread 类的实例。

【例 18.1】 在项目中创建 ThreadTest 类，该类继承 Thread 类方法创建线程。（**实例位置：光盘\TM\sl\18.01**）

```
public class ThreadTest extends Thread {    //指定类继承 Thread 类
    private int count = 10;
    public void run() {                      //重写 run()方法
        while (true) {
            System.out.print(count+" ");     //打印 count 变量
```

```
            if (--count == 0) {                    //使 count 变量自减，当自减为 0 时，退出循环
                return;
            }
        }
    }
    public static void main(String[] args) {
        new ThreadTest().start();
    }
}
```

在 Eclipse 中运行本实例，结果如图 18.2 所示。

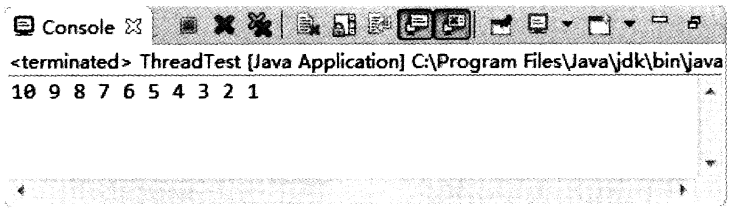


图 18.2　使用继承 Thread 类方法创建线程

在上述实例中，继承了 Thread 类，然后在类中覆盖了 run()方法。通常在 run()方法中使用无限循环的形式，使得线程一直运行下去，所以要指定一个跳出循环的条件，如本实例中使用变量 count 递减为 0 作为跳出循环的条件。

在 main 方法中，使线程执行需要调用 Thread 类中的 start()方法，start()方法调用被覆盖的 run()方法，如果不调用 start()方法，线程永远都不会启动，在主方法没有调用 start()方法之前，Thread 对象只是一个实例，而不是一个真正的线程。

18.2.2　实现 Runnable 接口

视频讲解：光盘\TM\lx\18\实现 Runnable 接口.exe

到目前为止，线程都是通过扩展 Thread 类来创建的，如果程序员需要继承其他类（非 Thread 类），而且还要使当前类实现多线程，那么可以通过 Runnable 接口来实现。例如，一个扩展 JFrame 类的 GUI 程序不可能再继承 Thread 类，因为 Java 语言中不支持多继承，这时该类就需要实现 Runnable 接口使其具有使用线程的功能。

实现 Runnable 接口的语法如下：

```
public class Thread extends Object implements Runnable
```

说明

有兴趣的读者可以查询 API，从中可以发现，实质上 Thread 类实现了 Runnable 接口，其中的 run()方法正是对 Runnable 接口中的 run()方法的具体实现。

实现Runnable接口的程序会创建一个Thread对象，并将Runnable对象与Thread对象相关联。Thread

类中有以下两个构造方法：

☑ public Thread(Runnable target)。

☑ public Thread(Runnable target,String name)。

这两个构造方法的参数中都存在 Runnable 实例，使用以上构造方法就可以将 Runnable 实例与 Thread 实例相关联。

使用 Runnable 接口启动新的线程的步骤如下：

（1）建立 Runnable 对象。

（2）使用参数为 Runnable 对象的构造方法创建 Thread 实例。

（3）调用 start()方法启动线程。

通过 Runnable 接口创建线程时程序员首先需要编写一个实现 Runnable 接口的类，然后实例化该类的对象，这样就建立了 Runnable 对象；接下来使用相应的构造方法创建 Thread 实例；最后使用该实例调用 Thread 类中的 start()方法启动线程。图 18.3 表明了实现 Runnable 接口创建线程的流程。

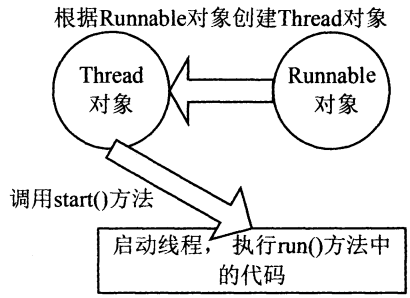


图 18.3 实现 Runnable 接口创建线程的流程

线程最引人注目的部分应该是与 Swing 相结合创建 GUI 程序，下面演示一个 GUI 程序，该程序实现了图标滚动的功能。

【例 18.2】 在项目中创建 SwingAndThread 类，该类继承了 JFrame 类，实现图标移动的功能，其中使用了 Swing 与线程相结合的技术。（**实例位置：光盘\TM\sl\18.02**）

```
import java.awt.Container;
import java.net.URL;
import javax.swing.*;

public class SwingAndThread extends JFrame {
    private JLabel jl = new JLabel();                          //声明 JLabel 对象
    private static Thread t;                                   //声明线程对象
    private int count = 0;                                     //声明计数变量
    private Container container = getContentPane();            //声明容器

    public SwingAndThread() {
        setBounds(300, 200, 250, 100);                         //绝对定位窗体大小与位置
        container.setLayout(null);                             //使窗体不使用任何布局管理器
        URL url = SwingAndThread.class.getResource("/1.gif");  //获取图片的 URL
        Icon icon = new ImageIcon(url);                        //实例化一个 Icon
        jl.setIcon(icon);                                      //将图标放置在标签中
         //设置图片在标签的最左方
        jl.setHorizontalAlignment(SwingConstants.LEFT);
        jl.setBounds(10, 10, 200, 50);                         //设置标签的位置与大小
        jl.setOpaque(true);
        t = new Thread(new Runnable() {                        //定义匿名内部类，该类实现 Runnable 接口
                public void run() {                            //重写 run()方法
                    while (count <= 200) {                     //设置循环条件
                        //将标签的横坐标用变量表示
```

```
                    jl.setBounds(count, 10, 200, 50);
                    try {
                        Thread.sleep(1000);                    //使线程休眠 1000 毫秒
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    count += 4;                                //使横坐标每次增加 4
                    if (count == 200) {
                        //当图标到达标签的最右边时，使其回到标签最左边
                        count = 10;
                    }
                }
            }
        });
        t.start();                                             //启动线程
        container.add(jl);                                     //将标签添加到容器中
        setVisible(true);                                      //使窗体可见
        //设置窗体的关闭方式
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    }
    public static void main(String[] args) {
        new SwingAndThread();  //实例化一个 SwingAndThread 对象
    }
}
```

运行本实例，结果如图 18.4 所示。

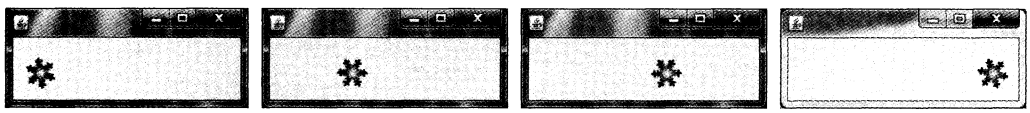


图 18.4　使图标移动

在本实例中，为了使图标具有滚动功能，需要在类的构造方法中创建 Thread 实例。在创建该实例的同时需要 Runnable 对象作为 Thread 类构造方法的参数，然后使用内部类形式实现 run()方法。在 run()方法中主要循环图标的横坐标位置，当图标横坐标到达标签的最右方时，再次将图标的横坐标置于图标滚动的初始位置。

注意

启动一个新的线程，不是直接调用 Thread 子类对象的 run()方法，而是调用 Thread 子类的 start()方法，Thread 类的 start()方法产生一个新的线程，该线程运行 Thread 子类的 run()方法。

18.3　线程的生命周期

视频讲解：光盘\TM\lx\18\线程的生命周期.exe

线程具有生命周期，其中包含 7 种状态，分别为出生状态、就绪状态、运行状态、等待状态、休眠状态、阻塞状态和死亡状态。出生状态就是线程被创建时处于的状态，在用户使用该线程实例调用

start()方法之前线程都处于出生状态；当用户调用 start()方法后，线程处于就绪状态（又被称为可执行状态）；当线程得到系统资源后就进入运行状态。

一旦线程进入可执行状态，它会在就绪与运行状态下转换，同时也有可能进入等待、休眠、阻塞或死亡状态。当处于运行状态下的线程调用 Thread 类中的 wait()方法时，该线程便进入等待状态，进入等待状态的线程必须调用 Thread 类中的 notify()方法才能被唤醒，而 notifyAll()方法是将所有处于等待状态下的线程唤醒；当线程调用 Thread 类中的 sleep()方法时，则会进入休眠状态。如果一个线程在运行状态下发出输入/输出请求，该线程将进入阻塞状态，在其等待输入/输出结束时线程进入就绪状态，对于阻塞的线程来说，即使系统资源空闲，线程依然不能回到运行状态。当线程的 run()方法执行完毕时，线程进入死亡状态。

说明

使线程处于不同状态下的方法会在 18.4 节中进行介绍，在此读者只需了解线程的多个状态即可。

图 18.5 描述了线程生命周期中的各种状态。

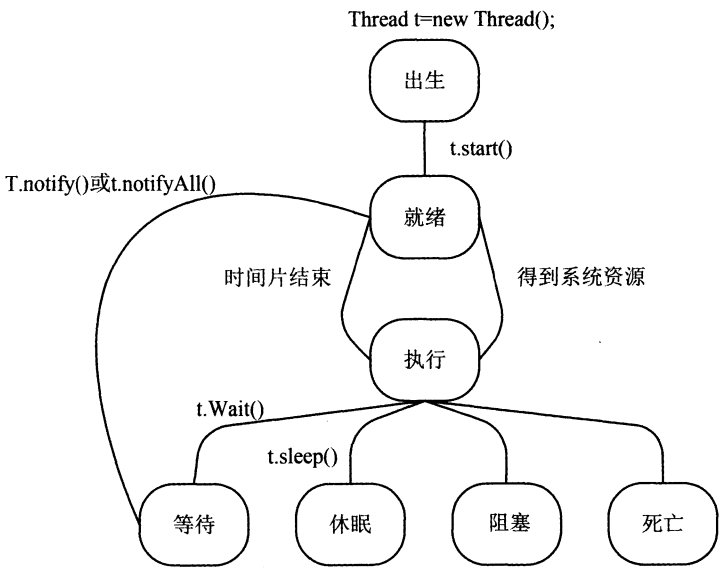


图 18.5　线程的生命周期状态图

虽然多线程看起来像同时执行，但事实上在同一时间点上只有一个线程被执行，只是线程之间切换较快，所以才会使人产生线程是同时进行的假象。在 Windows 操作系统中，系统会为每个线程分配一小段 CPU 时间片，一旦 CPU 时间片结束就会将当前线程换为下一个线程，即使该线程没有结束。

根据图 18.5 所示，可以总结出使线程处于就绪状态有以下几种方法：

- ☑ 调用 sleep()方法。
- ☑ 调用 wait()方法。
- ☑ 等待输入/输出完成。

当线程处于就绪状态后，可以用以下几种方法使线程再次进入运行状态。

☑ 线程调用 notify()方法。
☑ 线程调用 notifyAll()方法。
☑ 线程调用 interrupt()方法。
☑ 线程的休眠时间结束。
☑ 输入/输出结束。

图 18.5 中描述了线程的生命周期状态，下面将着重讲解使线程处于各种状态的方法。

18.4　操作线程的方法

操作线程有很多方法，这些方法可以使线程从某一种状态过渡到另一种状态。

18.4.1　线程的休眠

视频讲解：光盘\TM\lx\18\线程的休眠.exe

一种能控制线程行为的方法是调用 sleep()方法，sleep()方法需要一个参数用于指定该线程休眠的时间，该时间以毫秒为单位。在前面的实例中已经演示过 sleep()方法，它通常是在 run()方法内的循环中被使用。

sleep()方法的语法如下：

```
try{
    Thread.sleep(2000);
}catch(InterruptedException e){
    e.printStackTrace();
}
```

上述代码会使线程在 2 秒之内不会进入就绪状态。由于 sleep()方法的执行有可能抛出 InterruptedException 异常，所以将 sleep()方法的调用放在 try-catch 块中。虽然使用了 sleep()方法的线程在一段时间内会醒来，但是并不能保证它醒来后进入运行状态，只能保证它进入就绪状态。

为了使读者更深入地了解线程的休眠方法，来看下面的实例。

【例 18.3】 在项目中创建 SleepMethodTest 类，该类继承了 JFrame 类，实现在窗体中自动画线段的功能，并且为线段设置颜色，颜色是随机产生的。（**实例位置：光盘\TM\sl\18.03**）

```
import java.awt.*;
import java.util.Random;
import javax.swing.*;
public class SleepMethodTest extends JFrame {
    private Thread t;
    //定义颜色数组
    private static Color[] color = { Color.BLACK, Color.BLUE, Color.CYAN,
            Color.GREEN, Color.ORANGE, Color.YELLOW, Color.RED,
```

```
            Color.PINK, Color.LIGHT_GRAY };
    private static final Random rand = new Random();          //创建随机对象
    private static Color getC() {                             //获取随机颜色值的方法
        return color[rand.nextInt(color.length)];
    }
    public SleepMethodTest() {
        t = new Thread(new Runnable() {                       //创建匿名线程对象
            int x = 30;                                       //定义初始坐标
            int y = 50;
            public void run() {                               //覆盖线程接口方法
                while (true) {                                //无限循环
                    try {
                        Thread.sleep(100);                    //线程休眠 0.1 秒
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //获取组件绘图上下文对象
                    Graphics graphics = getGraphics();
                    graphics.setColor(getC());                //设置绘图颜色
                    //绘制直线并递增垂直坐标
                    graphics.drawLine(x, y, 100, y++);
                    if (y >= 80) {
                        y = 50;
                    }
                }
            }
        });
        t.start();                                            //启动线程
    }
    public static void main(String[] args) {
        init(new SleepMethodTest(), 100, 100);
    }
    //初始化程序界面的方法
    public static void init(JFrame frame, int width, int height) {
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(width, height);
        frame.setVisible(true);
    }
}
```

运行本实例，结果如图 18.6 所示。

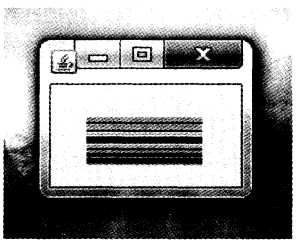


图 18.6　线程的休眠

在本实例中定义了 getC()方法，该方法用于随机产生 Color 类型的对象，并且在产生线程的匿名内部类中使用 getGraphics()方法获取 Graphics 对象，使用该对象调用 setColor()方法为图形设置颜色；调用 drawLine()方法绘制一条线段，同时线段会根据纵坐标的变化自动调整。

18.4.2　线程的加入

视频讲解：光盘\TM\lx\18\线程的加入.exe

如果当前某程序为多线程程序，假如存在一个线程 A，现在需要插入线程 B，并要求线程 B 先执行完毕，然后再继续执行线程 A，此时可以使用 Thread 类中的 join()方法来完成。这就好比此时读者正在看电视，突然有人上门收水费，读者必须付完水费后才能继续看电视。

当某个线程使用 join()方法加入到另外一个线程时，另一个线程会等待该线程执行完毕后再继续执行。

下面来看一个使用 join()方法的实例。

【例 18.4】 在项目中创建 JoinTest 类，该类继承了 JFrame 类。该实例包括两个进度条，进度条的进度由线程来控制，通过使用 join()方法使上面的进度条必须等待下面的进度条完成后才可以继续。**（实例位置：光盘\TM\sl\18.04）**

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JProgressBar;
public class JoinTest extends JFrame {
    private Thread threadA;                                          //定义两个线程
    private Thread threadB;
    final JProgressBar progressBar = new JProgressBar();             //定义两个进度条组件
    final JProgressBar progressBar2 = new JProgressBar();
    int count = 0;
    public static void main(String[] args) {
        init(new JoinTest(), 100, 100);
    }
    public JoinTest() {
        super();
        //将进度条设置在窗体最北面
        getContentPane().add(progressBar, BorderLayout.NORTH);
        //将进度条设置在窗体最南面
        getContentPane().add(progressBar2, BorderLayout.SOUTH);
        progressBar.setStringPainted(true);                          //设置进度条显示数字字符
        progressBar2.setStringPainted(true);
        //使用匿名内部类形式初始化 Thread 实例
        threadA = new Thread(new Runnable() {
            int count = 0;
            public void run() {                                      //重写 run()方法
                while (true) {
                    progressBar.setValue(++count);                   //设置进度条的当前值
                    try {
                        Thread.sleep(100);                           //使线程 A 休眠 100 毫秒
                        threadB.join();                              //使线程 B 调用 join()方法
```

```
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        threadA.start();                                    //启动线程 A
        threadB = new Thread(new Runnable() {
            int count = 0;
            public void run() {
                while (true) {
                    progressBar2.setValue(++count);         //设置进度条的当前值
                    try {
                        Thread.sleep(100);                  //使线程 B 休眠 100 毫秒
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    if (count == 100)                       //当 count 变量增长为 100 时
                        break;                              //跳出循环
                }
            }
        });
        threadB.start();                                    //启动线程 B
    }
    //设置窗体的各种属性的方法
    public static void init(JFrame frame, int width, int height) {
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(width, height);
        frame.setVisible(true);
    }
}
```

运行本实例，结果如图 18.7 所示。

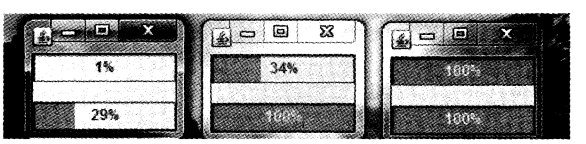


图 18.7　使用 join()方法控制进度条的滚动

在本实例中同时创建了两个线程，这两个线程分别负责进度条的滚动。在线程 A 的 run()方法中使线程 B 的对象调用 join()方法，而 join()方法使当前运行线程暂停，直到调用 join()方法的线程执行完毕后再执行，所以线程 A 等待线程 B 执行完毕后再开始执行，即下面的进度条滚动完毕后上面的进度条才开始滚动。

18.4.3　线程的中断

视频讲解：光盘\TM\lx\18\线程的中断.exe

以往有的时候会使用 stop()方法停止线程，但当前版本的 JDK 早已废除了 stop()方法，不建议使用

stop()方法来停止一个线程的运行。现在提倡在 run()方法中使用无限循环的形式，然后使用一个布尔型标记控制循环的停止。

【例 18.5】 在项目中创建 InterruptedTest 类，该类实现了 Runnable 接口，并设置线程正确的停止方式。

```
public class InterruptedTest implements Runnable {
    private boolean isContinue = false;        //设置一个标记变量，默认值为 false

    public void run() {                        //重写 run()方法
        while (true) {
            //…
            if (isContinue)                    //当 isContinue 变量为 true 时，停止线程
              break;
        }
    }

    public void setContinue() {                //定义设置 isContinue 变量为 true 的方法
        this.isContinue = true;
    }
}
```

如果线程是因为使用了 sleep()或 wait()方法进入了就绪状态，可以使用 Thread 类中 interrupt()方法使线程离开 run()方法，同时结束线程，但程序会抛出 InterruptedException 异常，用户可以在处理该异常时完成线程的中断业务处理，如终止 while 循环。

下面的实例演示了某个线程使用 interrupted()方法，同时程序抛出了 InterruptedException 异常，在异常处理时结束了 while 循环。在项目中，经常在这里执行关闭数据库连接和关闭 Socket 连接等操作。

【例 18.6】 在项目中创建 InterruptedSwing 类，该类实现了 Runnable 接口，创建一个进度条，在表示进度条的线程中使用 interrupted()方法。（**实例位置：光盘\TM\sl\18.05**）

```
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JProgressBar;
public class InterruptedSwing extends JFrame {
    Thread thread;
    public static void main(String[] args) {
        init(new InterruptedSwing(), 100, 100);
    }
    public InterruptedSwing() {
        super();
        final JProgressBar progressBar = new JProgressBar();   //创建进度条
        //将进度条放置在窗体合适位置
        getContentPane().add(progressBar, BorderLayout.NORTH);
        progressBar.setStringPainted(true);                    //设置进度条上显示数字
        thread = new Thread(new Runnable() {
            int count = 0;

            public void run() {
                while (true) {
                    progressBar.setValue(++count);             //设置进度条的当前值
```

```
                try {
                    thread.sleep(1000);              //使线程休眠 1000 毫秒
                    //捕捉 InterruptedException 异常
                } catch (InterruptedException e) {
                    System.out.println("当前线程程序被中断");
                    break;
                }
            }
        }
    });
    thread.start();                                  //启动线程
    thread.interrupt();                              //中断线程
  }
  public static void init(JFrame frame, int width, int height) {
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
}
```

运行本实例，结果如图 18.8 所示。

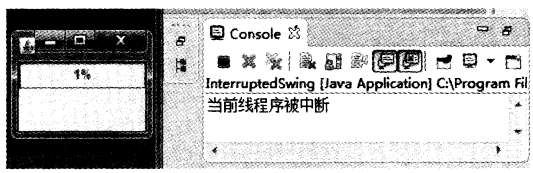


图 18.8　线程的中断

在本实例中，由于调用了 interrupted()方法，所以抛出了 InterruptedException 异常。

18.4.4　线程的礼让

视频讲解：光盘\TM\lx\18\线程的礼让.exe

Thread 类中提供了一种礼让方法，使用 yield()方法表示，它只是给当前正处于运行状态的线程一个提醒，告知它可以将资源礼让给其他线程，但这仅是一种暗示，没有任何一种机制保证当前线程会将资源礼让。

yield()方法使具有同样优先级的线程有进入可执行状态的机会，当当前线程放弃执行权时会再度回到就绪状态。对于支持多任务的操作系统来说，不需要调用 yield()方法，因为操作系统会为线程自动分配 CPU 时间片来执行。

18.5　线程的优先级

视频讲解：光盘\TM\lx\18\线程的优先级.exe

每个线程都具有各自的优先级，线程的优先级可以表明在程序中该线程的重要性，如果有很多线

程处于就绪状态，系统会根据优先级来决定首先使哪个线程进入运行状态。但这并不意味着低优先级的线程得不到运行，而只是它运行的几率比较小，如垃圾回收线程的优先级就较低。

Thread 类中包含的成员变量代表了线程的某些优先级，如 Thread.MIN_PRIORITY（常数 1）、Thread.MAX_PRIORITY（常数 10）、Thread.NORM_PRIORITY（常数 5）。其中每个线程的优先级都在 Thread.MIN_PRIORITY~Thread.MAX_PRIORITY 之间，在默认情况下其优先级都是 Thread.NORM_PRIORITY。每个新产生的线程都继承了父线程的优先级。

在多任务操作系统中，每个线程都会得到一小段 CPU 时间片运行，在时间结束时，将轮换另一个线程进入运行状态，这时系统会选择与当前线程优先级相同的线程予以运行。系统始终选择就绪状态下优先级较高的线程进入运行状态。处于各个优先级状态下的线程的运行顺序如图 18.9 所示。

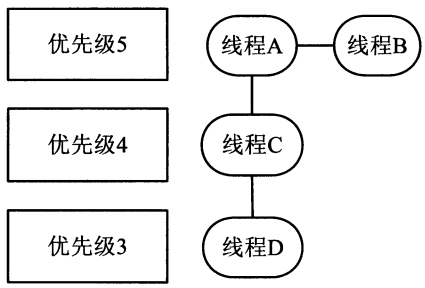


图 18.9　处于各个优先级状态下的线程的运行顺序

在图 18.9 中，优先级为 5 的线程 A 首先得到 CPU 时间片；当该时间结束后，轮换到与线程 A 相同优先级的线程 B；当线程 B 的运行时间结束后，会继续轮换到线程 A，直到线程 A 与线程 B 都执行完毕，才会轮换到线程 C；当线程 C 结束后，才会轮换到线程 D。

线程的优先级可以使用 setPriority()方法调整，如果使用该方法设置的优先级不在 1~10 之内，将产生 IllegalArgumentException 异常。

下面的实例演示了图 18.9 描述的状况，依然以进度条为例说明。

【例 18.7】 在项目中创建 PriorityTest 类，该类实现了 Runnable 接口。创建 4 个进度条，分别由 4 个线程来控制，并且为这 4 个线程设置不同的优先级。本实例关键代码如下：（**实例位置：光盘\TM\sl\18.06**）

