```
            nf[i * 3 + 1] = NumberFormat.getPercentInstance(locales[i]);
            nf[i * 3 + 2] = NumberFormat.getCurrencyInstance(locales[i]);
        }
        for (int i = 0 ; i < locales.length ; i++)
        {
            String tip = i == 0 ? "----中国的格式----" :
                i == 1 ? "----日本的格式----" :
                i == 2 ? "----德国的格式----" :"----美国的格式----";
            System.out.println(tip);
            System.out.println("通用数值格式："
                + nf[i * 3].format(db));
            System.out.println("百分比数值格式："
                + nf[i * 3 + 1].format(db));
            System.out.println("货币数值格式："
                + nf[i * 3 + 2].format(db));
        }
    }
}
```

运行上面程序，将看到如图 7.10 所示的结果。

从图 7.10 中可以看出，德国的小数点比较特殊，它们采用逗号（,）作为小数点；中国、日本使用￥作为货币符号，而美国则采用$作为货币符号。细心的读者可能会发现，NumberFormat 其实也有国际化的作用！没错，同样的数值在不同国家的写法是不同的，而 NumberFormat 的作用就是把数值转换成不同国家的本地写法。

至于使用 NumberFormat 类将字符串解析成数值的意义不大（因为可以使用 Integer、Double 等包装类完成这种解析），故此处不再赘述。

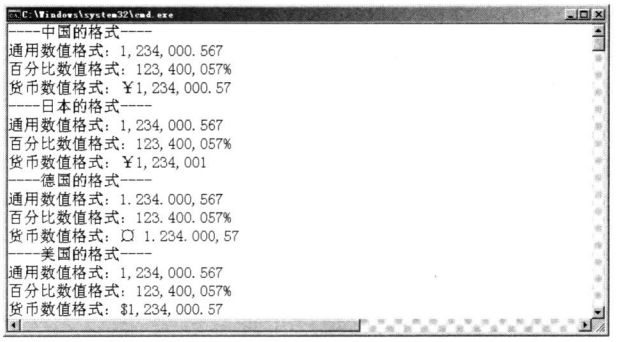


图 7.10　不同 Locale、不同类型的 NumberFormat

7.7.8　使用 DateFormat 格式化日期、时间

与 NumberFormat 相似的是，DateFormat 也是一个抽象类，它也提供了如下几个类方法用于获取 DateFormat 对象。

- getDateInstance()：返回一个日期格式器，它格式化后的字符串只有日期，没有时间。该方法可以传入多个参数，用于指定日期样式和 Locale 等参数；如果不指定这些参数，则使用默认参数。
- getTimeInstance()：返回一个时间格式器，它格式化后的字符串只有时间，没有日期。该方法可以传入多个参数，用于指定时间样式和 Locale 等参数；如果不指定这些参数，则使用默认参数。
- getDateTimeInstance()：返回一个日期、时间格式器，它格式化后的字符串既有日期，也有时间。该方法可以传入多个参数，用于指定日期样式、时间样式和 Locale 等参数；如果不指定这些参数，则使用默认参数。

上面三个方法可以指定日期样式、时间样式参数，它们是 DateFormat 的 4 个静态常量：FULL、LONG、MEDIUM 和 SHORT，通过这 4 个样式参数可以控制生成的格式化字符串。看如下例子程序。

程序清单：codes\07\7.7\DateFormatTest.java

```
public class DateFormatTest
{
    public static void main(String[] args)
        throws ParseException
    {
        // 需要被格式化的时间
        Date dt = new Date();
        // 创建两个Locale，分别代表中国、美国
        Locale[] locales = {Locale.CHINA, Locale.US};
        DateFormat[] df = new DateFormat[16];
```

```
        // 为上面两个 Locale 创建 16 个 DateFormat 对象
        for (int i = 0 ; i < locales.length ; i++)
        {
            df[i * 8] = DateFormat.getDateInstance(SHORT, locales[i]);
            df[i * 8 + 1] = DateFormat.getDateInstance(MEDIUM, locales[i]);
            df[i * 8 + 2] = DateFormat.getDateInstance(LONG, locales[i]);
            df[i * 8 + 3] = DateFormat.getDateInstance(FULL, locales[i]);
            df[i * 8 + 4] = DateFormat.getTimeInstance(SHORT, locales[i]);
            df[i * 8 + 5] = DateFormat.getTimeInstance(MEDIUM , locales[i]);
            df[i * 8 + 6] = DateFormat.getTimeInstance(LONG , locales[i]);
            df[i * 8 + 7] = DateFormat.getTimeInstance(FULL , locales[i]);
        }
        for (int i = 0 ; i < locales.length ; i++)
        {
            String tip = i == 0 ? "----中国日期格式----":"----美国日期格式----";
            System.out.println(tip);
            System.out.println("SHORT 格式的日期格式: "
                + df[i * 8].format(dt));
            System.out.println("MEDIUM 格式的日期格式: "
                + df[i * 8 + 1].format(dt));
            System.out.println("LONG 格式的日期格式: "
                + df[i * 8 + 2].format(dt));
            System.out.println("FULL 格式的日期格式: "
                + df[i * 8 + 3].format(dt));
            System.out.println("SHORT 格式的时间格式: "
                + df[i * 8 + 4].format(dt));
            System.out.println("MEDIUM 格式的时间格式: "
                + df[i * 8 + 5].format(dt));
            System.out.println("LONG 格式的时间格式: "
                + df[i * 8 + 6].format(dt));
            System.out.println("FULL 格式的时间格式: "
                + df[i * 8 + 7].format(dt));
        }
    }
}
```