```
import java.awt.*;
import javax.swing.*;
public class PriorityTest extends JFrame{
    …//非关键代码省略
    public PriorityTest() {
        …//非关键代码省略
        threadA=new Thread(new MyThread(progressBar));//分别实例化 4 个线程
        threadB=new Thread(new MyThread(progressBar2));
        threadC=new Thread(new MyThread(progressBar3));
        threadD=new Thread(new MyThread(progressBar4));
        setPriority("threadA", 5, threadA);
        setPriority("threadB", 5, threadB);
        setPriority("threadC", 4, threadC);
        setPriority("threadD", 3, threadD);
    }
    //定义设置线程的名称、优先级的方法
```

```
    public static void setPriority(String threadName,int priority,Thread t){
        t.setPriority(priority);                    //设置线程的优先级
        t.setName(threadName);                      //设置线程的名称
        t.start();                                  //启动线程
    }
    public static void main(String[] args) {
        init(new PriorityTest(),100,100);
    }
    ...//非关键代码省略
    //定义一个实现 Runnable 接口的类
    private final class MyThread implements Runnable {
        private final JProgressBar bar;
        int count=0;
        private MyThread(JProgressBar bar) {
            this.bar = bar;
        }
        public void run(){                          //重写 run()方法
            while(true){
                bar.setValue(count+=10);            //设置滚动条的值每次自增 10
                try{
                    Thread.sleep(1000);
                }catch(InterruptedException e){
                    System.out.println("当前线程被中断");
                }
            }
        }
    }
}
```

运行上述代码，结果如图 18.10 所示。

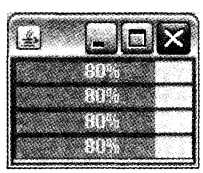


图 18.10　线程的优先级

在本实例中定义了 4 个线程，这 4 个线程用于设置 4 个进度条的进度。这里定义了 setPriority()方法，该方法设置了每个线程的优先级和名称等。虽然在图 18.10 中看这 4 个进度条好像是在一起滚动，但如果仔细观察还是可以看出细微差别，可以看到第一个进度条总是最先变化。由于 threadA 线程和 threadB 线程优先级最高，所以系统首先处理这两个线程，然后是 threadC 和 threadD 这两个线程。

18.6　线 程 同 步

在单线程程序中，每次只能做一件事情，后面的事情需要等待前面的事情完成后才可以进行，但是如果使用多线程程序，就会发生两个线程抢占资源的问题，如两个人同时说话、两个人同时过同一

个独木桥等。所以在多线程编程中需要防止这些资源访问的冲突。Java 提供了线程同步的机制来防止资源访问的冲突。

18.6.1　线程安全

视频讲解：光盘\TM\lx\18\线程安全.exe

实际开发中，使用多线程程序的情况很多，如银行排号系统、火车站售票系统等。这种多线程的程序通常会发生问题，以火车站售票系统为例，在代码中判断当前票数是否大于 0，如果大于 0 则执行将该票出售给乘客的功能，但当两个线程同时访问这段代码时（假如这时只剩下一张票），第一个线程将票售出，与此同时第二个线程也已经执行完成判断是否有票的操作，并得出票数大于 0 的结论，于是它也执行售出操作，这样就会产生负数。所以在编写多线程程序时，应该考虑到线程安全问题。实质上线程安全问题来源于两个线程同时存取单一对象的数据。

【例 18.8】 在项目中创建 ThreadSafeTest 类，该类实现了 Runnable 接口，主要实现模拟火车站售票系统的功能。（**实例位置：光盘\TM\sl\18.07**）

```
public class ThreadSafeTest implements Runnable {
    int num = 10;                                          //设置当前总票数
    public void run() {
        while (true) {
            if (num > 0) {
                try {
                    Thread.sleep(100);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                System.out.println("tickets" + num--);
            }
        }
    }
    public static void main(String[] args) {
        ThreadSafeTest t = new ThreadSafeTest();           //实例化类对象
        Thread tA = new Thread(t);                         //以该类对象分别实例化 4 个线程
        Thread tB = new Thread(t);
        Thread tC = new Thread(t);
        Thread tD = new Thread(t);
        tA.start();                                        //分别启动线程
        tB.start();
        tC.start();
        tD.start();
    }
}
```

运行本实例，最后几行结果如图 18.11 所示。

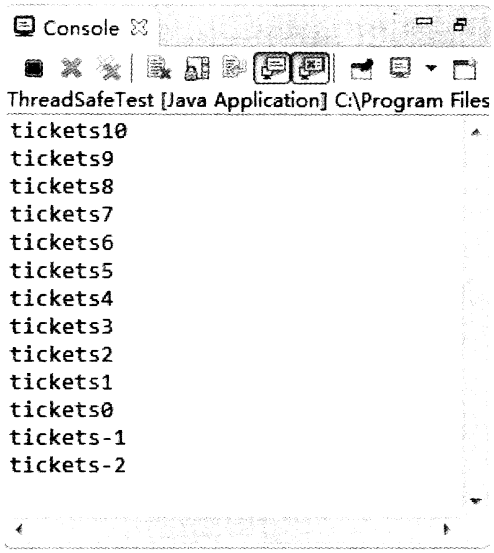


图 18.11　资源共享冲突后出现的问题

从图 18.11 中可以看出，最后打印售剩下的票为负值，这样就出现了问题。这是由于同时创建了 4 个线程，这 4 个线程执行 run()方法，在 num 变量为 1 时，线程 1、线程 2、线程 3、线程 4 都对 num 变量有存储功能，当线程 1 执行 run()方法时，还没有来得及做递减操作，就指定它调用 sleep()方法进入就绪状态，这时线程 2、线程 3 和线程 4 都进入了 run()方法，发现 num 变量依然大于 0，但此时线程 1 休眠时间已到，将 num 变量值递减，同时线程 2、线程 3、线程 4 也都对 num 变量进行递减操作，从而产生了负值。

18.6.2　线程同步机制

视频讲解：光盘\TM\lx\18\线程同步机制.exe

那么该如何解决资源共享的问题呢？基本上所有解决多线程资源冲突问题的方法都是采用给定时间只允许一个线程访问共享资源，这时就需要给共享资源上一道锁。这就好比一个人上洗手间时，他进入洗手间后会将门锁上，出来时再将锁打开，然后其他人才可以进入。

1. 同步块

在 Java 中提供了同步机制，可以有效地防止资源冲突。同步机制使用 synchronized 关键字。

【例 18.9】 在本实例中，创建类 ThreadSafeTest.java，在该类中修改例 18.8 中的 run()方法，把对 num 操作的代码设置在同步块中。本实例关键代码如下：（**实例位置：光盘\TM\sl\18.08**）

```
public class ThreadSafeTest implements Runnable {
    int num = 10;
    public void run() {
        while (true) {
            synchronized ("") {
                if (num > 0) {
```

```
                    try {
                        Thread.sleep(1000);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    System.out.println("tickets" + --num);
                }
            }
        }
    }
    public static void main(String[] args) {
        ThreadSafeTest t = new ThreadSafeTest();
        Thread tA = new Thread(t);
        Thread tB = new Thread(t);
        Thread tC = new Thread(t);
        Thread tD = new Thread(t);
        tA.start();
        tB.start();
        tC.start();
        tD.start();
    }
}
```

运行本实例，结果如图 18.12 所示。

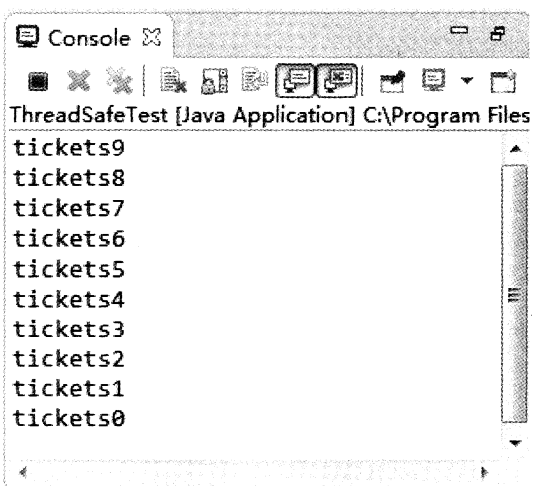


图 18.12　修改例 18.8 中的 run()方法

从图 18.12 中可以看出，打印到最后票数没有出现负数，这是因为将资源放置在了同步块中。这个同步块也被称为临界区，它使用 synchronized 关键字建立，其语法如下：

```
synchronized(Object){
}
```

通常将共享资源的操作放置在 synchronized 定义的区域内，这样当其他线程也获取到这个锁时，

必须等待锁被释放时才能进入该区域。Object 为任意一个对象，每个对象都存在一个标志位，并具有两个值，分别为 0 和 1。一个线程运行到同步块时首先检查该对象的标志位，如果为 0 状态，表明此同步块中存在其他线程在运行。这时该线程处于就绪状态，直到处于同步块中的线程执行完同步块中的代码为止。这时该对象的标志位被设置为 1，该线程才能执行同步块中的代码，并将 Object 对象的标志位设置为 0，防止其他线程执行同步块中的代码。

2．同步方法

同步方法就是在方法前面修饰 synchronized 关键字的方法，其语法如下：

```
synchronized void f(){ }
```

当某个对象调用了同步方法时，该对象上的其他同步方法必须等待该同步方法执行完毕后才能被执行。必须将每个能访问共享资源的方法修饰为 synchronized，否则就会出错。

修改例 18.9，将共享资源操作放置在一个同步方法中，如例 18.10 所示。

【例 18.10】 在项目中创建一个类文件，在该类中定义同步方法。

```
public synchronized void doit() {          //定义同步方法
    if(num>0){
        try{
            Thread.sleep(10);
        }catch(Exception e){
            e.printStackTrace();
        }
        System.out.println("tickets"+--num);
    }
}
public void run(){
    while(true){
        doit();                            //在 run()方法中调用该同步方法
    }
}
```

将共享资源的操作放置在同步方法中，运行结果与使用同步块的结果一致。

18.7 小　　结

本章讲述了线程，通过对其学习读者应该掌握线程与 Swing 技术相结合使用的方法。

学习多线程编程就像进入了一个全新的领域，它与以往的编程思想截然不同，随着大多数操作系统对多线程的支持，很多程序语言都已支持和扩展多线程，初学者应该积极转换编程思维，以进入多线程编程的思维方式。多线程本身是一种非常复杂的机制，完全理解它也需要一段时间，并且需要深入地学习。本章将多线程与 Swing 技术联系在一起，列举了大量实例，使读者从实例中体会多线程机制，为读者掌握多线程的基础知识打下了坚实的基础，从而深刻理解其概念并编写出合理的多线程程序。

18.8　实践与练习

1．尝试定义一个继承 Thread 类的类，并覆盖 run()方法，在 run()方法中每隔 100 毫秒打印一句话。（**答案位置：光盘\TM\sl\18.09**）

2．尝试开发一个窗体，在窗体中有两个按钮，一个是“开始”按钮，另一个是“结束”按钮。当用户单击“开始”按钮时，在控制台中持续打印一段话；当用户单击“停止”按钮时，控制台结束打印。（**答案位置：光盘\TM\sl\18.10**）

3．尝试开发一个窗体，在窗体中设计一个进度条，使进度条每次递增滚动。（**答案位置：光盘\TM\sl\18.11**）

第19章

网络通信

（视频讲解：24分钟）

Internet提供了大量、多样的信息，很少有人能在接触过Internet后拒绝它的诱惑。计算机网络实现了多个计算机互连系统，相互连接的计算机之间彼此能够进行数据交流。网络应用程序就是在已连接的不同计算机上运行的程序，这些程序相互之间可以交换数据。而编写网络应用程序，首先必须明确网络应用程序所要使用的网络协议，TCP/IP协议是网络应用程序的首选。本章将从介绍网络协议开始，向读者介绍TCP网络程序和UDP网络程序。

通过阅读本章，您可以：

- 了解网络程序设计基础
- 学会编写TCP程序
- 学会编写UDP程序

19.1　网络程序设计基础

网络程序设计是指编写与其他计算机进行通信的程序。Java 已经将网络程序所需要的东西封装成不同的类。只要创建这些类的对象，使用相应的方法，即使设计人员不具备有关的网络知识，也可以编写出高质量的网络通信程序。

19.1.1　局域网与因特网

视频讲解：光盘\TM\lx\19\局域网与因特网.exe

为了实现两台计算机的通信，必须用一个网络线路连接两台计算机，如图 19.1 所示。

图 19.1　服务器、客户机和网络

服务器是指提供信息的计算机或程序，客户机是指请求信息的计算机或程序，而网络用于连接服务器与客户机，实现两者相互通信。但有时在某个网络中很难将服务器与客户机区分开。我们通常所说的局域网（Local Area Network，LAN），就是一群通过一定形式连接起来的计算机。它可以由两台计算机组成，也可以由同一区域内的上千台计算机组成。由 LAN 延伸到更大的范围，这样的网络称为广域网（Wide Area Network，WAN）。我们熟悉的因特网（Internet），就是由无数的 LAN 和 WAN 组成的。

19.1.2　网络协议

视频讲解：光盘\TM\lx\19\网络协议.exe

网络协议规定了计算机之间连接的物理、机械（网线与网卡的连接规定）、电气（有效的电平范围）等特征以及计算机之间的相互寻址规则、数据发送冲突的解决、长的数据如何分段传送与接收等。就像不同的国家有不同的法律一样，目前网络协议也有多种，下面简单地介绍几个常用的网络协议。

1．IP 协议

IP 是 Internet Protocol 的简称，它是一种网络协议。Internet 网络采用的协议是 TCP/IP 协议，其全称是 Transmission Control Protocol/Internet Protocol。Internet 依靠 TCP/IP 协议，在全球范围内实现不同硬件结构、不同操作系统、不同网络系统的互联。在 Internet 网络上存在数以亿计的主机，每一台主机在网络上用为其分配的 Internet 地址代表自己，这个地址就是 IP 地址。到目前为止，IP 地址用 4 个字节，也就是 32 位的二进制数来表示，称为 IPv4。为了便于使用，通常取用每个字节的十进制数，并且每个字节之间用圆点隔开来表示 IP 地址，如 192.168.1.1。现在人们正在试验使用 16 个字节来表示 IP 地址，这就是 IPv6，但 IPv6 还没有投入使用。

TCP/IP 模式是一种层次结构，共分为 4 层，分别为应用层、传输层、互联网层和网络层。各层实现特定的功能，提供特定的服务和访问接口，并具有相对的独立性，如图 19.2 所示。

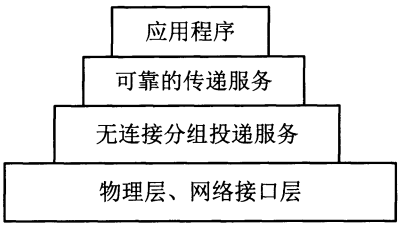


图 19.2　TCP/IP 层次结构

2. TCP 与 UDP 协议

在 TCP/IP 协议栈中，有两个高级协议是网络应用程序编写者应该了解的，即传输控制协议（Transmission Control Protocol，TCP）与用户数据报协议（User Datagram Protocol，UDP）。

TCP 协议是一种以固接连线为基础的协议，它提供两台计算机间可靠的数据传送。TCP 可以保证从一端数据送至连接的另一端时，数据能够确实送达，而且抵达的数据的排列顺序和送出时的顺序相同，因此，TCP 协议适合可靠性要求比较高的场合。就像拨打电话，必须先拨号给对方，等两端确定连接后，相互才能听到对方说话，也知道对方回应的是什么。

HTTP、FTP 和 Telnet 等都需要使用可靠的通信频道。例如，HTTP 从某个 URL 读取数据时，如果收到的数据顺序与发送时不相同，可能就会出现一个混乱的 HTML 文件或是一些无效的信息。

UDP 是无连接通信协议，不保证可靠数据的传输，但能够向若干个目标发送数据，接收发自若干个源的数据。UDP 是以独立发送数据包的方式进行。这种方式就像邮递员送信给收信人，可以寄出很多信给同一个人，而每一封信都是相对独立的，各封信送达的顺序并不重要，收信人接收信件的顺序也不能保证与寄出信件的顺序相同。

UDP 协议适合于一些对数据准确性要求不高的场合，如网络聊天室、在线影片等。这是由于 TCP 协议在认证上存在额外耗费，可能使传输速度减慢，而 UDP 协议可能会更适合这些对传输速度和时效要求非常高的网站，即使有一小部分数据包遗失或传送顺序有所不同，也不会严重危害该项通信。

注意

一些防火墙和路由器会设置成不允许 UDP 数据包传输，因此，若遇到 UDP 连接方面的问题，应先确定所在网络是否允许 UDP 协议。

19.1.3　端口和套接字

视频讲解：光盘\TM\lx\19\端口和套接字.exe

一般而言，一台计算机只有单一的连到网络的物理连接（Physical Connection），所有的数据都通过此连接对内、对外送达特定的计算机，这就是端口。网络程序设计中的端口（port）并非真实的物理存在，而是一个假想的连接装置。端口被规定为一个在 0~65535 之间的整数。HTTP 服务一般使用 80 端口，FTP 服务使用 21 端口。假如一台计算机提供了 HTTP、FTP 等多种服务，那么客户机会通过不同的端口来确定连接到服务器的哪项服务上，如图 19.3 所示。

通常，0~1023 之间的端口数用于一些知名的网络服务和应用，用户的普通网络应用程序应该使用 1024 以上的端口数，以避免端口号与另一个应用或系统服务所用端口冲突。

网络程序中的套接字（Socket）用于将应用程序与端口连接起来。套接字是一个假想的连接装置，

就像插插头的设备“插座”用于连接电器与电线一样，如图 19.4 所示。Java 将套接字抽象化为类，程序设计者只需创建 Socket 类对象，即可使用套接字。

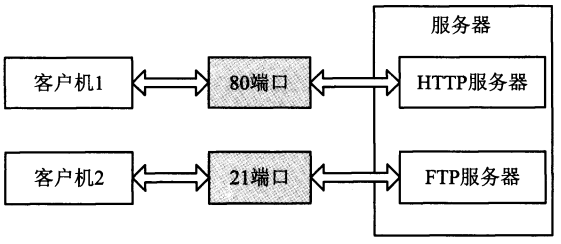


图 19.3　端口

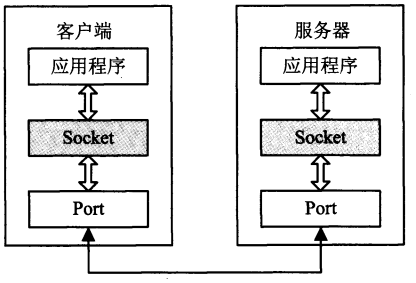


图 19.4　套接字

19.2　TCP 程序设计基础

TCP 网络程序设计是指利用 Socket 类编写通信程序。利用 TCP 协议进行通信的两个应用程序是有主次之分的，一个称为服务器程序，另一个称为客户机程序，两者的功能和编写方法大不一样。服务器端与客户端的交互过程如图 19.5 所示。

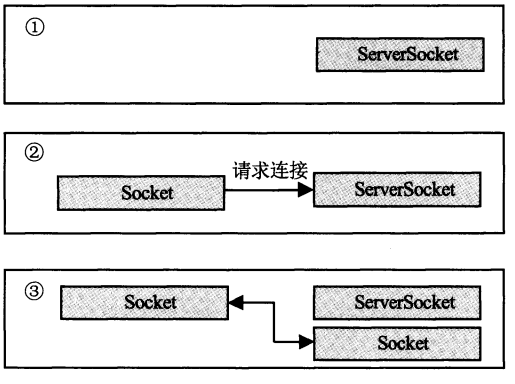


图 19.5　服务器端与客户端的交互

①——服务器程序创建一个 ServerSocket（服务器端套接字），调用 accept()方法等待客户机来连接

②——客户端程序创建一个 Socket，请求与服务器建立连接

③——服务器接收客户机的连接请求，同时创建一个新的 Socket 与客户建立连接。服务器继续等待新的请求

19.2.1　InetAddress 类

视频讲解：光盘\TM\lx\19\InetAddress 类.exe

java.net 包中的 InetAddress 类是与 IP 地址相关的类，利用该类可以获取 IP 地址、主机地址等信息。InetAddress 类的常用方法如表 19.1 所示。

表 19.1　InetAddress 类的常用方法

方　法	返 回 值	说　明
getByName(String host)	InetAddress	获取与 Host 相对应的 InetAddress 对象
getHostAddress()	String	获取 InetAddress 对象所包含的 IP 地址
getHostName()	String	获取此 IP 地址的主机名
getLocalHost()	InetAddress	返回本地主机的 InetAddress 对象

【例 19.1】 使用 InetAddress 类的 getHostName()和 getHostAddress()方法获得本地主机的本机名、本机 IP 地址。（实例位置：光盘\TM\sl\19.01）

```
import java.net.*;                                          //导入 java.net 包
public class Address {                                      //创建类
    public static void main(String[] args) {
        InetAddress ip;                                     //创建 InetAddress 对象
        try {                                               //使用 try 语句块捕捉可能出现的异常
            ip = InetAddress.getLocalHost();                //实例化对象
            String localname = ip.getHostName();            //获取本机名
            String localip = ip.getHostAddress();           //获取本机 IP 地址
            System.out.println("本机名：" + localname);      //将本机名输出
            System.out.println("本机 IP 地址：" + localip);  //将本机 IP 地址输出
        } catch (UnknownHostException e) {
            e.printStackTrace();                            //输出异常信息
        }
    }
}
```

运行结果如图 19.6 所示。

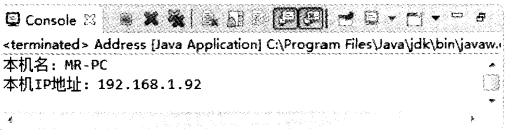


图 19.6　例 19.1 的运行结果

注意

InetAddress 类的方法会抛出 UnknownHostException 异常，所以必须进行异常处理。这个异常在主机不存在或网络连接错误时发生。

19.2.2　ServerSocket 类

视频讲解：光盘\TM\lx\19\ServerSocket 类.exe

java.net 包中的 ServerSocket 类用于表示服务器套接字，其主要功能是等待来自网络上的“请求”，它可通过指定的端口来等待连接的套接字。服务器套接字一次可以与一个套接字连接。如果多台客户机同时提出连接请求，服务器套接字会将请求连接的客户机存入列队中，然后从中取出一个套接字，

与服务器新建的套接字连接起来。若请求连接数大于最大容纳数，则多出的连接请求被拒绝。队列的默认大小是 50。

ServerSocket 类的构造方法都抛出 IOException 异常，分别有以下几种形式。

- ☑ ServerSocket()：创建非绑定服务器套接字。
- ☑ ServerSocket(int port)：创建绑定到特定端口的服务器套接字。
- ☑ ServerSocket(int port, int backlog)：利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。
- ☑ ServerSocket(int port, int backlog, InetAddress bindAddress)：使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。这种情况适用于计算机上有多块网卡和多个 IP 地址的情况，用于可以明确规定 ServerSocket 在哪块网卡或 IP 地址上等待客户的连接请求。

ServerSocket 类的常用方法如表 19.2 所示。

表 19.2　ServerSocket 类的常用方法

方　法	返 回 值	说　明
accept()	Socket	等待客户机的连接。若连接，则创建一套接字
isBound()	boolean	判断 ServerSocket 的绑定状态
getInetAddress()	InetAddress	返回此服务器套接字的本地地址
isClosed()	boolean	返回服务器套接字的关闭状态
close()	void	关闭服务器套接字
bind(SocketAddress endpoint)	void	将 ServerSocket 绑定到特定地址（IP 地址和端口号）
getInetAddress()	int	返回服务器套接字等待的端口号

调用 ServerSocket 类的 accept()方法会返回一个和客户端 Socket 对象相连接的 Socket 对象，服务器端的 Socket 对象使用 getOutputStream()方法获得的输出流将指向客户端 Socket 对象使用 getInputStream()方法获得的那个输入流；同样，服务器端的 Socket 对象使用 getInputStream()方法获得的输入流将指向客户端 Socket 对象使用 getOutputStream()方法获得的那个输出流。也就是说，当服务器向输出流写入信息时，客户端通过相应的输入流就能读取，反之亦然。

注意

accept()方法会阻塞线程的继续执行，直到接收到客户的呼叫。如果没有客户呼叫服务器，那么 System.out.println("连接中")语句将不会执行。语句如果没有客户请求，accept()方法没有发生阻塞，肯定是程序出现了问题。通常是使用了一个还在被其他程序占用的端口号，ServerSocket 绑定没有成功。

```
yu = server.accept();
System.out.println("连接中");
```

19.2.3　TCP 网络程序

视频讲解：光盘\TM\lx\19\TCP 网络程序.exe

明白了 TCP 程序工作的过程，就可以编写 TCP 服务器程序了。在网络编程中如果只要求客户机向

服务器发送消息，不要求服务器向客户机发送消息，称为单向通信。客户机套接字和服务器套接字连接成功后，客户机通过输出流发送数据，服务器则通过输入流接收数据。下面是简单的单向通信的实例。

【例 19.2】 本实例是一个 TCP 服务器端程序，在 getserver()方法中建立服务器套接字，调用 getClientMessage()方法获取客户端信息。（**实例位置：光盘\TM\sl\19.02**）

```
import java.io.*;                                          //导入 java.io 包
import java.net.*;                                         //导入 java.net 包
public class MyTcp {                                       //创建类 MyTcp
    private BufferedReader reader;                         //创建 BufferedReader 对象
    private ServerSocket server;                           //创建 ServerSocket 对象
    private Socket socket;                                 //创建 Socket 对象 socket
    void getserver() {
        try {
            server = new ServerSocket(8998);               //实例化 Socket 对象
            System.out.println("服务器套接字已经创建成功");    //输出信息
            while (true) {                                 //如果套接字是连接状态
                System.out.println("等待客户机的连接");        //输出信息
                socket = server.accept();                  //实例化 Socket 对象
                reader = new BufferedReader(new InputStreamReader(socket
                        .getInputStream()));               //实例化 BufferedReader 对象
                getClientMessage();                        //调用 getClientMessage()方法
            }
        } catch (Exception e) {
            e.printStackTrace();                           //输出异常信息
        }
    }
    private void getClientMessage() {
        try {
            while (true) {                                 //如果套接字是连接状态
                //获得客户端信息
                System.out.println("客户机:" + reader.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();                           //输出异常信息
        }
        try {
            if (reader != null) {
                reader.close();                            //关闭流
            }
            if (socket != null) {
                socket.close();                            //关闭套接字
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {               //主方法
        MyTcp tcp = new MyTcp();                           //创建本类对象
        tcp.getserver();                                   //调用方法
    }
}
```

运行结果如图 19.7 所示。

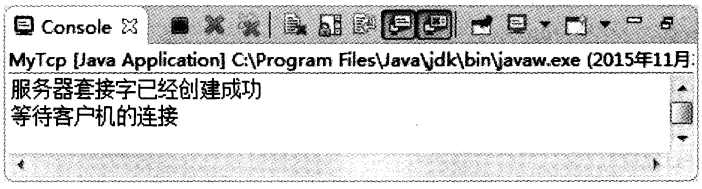


图 19.7　例 19.2 的运行结果

运行服务器端程序，将输出提示信息，等待客户呼叫。下面再来看一下客户端程序。

【例 19.3】 客户端程序，实现将用户在文本框中输入的信息发送至服务器端，并将文本框中输入的信息显示在客户端的文本域中。（**实例位置：光盘\TM\sl\19.03**）

```
package com.lzw;
public class MyClien extends JFrame {                          //创建类继承 JFrame 类
    private PrintWriter writer;                                //声明 PrintWriter 类对象
    Socket socket;                                             //声明 Socket 对象
    private JTextArea ta = new JTextArea();                    //创建 JtextArea 对象
    private JTextField tf = new JTextField();                  //创建 JtextField 对象
    Container cc;                                              //声明 Container 对象
    public MyClien(String title) {                             //构造方法
        super(title);                                          //调用父类的构造方法
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cc = this.getContentPane();                            //实例化对象
        final JScrollPane scrollPane = new JScrollPane();
        scrollPane.setBorder(new BevelBorder(BevelBorder.RAISED));
        getContentPane().add(scrollPane, BorderLayout.CENTER);
        scrollPane.setViewportView(ta);
        cc.add(tf, "South");                                   //将文本框放在窗体的下部
        tf.addActionListener(new ActionListener() {
            //绑定事件
            public void actionPerformed(ActionEvent e) {
                //将文本框中的信息写入流
                writer.println(tf.getText());
                //将文本框中的信息显示在文本域中
                ta.append(tf.getText() + '\n');
                ta.setSelectionEnd(ta.getText().length());
                tf.setText("");                                //将文本框清空
            }
        });
    }
    private void connect() {                                   //连接套接字方法
        ta.append("尝试连接\n");                                //文本域中提示信息
        try {                                                  //捕捉异常
            socket = new Socket("127.0.0.1", 8998);            //实例化 Socket 对象
            writer = new PrintWriter(socket.getOutputStream(), true);
            ta.append("完成连接\n");                            //文本域中提示信息
        } catch (Exception e) {
            e.printStackTrace();                               //输出异常信息
```

```
        }
    }
    public static void main(String[] args) {                    //主方法
        MyClien clien = new MyClien("向服务器送数据");          //创建本例对象
        clien.setSize(200, 200);                                 //设置窗体大小
        clien.setVisible(true);                                  //将窗体显示
        clien.connect();                                         //调用连接方法
    }
}
```

运行服务器端，再运行这个客户端，运行结果如图 19.8 所示。

从图 19.8 中可以看出，客户端与服务器端已经创建了连接。向文本框中输入信息，会发现输入的信息在服务器端输出，并在客户端的文本域中显示，如图 19.9 和图 19.10 所示。

说明

当一台机器上安装了多个网络应用程序时，很可能指定的端口号已被占用。还可能遇到以前运行良好的网络程序突然运行不了的情况，这种情况很可能也是由于端口被别的程序占用了。此时可以运行 netstat-help 来获得帮助，使用命令 netstat-an 来查看该程序所使用的端口，如图 19.11 所示。

图 19.8　例 19.3 的运行结果

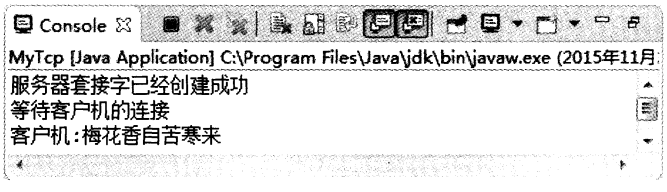


图 19.9　服务器端运行结果

图 19.10　客户端运行结果

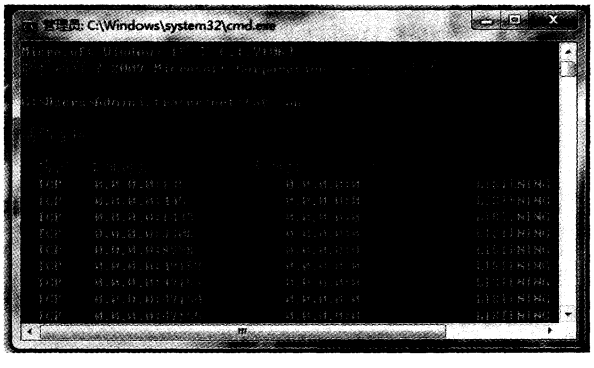


图 19.11　查看端口

19.3　UDP 程序设计基础

用户数据报协议（UDP）是网络信息传输的另一种形式。基于 UDP 的通信和基于 TCP 的通信不同，基于 UDP 的信息传递更快，但不提供可靠的保证。使用 UDP 传递数据时，用户无法知道数据能否正确地到达主机，也不能确定到达目的地的顺序是否和发送的顺序相同。虽然 UDP 是一种不可靠的协议，但如果需要较快地传输信息，并能容忍小的错误，可以考虑使用 UDP。

基于 UDP 通信的基本模式如下：

☑ 将数据打包（称为数据包），然后将数据包发往目的地。

☑ 接收别人发来的数据包，然后查看数据包。

下面是总结的 UDP 程序的步骤。

发送数据包：

（1）使用 DatagramSocket()创建一个数据包套接字。

（2）使用 DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)创建要发送的数据包。

（3）使用 DatagramSocket 类的 send()方法发送数据包。

接收数据包：

（1）使用 DatagramSocket(int port)创建数据包套接字，绑定到指定的端口。

（2）使用 DatagramPacket(byte[] buf, int length)创建字节数组来接收数据包。

（3）使用 DatagramPacket 类的 receive()方法接收 UDP 包。

注意

DatagramSocket 类的 receive()方法接收数据时，如果还没有可以接收的数据，在正常情况下 receive()方法将阻塞，一直等到网络上有数据传来，receive()方法接收该数据并返回。如果网络上没有数据发送过来，receive()方法也没有阻塞，肯定是程序有问题，大多数是使用了一个被其他程序占用的端口号。

19.3.1　DatagramPacket 类

视频讲解：光盘\TM\lx\19\DatagramPacket 类.exe

java.net 包的 DatagramPacket 类用来表示数据包。DatagramPacket 类的构造函数有：

☑ DatagramPacket(byte[] buf, int length)。

☑ DatagramPacket(byte[] buf, int length, InetAddress address, int port)。

第一种构造函数创建 DatagramPacket 对象，指定了数据包的内存空间和大小。第二种构造函数不仅指定了数据包的内存空间和大小，还指定了数据包的目标地址和端口。在发送数据时，必须指定接收方的 Socket 地址和端口号，因此使用第二种构造函数可创建发送数据的 DatagramPacket 对象。

19.3.2 DatagramSocket 类

视频讲解：光盘\TM\lx\19\DatagramSocket 类.exe

java.net 包中的 DatagramSocket 类用于表示发送和接收数据包的套接字。该类的构造函数有：

☑ DatagramSocket()。

☑ DatagramSocket(int port)。

☑ DatagramSocket(int port, InetAddress addr)。

第一种构造函数创建 DatagramSocket 对象，构造数据报套接字并将其绑定到本地主机上任何可用的端口。第二种构造函数创建 DatagramSocket 对象，创建数据报套接字并将其绑定到本地主机上的指定端口。第三种构造函数创建 DatagramSocket 对象，创建数据报套接字并将其绑定到指定的本地地址。第三种构造函数适用于有多块网卡和多个 IP 地址的情况。

在接收程序时，必须指定一个端口号，不要让系统随机产生，此时可以使用第二种构造函数。比如有个朋友要你给他写信，可他的地址不确定是不行的。在发送程序时，通常使用第一种构造函数，不指定端口号，这样系统就会为我们分配一个端口号。就像寄信不需要到指定的邮局去寄一样。

19.3.3 UDP 网络程序

视频讲解：光盘\TM\lx\19\UDP 网络程序.exe

根据前面所讲的网络编程的基本知识，以及 UDP 网络编程的特点，下面创建一个广播数据报程序。广播数据报是一种较新的技术，类似于电台广播，广播电台需要在指定的波段和频率上广播信息，收听者也要将收音机调到指定的波段、频率才可以收听广播内容。

【例 19.4】 主机不断地重复播出节目预报，可以保证加入到同一组的主机随时可接收到广播信息。接收者将正在接收的信息放在一个文本域中，并将接收的全部信息放在另一个文本域中。（**实例位置：光盘\TM\sl\19.04**）

（1）广播主机程序不断地向外播出信息，代码如下：

```
import java.net.*;
public class Weather extends Thread {                    //创建类。该类为多线程执行程序
    String weather = "节目预报：八点有大型晚会，请收听";
    int port = 9898;                                     //定义端口
    InetAddress iaddress = null;                         //创建 InetAddress 对象
    MulticastSocket socket = null;                       //声明多点广播套接字
    Weather() {                                          //构造方法
        try {
            //实例化 InetAddress，指定地址
            iaddress = InetAddress.getByName("224.255.10.0");
            socket = new MulticastSocket(port);          //实例化多点广播套接字
            socket.setTimeToLive(1);                     //指定发送范围是本地网络
            socket.joinGroup(iaddress);                  //加入广播组
        } catch (Exception e) {
```

```
            e.printStackTrace();                    //输出异常信息
        }
    }
    public void run() {                             //run()方法
        while (true) {
            DatagramPacket packet = null;           //声明 DatagramPacket 对象
            byte data[] = weather.getBytes();       //声明字节数组
            //将数据打包
            packet = new DatagramPacket(data, data.length, iaddress, port);
            System.out.println(new String(data));   //将广播信息输出
            try {
                socket.send(packet);                //发送数据
                sleep(3000);                        //线程休眠
            } catch (Exception e) {
                e.printStackTrace();                //输出异常信息
            }
        }
    }
    public static void main(String[] args) {        //主方法
        Weather w = new Weather();                  //创建本类对象
        w.start();                                  //启动线程
    }
}
```

运行结果如图 19.12 所示。

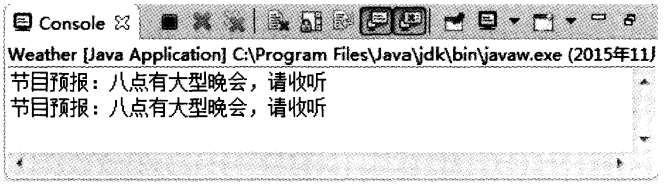


图 19.12　广播主机程序的运行结果

（2）接收广播程序：单击“开始接收”按钮，系统开始接收主机播出的信息；单击“停止接收”按钮，系统会停止接收广播主机播出的信息。

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;
public class Receive extends JFrame implements Runnable, ActionListener {
    int port;                                       //定义 int 型变量
    InetAddress group = null;                       //声明 InetAddress 对象
    MulticastSocket socket = null;                  //创建多点广播套接字对象
    JButton ince = new JButton("开始接收");          //创建按钮对象
    JButton stop = new JButton("停止接收");
    JTextArea inceAr = new JTextArea(10, 10);       //显示接收广播的文本域
    JTextArea inced = new JTextArea(10, 10);
    Thread thread;                                  //创建 Thread 对象
```

```
boolean b = false;                                          //创建 boolean 型变量
public Receive() {                                          //构造方法
    super("广播数据报");                                     //调用父类方法
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    thread = new Thread(this);
    ince.addActionListener(this);                           //绑定按钮 ince 的单击事件
    stop.addActionListener(this);                           //绑定按钮 stop 的单击事件
    inceAr.setForeground(Color.blue);                       //指定文本域中文字的颜色
    JPanel north = new JPanel();                            //创建 Jpanel 对象
    north.add(ince);                                        //将按钮添加到面板 north 上
    north.add(stop);
    add(north, BorderLayout.NORTH);                         //将 north 放置在窗体的上部
    JPanel center = new JPanel();                           //创建面板对象 center
    center.setLayout(new GridLayout(1, 2));                 //设置面板布局
    center.add(inceAr);                                     //将文本域添加到面板上
    center.add(inced);
    add(center, BorderLayout.CENTER);                       //设置面板布局
    validate();                                             //刷新
    port = 9898;                                            //设置端口号
    try {
        group = InetAddress.getByName("224.255.10.0");      //指定接收地址
        socket = new MulticastSocket(port);                 //绑定多点广播套接字
        socket.joinGroup(group);                            //加入广播组
    } catch (Exception e) {
        e.printStackTrace();                                //输出异常信息
    }
    setBounds(100, 50, 360, 380);                           //设置布局
    setVisible(true);                                       //将窗体设置为显示状态
}
public void run() {                                         //run()方法
    while (true) {
        byte data[] = new byte[1024];                       //创建 byte 数组
        DatagramPacket packet = null;                       //创建 DatagramPacket 对象
        //待接收的数据包
        packet = new DatagramPacket(data, data.length, group, port);
        try {
            socket.receive(packet);                         //接收数据包
            String message = new String(packet.getData(), 0, packet
                .getLength());                              //获取数据包中的内容
            //将接收内容显示在文本域中
            inceAr.setText("正在接收的内容：\n" + message);
            inced.append(message + "\n");                   //每条信息为一行
        } catch (Exception e) {
            e.printStackTrace();                            //输出异常信息
        }
        if (b == true) {                                    //当变量等于 true 时，退出循环
            break;
        }
    }
```

```
    }
    public void actionPerformed(ActionEvent e) {                    //单击事件
        if (e.getSource() == ince) {                                //单击按钮 ince 触发的事件
            ince.setBackground(Color.red);                          //设置按钮颜色
            stop.setBackground(Color.yellow);
            if (!(thread.isAlive())) {                              //如线程不处于“新建状态”
                thread = new Thread(this);                          //实例化 Thread 对象
            }
            thread.start();                                         //启动线程
            b = false;                                              //设置变量值
        }
        if (e.getSource() == stop) {                                //单击按钮 stop 触发的事件
            ince.setBackground(Color.yellow);                       //设置按钮颜色
            stop.setBackground(Color.red);
            b = true;                                               //设置变量值 s
        }
    }
    public static void main(String[] args) {                       //主方法
        Receive rec = new Receive();                                //创建本类对象
        rec.setSize(460, 200);                                      //设置窗体大小
    }
}
```

运行结果如图 19.13 所示。

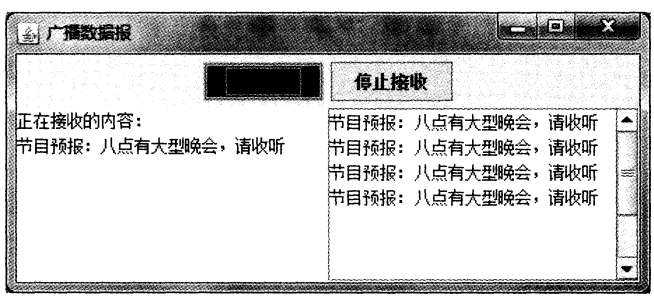


图 19.13　接收广播的运行结果

说明

要广播或接收广播的主机地址必须加入到一个组内，地址范围为 224.0.0.0~224.255.255.255，这类地址并不代表某个特定主机的位置。加入到同一个组的主机可以在某个端口上广播信息，也可以在某个端口上接收信息。

19.4　小　　结

本章介绍了 Java 网络编程知识，对于网络协议等内容，程序设计人员都应该有所了解，有兴趣的

读者还可以查阅其他资料来获取更详细的信息。TCP 与 UDP 网络编程的区别、java.net 包中提供的网络应用程序的常用类，以及这些类中的常用方法是本章的重点。通过本章的学习，读者应该能够自己尝试着编写一些网络程序来巩固所学知识。

19.5 实践与练习

1．编写 Java 程序，获得指定端口的主机名、主机地址和本机地址。（**答案位置：光盘\TM\sl\19.05**）

2．编写 TCP 服务器程序，实现创建一个在 8001 端口上等待的 ServerSocket 对象，当接收到一个客户机的连接请求后，程序从与客户机建立了连接的 Socket 对象中获得输入/输出流。通过输出流向客户机发送信息。（**答案位置：光盘\TM\sl\19.06**）

3．编写聊天室程序。（**答案位置：光盘\TM\sl\19.07**）

第20章

数据库操作

（视频讲解：48分钟）

数据库系统是由数据库、数据库管理系统和应用系统、数据库管理员构成的。数据库管理系统简称DBMS，是数据库系统的关键组成部分，包括数据库定义、数据查询、数据维护等。JDBC技术是连接数据库与应用程序的纽带。学习Java语言，必须学习JDBC技术，因为JDBC技术是在Java语言中被广泛使用的一种操作数据库的技术。每个应用程序的开发都是使用数据库保存数据，而使用JDBC技术访问数据库可达到查找满足条件的记录，或者向数据库添加、修改、删除数据。本章将向读者介绍Java语言的数据库操作部分。

通过阅读本章，您可以：

- 了解数据库的基础知识
- 了解JDBC技术的概念
- 掌握JDBC中常用的类和接口
- 掌握数据库操作的步骤

20.1 数据库基础知识

视频讲解：光盘\TM\lx\20\数据库基础.mp4

数据库在应用程序中占据着非常重要的地位。从原来的 Sybase 数据库，发展到今天的 SQL Server、MySQL、Oracle 等高级数据库，数据库已经相当成熟了。

20.1.1 什么是数据库

数据库是一种存储结构，它允许使用各种格式输入、处理和检索数据，不必在每次需要数据时重新输入。例如，当需要某人的电话号码时，需要查看电话簿，按照姓名来查阅，这个电话簿就是一个数据库。数据库具有以下主要特点。