上面程序共创建了 16 个 DateFormat 对象，分别为中国、美国两个 Locale 各创建 8 个 DateFormat 对象，分别是 SHORT、MEDIUM、LONG、FULL 四种样式的日期格式器、时间格式器。运行上面程序，会看到如图 7.11 所示的效果。

从图 7.11 中可以看出，正如 NumberFormat 提供了国际化的能力一样，DateFormat 也具有国际化的能力，同一个日期使用不同的 Locale 格式器格式化的效果完全不同，格式化后的字符串正好符合 Locale 对应的本地习惯。

图 7.11 16 种 DateFormat 格式化的效果

获得了 DateFormat 之后，还可以调用它的 setLenient(boolean lenient)方法来设置该格式器是否采用严格语法。举例来说，如果采用不严格的日期语法（该方法的参数为 true），对于字符串"2004-2-31"将会转换成 2004 年 3 月 2 日；如果采用严格的日期语法，解析该字符串时将抛出异常。

DateFormat 的 parse()方法可以把一个字符串解析成 Date 对象，但它要求被解析的字符串必须符合日期字符串的要求，否则可能抛出 ParseException 异常。例如，如下代码片段：

```
String str1 = "2017/10/07";
String str2 = "2017 年 10 月 07 日";
// 下面输出 Sat Oct 07 00:00:00 CST 2017
System.out.println(DateFormat.getDateInstance().parse(str2));
// 下面输出 Sat Oct 07 00:00:00 CST 2017
System.out.println(DateFormat.getDateInstance(SHORT).parse(str1));
// 下面抛出 ParseException 异常
```

```
System.out.println(DateFormat.getDateInstance().parse(str1));
```

上面代码中最后一行代码解析日期字符串时引发 ParseException 异常，因为"2017/10/07"是一个 SHORT 样式的日期字符串，必须用 SHORT 样式的 DateFormat 实例解析，否则将抛出异常。

7.7.9 使用 SimpleDateFormat 格式化日期

前面介绍的DateFormat的parse()方法可以把字符串解析成Date对象，但实际上DateFormat的parse()方法不够灵活——它要求被解析的字符串必须满足特定的格式！为了更好地格式化日期、解析日期字符串，Java 提供了 SimpleDateFormat 类。

SimpleDateFormat 是 DateFormat 的子类，正如它的名字所暗示的，它是“简单”的日期格式器。很多读者对“简单”的日期格式器不屑一顾，实际上 SimpleDateFormat 比 DateFormat 更简单，功能更强大。

提示： 有一封读者来信让笔者记忆很深刻，他说：“相对于有些人喜欢深奥的图书，他更喜欢“简单”的 IT 图书，“简单”的东西很清晰、明确，下一步该怎么做，为什么这样做，一切都清清楚楚，无须任何猜测、想象——正好符合计算机哲学——0 就是 0，1 就是 1，中间没有任何回旋的余地。如果喜欢深奥的书籍，那就看《老子》吧！够深奥，几乎可以包罗万象，但有人是通过《老子》开始学习编程的吗……”

SimpleDateFormat 可以非常灵活地格式化 Date，也可以用于解析各种格式的日期字符串。创建 SimpleDateFormat 对象时需要传入一个 pattern 字符串，这个 pattern 不是正则表达式，而是一个日期模板字符串。

程序清单：codes\07\7.7\SimpleDateFormatTest.java

```
public class SimpleDateFormatTest
{
    public static void main(String[] args)
        throws ParseException
    {
        Date d = new Date();
        // 创建一个 SimpleDateFormat 对象
        SimpleDateFormat sdf1 = new SimpleDateFormat("Gyyyy 年中第 D 天");
        // 将 d 格式化成日期，输出：公元 2017 年中第 281 天
        String dateStr = sdf1.format(d);
        System.out.println(dateStr);
        // 一个非常特殊的日期字符串
        String str = "14###3 月##21";
        SimpleDateFormat sdf2 = new SimpleDateFormat("y###MMM##d");
        // 将日期字符串解析成日期，输出：Fri Mar 21 00:00:00 CST 2014
        System.out.println(sdf2.parse(str));
    }
}
```