- ☑ 实现数据共享。数据共享包含所有用户可同时存取数据库中的数据，也包括用户可以用各种方式通过接口使用数据库，并提供数据共享。
- ☑ 减少数据的冗余度。同文件系统相比，数据库实现了数据共享，从而避免了用户各自建立应用文件，减少了大量重复数据，减少了数据冗余，维护了数据的一致性。
- ☑ 数据的独立性。数据的独立性包括数据库中数据库的逻辑结构和应用程序相互独立，也包括数据物理结构的变化不影响数据的逻辑结构。
- ☑ 数据实现集中控制。文件管理方式中，数据处于一种分散的状态，不同的用户或同一用户在不同处理中其文件之间毫无关系。利用数据库可对数据进行集中控制和管理，并通过数据模型表示各种数据的组织以及数据间的联系。
- ☑ 数据的一致性和可维护性，以确保数据的安全性和可靠性。主要包括：
  - ➢ 安全性控制，以防止数据丢失、错误更新和越权使用。
  - ➢ 完整性控制，保证数据的正确性、有效性和相容性。
  - ➢ 并发控制，使在同一时间周期内，允许对数据实现多路存取，又能防止用户之间的不正常交互作用。
  - ➢ 故障的发现和恢复。

从发展的历史来看，数据库是数据管理的高级阶段，是由文件管理系统发展起来的。数据库的基本结构分为 3 个层次。

- ☑ 物理数据层：它是数据库的最内层，是物理存储设备上实际存储的数据集合。这些数据是原始数据，是用户加工的对象，由内部模式描述的指令操作处理的字符和字组成。
- ☑ 概念数据层：它是数据库的中间一层，是数据库的整体逻辑表示，指出了每个数据的逻辑定义及数据间的逻辑联系，是存储记录的集合。它所涉及的是数据库所有对象的逻辑关系，而不是它们的物理情况，是数据库管理员概念下的数据库。
- ☑ 逻辑数据层：它是用户所看到和使用的数据库，是一个或一些特定用户使用的数据集合，即逻辑记录的集合。

20.1.2　数据库的种类及功能

数据库系统一般基于某种数据模型，可以分为层次型、网状型、关系型及面向对象型等。

☑ 层次型数据库：层次型数据库类似于树结构，是一组通过链接而相互联系在一起的记录。层次模型的特点是记录之间的联系通过指针实现。由于层次模型层次顺序严格而且复杂，因此对数据进行各项操作都很困难。层次型数据库如图 20.1 所示。

☑ 网状型数据库：网络模型是使用网络结构表示实体类型、实体间联系的数据模型。网络模型容易实现多对多的联系。但在编写应用程序时，程序员必须熟悉数据库的逻辑结构，如图 20.2 所示。

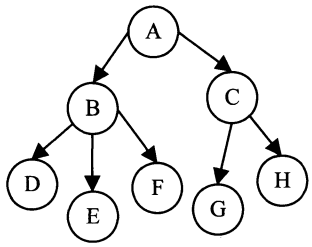


图 20.1　层次型数据库

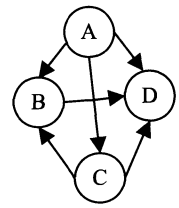


图 20.2　网状型数据库

☑ 面向对象型数据库：建立在面向对象模型基础上。

☑ 关系型数据库：关系型数据库是目前最流行的数据库，是基于关系模型建立的数据库，关系模型是由一系列表格组成的。后面会详细地讲解它。

在当前比较流行的数据库中，MySQL 数据库是开发源代码的软件，具有功能强、使用简便、管理方便、运行速度快、安全可靠性强等优点，同时也是具有客户机/服务器体系结构的分布式数据库管理系统。MySQL 是完全网络化的跨平台关系型数据库系统，它还支持多种平台，在 UNIX/Linux 系统上 MySQL 支持多线程运行方式，从而能获得相当好的性能。对于不使用 UNIX 系统的用户，可以在 Windows NT 系统上以系统服务方式运行，或者在 Windows 95/98 系统上以普通进程方式运行。

从 JDK 6 开始，在 JDK 的安装目录中，除了传统的 bin、jre 等目录，还新增了一个名为 db 的目录，这便是 Java DB。这是一个纯 Java 实现的、开源的数据库管理系统（DBMS），源于 Apache 软件基金会（ASF）名下的项目 Derby。它只有 2MB 大小，但这并不妨碍 Derby 功能齐备、支持几乎大部分的数据库应用所需要的特性。更难能可贵的是，作为内嵌的数据库，Derby 得到了包括 IBM 和 Sun 等大公司以及全世界优秀程序员们的支持。这就好像为 JDK 注入了一股全新的活力，Java 程序员不再需要耗费大量精力安装和配置数据库，就能进行安全、易用、标准且免费的数据库编程了。

20.1.3　SQL 语言

SQL（Structure Query Language，结构化查询语言）被广泛地应用于大多数数据库中，使用 SQL 语言可以方便地查询、操作、定义和控制数据库中的数据。SQL 语言主要由以下几部分组成。

☑ 数据定义语言（Data Definition Language，DDL），如 create、alter、drop 等。

☑ 数据操纵语言（Data Manipulation Language，DML），如 select、insert、update、delete 等。

☑ 数据控制语言（Data Control Language，DCL），如 grant、revoke 等。

☑ 事务控制语言（Transaction Control Language），如 commit、rollback 等。

在应用程序中使用最多的就是数据操纵语言，它也是最常用的核心 SQL 语言。下面对数据操纵语言进行简单的介绍。

1. select 语句

select 语句用于从数据表中检索数据。

语法如下：

```
SELECT 所选字段列表 FROM 数据表名
WHERE 条件表达式 GROUP BY 字段名 HAVING 条件表达式(指定分组的条件)
ORDER BY 字段名[ASC|DESC]
```

假设数据表名称是 tb_emp，要检索出 tb_emp 表中所有女员工的姓名、年龄，并按年龄升序排序，代码如例 20.1 所示。

【例 20.1】 将 tb_emp 表中所有女员工的姓名、年龄按年龄升序的形式检索出来。

```
select name,age form tb_emp where sex = '女' order by age;
```

2. insert 语句

insert 语句用于向表中插入新数据。

语法如下：

```
insert into 表名[(字段名 1,字段名 2…)]
values(属性值 1,属性值 2 …)
```

假设要向数据表 tb_emp（包含字段 id、name、sex、department）中插入数据，代码如例 20.2 所示。

【例 20.2】 向 tb_emp 表中插入数据。

```
insert into tb_emp values(2,'lili','女','销售部');
```

3. update 语句

update 语句用于更新数据表中的某些记录。

语法如下：

```
UPDATE 数据表名 SET 字段名 = 新的字段值 WHERE 条件表达式
```

假设要将数据表 tb_emp 中 2 号员工的年龄修改为 24，代码如例 20.3 所示。

【例 20.3】 修改 tb_emp 表中编号是 2 的员工年龄。

```
update tb_emp set age = 24 where id = 2;
```

4. delete 语句

delete 语句用于删除数据。

语法如下：

```
delete from 数据表名 where 条件表达式
```

假设要删除数据表 tb_emp 中编号为 1024 的员工，代码如例 20.4 所示。

【例 20.4】 将 tb_emp 表中编号为 1024 的员工删除。

```
delete from tb_emp where id = 1024;
```

20.2　JDBC 概述

视频讲解：光盘\TM\lx\20\JDBC 概述.mp4

JDBC 是一种可用于执行 SQL 语句的 Java API（Application Programming Interface，应用程序设计接口），是连接数据库和 Java 应用程序的纽带。

20.2.1　JDBC-ODBC 桥

JDBC-ODBC 桥是一个 JDBC 驱动程序，完成了从 JDBC 操作到 ODBC 操作之间的转换工作，允许 JDBC 驱动程序被用作 ODBC 的驱动程序。使用 JDBC-ODBC 桥连接数据库的步骤如下：

（1）首先加载 JDBC-ODBC 桥的驱动程序。代码如下：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Class 类是 java.lang 包中的一个类，通过该类的静态方法 forName()可加载 sun.jdbc.odbc 包中的 JdbcOdbcDriver 类来建立 JDBC-ODBC 桥连接器。

（2）使用 java.sql 包中的 Connection 接口，并通过 DriverManager 类的静态方法 getConnection()创建连接对象。代码如下：

```
Connection conn = DriverManager.getConnection("jdbc:odbc:数据源名字" , "user name" , "password");
```

数据源必须给出一个简短的描述名。假设没有设置 user name 和 password，则要与数据源 tom 交换数据。建立 Connection 对象的代码如下：

```
Connection conn = DriverManager.getConnection("jdbc.odbc:tom","","");
```

（3）向数据库发送 SQL 语句。使用 Statement 接口声明一个 SQL 语句对象，并通过刚才创建的连接数据库对象 conn 的 createStatement()方法创建这个 SQL 对象。代码如下：

```
Statement sql = conn.createStatement();
```

JDBC-ODBC 桥作为连接数据库的过渡性技术，现在已经不被 Java 广泛应用了，现在被广泛应用的是 JDBC 技术。但这并不表示 JDBC-ODBC 桥技术已经被淘汰，由于 ODBC 技术被广泛地使用，使得 Java 可以利用 JDBC-ODBC 桥访问几乎所有的数据库。JDBC-ODBC 桥作为 sun.jdbc.odbc 包与 JDK 一起自动安装，不需要特殊配置。

20.2.2 JDBC 技术

JDBC 的全称是 Java DataBase Connectivity，是一套面向对象的应用程序接口，指定了统一的访问各种关系型数据库的标准接口。JDBC 是一种底层的 API，因此访问数据库时需要在业务逻辑层中嵌入 SQL 语句。SQL 语句是面向关系的，依赖于关系模型，所以通过 JDBC 技术访问数据库也是面向关系的。JDBC 技术主要完成以下几个任务：

- ☑ 与数据库建立一个连接。
- ☑ 向数据库发送 SQL 语句。
- ☑ 处理从数据库返回的结果。

需要注意的是，JDBC 并不能直接访问数据库，必须依赖于数据库厂商提供的 JDBC 驱动程序。下面详细介绍 JDBC 驱动程序的分类。

20.2.3 JDBC 驱动程序的类型

JDBC 的总体结构由 4 个组件——应用程序、驱动程序管理器、驱动程序和数据源组成。JDBC 驱动基本上分为以下 4 种。

- ☑ JDBC-ODBC 桥：依靠 ODBC 驱动器和数据库通信。这种连接方式必须将 ODBC 二进制代码加载到使用该驱动程序的每台客户机上。这种类型的驱动程序最适合于企业网或者用 Java 编写的三层结构的应用程序服务器代码。
- ☑ 本地 API 一部分用 Java 编写的驱动程序：这类驱动程序把客户机的 API 上的 JDBC 调用转换为 Oracle、DB2、Sybase 或其他 DBMS 的调用。这种驱动程序也需要将某些二进制代码加载到每台客户机上。
- ☑ JDBC 网络驱动：这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，又被某个服务器转换为一种 DBMS 协议，是一种利用 Java 编写的 JDBC 驱动程序，也是最为灵活的 JDBC 驱动程序。这种方案的提供者提供了适合于企业内部互联网（Intranet）用的产品。为使这种产品支持 Internet 访问，需要处理 Web 提出的安全性、通过防火墙的访问等额外的要求。
- ☑ 本地协议驱动：这是一种纯 Java 的驱动程序。这种驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议，允许从客户机上直接调用 DBMS 服务器，是一种很实用的访问 Intranet 的解决方法。

JDBC 网络驱动和本地协议驱动是 JDBC 访问数据库的首选，这两类驱动程序提供了 Java 的所有优点。

20.3 JDBC 中常用的类和接口

在 Java 语言中提供了丰富的类和接口用于数据库编程，利用这些类和接口可以方便地进行数据访问和处理。本节将介绍一些常用的 JDBC 接口和类，这些类或接口都在 java.sql 包中。

20.3.1　Connection 接口

Connection 接口代表与特定的数据库的连接，在连接上下文中执行 SQL 语句并返回结果。Connection 接口的常用方法如表 20.1 所示。

表 20.1　Connection 接口的常用方法

方　法	功 能 描 述
createStatement()	创建 Statement 对象
createStatement(int resultSetType, int resultSetConcurrency)	创建一个 Statement 对象，该对象将生成具有给定类型、并发性和可保存性的 ResultSet 对象
preparedStatement()	创建预处理对象 preparedStatement
isReadOnly()	查看当前 Connection 对象的读取模式是否为只读形式
setReadOnly()	设置当前 Connection 对象的读写模式，默认是非只读模式
commit()	使所有上一次提交/回滚后进行的更改成为持久更改，并释放此 Connection 对象当前持有的所有数据库锁
roolback()	取消在当前事务中进行的所有更改，并释放此 Connection 对象当前持有的所有数据库锁
close()	立即释放此 Connection 对象的数据库和 JDBC 资源，而不是等待它们被自动释放

20.3.2　Statement 接口

Statement 接口用于在已经建立连接的基础上向数据库发送 SQL 语句。在 JDBC 中有 3 种 Statement 对象，分别是 Statement、PreparedStatement 和 CallableStatement。Statement 对象用于执行不带参数的简单的 SQL 语句；PreparedStatement 继承了 Statement，用来执行动态的 SQL 语句；CallableStatement 继承了 PreparedStatement，用于执行对数据库的存储过程的调用。Statement 接口的常用方法如表 20.2 所示。

表 20.2　Statement 接口中常用的方法

方　法	功 能 描 述
execute(String sql)	执行静态的 SELECT 语句，该语句可能返回多个结果集
executeQuery(String sql)	执行给定的 SQL 语句，该语句返回单个 ResultSet 对象
clearBatch()	清空此 Statement 对象的当前 SQL 命令列表
executeBatch()	将一批命令提交给数据库来执行，如果全部命令执行成功，则返回更新计数组成的数组。数组元素的排序与 SQL 语句的添加顺序对应
addBatch(String sql)	将给定的 SQL 命令添加到此 Statement 对象的当前命令列表中。如果驱动程序不支持批量处理，将抛出异常
close()	释放 Statement 实例占用的数据库和 JDBC 资源

20.3.3 PreparedStatement 接口

PreparedStatement 接口用来动态地执行 SQL 语句。通过 PreparedStatement 实例执行的动态 SQL 语句，将被预编译并保存到 PreparedStatement 实例中，从而可以反复地执行该 SQL 语句。PreparedStatement 接口的常用方法如表 20.3 所示。

表 20.3 PreparedStatement 接口提供的常用方法

方 法	功 能 描 述
setInt(int index , int k)	将指定位置的参数设置为 int 值
setFloat(int index , float f)	将指定位置的参数设置为 float 值
setLong(int index,long l)	将指定位置的参数设置为 long 值
setDouble(int index , double d)	将指定位置的参数设置为 double 值
setBoolean(int index ,boolean b)	将指定位置的参数设置为 boolean 值
setDate(int index , date date)	将指定位置的参数设置为对应的 date 值
executeQuery()	在此 PreparedStatement 对象中执行 SQL 查询，并返回该查询生成的 ResultSet 对象
setString(int index String s)	将指定位置的参数设置为对应的 String 值
setNull(int index , intsqlType)	将指定位置的参数设置为 SQL NULL
executeUpdate()	执行前面包含的参数的动态 INSERT、UPDATE 或 DELETE 语句
clearParameters()	清除当前所有参数的值

20.3.4 DriverManager 类

DriverManager 类用来管理数据库中的所有驱动程序。它是 JDBC 的管理层，作用于用户和驱动程序之间，跟踪可用的驱动程序，并在数据库的驱动程序之间建立连接。如果通过 getConnection()方法可以建立连接，则经连接返回，否则抛出 SQLException 异常。DriverManager 类的常用方法如表 20.4 所示。

表 20.4 DriverManager 类的常用方法

方 法	功 能 描 述
getConnection(String url, String user, String password)	指定 3 个入口参数（依次是连接数据库的 URL、用户名、密码）来获取与数据库的连接
setLoginTimeout()	获取驱动程序试图登录到某一数据库时可以等待的最长时间，以秒为单位
println(String message)	将一条消息打印到当前 JDBC 日志流中

20.3.5 ResultSet 接口

ResultSet 接口类似于一个临时表，用来暂时存放数据库查询操作所获得的结果集。ResultSet 实例具有指向当前数据行的指针，指针开始的位置在第一条记录的前面，通过 next()方法可将指针向下移。

在 JDBC 2.0（JDK 1.2）之后，该接口添加了一组更新方法 updateXXX()，该方法有两个重载方法，可根据列的索引号和列的名称来更新指定列。但该方法并没有将对数据进行的操作同步到数据库中，

需要执行 updateRow()或 insertRow()方法更新数据库。ResultSet 接口的常用方法如表 20.5 所示。

表 20.5　ResultSet 接口提供的常用方法

方　法	功 能 描 述
getInt()	以 int 形式获取此 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回值是 0
getFloat()	以 float 形式获取此 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回值是 0
getDate()	以 data 形式获取 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回值是 null
getBoolean()	以 boolean 形式获取 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回 null
getString()	以 String 形式获取 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回 null
getObject()	以 Object 形式获取 ResultSet 对象的当前行的指定列值。如果列值是 NULL，则返回 null
first()	将指针移到当前记录的第一行
last()	将指针移到当前记录的最后一行
next()	将指针向下移一行
beforeFirst()	将指针移到集合的开头（第一行位置）
afterLast()	将指针移到集合的尾部（最后一行位置）
absolute(int index)	将指针移到 ResultSet 给定编号的行
isFrist()	判断指针是否位于当前 ResultSet 集合的第一行。如果是返回 true，否则返回 false
isLast()	判断指针是否位于当前 ResultSet 集合的最后一行。如果是返回 true，否则返回 false
updateInt()	用 int 值更新指定列
updateFloat()	用 float 值更新指定列
updateLong()	用指定的 long 值更新指定列
updateString()	用指定的 String 值更新指定列
updateObject()	用 Object 值更新指定列
updateNull()	将指定的列值修改为 NULL
updateDate()	用指定的 date 值更新指定列
updateDouble()	用指定的 double 值更新指定列
getrow()	查看当前行的索引号
insertRow()	将插入行的内容插入到数据库
updateRow()	将当前行的内容同步到数据表
deleteRow()	删除当前行，但并不同步到数据库中，而是在执行 close()方法后同步到数据库

20.4　数据库操作

要对数据库表中的数据进行操作，应该首先建立与数据库的连接。通过 JDBC 的 API 中提供的各种类可实现对数据表中的数据进行查找、添加、修改、删除等操作。本节以操作 MySQL 数据库为例，介绍几种常见的数据库操作。

20.4.1 连接数据库

视频讲解：光盘\TM\lx\20\连接数据库.mp4

要访问数据库，首先要加载数据库的驱动程序（只需要在第一次访问数据库时加载一次），然后每次访问数据时创建一个 Connection 对象，接着执行操作数据库的 SQL 语句，最后在完成数据库操作后销毁前面创建的 Connection 对象，释放与数据库的连接。

【例 20.5】 在项目中创建类 Conn，并创建 getConnection()方法，获取与 MySQL 数据库的连接，在主方法中调用该方法。（**实例位置：光盘\TM\sl\20.01**）

```
import java.sql.*;                                  //导入 java.sql 包
public class Conn {                                 //创建类 Conn
    Connection con;                                 //声明 Connection 对象
    public Connection getConnection(){              //建立返回值为 Connection 的方法
        try {                                       //加载数据库驱动类
            Class.forName("com.mysql.jdbc.Driver");      System.out.println("数据库驱动加载成功");
        } catch (ClassNotFoundException e) {                                          (1)
            e.printStackTrace();
        }
        try {                                       //通过访问数据库的 URL 获取数据库连接对象

            con=DriverManager.getConnection("jdbc:mysql:" +
                    "//127.0.0.1:3306/test"
                            ,"root","123456");                          (2)
            System.out.println("数据库连接成功");
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return con;                                 //按方法要求返回一个 Connection 对象
    }
    public static void main(String[] args) {        //主方法
        Conn c = new Conn();                        //创建本类对象
        c.getConnection();                          //调用连接数据库的方法
    }
}
```

运行结果如图 20.3 所示。

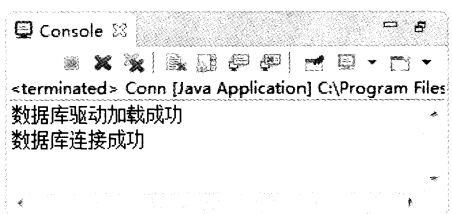


图 20.3 例 20.5 的运行结果

代码说明：

☑ 代码块（1）：通过 java.lang 包的静态方法 forName()来加载 JDBC 驱动程序，如果加载失败会抛出 ClassNotFoundException 异常。应该确定数据库驱动类是否成功加载到程序中。

☑ 代码块（2）：通过 java.sql 包中类 DriverManager 的静态方法 getConnection(String url, String user, String password)建立数据库连接。该方法的 3 个参数依次指定预连接数据库的路径、用户名和密码。返回 Connection 对象。如果连接失败，则抛出 SQLException 异常。

注意

本实例中将连接数据库作为单独的一个方法，并以 Connection 对象作为返回值。这样写的好处是在遇到对数据库执行操作的程序时可直接调用 Conn 类的 getConnection()方法获取连接，增加了代码的重用性。

20.4.2　向数据库发送 SQL 语句

例 20.5 中的 getConnection()方法只是获取与数据库的连接，要执行 SQL 语句首先要获得 Statement 类对象。通过例 20.5 创建的连接数据库对象 con 的 createStatement()方法可获得 Statement 对象。

【例 20.6】 创建 Statement 类对象 sql。代码如下：

```
try {
    Statement sql = con.createStatement();
} catch (SQLException e) {
    e.printStackTrace();
}
```

20.4.3　处理查询结果集

有了 Statement 对象以后，可调用相应的方法实现对数据库的查询和修改，并将查询的结果集存放在 ResultSet 类的对象中。

【例 20.7】 获取查询结果集。代码如下：

```
ResultSet res = sql.executeQuery("select * from tb_emp");
```

运行结果为返回一个 ResultSet 对象，ResultSet 对象一次只可以看到结果集中的一行数据，使用该类的 next()方法可将光标从当前位置移向下一行（关于 Statement 类的其他方法可参见 20.3 节）。

20.4.4 顺序查询

视频讲解：光盘\TM\lx\20\数据库查询.mp4

ResultSet 类的 next()方法的返回值是 boolean 类型的数据，当游标移动到最后一行之后会返回 false。下面的实例就是将数据表 tb_emp 中的全部信息显示在控制台上。

【例 20.8】 本实例在 getConnection()方法中获取与数据库的连接，在主方法中将数据表 tb_stu 中的数据检索出来，保存在遍历查询结果集 ResultSet 中，并遍历该结果集。（**实例位置：光盘\TM\ sl\20.02**）

```
import java.sql.*;                                    //导入 java.sql 包
public class Gradation {                              //创建类
    static Connection con;                            //声明 Connection 对象
    static Statement sql;                             //声明 Statement 对象
    static ResultSet res;                             //声明 ResultSet 对象
    public Connection getConnection() {               //连接数据库方法
        /********* 省略了获取数据库连接的代码，读者可参考 20.4.1 节的内容 ******/
        return con;                                   //返回 Connection 对象
    }
    public static void main(String[] args) {          //主方法
        Gradation c = new Gradation();                //创建本类对象
        con = c.getConnection();                      //与数据库建立连接
        try {
            sql = con.createStatement();              //实例化 Statement 对象
            //执行 SQL 语句，返回结果集
            res = sql.executeQuery("select * from tb_stu");
            while (res.next()) {                      //如果当前语句不是最后一条，则进入循环
                String id = res.getString("id");      //获取列名是 id 的字段值
                //获取列名是 name 的字段值
                String name = res.getString("name");
                //获取列名是 sex 的字段值
                String sex = res.getString("sex");
                //获取列名是 birthday 的字段值
                String birthday = res.getString("birthday");
                System.out.print("编号：" + id);      //将列值输出
                System.out.print(" 姓名:" + name);
                System.out.print(" 性别:" + sex);
                System.out.println(" 生日：" + birthday);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如图 20.4 所示。

注意

可以通过列的序号来获取结果集中指定的列值。例如，获取结果集中 id 列的列值，可以写成 getString("id")。由于 id 列是数据表中的第一列，所以也可以写成 getString(1)来获取。结果集 res 的结构如图 20.5 所示。

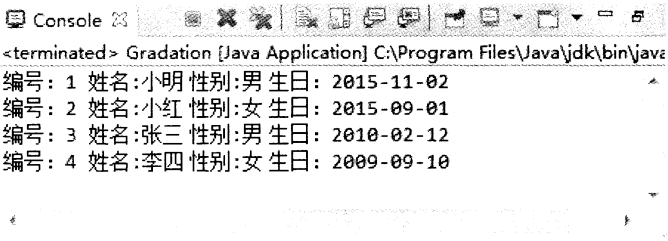


图 20.4　例 20.8 的运行结果

```
mysql> select * from tb_stu;
+----+------+-----+------------+
| id | name | sex | birthday   |
+----+------+-----+------------+
|  1 | 小明 | 男  | 2015-11-02 |
|  2 | 小红 | 女  | 2015-09-01 |
|  3 | 张三 | 男  | 2010-02-12 |
|  4 | 李四 | 女  | 2009-09-10 |
+----+------+-----+------------+
4 rows in set
```

图 20.5　结果集结构

20.4.5　模糊查询

SQL 语句中提供了 LIKE 操作符用于模糊查询，可使用“%”来代替 0 个或多个字符，使用下划线“_”来代替一个字符。例如，在查询姓张的同学的信息时，可使用以下 SQL 语句：

```
select * from tb_stu where name like '张%'
```

【例 20.9】 本实例在 getConnection()方法中获取与数据库的连接，在主方法中将数据表 tb_stu 中姓张的同学的信息检索出来，保存在 ResultSet 结果集中，并遍历该集合。（**实例位置：光盘\TM\sl\20.03**）

```
import java.sql.*;                                    //导入 java.sql 包
public class Train {                                  //创建类 Train
    static Connection con;                            //声明 Connection 对象
    static Statement sql;                             //声明 Statement 对象
    static ResultSet res;                             //声明 ResultSet 对象
    public Connection getConnection() {               //与数据库连接方法
        /******* 省略了获取数据库连接的代码，读者可参考 20.4.1 节的内容 ********/
        return con;                                   //返回 Connection 对象
    }
    public static void main(String[] args) {          //主方法
        Train c = new Train();                        //创建本类对象
        con = c.getConnection();                      //获取与数据库的连接
        try {                                         //try 语句捕捉异常
            sql = con.createStatement();              //实例化 Statement 对象
            //执行 SQL 语句
            res = sql.executeQuery("select * from tb_stu where"
                    + " name like '张%'");
            //如果当前记录不是结果集中的最后一条，进入循环体
            while (res.next()) {
                String id = res.getString(1);         //获取 id 字段值
```

```
                String name = res.getString("name");        //获取 name 字段值
                String sex = res.getString("sex");          //获取 sex 字段值
                //获取 birthday 字段值
                String birthday = res.getString("birthday");
                System.out.print("编号：" + id);             //输出信息
                System.out.print(" 姓名：" + name);
                System.out.print(" 性别:" + sex);
                System.out.println(" 生日：" + birthday);
            }
        } catch (Exception e) {                             //处理异常
            e.printStackTrace();                            //输出异常信息
        }
    }
}
```

运行结果如图 20.6 所示。

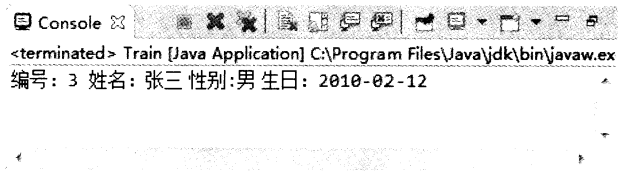


图 20.6　例 20.9 的运行结果

20.4.6　预处理语句

视频讲解：光盘\TM\lx\20\动态查询.mp4

向数据库发送一个 SQL 语句，数据库中的 SQL 解释器负责把 SQL 语句生成底层的内部命令，然后执行该命令，完成相关的数据操作。如果不断地向数据库提交 SQL 语句，肯定会增加数据库中 SQL 解释器的负担，影响执行的速度。

对于 JDBC，可以通过 Connection 对象的 preparedStatement(String sql)方法对 SQL 语句进行预处理，生成数据库底层的内部命令，并将该命令封装在 PreparedStatement 对象中，通过调用该对象的相应方法执行底层数据库命令。这样应用程序能针对连接的数据库，实现将 SQL 语句解释为数据库底层的内部命令，然后让数据库执行这个命令，这样可以减轻数据库的负担，提高访问数据库的速度。

对 SQL 进行预处理时可以使用通配符“?”来代替任何的字段值。例如：

```
sql = con.prepareStatement("select * from tb_stu where id = ?");
```

在执行预处理语句前，必须用相应方法来设置通配符所表示的值。例如：

```
sql.setInt(1,2);
```

上述语句中的“1”表示从左向右的第几个通配符，“2”表示设置的通配符的值。将通配符的值设置为 2 后，功能等同于：

```
sql = con.prepareStatement("select * from tb_stu where id = 2");
```

尽管书写两条语句看似麻烦了一些，但使用预处理语句可使应用程序更容易动态地改变 SQL 语句中关于字段值条件的设定。

注意

通过 setXXX()方法为 SQL 语句中的参数赋值时，建议利用与参数匹配的方法，也可以利用 setObject()方法为各种类型的参数赋值。例如：

```
sql.setObject(2,'李丽');
```

【例 20.10】本实例预处理语句动态地获取指定编号的同学的信息，在此以查询编号为 19 的同学的信息为例介绍预处理语句的用法。（实例位置：光盘\TM\sl\20.04）

```
import java.sql.*;                                          //导入 java.sql 包
public class Prep {                                         //创建类 Perp
    static Connection con;                                  //声明 Connection 对象
    static PreparedStatement sql;                           //声明预处理对象
    static ResultSet res;                                   //声明结果集对象
    public Connection getConnection() {                     //与数据库连接的方法
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection("jdbc:mysql:"
                    + "//127.0.0.1:3306/test", "root", "123456");          } catch (Exception e) {
            e.printStackTrace();
        }
        return con;                                         //返回 Connection 对象
    }
    public static void main(String[] args) {                //主方法
        Prep c = new Prep();                                //创建本类对象
        con = c.getConnection();                            //获取与数据库的连接
        try {
            //实例化预处理对象
            sql = con.prepareStatement("select * from tb_stu"
                    + " where id = ?");
            sql.setInt(1, 19);                              //设置参数
            res = sql.executeQuery();                       //执行预处理语句
            //如果当前记录不是结果集中的最后一行，则进入循环体
            while (res.next()) {
                String id = res.getString(1);               //获取结果集中第一列的值
                String name = res.getString("name");        //获取 name 列的列值
                String sex = res.getString("sex");          //获取 sex 列的列值
                //获取 birthday 列的列值
                String birthday = res.getString("birthday");
                System.out.print("编号：" + id);             //输出信息
                System.out.print(" 姓名：" + name);
                System.out.print(" 性别:" + sex);
                System.out.println(" 生日：" + birthday);
            }
        } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
}
```

运行结果如图 20.7 所示。

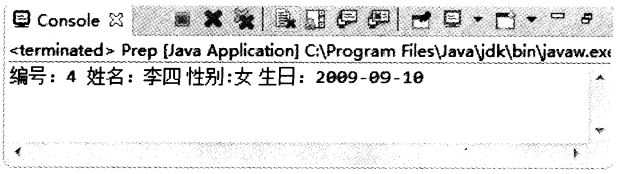


图 20.7　例 20.10 的运行结果

20.4.7　添加、修改、删除记录

视频讲解：光盘\TM\lx\20\添加、修改、删除记录.mp4

通过 SQL 语句可以对数据执行添加、修改和删除操作。可通过 PreparedStatement 类的指定参数动态地对数据表中原有数据进行修改操作，并通过 executeUpdate()方法执行更新语句操作。

【例 20.11】 本实例通过预处理语句动态地对数据表 tb_stu 中的数据执行添加、修改、删除操作，并遍历对数据进行操作之前与对数据进行操作之后的 tb_stu 表中的数据。（**实例位置：光盘\TM\sl\20.05**）

```
import java.sql.*;                                          //导入 java.sql 包
public class Renewal {                                      //创建类
    static Connection con;                                  //声明 Connection 对象
    static PreparedStatement sql;                           //声明 PreparedStatement 对象
    static ResultSet res;                                   //声明 ResultSet 对象
    public Connection getConnection() {
        /********* 省略了获取数据库连接的代码，读者可参考 20.4.1 节的内容 *******/
        return con;
    }
    public static void main(String[] args) {
        Renewal c = new Renewal();                          //创建本类对象
        con = c.getConnection();                            //调用连接数据库的方法
        try {
            sql = con.prepareStatement("select * from tb_stu");   //查询数据库
            res = sql.executeQuery();                       //执行 SQL 语句
            System.out.println("执行增加、修改、删除前数据:");
            while (res.next()) {
                String id = res.getString(1);
                String name = res.getString("name");
                String sex = res.getString("sex");
                String brithday = res.getString("birthday");      //遍历查询结果集
                System.out.print("编号：" + id);
                System.out.print(" 姓名：" + name);
                System.out.print(" 性别:" + sex);
                System.out.println(" 生日：" + birthday);
```

```
            }
            sql = con.prepareStatement("insert into tb_stu"
                    + " values(?,?,?)");
            sql.setString(1, "张一");
            sql.setString(2, "女");                                    //预处理添加数据
            sql.setString(3, "2012-12-1");
            sql.executeUpdate();
            sql = con.prepareStatement("update tb_stu set birthday "
                    + "= ? where id = (select min(id) from tb_stu)");
            sql.setString(1, " 2012-12-02");                           //更新数据
            sql.executeUpdate();
            stmt.executeUpdate("delete from tb_stu where id = "
                    + "(select min(id)from tb_stu)")
            sql.setInt(1, 1);
            sql.executeUpdate();                                       //删除数据
            //查询修改数据后 tb_stu 表中的数据
            sql = con.prepareStatement("select * from tb_stu");
            res = sql.executeQuery();                                  //执行 SQL 语句
            System.out.println("执行增加、修改、删除后的数据:");
            while (res.next()) {
                /************** 省略了数据修改之后遍历结果集的代码 **********/
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如图 20.8 所示。

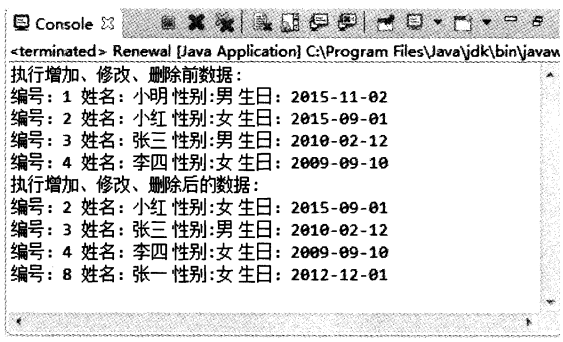


图 20.8　例 20.11 的运行结果

说明

executeQuery()方法是在 PreparedStatement 对象中执行 SQL 查询，并返回该查询生成的 ResultSet 对象，而 executeUpdate()方法是在 PreparedStatement 对象中执行 SQL 语句，该语句必须是一个 SQL 数据操作语言（Data Manipulation Language，DML）语句，如 INSERT、UPDATE 或 DELETE 语句，或者是无返回内容的 SQL 语句，如 DDL 语句。

20.5 小　　结

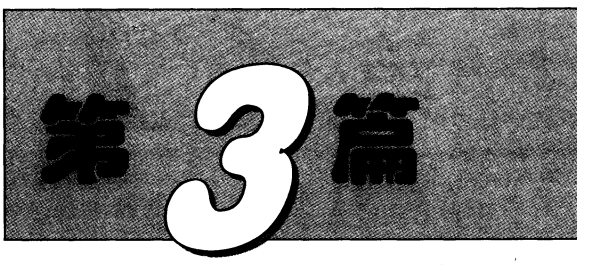
本章主要介绍了 Java 程序中的数据库操作部分。通过对本章的学习，读者应该了解什么是数据库，数据库的种类、功能以及常用的 SQL 语言的基本语法，重点要掌握使用 JDBC 技术操作数据库，以及对数据执行增加、删除、修改、查找操作的方法。

20.6 实践与练习

1．创建类 SearchEmp，实现查找数据表 tb_emp 中销售部的所有成员的功能。（**答案位置：光盘\TM\sl\20.06**）

2．编写程序，实现向数据表 tb_stu 中添加数据的功能，要求姓名为“李某”，性别是“女”，出生日期是“1999-10-20”。（**答案位置：光盘\TM\sl\20.07**）

3．编写程序，实现删除出生日期在“2010-01-01”之前的学生的功能。（**答案位置：光盘\TM\sl\20.08**）



高级应用

本篇介绍了 Swing 表格组件、Swing 树组件、Swing 其他高级组件、高级布局管理器、高级事件处理、AWT 绘图与音频播放、打印技术等内容。学习完本篇，读者将能够开发高级的桌面应用程序、多媒体程序和打印程序等。

第21章

Swing 表格组件

（视频讲解：20 分钟）

表格是最常用的数据统计形式之一，在日常生活中经常需要使用表格统计数据，如对销售数据的统计、日常开销的统计，以及生成员工待遇报表等。本章将介绍 Swing 表格的使用方法，在最后还将讲解提供行标题栏表格的实现思路和方法。在讲解过程中，为了便于读者理解结合了大量的实例。

通过阅读本章，您可以：

- 学会创建 Swing 表格的方法
- 学会定义和操纵表格的常用方法
- 学会维护表格模型的常用方法
- 掌握 JTable 和 DefaultTableModel 类的主要功能
- 掌握提供行标题栏表格的开发思路
- 了解 Swing 表格的设计思路

21.1　利用 JTable 类直接创建表格

表格是最常用的数据统计形式之一，在 Swing 中由 JTable 类实现表格。本节将学习利用 JTable 类创建和定义表格，以及操纵表格的方法。

21.1.1　创建表格

视频讲解：光盘\TM\lx\21\创建表格.exe

在 JTable 类中除了默认的构造方法外，还提供了利用指定表格列名数组和表格数据数组创建表格的构造方法，代码如下：

```
JTable(Object[][] rowData, Object[] columnNames)
```

☑　rowData：封装表格数据的数组。

☑　columnNames：封装表格列名的数组。

在使用表格时，通常将其添加到滚动面板中，然后将滚动面板添加到相应的位置。下面看一个这样的例子。

【例 21.1】 创建可以滚动的表格。（**实例位置：光盘\TM\sl\21.01**）

本例利用构造方法 JTable(Object[][] rowData, Object[] columnNames)创建了一个表格，并将表格添加到了滚动面板中。本例的完整代码如下：

```
import java.awt.*;
import javax.swing.*;
public class ExampleFrame_01 extends JFrame {
    public static void main(String args[]) {
        ExampleFrame_01 frame = new ExampleFrame_01();
        frame.setVisible(true);
    }
    public ExampleFrame_01() {
        super();
        setTitle("创建可以滚动的表格");
        setBounds(100, 100, 240, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        String[] columnNames = { "A", "B" }; //定义表格列名数组
        //定义表格数据数组
        String[][] tableValues = { { "A1", "B1" }, { "A2", "B2" },
                { "A3", "B3" }, { "A4", "B4" }, { "A5", "B5" } };
        //创建指定列名和数据的表格
        JTable table = new JTable(tableValues, columnNames);
        //创建显示表格的滚动面板
        JScrollPane scrollPane = new JScrollPane(table);
        //将滚动面板添加到边界布局的中间
```

```
        getContentPane().add(scrollPane, BorderLayout.CENTER);
    }
}
```

运行本实例，将得到如图 21.1 所示的窗体；当调小窗体的高度时，将出现滚动条，如图 21.2 所示。

图 21.1　创建可以滚动的表格

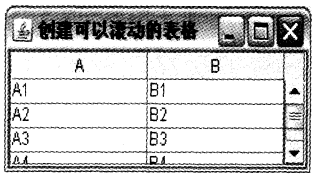


图 21.2　出现滚动条的表格

在 JTable 类中还提供了利用指定表格列名向量和表格数据向量创建表格的构造方法，代码如下：

```
JTable(Vector rowData, Vector columnNames)
```

☑ rowData：封装表格数据的向量。

☑ columnNames：封装表格列名的向量。

在使用表格时，有时并不需要使用滚动条，即在窗体中可以显示出整个表格，在这种情况下，也可以直接将表格添加到相应的容器中。

注意

如果是直接将表格添加到相应的容器中，则首先需要通过 JTable 类的 getTableHeader()方法获得 JTableHeader 类的对象，然后再将该对象添加到容器的相应位置，否则表格将没有列名。

【例 21.2】 创建不可滚动的表格。（实例位置：光盘\TM\sl\21.02）

本例利用构造方法 JTable(Vector rowData, Vector columnNames)创建了一个表格，并将表格直接添加到了容器中。本例的关键代码如下：

```
public class ExampleFrame_02 extends JFrame {
    public static void main(String args[]) {
        ExampleFrame_02 frame = new ExampleFrame_02();
        frame.setVisible(true);
    }
    public ExampleFrame_02() {
        super();
        setTitle("创建不可滚动的表格");
        setBounds(100, 100, 240, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Vector<String> columnNameV = new Vector<>();          //定义表格列名向量
        columnNameV.add("A");                                 //添加列名
        columnNameV.add("B");                                 //添加列名
        Vector<Vector<String>> tableValueV = new Vector<>();  //定义表格数据向量
        for (int row = 1; row < 6; row++) {
            Vector<String> rowV = new Vector<>();             //定义表格行向量
            rowV.add("A" + row);                              //添加单元格数据
```

```
            rowV.add("B" + row);                                    //添加单元格数据
            tableValueV.add(rowV);                                  //将表格行向量添加到表格数据向量中
        }
        //创建指定表格列名和表格数据的表格
        JTable table = new JTable(tableValueV, columnNameV);
        //将表格添加到边界布局的中间
        getContentPane().add(table, BorderLayout.CENTER);
        JTableHeader tableHeader = table.getTableHeader();      //获得表格头对象
        //将表格头添加到边界布局的上方
        getContentPane().add(tableHeader, BorderLayout.NORTH);
    }
}
```

运行本例，将得到如图 21.3 所示的窗体；当调小窗体的高度时，不会出现滚动条，如图 21.4 所示。如果将上面代码中的最后两行去掉，再次运行本例，将得到如图 21.5 所示的窗体，会发现表格没有列名。

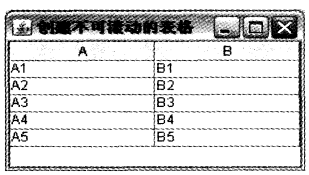


图 21.3　创建不可滚动的表格

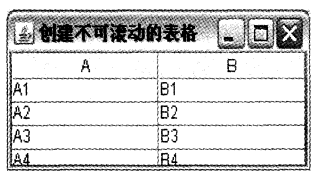


图 21.4　不出现滚动条的表格

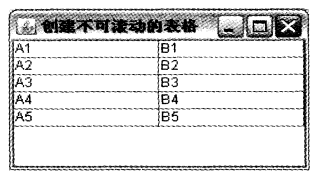


图 21.5　没有表格列名的表格

21.1.2　定制表格

视频讲解：光盘\TM\lx\21\定制表格.exe

表格创建完成后，还需要对其进行一系列的定义，以便适合于具体的使用情况。默认情况下通过双击表格中的单元格就可以对其进行编辑，如图21.6所示。如果不需要提供该功能，可以通过重构JTable类的 isCellEditable(int row, int column)方法实现。默认情况下该方法返回 boolean 型值 true，表示指定单元格可编辑，如果返回 false 则表示不可编辑。

如果表格只有几列，通常不需要表格列的可重新排列功能。在创建不支持滚动条的表格时已经使用了 JTableHeader 类的对象，通过该类的 setReorderingAllowed(boolean reorderingAllowed)方法即可设置表格是否支持重新排列功能，设为 false 表示不支持重新排列功能，如图 21.7 所示。

默认情况下单元格中的内容靠左侧显示，如果需要令单元格中的内容居中显示，如图 21.8 所示，可以通过重构 JTable 类的 getDefaultRenderer(Class<?> columnClass)方法来实现。下面是重构后的代码：

```
public TableCellRenderer getDefaultRenderer(Class<?> columnClass) {
    DefaultTableCellRenderer cr = (DefaultTableCellRenderer)
                                        super.getDefaultRenderer(columnClass);
    cr.setHorizontalAlignment(DefaultTableCellRenderer.CENTER);
    return cr;
}
```

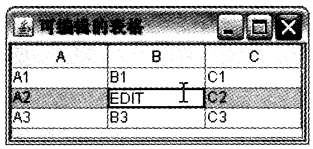


图 21.6　可编辑的表格

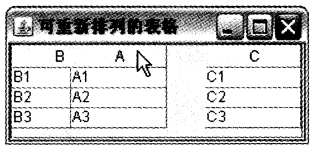


图 21.7　可重新排列的表格

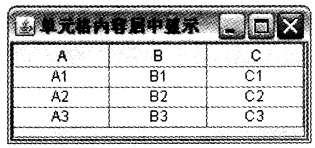


图 21.8　单元格内容居中显示

表 21.1 中列出了 JTable 类中用来定义表格的常用方法。

表 21.1　JTable 类中用来定义表格的常用方法

方　法	说　明
setRowHeight(int rowHeight)	设置表格的行高，默认为 16 像素
setRowSelectionAllowed(boolean sa)	设置是否允许选中表格行，默认为允许选中，设为 false 表示不允许选中
setSelectionMode(int sm)	设置表格行的选择模式
setSelectionBackground(Color bc)	设置表格选中行的背景色
setSelectionForeground(Color fc)	设置表格选中行的前景色（通常情况下为文字的颜色）
setAutoResizeMode(int mode)	设置表格的自动调整模式