从上面程序中可以看出，使用 SimpleDateFormat 可以将日期格式化成形如“公元 2014 年中第 101 天”这样的字符串，也可以把形如“14###三月##21”这样的字符串解析成日期，功能非常强大。SimpleDateFormat 把日期格式化成怎样的字符串，以及能把怎样的字符串解析成 Date，完全取决于创建该对象时指定的 pattern 参数，pattern 是一个使用日期字段占位符的日期模板。

如果读者想知道 SimpleDateFormat 支持哪些日期、时间占位符，可以查阅 API 文档中 SimpleDateFormat 类的说明，此处不再赘述。

7.8 Java 8 新增的日期、时间格式器

Java 8 新增的日期、时间 API 里不仅包括了 Instant、LocalDate、LocalDateTime、LocalTime 等代表日期、时间的类，而且在 java.time.format 包下提供了一个 DateTimeFormatter 格式器类，该类相当于前

面介绍的 DateFormat 和 SimpleDateFormat 的合体，功能非常强大。

与 DateFormat、SimpleDateFormat 类似，DateTimeFormatter 不仅可以将日期、时间对象格式化成字符串，也可以将特定格式的字符串解析成日期、时间对象。

为了使用 DateTimeFormatter 进行格式化或解析，必须先获取 DateTimeFormatter 对象，获取 DateTimeFormatter 对象有如下三种常见的方式。

- 直接使用静态常量创建 DateTimeFormatter 格式器。DateTimeFormatter 类中包含了大量形如 ISO_LOCAL_DATE、ISO_LOCAL_TIME、ISO_LOCAL_DATE_TIME 等静态常量，这些静态常量本身就是 DateTimeFormatter 实例。
- 使用代表不同风格的枚举值来创建 DateTimeFormatter 格式器。在 FormatStyle 枚举类中定义了 FULL、LONG、MEDIUM、SHORT 四个枚举值，它们代表日期、时间的不同风格。
- 根据模式字符串来创建 DateTimeFormatter 格式器。类似于 SimpleDateFormat，可以采用模式字符串来创建 DateTimeFormatter，如果需要了解 DateTimeFormatter 支持哪些模式字符串，则需要参考该类的 API 文档。

注意：

本书第 3 版曾提醒大家，Java 8 的 DateTimeFormatter 的官方 API 文档有错，后来我们也把错误反馈给 Oracle 官方，现在 Java 9 的 DateTimeFormatter 的官方 API 文档已经改正了。

7.8.1 使用 DateTimeFormatter 完成格式化

使用 DateTimeFormatter 将日期、时间（LocalDate、LocalDateTime、LocalTime 等实例）格式化为字符串，可通过如下两种方式。

- 调用 DateTimeFormatter 的 format(TemporalAccessor temporal)方法执行格式化，其中 LocalDate、LocalDateTime、LocalTime 等类都是 TemporalAccessor 接口的实现类。
- 调用 LocalDate、LocalDateTime、LocalTime 等日期、时间对象的 format(DateTimeFormatter formatter)方法执行格式化。

上面两种方式的功能相同，用法也基本相似，如下程序示范了使用 DateTimeFormatter 来格式化日期、时间。

程序清单：codes\07\7.8\NewFormatterTest.java

```
public class NewFormatterTest
{
    public static void main(String[] args)
    {
        DateTimeFormatter[] formatters = new DateTimeFormatter[]{
            // 直接使用常量创建 DateTimeFormatter 格式器
            DateTimeFormatter.ISO_LOCAL_DATE,
            DateTimeFormatter.ISO_LOCAL_TIME,
            DateTimeFormatter.ISO_LOCAL_DATE_TIME,
            // 使用本地化的不同风格来创建 DateTimeFormatter 格式器
            DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL, FormatStyle.MEDIUM),
            DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG),
            // 根据模式字符串来创建 DateTimeFormatter 格式器
            DateTimeFormatter.ofPattern("Gyyyy%%MMM%%dd HH:mm:ss")
        };
        LocalDateTime date = LocalDateTime.now();
        // 依次使用不同的格式器对 LocalDateTime 进行格式化
        for(int i = 0 ; i < formatters.length ; i++)
        {
            // 下面两行代码的作用相同
            System.out.println(date.format(formatters[i]));
            System.out.println(formatters[i].format(date));
        }
    }
}
```

上面程序使用三种方式创建了 6 个 DateTimeFormatter 对象，然后程序中两行粗体字代码分别使用不同方式来格式化日期。运行上面程序，会看到如图 7.12 所示的效果。