在利用 setSelectionMode(int sm)方法设置表格行的选择模式时，它的入口参数可以从表 21.2 列出的 ListSelectionModel 类的静态常量中选择。

表 21.2　ListSelectionModel 类中用来设置选择模式的静态常量

静 态 常 量	常 量 值	代表的选择模式
SINGLE_SELECTION	0	只允许选择一个，如图 21.9 所示
SINGLE_INTERVAL_SELECTION	1	允许连续选择多个，如图 21.10 所示
MULTIPLE_INTERVAL_SELECTION	2	可以随意选择多个，如图 21.11 所示

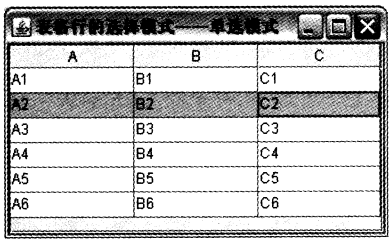


图 21.9　单选模式

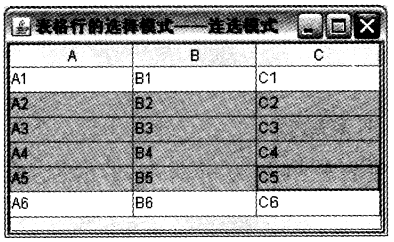


图 21.10　连选模式

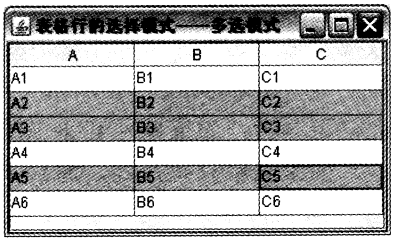


图 21.11　多选模式

在利用 setAutoResizeMode(int mode)方法设置表格的自动调整模式时，它的入口参数可以从表 21.3 列出的 JTable 类的静态常量中选择。

说明

所谓表格的自动调整模式，就是在调整表格某一列的宽度时，表格采用何种方式保持其总宽度不变。

表 21.3　JTable 类中用来设置自动调整模式的静态常量

静 态 常 量	常　量　值	代表的自动调整模式
AUTO_RESIZE_OFF	0	关闭自动调整功能，使用水平滚动条时的必要设置，如图 21.12 所示
AUTO_RESIZE_NEXT_COLUMN	1	只调整其下一列的宽度，如图 21.13 所示
AUTO_RESIZE_SUBSEQUENT_COLUMNS	2	按比例调整其后所有列的宽度，为默认设置，如图 21.14 所示
AUTO_RESIZE_LAST_COLUMN	3	只调整最后一列的宽度，如图 21.15 所示
AUTO_RESIZE_ALL_COLUMNS	4	按比例调整表格所有列的宽度，如图 21.16 所示

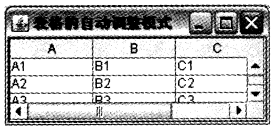


图 21.12　关闭调整功能

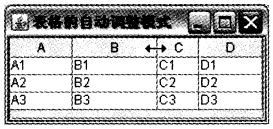


图 21.13　只调整其下一列的宽度

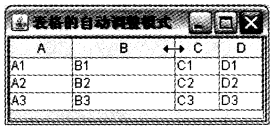


图 21.14　按比例调整其后所有列的宽度

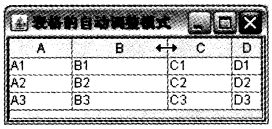


图 21.15　只调整最后一列的宽度

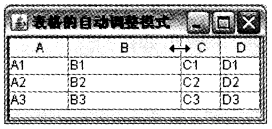


图 21.16　按比例调整表格所有列的宽度

说明

调整表格所在窗体的宽度时，如果关闭了表格的自动调整功能，表格的总宽度仍保持不变，如图 21.17 所示；如果开启了表格的自动调整功能，表格将按比例调整所有列的宽度至适合窗体的宽度，如图 21.18 所示。

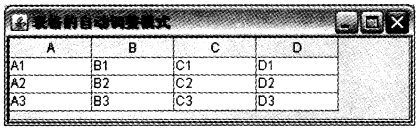


图 21.17　关闭的情况下调整窗体宽度

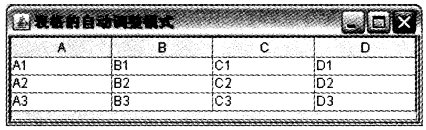


图 21.18　开启的情况下调整窗体宽度

【例 21.3】 定义表格。（**实例位置：光盘\TM\sl\21.03**）

本例利用本节所讲的全部知识对表格进行了定义，完整代码如下：

```
public class ExampleFrame_03 extends JFrame {
    public static void main(String args[]) {
        ExampleFrame_03 frame = new ExampleFrame_03();
        frame.setVisible(true);
    }
    public ExampleFrame_03() {
        super();
        setTitle("定义表格");
        setBounds(100, 100, 500, 375);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        final JScrollPane scrollPane = new JScrollPane();
        getContentPane().add(scrollPane, BorderLayout.CENTER);
        String[] columnNames = { "A", "B", "C", "D", "E", "F", "G" };
        Vector<String> columnNameV = new Vector<>();
        for (int column = 0; column < columnNames.length; column++) {
            columnNameV.add(columnNames[column]);
        }
        Vector<Vector<String>> tableValueV = new Vector<>();
        for (int row = 1; row < 21; row++) {
            Vector<String> rowV = new Vector<String>();
            for (int column = 0; column < columnNames.length; column++) {
                rowV.add(columnNames[column] + row);
            }
            tableValueV.add(rowV);
        }
        JTable table = new MTable(tableValueV, columnNameV);
        //关闭表格列的自动调整功能
        table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        //选择模式为单选
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        //被选择行的背景色为黄色
        table.setSelectionBackground(Color.YELLOW);
        //被选择行的前景色（文字颜色）为红色
        table.setSelectionForeground(Color.RED);
        table.setRowHeight(30);                                    //表格的行高为 30 像素
        scrollPane.setViewportView(table);
    }
    private class MTable extends JTable {                          //实现自己的表格类
        public MTable(Vector<Vector<String>> rowData, Vector<String> columnNames) {
            super(rowData, columnNames);
        }
        @Override
        public JTableHeader getTableHeader() {                     //定义表格头
            //获得表格头对象
            JTableHeader tableHeader = super.getTableHeader();
            tableHeader.setReorderingAllowed(false);               //设置表格列不可重排
            DefaultTableCellRenderer hr = (DefaultTableCellRenderer)
                    tableHeader.getDefaultRenderer();              //获得表格头的单元格对象
            //设置列名居中显示
            hr.setHorizontalAlignment(DefaultTableCellRenderer.CENTER);
            return tableHeader;
        }
        //定义单元格
        @Override
        public TableCellRenderer getDefaultRenderer(Class<?> columnClass) {
            DefaultTableCellRenderer cr = (DefaultTableCellRenderer) super
                    .getDefaultRenderer(columnClass);              //获得表格的单元格对象
            //设置单元格内容居中显示
            cr.setHorizontalAlignment(DefaultTableCellRenderer.CENTER);
            return cr;
        }
        @Override
```

```
        public boolean isCellEditable(int row, int column) { //表格不可编辑
            return false;
        }
    }
}
```

运行本例，选中表格的第 2 行，将得到如图 21.19 所示的效果。选中行的背景色为黄色，文字颜色为红色，并且所有单元格的内容均居中显示。

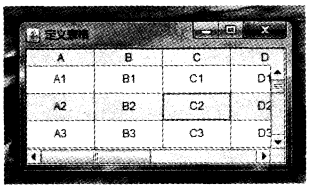


图 21.19　定义表格

21.1.3　操纵表格

视频讲解：光盘\TM\lx\21\操纵表格.exe

在编写应用表格的程序时，经常需要获得表格的一些信息，如表格拥有的行数和列数。下面是 JTable 类中 3 个经常用来获得表格信息的方法。

☑ getRowCount()：获得表格拥有的行数，返回值为 int 型。

☑ getColumnCount()：获得表格拥有的列数，返回值为 int 型。

☑ getColumnName(int column)：获得位于指定索引位置的列的名称，返回值为 String 型。

在表 21.4 中列出了经常用来操纵表格选中行的方法，包括设置、查看、统计、获取和取消选中行的方法。

表 21.4　JTable 类中经常用来操纵表格选中行的方法

方　法	说　明
setRowSelectionInterval(int from, int to)	选中行索引从 from 到 to 的所有行（包括索引为 from 和 to 的行）
addRowSelectionInterval(int from, int to)	将行索引从 from 到 to 的所有行追加为表格的选中行
isRowSelected(int row)	查看行索引为 row 的行是否被选中
selectAll()	选中表格中的所有行
clearSelection()	取消所有选中行的选择状态
getSelectedRowCount()	获得表格中被选中行的数量，返回值为 int 型，如果没有被选中的行，则返回-1
getSelectedRow()	获得被选中行中最小的行索引值，返回值为 int 型，如果没有被选中的行，则返回-1
getSelectedRows()	获得所有被选中行的索引值，返回值为 int 型数组

注意

由 JTable 类实现的表格的行索引和列索引均从 0 开始，即第一行的索引为 0，第二行的索引为 1，依此类推。

在 JTable 类中还提供了一个用来移动表格列位置的方法 moveColumn(int column, int targetColumn)，其中 column 为欲移动列的索引值，targetColumn 为目的列的索引值。移动表格列的具体执行方式如图 21.20 所示。

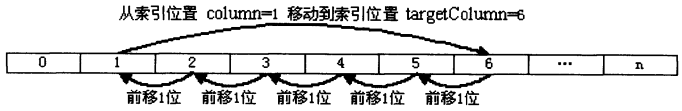


图 21.20　移动表格列的具体执行方式

【例 21.4】 操纵表格。（实例位置：光盘\TM\sl\21.04）

本例展示了本节讲到的所有方法的功能。关键代码如下：

```
table = new JTable(tableValueV, columnNameV);
table.setRowSelectionInterval(1, 3);                    //设置选中行
table.addRowSelectionInterval(5, 5);                    //添加选中行
scrollPane.setViewportView(table);
JPanel buttonPanel = new JPanel();
getContentPane().add(buttonPanel, BorderLayout.SOUTH);
JButton selectAllButton = new JButton("全部选择");
selectAllButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        table.selectAll();                              //选中所有行
    }
});
buttonPanel.add(selectAllButton);
JButton clearSelectionButton = new JButton("取消选择");
clearSelectionButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        table.clearSelection();                         //取消所有选中行的选择状态
    }
});
buttonPanel.add(clearSelectionButton);
System.out.println("表格共有" + table.getRowCount() + "行"
        + table.getColumnCount() + "列");
System.out.println("共有" + table.getSelectedRowCount() + "行被选中");
System.out.println("第 3 行的选择状态为：" + table.isRowSelected(2));
System.out.println("第 5 行的选择状态为：" + table.isRowSelected(4));
System.out.println("被选中的第一行的索引是：" + table.getSelectedRow());
int[] selectedRows = table.getSelectedRows();           //获得所有被选中行的索引
System.out.print("所有被选中行的索引是：");
for (int row = 0; row < selectedRows.length; row++) {
    System.out.print(selectedRows[row] + "  ");
}
System.out.println();
System.out.println("列移动前第 2 列的名称是：" + table.getColumnName(1));
System.out.println("列移动前第 2 行第 2 列的值是：" + table.getValueAt(1, 1));
table.moveColumn(1, 5);                                 //将位于索引 1 的列移动到索引 5 处
System.out.println("列移动后第 2 列的名称是：" + table.getColumnName(1));
System.out.println("列移动后第 2 行第 2 列的值是：" + table.getValueAt(1, 1));
```

运行本例，将得到如图 21.21 所示的窗体，其中表格的第 2、3、4、6 行被选中，并且列名为 B 的列从索引 1 处移动到了索引 5 处。单击“全部选择”按钮将选中表格的所有行，单击“取消选择”按钮将取消所有选中行的选择状态。运行本例后在控制台将输出如图 21.22 所示的信息。

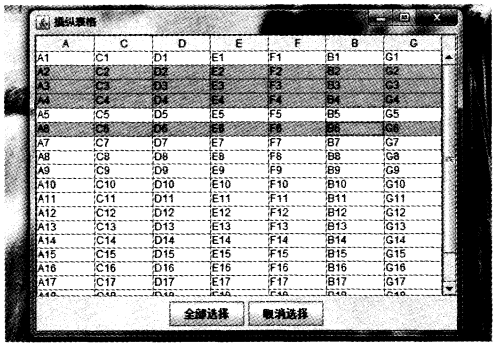


图 21.21　被选中指定行的表格

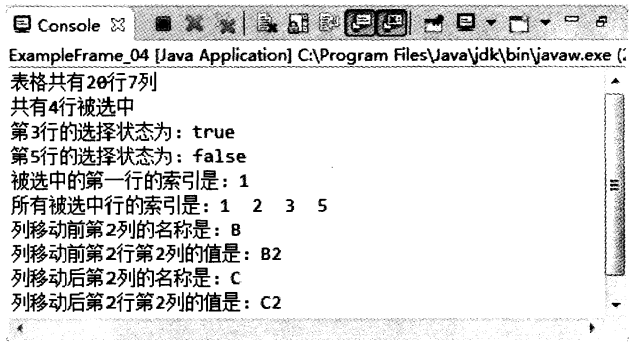


图 21.22　输出到控制台的信息

21.2　表格模型与表格

用来创建表格的 JTable 类并不负责存储表格中的数据，而是由表格模型负责存储。当利用 JTable 类直接创建表格时，只是将数据封装到了默认的表格模型中。本节将学习表格模型的使用方法。

21.2.1　利用表格模型创建表格

视频讲解：光盘\TM\lx\21\利用表格模型创建表格.exe

接口 TableModel 定义了一个表格模型，抽象类 AbstractTableModel 实现了 TableModel 接口的大部分方法，只有以下 3 个抽象方法没有实现。

- ☑ public int getRowCount()。
- ☑ public int getColumnCount()。
- ☑ public Object getValueAt(int rowIndex, int columnIndex)。

通过继承 AbstractTableModel 类实现上面 3 个抽象方法可以创建自己的表格模型类。DefaultTableModel 类便是由 Swing 提供的继承了 AbstractTableModel 类并实现了上面 3 个抽象方法的表格模型类。DefaultTableModel 类提供的常用构造方法如表 21.5 所示。

表 21.5　DefaultTableModel 类提供的常用构造方法

构 造 方 法	说　明
DefaultTableModel()	创建一个 0 行 0 列的表格模型
DefaultTableModel(int rowCount, int columnCount)	创建一个 rowCount 行 columnCount 列的表格模型
DefaultTableModel(Object[][] data, Object[] columnNames)	按照数组中指定的数据和列名创建一个表格模型
DefaultTableModel(Vector data, Vector columnNames)	按照向量中指定的数据和列名创建一个表格模型

表格模型创建完成后，通过 JTable 类的构造方法 JTable(TableModel dm)创建表格，就实现了利用表格模型创建表格。

从 JDK 1.6 开始，提供了对表格进行排序的功能。通过 JTable 类的 setRowSorter(RowSorter<? extends TableModel> sorter)方法可以为表格设置排序器。TableRowSorter 类是由 Swing 提供的排序器类。为表格设置排序器的典型代码如下：

```
DefaultTableModel tableModel = new DefaultTableModel();      //创建表格模型
JTable table = new JTable(tableModel);                       //创建表格
table.setRowSorter(new TableRowSorter(tableModel));          //设置排序器
```

如果为表格设置了排序器，当单击表格的某一列头时，在该列名称的后面将出现▲标记，说明按该列升序排列表格中的所有行，如图 21.23 所示；当再次单击该列头时，标记将变为▼，说明按该列降序排列表格中的所有行，如图 21.24 所示。

注意

在使用表格排序器时，通常要为其设置表格模型，否则将出现如图 21.25 所示的效果。一种方法是通过构造方法 TableRowSorter(TableModel model)创建排序器；另一种方法是通过 setModel (TableModel model)方法为排序器设置表格模型。

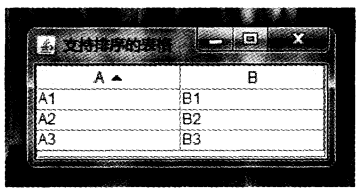


图 21.23　按升序排列

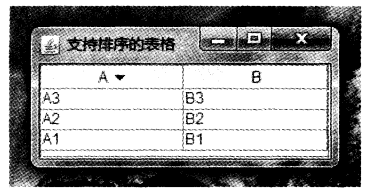


图 21.24　按降序排列

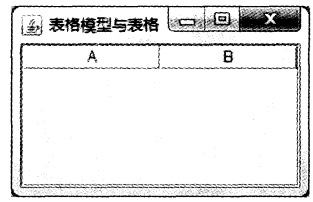


图 21.25　未设置表格模型

【例 21.5】 利用表格模型创建表格，并使用表格排序器。（**实例位置：光盘\TM\sl\21.05**）

本例利用表格模型创建了一个表格，并对表格使用了表格排序器。本例的关键代码如下：

```
JScrollPane scrollPane = new JScrollPane();
getContentPane().add(scrollPane, BorderLayout.CENTER);
String[] columnNames = { "A", "B" };          //定义表格列名数组
String[][] tableValues = { { "A1", "B1" }, { "A2", "B2" },
      { "A3", "B3" } };                       //定义表格数据数组
DefaultTableModel tableModel = new DefaultTableModel(tableValues,
      columnNames);                           //创建指定表格列名和表格数据的表格模型
JTable table = new JTable(tableModel);        //创建指定表格模型的表格
table.setRowSorter(new TableRowSorter<>(tableModel));
scrollPane.setViewportView(table);
```

运行本例，将得到如图 21.26 所示的窗体；单击名称为 B 列的列头后，将得到如图 21.27 所示的效果，表格按 B 列升序排列；再次单击名称为 B 列的列头后，将得到如图 21.28 所示的效果，表格按 B 列降序排列。

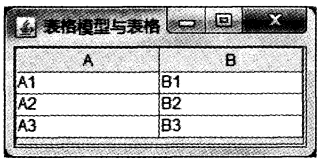


图 21.26　运行效果

图 21.27　升序排列

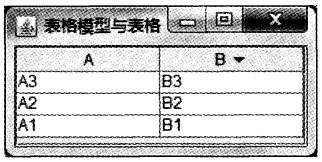


图 21.28　降序排列

21.2.2　维护表格模型

视频讲解：光盘\TM\lx\21\维护表格模型.exe

使用表格时，经常需要对表格中的内容进行维护，如向表格中添加新的数据行、修改表格中某一单元格的值、从表格中删除指定的数据行等，这些操作均可以通过维护表格模型来完成。

在向表格模型中添加新的数据行时有两种情况：一种是添加到表格模型的尾部，另一种是添加到表格模型的指定索引位置。

（1）添加到表格模型的尾部，可以通过 addRow()方法完成。它的两个重载方法如下。

☑ addRow(Object[] rowData)：将由数组封装的数据添加到表格模型的尾部。

☑ addRow(Vector rowData)：将由向量封装的数据添加到表格模型的尾部。

（2）添加到表格模型的指定位置，可以通过 insertRow()方法完成。它的两个重载方法如下。

☑ insertRow(int row, Object[] rowData)：将由数组封装的数据添加到表格模型的指定索引位置。

☑ insertRow(int row, Vector rowData)：将由向量封装的数据添加到表格模型的指定索引位置。

如果需要修改表格模型中某一单元格的数据，可以通过方法 setValueAt(Object aValue, int row, int column)完成，其中 aValue 为单元格修改后的值，row 为单元格所在行的索引，column 为单元格所在列的索引；可以通过方法 getValueAt(int row, int column)获得指定单元格的值，该方法的返回值类型为 Object。

如果需要删除表格模型中某一行的数据，可以通过方法 removeRow(int row)完成，其中 row 为欲删除行的索引。

注意

在删除表格模型中的数据时，每删除一行，其后所有行的索引值将相应地减 1，所以当连续删除多行时，需要注意对删除行索引的处理。

【例 21.6】 维护表格模型。（**实例位置：光盘\TM\sl\21.06**）

本例通过维护表格模型，实现了向表格中添加新的数据行、修改表格中某一单元格的值，以及从表格中删除指定的数据行。本例的完整代码如下：

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
public class ExampleFrame_06 extends JFrame {
    private DefaultTableModel tableModel;   //定义表格模型对象
    private JTable table;                   //定义表格对象
```

```
private JTextField aTextField;
private JTextField bTextField;
public static void main(String args[]) {
    ExampleFrame_06 frame = new ExampleFrame_06();
    frame.setVisible(true);
}
public ExampleFrame_06() {
    super();
    setTitle("维护表格模型");
    setBounds(100, 100, 510, 375);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    final JScrollPane scrollPane = new JScrollPane();
    getContentPane().add(scrollPane, BorderLayout.CENTER);
    String[] columnNames = { "A", "B" };                          //定义表格列名数组
    String[][] tableValues = { { "A1", "B1" }, { "A2", "B2" },
            { "A3", "B3" } };                                     //定义表格数据数组
    //创建指定表格列名和表格数据的表格模型
    tableModel = new DefaultTableModel(tableValues, columnNames);

    table = new JTable(tableModel);                               //创建指定表格模型的表格
    table.setRowSorter(new TableRowSorter<>(tableModel));         //设置表格的排序器
    //设置表格的选择模式为单选
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    //为表格添加鼠标事件监听器
    table.addMouseListener(new MouseAdapter() {
        //发生了单击事件
        public void mouseClicked(MouseEvent e) {
            //获得被选中行的索引
            int selectedRow = table.getSelectedRow();
            //从表格模型中获得指定单元格的值
            Object oa = tableModel.getValueAt(selectedRow, 0);
            //从表格模型中获得指定单元格的值
            Object ob = tableModel.getValueAt(selectedRow, 1);
            aTextField.setText(oa.toString());                    //将值赋值给文本框
            bTextField.setText(ob.toString());                    //将值赋值给文本框
        }
    });
    scrollPane.setViewportView(table);
    final JPanel panel = new JPanel();
    getContentPane().add(panel, BorderLayout.SOUTH);
    panel.add(new JLabel("A："));
    aTextField = new JTextField("A4", 10);
    panel.add(aTextField);
    panel.add(new JLabel("B："));
    bTextField = new JTextField("B4", 10);
    panel.add(bTextField);
    final JButton addButton = new JButton("添加");
    addButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String[] rowValues = { aTextField.getText(),
```

```
                    bTextField.getText() };                    //创建表格行数组
                tableModel.addRow(rowValues);                  //向表格模型中添加一行
                int rowCount = table.getRowCount() + 1;
                aTextField.setText("A" + rowCount);
                bTextField.setText("B" + rowCount);
            }
        });
        panel.add(addButton);
        final JButton updButton = new JButton("修改");
        updButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int selectedRow = table.getSelectedRow();      //获得被选中行的索引
                if (selectedRow != -1) {                       //判断是否存在被选中行
                    tableModel.setValueAt(aTextField.getText(),
                            selectedRow, 0);                   //修改表格模型中的指定值
                    tableModel.setValueAt(bTextField.getText(),
                            selectedRow, 1);                   //修改表格模型中的指定值
                }
            }
        });
        panel.add(updButton);
        final JButton delButton = new JButton("删除");
        delButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int selectedRow = table.getSelectedRow();      //获得被选中行的索引
                if (selectedRow != -1)                         //判断是否存在被选中行
                    //从表格模型当中删除指定行
                    tableModel.removeRow(selectedRow);
            }
        });
        panel.add(delButton);
    }
}
```

运行本例，将得到如图 21.29 所示的窗体，其中 A、B 文本框分别用来编辑 A、B 列的信息。单击“添加”按钮可以将编辑好的信息添加到表格中，选中表格的某一行后，在 A、B 文本框中将显示该行对应列的信息。重新编辑后单击“修改”按钮可以修改表格中的信息，单击“删除”按钮可以删除表格中被选中的行。

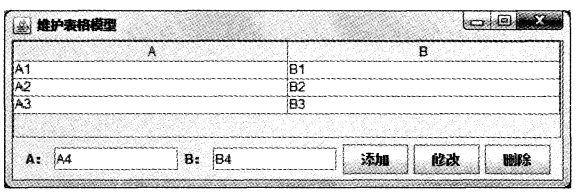


图 21.29　维护表格模型

21.3　提供行标题栏的表格

视频讲解：光盘\TM\lx\21\提供行标题栏的表格.exe

通过 JTable 类创建的表格的列标题栏是永远可见的，即使是向下滚动了垂直滚动条，这就大大增

强了表格的可读性。但是当窗体不能显示出表格的所有列时，如果向右滚动水平滚动条则会导致表格左侧的部分列不可见，而通常情况下表格左侧的一列或几列为表格的基本数据，如图 21.30 所示。如果通过移动滚动条查看未显示出的列数据时，则会导致如图 21.31 所示的效果，即不知道每一行的具体销售日期，但是针对表格列则不会出现这样的问题。如果能够使表格左侧的一列或几列不随着水平滚动条滚动，也能够永远可见，就解决了上面的问题。

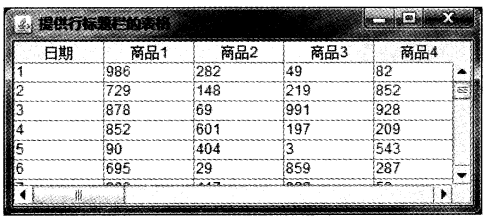


图 21.30　表格左侧的一列为表格的基本数据

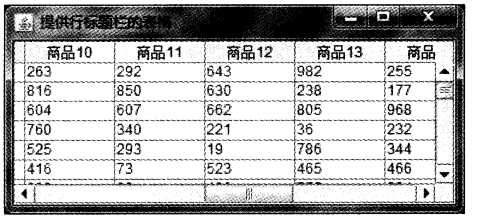


图 21.31　移动滚动条查看未显示出的列数据

可以通过两个并列显示的表格实现这样的效果，其中左侧的表格用来显示永远可见的一列或几列，右侧的表格则用来显示其他的表格列。下面来看一个实现该效果的例子。

【例 21.7】 提供行标题栏的表格。（**实例位置：光盘\TM\sl\21.07**）

本例实现了一个提供行标题栏的表格，运行本例后将得到如图 21.32 所示的窗体，在表格最左侧的“日期”列下方并没有滚动条，移动水平滚动条后将得到如图 21.33 所示的效果，表格最左侧的“日期”列仍然可见。

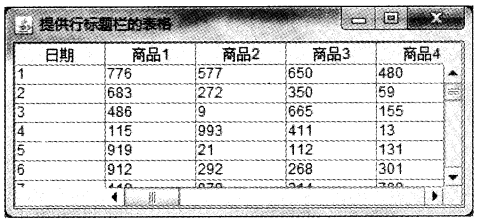


图 21.32　提供行标题栏的表格

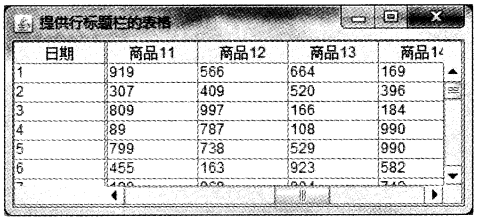


图 21.33　移动滚动条后的效果

实现本例的基本步骤如下：

（1）创建 MfixedColumnTable 类，该类继承了 JPanel 类，并声明 3 个属性。具体代码如下：

```
public class MfixedColumnTable extends JPanel {
    private Vector<String> columnNameV;                    //表格列名数组
    private Vector<Vector<Object>> tableValueV;            //表格数据数组
    private int fixedColumn = 1;                           //固定列数量
}
```

（2）创建用于左侧固定列表格的模型类 FixedColumnTableModel，该类继承了 AbstractTableModel 类，并且为 MfixedColumnTable 类的内部类。FixedColumnTableModel 类除了需要实现 AbstractTableModel 类的 3 个抽象方法外，还需要重构 getColumnName(int columnIndex)方法。具体代码如下：

```
private class FixedColumnTableModel extends AbstractTableModel {
    public int getColumnCount() {                          //返回固定列的数量
        return fixedColumn;
```

```
    }
    public int getRowCount() {                          //返回行数
        return tableValueV.size();
    }
    //返回指定单元格的值
    public Object getValueAt(int rowIndex, int columnIndex) {
        return tableValueV.get(rowIndex).get(columnIndex);
    }
    public String getColumnName(int columnIndex) {    //返回指定列的名称
        return columnNameV.get(columnIndex);
    }
}
```

（3）创建用于右侧可移动列表格的模型类 FloatingColumnTableModel，该类继承了 AbstractTableModel 类，并且为 MfixedColumnTable 类的内部类。FixedColumnTableModel 类除了需要实现 AbstractTableModel 类的 3 个抽象方法外，还需要重构 getColumnName(int columnIndex)方法。具体代码如下：

```
private class FloatingColumnTableModel extends AbstractTableModel {
    public int getColumnCount() {                       //返回可移动列的数量
        return columnNameV.size() - fixedColumn;      //需要扣除固定列的数量
    }
    public int getRowCount() {                          //返回行数
        return tableValueV.size();
    }
    //返回指定单元格的值
    public Object getValueAt(int rowIndex, int columnIndex) {
        //为列索引加上固定列的数量
        return tableValueV.get(rowIndex).get(columnIndex + fixedColumn);
    }
    public String getColumnName(int columnIndex) {    //返回指定列的名称
        //需要为列索引加上固定列的数量
        return columnNameV.get(columnIndex + fixedColumn);
    }
}
```

注意

在处理与表格列有关的信息时，均需要在表格总列数的基础上减去固定列的数量。

（4）在 MfixedColumnTable 类中再声明以下 4 个属性。

```
private JTable fixedColumnTable;                          //固定列表格对象
private FixedColumnTableModel fixedColumnTableModel;      //固定列表格模型对象
private JTable floatingColumnTable;                       //移动列表格对象
//移动列表格模型对象
private FloatingColumnTableModel floatingColumnTableModel;
```

（5）创建用于同步两个表格中被选中行的事件监听器类 MListSelectionListener，即当选中左侧固定列表格中的某一行时，监听器会同步选中右侧可移动列表格中的对应行；同样，当选中右侧可移动

列表格中的某一行时，监听器会同步选中左侧固定列表格中的对应行。该类继承了 ListSelectionListener 类，并且为 MfixedColumnTable 类的内部类。具体代码如下：

```
private class MListSelectionListener implements ListSelectionListener {
    boolean isFixedColumnTable = true;      //默认由选中固定列表格中的行触发
    public MListSelectionListener(boolean isFixedColumnTable) {
        this.isFixedColumnTable = isFixedColumnTable;
    }
    public void valueChanged(ListSelectionEvent e) {
        if (isFixedColumnTable) {                //由选中固定列表格中的行触发
            //获得固定列表格中的选中行
            int row = fixedColumnTable.getSelectedRow();
            //同时选中右侧可移动列表格中的相应行
            floatingColumnTable.setRowSelectionInterval(row, row);
        } else {                                 //由选中可移动列表格中的行触发
            //获得可移动列表格中的选中行
            int row = floatingColumnTable.getSelectedRow();
            //同时选中左侧固定列表格中的相应行
            fixedColumnTable.setRowSelectionInterval(row, row);
        }
    }
}
```

注意

这里实现的事件监听器要求两个表格必须均是单选模式的，即一次只允许选中一行。

（6）编写 MfixedColumnTable 类的构造方法，需要传入 3 个参数，分别为表格列名数组、表格数据数组和固定列数量，之后便是创建固定列表格、可移动列表格和滚动面板。具体代码如下：

```
public MfixedColumnTable(Vector<String> columnNameV,
        Vector<Vector<Object>> tableValueV, int fixedColumn) {
    super();
    setLayout(new BorderLayout());
    this.columnNameV = columnNameV;
    this.tableValueV = tableValueV;
    this.fixedColumn = fixedColumn;
    //创建固定列表格模型对象
    fixedColumnTableModel = new FixedColumnTableModel();
    //创建固定列表格对象
    fixedColumnTable = new JTable(fixedColumnTableModel);
    //获得选择模型对象
    ListSelectionModel fixed = fixedColumnTable.getSelectionModel();
    //选择模式为单选
    fixed.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    //添加行被选中的事件监听器
    fixed.addListSelectionListener(new MListSelectionListener(true));
    //创建可移动列表格模型对象
    floatingColumnTableModel = new FloatingColumnTableModel();
```

```
        //创建可移动列表格对象
        floatingColumnTable = new JTable(floatingColumnTableModel);
        //关闭表格的自动调整功能
        floatingColumnTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        ListSelectionModel floating = floatingColumnTable
                .getSelectionModel();                    //获得选择模型对象
        //选择模式为单选
        floating.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        //添加行被选中的事件监听器
        MListSelectionListener listener = new MListSelectionListener(false);
        floating.addListSelectionListener(listener);
        JScrollPane scrollPane = new JScrollPane();          //创建一个滚动面板对象
        //将固定列表格头放到滚动面板的左上方
        scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER,
                fixedColumnTable.getTableHeader());
        //创建一个用来显示基础信息的视图对象
        JViewport viewport = new JViewport();
        viewport.setView(fixedColumnTable);                  //将固定列表格添加到视图中
        //设置视图的首选大小为固定列表格的首选大小
        viewport.setPreferredSize(fixedColumnTable.getPreferredSize());
        //将视图添加到滚动面板的标题视图中
        scrollPane.setRowHeaderView(viewport);
        //将可移动表格添加到默认视图
        scrollPane.setViewportView(floatingColumnTable);
        add(scrollPane, BorderLayout.CENTER);
    }
```

（7）创建 ExampleFrame_07 类，编写测试提供行标题栏表格的代码，首先封装表格列名数组和表格数据数组，然后创建 MfixedColumnTable 类的对象，最后将其添加到窗体中。关键代码如下：

```
Vector<String> columnNameV = new Vector<>();
columnNameV.add("日期");
for (int i = 1; i < 21; i++) {
    columnNameV.add("商品" + i);
}
Vector<Vector<Object>> tableValueV = new Vector<>();
for (int row = 1; row < 31; row++) {
    Vector<Object> rowV = new Vector<>();
    rowV.add(row);
    for (int col = 0; col < 20; col++) {
        rowV.add((int) (Math.random() * 1000));
    }
    tableValueV.add(rowV);
}
final MfixedColumnTable panel
                = new MfixedColumnTable(columnNameV, tableValueV, 1);
getContentPane().add(panel, BorderLayout.CENTER);
```

21.4 小　　结

通过对本章的学习，相信读者已经可以熟练地使用 JTable 表格，包括通过各种方式创建表格、根据实际需要定制表格、通过编码操纵表格及维护表格模型。在本章的最后还讲解了提供行标题栏表格的实现方法，以帮助读者拓宽表格的设计思路，这也是一种很适用的表格形式。

21.5 实践与练习

1. 利用 Swing 表格设计一个用来选择日期的对话框。（**答案位置：光盘\TM\sl\21.08**）
2. 设计一个以多列为行标题栏的例子。（**答案位置：光盘\TM\sl\21.09**）

第22章

Swing 树组件

（视频讲解：20 分钟）

树状结构是一种常用的信息表现形式，它可以直观地显示出一组信息的层次结构。Swing 中的 JTree 类用来创建树。本章将深入学习该类的使用方法，以及一些相关类的使用方法，为了便于读者理解，在讲解过程中结合了大量的实例。

通过阅读本章，您可以：

- 学会创建树的方法
- 学会处理选中节点事件的方法
- 学会定制树的基本方法
- 掌握遍历树节点的方法
- 掌握维护树模型的方法
- 掌握处理展开节点事件的方法

22.1 简 单 的 树

视频讲解：光盘\TM\lx\22\简单的树.exe

树状结构是一种常用的信息表现形式，它可以直观地显示出一组信息的层次结构。Swing 中的 JTree 类用来创建树，该类的常用构造方法如表 22.1 所示。

表 22.1 JTree 类的常用构造方法

构 造 方 法	说 明
JTree()	创建一个默认的树
JTree(TreeNode root)	根据指定根节点创建树
JTree(TreeModel newModel)	根据指定树模型创建树

DefaultMutableTreeNode 类实现了 TreeNode 接口，用来创建树的节点。一个树只能有一个父节点，可以有 0 个或多个子节点，默认情况下每个节点都允许有子节点，如果需要可以设置为不允许。该类的常用构造方法如表 22.2 所示。

表 22.2 DefaultMutableTreeNode 类的常用构造方法

构 造 方 法	说 明
DefaultMutableTreeNode()	创建一个默认的节点，默认情况下允许有子节点
DefaultMutableTreeNode(Object userObject)	创建一个具有指定标签的节点
DefaultMutableTreeNode(Object userObject, boolean allowsChildren)	创建一个具有指定标签的节点，并且指定是否允许有子节点

利用 DefaultMutableTreeNode 类的 add(MutableTreeNode newChild)方法可以为该节点添加子节点，该节点则称为父节点，没有父节点的节点则称为根节点。可以通过根节点利用构造方法 JTree(TreeNode root)直接创建树，也可以先创建一个树模型 TreeModel，然后再通过树模型利用构造方法 JTree (TreeModel newModel)创建树。

DefaultTreeModel 类实现了 TreeModel 接口，该类仅提供了以下两个构造方法，所以在利用该类创建树模型时，必须指定树的根节点。

- ☑ DefaultTreeModel(TreeNode root)：创建一个采用默认方式判断节点是否为叶子节点的树模型。
- ☑ DefaultTreeModel(TreeNode root, boolean asksAllowsChildren)：创建一个采用指定方式判断节点是否为叶子节点的树模型。

由 DefaultTreeModel 类实现的树模型判断节点是否为叶子节点有两种方式：默认方式为如果节点不存在子节点则为叶子节点，如图 22.1 所示；另一种方式则是根据节点是否允许有子节点，只要不允许有子节点就是叶子节点，如果允许有子节点，即使并不包含任何子节点，也不是叶子节点，如图 22.2 所示。将入口参数 asksAllowsChildren 设置为 true 即表示采用后一种方式。

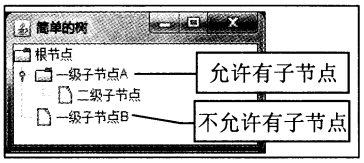


图 22.1　采用默认方式

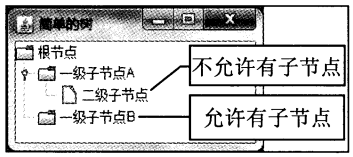


图 22.2　采用非默认方式

【例 22.1】 简单的树。（**实例位置：光盘\TM\sl\22.01**）

本例利用一个根节点创建了 3 个树，从左到右依次添加到了窗体中，其中的一个是利用构造方法 JTree(TreeNode root)直接创建的，其他两个是通过树模型创建的，分别采用默认和非默认的方式判断节点是否为叶子节点。下面是本例的关键代码：

```
//创建根节点
DefaultMutableTreeNode root = new DefaultMutableTreeNode("根节点");
//创建一级节点
DefaultMutableTreeNode nodeFirst = new DefaultMutableTreeNode(
        "一级子节点 A");
root.add(nodeFirst);                                    //将一级节点添加到根节点
DefaultMutableTreeNode nodeSecond = new DefaultMutableTreeNode(
        "二级子节点", false);                           //创建不允许有子节点的二级节点
nodeFirst.add(nodeSecond);                              //将二级节点添加到一级节点
root.add(new DefaultMutableTreeNode("一级子节点 B"));   //创建一级节点
JTree treeRoot = new JTree(root);                       //利用根节点直接创建树
getContentPane().add(treeRoot, BorderLayout.WEST);
//利用根节点创建树模型，采用默认的判断方式
DefaultTreeModel treeModelDefault = new DefaultTreeModel(root);
//利用树模型创建树
JTree treeDefault = new JTree(treeModelDefault);
getContentPane().add(treeDefault, BorderLayout.CENTER);
//利用根节点创建树模型，并采用非默认的判断方式
DefaultTreeModel treeModelPointed = new DefaultTreeModel(root,  true);
JTree treePointed = new JTree(treeModelPointed);        //利用树模型创建树
getContentPane().add(treePointed, BorderLayout.EAST);
```

运行本例，将得到如图 22.3 所示的窗体。窗体左侧的树是直接创建的，中间的树是采用默认方式判断节点的，这两个树中名称为“一级子节点 B”的节点图标均为叶子节点图标；右侧的树是采用非默认方式判断节点的，该树中名称为“一级子节点 B”的节点图标均为非叶子节点图标。

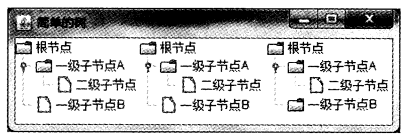


图 22.3　简单的树

22.2　处理选中节点事件

视频讲解：光盘\TM\lx\22\处理选中节点事件.exe

树的节点允许为被选中和取消选中状态，通过捕获树节点的选择事件，可以处理相应的操作。树

的选择模式有 3 种，通过 TreeSelectionModel 类的对象可以设置树的选择模式。可以通过 JTree 类的 getSelectionModel()方法获得 TreeSelectionModel 类的对象，然后通过 TreeSelectionModel 类的 setSelectionMode(int mode)方法设置选择模式。该方法的入口参数可以从表 22.3 列出的该类的静态常量中选择。

表 22.3 TreeSelectionModel 类中代表选择模式的静态常量

静 态 常 量	常 量 值	说 明
SINGLE_TREE_SELECTION	1	只允许选中一个，如图 22.4 所示
CONTIGUOUS_TREE_SELECTION	2	允许连续选中多个，如图 22.5 所示
DISCONTIGUOUS_TREE_SELECTION	4	允许任意选中多个，为树的默认模式，如图 22.6 所示

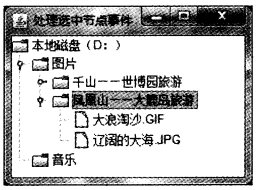


图 22.4 单选模式

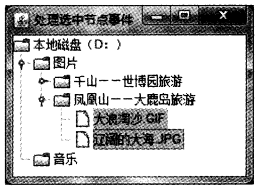


图 22.5 连选模式

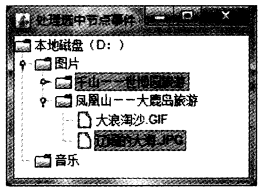


图 22.6 多选模式

当选中树节点和取消树节点的选中状态时，将发出 TreeSelectionEvent 事件，通过实现 TreeSelectionListener 接口可以捕获该事件。TreeSelectionListener 接口的具体定义如下：

```
public interface TreeSelectionListener extends EventListener {
    void valueChanged(TreeSelectionEvent e);
}
```

当捕获发出的 TreeSelectionEvent 事件时，valueChanged(TreeSelectionEvent e)方法将被触发执行，此时通过 JTree 类的 getSelectionPaths()方法可以获得所有被选中节点的路径，该方法将返回一个 TreePath 类型的数组；通过 getSelectionPath()方法将获得选中节点中索引值最小的节点的路径，即 TreePath 类的对象，也可以理解为选中节点中距离根节点最近的节点的路径。在获得选中节点的路径之前，可以通过 JTree 类的 isSelectionEmpty()方法查看是否存在被选中的节点，如果返回 false，则表示存在被选中的节点，通过 getSelectionCount()方法可以获得被选中节点的数量。

TreePath 类表示树节点的路径，即通过该类可以获得子节点所属的父节点，以及父节点所属的上级节点，直到树的根节点。TreePath 类的常用方法如表 22.4 所示。

表 22.4 TreePath 类的常用方法

方 法	说 明
getPath()	以 Object 数组的形式返回该路径中所有节点的对象，在数组中的顺序按照从根节点到子节点的顺序
getLastPathComponent()	获得路径中最后一个节点的对象
getParentPath()	获得路径中除了最后一个节点的路径
pathByAddingChild(Object child)	获得向路径中添加指定节点后的路径
getPathCount()	获得路径中包含节点的数量
getPathComponent(int element)	获得路径中指定索引位置的节点对象

【例 22.2】 处理选中节点事件。（实例位置：光盘\TM\sl\22.02）

本例利用 TreeSelectionListener 监听器捕获了选中树节点和取消选中树节点的事件，并将选中节点的路径信息全部输出到控制台。下面是本例的关键代码：

```
TreeSelectionModel treeSelectionModel;                    //获得树的选择模式
treeSelectionModel = tree.getSelectionModel();
//设置树的选择模式为连选
treeSelectionModel.setSelectionMode(CONTIGUOUS_TREE_SELECTION);
tree.addTreeSelectionListener(new TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent e) {
        if (!tree.isSelectionEmpty()) {                   //查看是否存在被选中的节点
            //获得所有被选中节点的路径
            TreePath[] selectionPaths = tree.getSelectionPaths();
            for (int i = 0; i < selectionPaths.length; i++) {
                //获得被选中节点的路径
                TreePath treePath = selectionPaths[i];
                //以 Object 数组的形式返回该路径中所有节点的对象
                Object[] path = treePath.getPath();
                for (int j = 0; j < path.length; j++) {
                    DefaultMutableTreeNode node;          //获得节点
                    node = (DefaultMutableTreeNode) path[j];
        String s = node.getUserObject()+ (j == (path.length - 1) ? "" : "-->");
                    System.out.print(s);                  //输出节点标签
                }
                System.out.println();
            } System.out.println();
        }}});
```

运行本例，将得到如图 22.7 所示的窗体，首先展开树的所有节点，然后选中“千山——世博园旅游”节点，最后追加选中“凤凰山——大鹿岛旅游”节点，在控制台将输出如图 22.8 所示的信息。

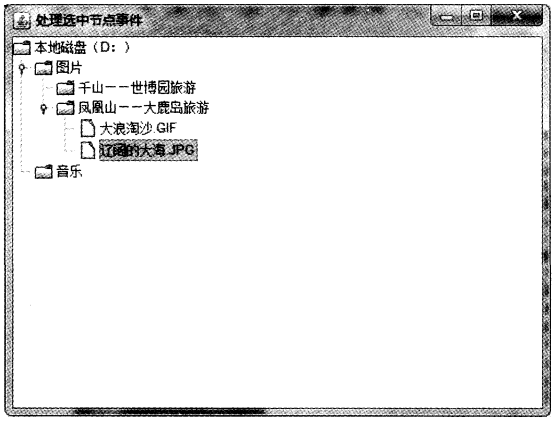


图 22.7　处理选中节点事件

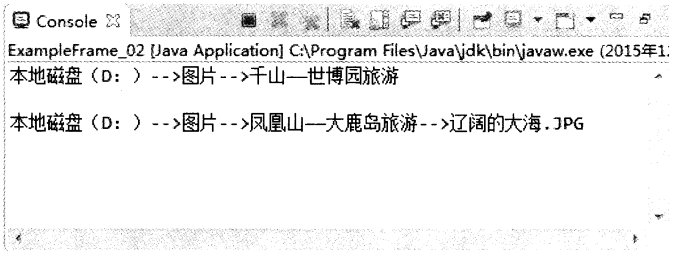


图 22.8　输出到控制台的信息

22.3 遍历树节点

视频讲解：光盘\TM\lx\22\遍历树节点.exe

有时需要对树进行遍历，也就是遍历树中的部分或全部节点，以便查找某一节点，或者是对树中的节点执行某一操作。DefaultMutableTreeNode 类提供了两组相对的遍历方式，下面详细介绍。

前序遍历和后序遍历是一组相对的遍历方式，按前序遍历树节点的顺序如图 22.9 所示，通过 preorderEnumeration()方法将返回按前序遍历的枚举对象；按后序遍历树节点的顺序如图 22.10 所示，通过 postorderEnumeration()方法将返回按后序遍历的枚举对象。

广度优先遍历和深度优先遍历是一组相对的遍历方式，以广度优先遍历树节点的顺序如图 22.11 所示，通过 breadthFirstEnumeration()方法将返回以广度优先遍历的枚举对象；以深度优先遍历树节点的顺序如图 22.12 所示，通过 depthFirstEnumeration()方法将返回以深度优先遍历的枚举对象。

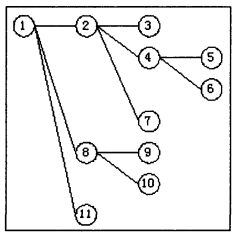


图 22.9 按前序遍历

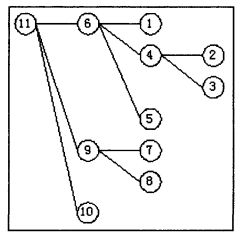


图 22.10 按后序遍历

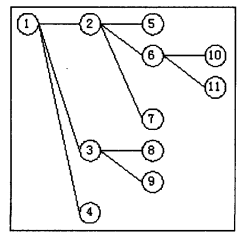


图 22.11 以广度优先遍历

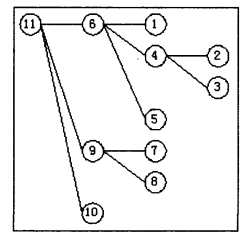


图 22.12 以深度优先遍历

说明

因为后序遍历和深度优先遍历这两种遍历方式的具体遍历方法相同，所以图 22.10 和图 22.12 是相同的，实际上方法 depthFirstEnumeration()只是调用了方法 postorderEnumeration()。

通过 DefaultMutableTreeNode 类的 children()方法，可以得到仅包含该节点子节点的枚举对象，以便快速遍历节点的子节点。在 DefaultMutableTreeNode 类中还提供了一些常用方法，如表 22.5 所示。

表 22.5 DefaultMutableTreeNode 类的常用方法

方 法	说 明
getLevel()	获得该节点相对于根节点的级别值，如根节点的子节点的级别值为 1
getDepth()	获得以此节点为根节点的树的深度，如果该节点没有子节点，则深度为 0
getParent()	获得该节点的父节点对象
getChildCount()	获得该节点拥有子节点的个数
getFirstChild()	获得该节点的第一个子节点对象
getSiblingCount()	获得该节点拥有兄弟节点的个数
getNextSibling()	获得该节点的后一个兄弟节点
getPreviousSibling()	获得该节点的前一个兄弟节点
getPath()	获得该节点的路径

续表

方　法	说　明
isRoot()	判断该节点是否为根节点
isLeaf()	判断该节点是否为叶子节点

【例 22.3】 遍历树节点。（实例位置：光盘\TM\sl\22.03）

本例以按钮的方式提供了本节讲解的 5 种遍历方式，通过单击相应的按钮可以在控制台查看具体的遍历方式。下面是本例的关键代码：

```
public void actionPerformed(ActionEvent e) {
    Enumeration<?> enumeration;                          //声明节点枚举对象
    if (mode.equals("按前序遍历"))
        //按前序遍历所有树节点
        enumeration = root.preorderEnumeration();
    else if (mode.equals("按后序遍历"))
        //按后序遍历所有树节点
        enumeration = root.postorderEnumeration();
    else if (mode.equals("以广度优先遍历"))
        //以广度优先遍历所有树节点
        enumeration = root.breadthFirstEnumeration();
    else if (mode.equals("以深度优先遍历"))
        //以深度优先遍历所有树节点
        enumeration = root.depthFirstEnumeration();
    else
        enumeration = root.children();                   //遍历该节点的子节点
    while (enumeration.hasMoreElements()) {              //遍历节点枚举对象
        DefaultMutableTreeNode node;                     //获得节点
        node = (DefaultMutableTreeNode) enumeration.nextElement();
        //根据节点级别输出占位符
        for (int i = 0; i < node.getLevel(); i++) {
            System.out.print("----");
        }
        System.out.println(node.getUserObject());       //输出节点标签
    }
}
```

程序运行界面如图 22.13 所示。

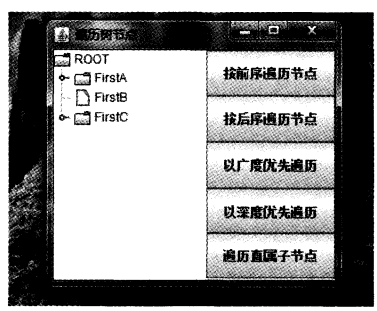


图 22.13　遍历树节点

22.4 定 制 树

视频讲解：光盘\TM\lx\22\定制树.exe

在使用树时，经常需要对树进行一系列的设置，例如，对节点图标的设置，对节点间连接线的设置，以及对树展开状况的设置等，以便实现实际需要的视觉效果。本节将学习利用 JTree 类的一些方法定制树。

默认情况下显示树的根节点，但是不显示根节点手柄，如图 22.14 所示。如果并不希望显示树的根节点，则可以调用 setRootVisible(boolean rootVisible)方法，并将入口参数设为 false，效果如图 22.15 所示。如果希望在树的根节点前面也显示手柄，可以调用 setShowsRootHandles(boolean newValue)方法，并将入口参数设为 true，效果如图 22.16 所示。

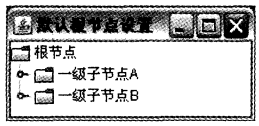


图 22.14 默认根节点设置

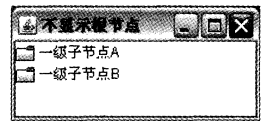


图 22.15 不显示根节点

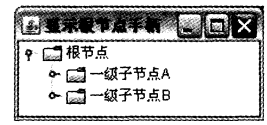


图 22.16 显示根节点手柄

默认情况下树节点的图标效果如图 22.17 所示，其中为非叶子节点的图标，为叶子节点的图标。通过 DefaultTreeCellRenderer 类的对象可以修改节点图标，通过 JTree 类的 getCellRenderer()方法可以得到该对象。方法 setLeafIcon(Icon newIcon)用来设置叶子节点图标，如图 22.18 所示为不采用叶子节点图标的效果；方法 setClosedIcon(Icon newIcon)用来设置节点处于折叠状态时采用的图标，如图 22.19 所示为不采用节点折叠图标的效果；方法 setOpenIcon(Icon newIcon)用来设置节点处于展开状态时采用的图标，如图 22.20 所示为不采用节点展开图标的效果。

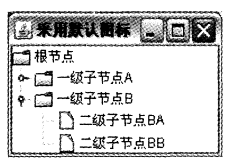


图 22.17 默认图标

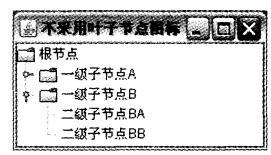


图 22.18 叶子节点图标

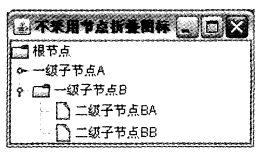


图 22.19 不采用节点折叠图标

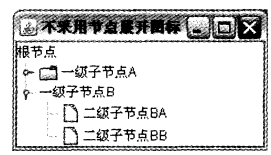


图 22.20 不采用节点展开图标

默认情况下在树节点之间绘制连接线，效果如图 22.21 所示。通过 putClientProperty(Object key, Object value)方法可以设置连接线的绘制方式，此时需要将入口参数 key 设置为 JTree.lineStyle；将入口参数 value 设置为 None 表示不绘制节点间的连接线，效果如图 22.22 所示；设置为 Horizontal 表示绘制水平分栏线，绘制方式为仅在根节点和一级节点之间，或者是一级节点和一级节点之间，效果如图 22.23 所示；设置为 Angled 则表示绘制节点间的连接线，等效于默认设置。

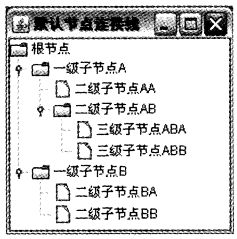


图 22.21　默认节点连接线

图 22.22　取消节点连接线

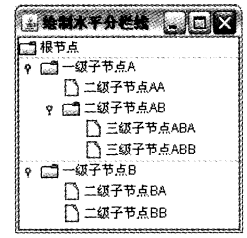


图 22.23　绘制水平分栏线

默认情况下只有树的根节点是展开的，其他子节点均处于折叠状态，如果希望在初次运行时树的某一节点就处于展开状态，可以通过 expandPath(TreePath path)方法实现。在调用该方法时需要传入要展开节点的路径。

【例 22.4】 定制树。（**实例位置：光盘\TM\sl\22.04**）

本例利用树实现了一个分层的导航栏，并且在初次运行时树的所有节点就处于展开状态，效果如图 22.24 所示。关键代码如下：

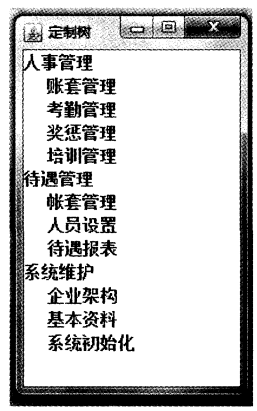


图 22.24　定制树

```
tree = new JTree(root);
tree.setRootVisible(false);                          //不显示树的根节点
tree.setRowHeight(20);                               //树节点的行高为 20 像素
tree.setFont(new Font("宋体", Font.BOLD, 14));       //设置树节点的字体
tree.putClientProperty("JTree.lineStyle", "None");   //节点间不采用连接线
DefaultTreeCellRenderer treeCellRenderer;            //获得树节点的绘制对象
treeCellRenderer = (DefaultTreeCellRenderer) tree.getCellRenderer();
treeCellRenderer.setLeafIcon(null);                  //设置叶子节点不采用图标
treeCellRenderer.setClosedIcon(null);                //设置节点折叠时不采用图标
treeCellRenderer.setOpenIcon(null);                  //设置节点展开时不采用图标
Enumeration<?> enumeration;                          //按前序遍历所有树节点
enumeration = root.preorderEnumeration();
while (enumeration.hasMoreElements()) {
    DefaultMutableTreeNode node;
    node = (DefaultMutableTreeNode) enumeration.nextElement();
    if (!node.isLeaf()) {                            //判断是否为叶子节点
        //创建该节点的路径
```

```
            TreePath path = new TreePath(node.getPath());
            tree.expandPath(path);                    //如果不是，则展开该节点
        }
    }
```

22.5 维护树模型

视频讲解：光盘\TM\lx\22\维护树模型.exe

在使用树时，有时需要提供对树的维护功能，包括向树中添加新节点，以及修改或删除树中的现有节点，这些操作需要通过树的模型类 DefaultTreeModel 实现。下面就来介绍维护树模型的方法。

（1）添加树节点

利用 DefaultTreeModel 类的 insertNodeInto()方法可以向树模型中添加新的节点。insertNodeInto()方法的具体定义如下：

```
insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)
```

insertNodeInto()方法中各个入口参数的说明如表 22.6 所示。

表 22.6 insertNodeInto()方法中各个入口参数的说明

入口参数	说明
newChild	新添加的节点对象
parent	新添加节点所属的父节点对象，该对象必须为树模型中的一个节点
index	新添加节点在其父节点中的索引位置，索引值从 0 开始

例如，假设要为节点 parentNode 添加一个子节点 treeNode，当前在父节点 parentNode（父节点 parentNode 为树模型 treeModel 中的一个节点）中已经存在 6 个子节点，现在要将该节点添加到所有子节点之后，典型代码如下：

```
treeModel.insertNodeInto(treeNode, parentNode, parentNode.getChildCount());
```

（2）修改树节点

DefaultTreeModel 类的 nodeChanged(TreeNode node)方法用来通知树模型某节点已经被修改，如果修改的是节点的用户对象，修改信息将不会被同步到 GUI 界面。其中入口参数为被修改的节点对象。

例如，假设现在已经修改了树模型 treeModel 中的节点 treeNode（修改的是节点 treeNode 的用户对象），通知树模型 treeModel 其组成节点 treeNode 已经被修改的典型代码如下：

```
treeModel.nodeChanged(treeNode);
```

（3）删除树节点

DefaultTreeModel 类的 removeNodeFromParent(MutableTreeNode node)方法用来从树模型中删除指定节点 node。

例如，假设要从树模型 treeModel 中删除节点 treeNode，典型代码如下：

```
treeModel.removeNodeFromParent(treeNode);
```

注意

树的根节点不允许删除，当试图删除根节点时，将抛出 java.lang.IllegalArgumentException: node does not have a parent.异常。

【例 22.5】 维护树模型。（实例位置：光盘\TM\sl\22.05）

本例通过维护树模型，实现了维护企业架构树。关键代码如下：

```
final JButton addButton = new JButton("添加");
addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        DefaultMutableTreeNode node = new DefaultMutableTreeNode(
                textField.getText());                                    //创建欲添加节点
        TreePath selectionPath = tree.getSelectionPath();                //获得选中的父节点路径
        DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)
                selectionPath.getLastPathComponent();                    //获得选中的父节点
        treeModel.insertNodeInto(node, parentNode, parentNode
                .getChildCount());                                       //插入节点到所有子节点之后
        //获得新添加节点的路径
        TreePath path = selectionPath.pathByAddingChild(node);
        if (!tree.isVisible(path))
            tree.makeVisible(path);                                      //如果该节点不可见，则令其可见
    }
});
panel.add(addButton);
final JButton updButton = new JButton("修改");
updButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //获得选中的要修改节点的路径
        TreePath selectionPath = tree.getSelectionPath();
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) selectionPath
                .getLastPathComponent();                                 //获得选中的要修改的节点
        node.setUserObject(textField.getText());                         //修改节点的用户标签
        treeModel.nodeChanged(node);                                     //通知树模型该节点已经被修改
        tree.setSelectionPath(selectionPath);                    //选中被修改的节点
    }
});
panel.add(updButton);
final JButton delButton = new JButton("删除");
delButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) tree
                .getLastSelectedPathComponent();          //获得选中的欲删除的节点
        //查看要删除的节点是否为根节点，根节点不允许删除
        if (!node.isRoot()) {
            DefaultMutableTreeNode nextSelectedNode = node
                    .getNextSibling();                           //获得下一个兄弟节点，以备选中
```

```
                if (nextSelectedNode == null)                    //查看是否存在兄弟节点
                    nextSelectedNode = (DefaultMutableTreeNode) node
                            .getParent();                        //如果不存在则选中其父节点
                treeModel.removeNodeFromParent(node);            //删除节点
                tree.setSelectionPath(new TreePath(nextSelectedNode
                            .getPath()));                        //选中节点
            }
        }
});
panel.add(delButton);
```

运行本例，将得到如图 22.25 所示的窗体，单击“添加”按钮后将向当前选中的节点中添加一个标签为“名称”文本框中内容的子节点；单击“修改”按钮后将把当前选中节点的标签修改为“名称”文本框中的内容；单击“删除”按钮后将删除当前选中的节点，根节点除外。

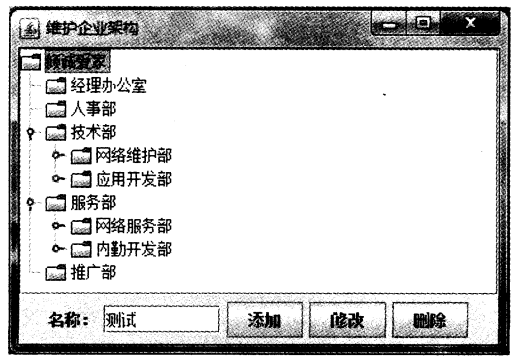


图 22.25　维护树模型

22.6　处理展开节点事件

视频讲解：光盘\TM\lx\22\处理展开节点事件.exe

有时需要捕获树节点被展开和折叠的事件，例如，需要验证用户的权限，如果用户没有权限查看该节点包含的子节点，则不允许树节点展开。

当展开和折叠树节点时，将发出 TreeExpansionEvent 事件，通过实现 TreeWillExpandListener 接口，可以在树节点展开和折叠之前捕获该事件。TreeWillExpandListener 接口的具体定义如下：

```
public interface TreeWillExpandListener extends EventListener {
    public void treeWillExpand(TreeExpansionEvent event)
            throws ExpandVetoException;
    public void treeWillCollapse(TreeExpansionEvent event)
            throws ExpandVetoException;
}
```

如果此次事件是由将要展开节点发出的，treeWillExpand()方法将被触发；如果此次事件是由将要折叠节点发出的，treeWillCollapse()方法将被触发。

通过实现 TreeExpansionListener 接口，可以在树节点展开和折叠时捕获该事件。TreeExpansionListener 接口的具体定义如下：

```
public interface TreeExpansionListener extends EventListener {
    public void treeExpanded(TreeExpansionEvent event);
    public void treeCollapsed(TreeExpansionEvent event);
}
```

如果此次事件是由展开节点发出的，treeExpanded()方法将被触发；如果此次事件是由折叠节点发出的，treeCollapsed()方法将被触发。

【例 22.6】 处理展开节点事件。（**实例位置：光盘\TM\sl\22.06**）

本例同时为树添加了 TreeWillExpandListener 和 TreeExpansionListener 监听器，目的是向控制台输出相应的提示信息，以展示这两个监听器的使用方法。关键代码如下：

```
//捕获树节点将要被展开或折叠的事件
tree.addTreeWillExpandListener(new TreeWillExpandListener() {
    //树节点将要被折叠时触发
    public void treeWillCollapse(TreeExpansionEvent e) {
        TreePath path = e.getPath();                    //获得将要被折叠节点的路径
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) path
                .getLastPathComponent();        //获得将要被折叠的节点
        System.out.println("节点"" + node + ""将要被折叠！");
    }
    //树节点将要被展开时触发
    public void treeWillExpand(TreeExpansionEvent e) {
        TreePath path = e.getPath();                    //获得将要被展开节点的路径
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) path
                .getLastPathComponent();        //获得将要被展开的节点
        System.out.println("节点"" + node + ""将要被展开！");
    }
});
//捕获树节点已经被展开或折叠的事件
tree.addTreeExpansionListener(new TreeExpansionListener() {
    //树节点已经折叠时触发
    public void treeCollapsed(TreeExpansionEvent e) {
        TreePath path = e.getPath();                    //获得已经被折叠节点的路径
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) path
                .getLastPathComponent();        //获得已经被折叠的节点
        System.out.println("节点"" + node + ""已经被折叠！");
        System.out.println();
    }
    //树节点已经被展开时触发
    public void treeExpanded(TreeExpansionEvent e) {
        TreePath path = e.getPath();                    //获得已经被展开节点的路径
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) path
                .getLastPathComponent();        //获得已经被展开的节点
        System.out.println("节点"" + node + ""已经被展开！");
        System.out.println();
    }
});
```

运行本例，将得到如图 22.26 所示的窗体，首先展开“技术部”节点，然后展开“服务部”节点，最后折叠“技术部”节点，在控制台将输出如图 22.27 所示的信息。

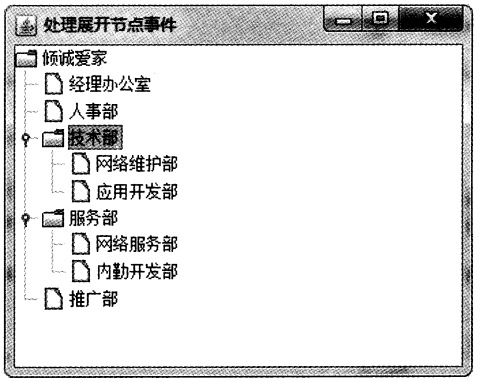


图 22.26　处理展开节点事件

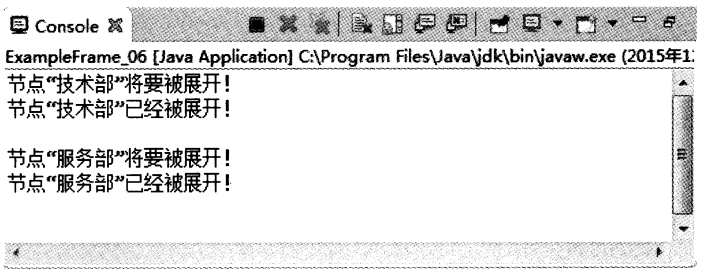


图 22.27　控制台的输出信息

22.7　小　　结

通过对本章的学习，相信读者已经能够熟练地使用 JTree 树，包括通过各种方式创建树、根据实际需要定制树、各种遍历树节点的方法、维护树模型的方法，以及处理选中节点事件和展开节点事件的方法。通过对 JTree 树的灵活使用，可以直观地显示出各种具有层次结构的信息。

22.8　实践与练习

1．尝试开发一个用来维护树结构的小程序。（**答案位置：光盘\TM\sl\22.07**）

2．尝试开发一个支持树状结构的下拉菜单。（**答案位置：光盘\TM\sl\22.08**）