图 7.12 DateTimeFormatter 格式化的效果

从图 7.12 可以看出，使用 DateTimeFormatter 进行格式化时不仅可按系统预置的格式对日期、时间进行格式化，也可使用模式字符串对日期、时间进行自定义格式化，由此可见，DateTimeFormatter 的功能完全覆盖了传统的 DateFormat、SimpleDateFormate 的功能。

提示： 有些时候，读者可能还需要使用传统的 DateFormat 来执行格式化，DateTimeFormatter 则提供了一个 toFormat()方法，该方法可以获取 DateTimeFormatter 对应的 Format 对象。

7.8.2 使用 DateTimeFormatter 解析字符串

为了使用 DateTimeFormatter 将指定格式的字符串解析成日期、时间对象（LocalDate、LocalDateTime、LocalTime 等实例），可通过日期、时间对象提供的 parse(CharSequence text, DateTimeFormatter formatter) 方法进行解析。

如下程序示范了使用 DateTimeFormatter 解析日期、时间字符串。

程序清单：codes\07\7.8\NewFormatterParse.java

```
public class NewFormatterParse
{
    public static void main(String[] args)
    {
        // 定义一个任意格式的日期、时间字符串
        String str1 = "2014==04==12 01 时 06 分 09 秒";
        // 根据需要解析的日期、时间字符串定义解析所用的格式器
        DateTimeFormatter fomatter1 = DateTimeFormatter
            .ofPattern("yyyy==MM==dd HH 时 mm 分 ss 秒");
        // 执行解析
        LocalDateTime dt1 = LocalDateTime.parse(str1, fomatter1);
        System.out.println(dt1); // 输出 2014-04-12T01:06:09
        // ---下面代码再次解析另一个字符串---
        String str2 = "2014$$$4 月$$$13 20 小时";
        DateTimeFormatter fomatter2 = DateTimeFormatter
            .ofPattern("yyy$$$MMM$$$dd HH 小时");
        LocalDateTime dt2 = LocalDateTime.parse(str2, fomatter2);
        System.out.println(dt2); // 输出 2014-04-13T20:00
    }
}
```

上面程序中定义了两个不同格式的日期、时间字符串，为了解析它们，程序分别使用对应的格式字符串创建了 DateTimeFormatter 对象，这样 DateTimeFormatter 即可按该格式字符串将日期、时间字符串解析成 LocalDateTime 对象。编译、运行该程序，即可看到两个日期、时间字符串都被成功地解析成 LocalDateTime。

7.9 本章小结

本章介绍了运行Java程序时的参数，并详细解释了main方法签名的含义。为了实现字符界面程序与用户交互功能，本章介绍了两种读取键盘输入的方法。本章还介绍了System、Runtime、String、StringBuffer、StringBuilder、Math、BigDecimal、Random、Date、Calendar和TimeZone等常用类的用法。本章重点介绍了JDK 1.4所新增的正则表达式支持，包括如何创建正则表达式，以及使用Pattern、Matcher、String等类来使用正则表达式。本章还详细介绍了程序国际化的相关知识，包括消息、日期、时间国际化以及格式化等内容。除此之外，本章详细介绍了Java 8新增的日期、时间包，以及Java 8新增的日期、时间格式器。

➤➤ 本章练习

1. 定义一个长度为10的整数数组，可用于保存用户通过控制台输入的10个整数。并计算它们的平均值、最大值、最小值。

2. 将字符串"ABCDEFG"中的"CD"截取出来；再将"B"、"F"截取出来。

3. 将A1B2C3D4E5F6G7H8拆分开来，并分别存入int[]和String[]数组。得到的结果为[1,2,3,4,5,6,7,8]和[A,B,C,D,E,F,G,H]。

4. 改写第4章练习中的五子棋游戏，通过正则表达式保证用户输入必须合法。

5. 改写第4章练习中的五子棋游戏，为该程序增加国际化功能。

CHAPTER

8

第8章 Java集合

本章要点

- 集合的概念和作用
- 使用 Lambda 表达式遍历集合
- Collection 集合的常规用法
- 使用 Predicate 操作集合
- 使用 Iterator 和 foreach 循环遍历 Collection 集合
- HashSet、LinkedHashSet 的用法
- 对集合使用 Stream 进行流式编程
- EnumSet 的用法
- TreeSet 的用法
- ArrayList 和 Vector
- List 集合的常规用法
- Queue 接口与 Deque 接口
- 固定长度的 List 集合
- ArrayDeque 的用法
- PriorityQueue 的用法
- Map 的概念和常规用法
- LinkedList 集合的用法
- TreeMap 的用法
- HashMap 和 Hashtable
- 几种特殊的 Map 实现类
- Hash 算法对 HashSet、HashMap 性能的影响
- Collections 工具类的用法
- Java 9 新增的不可变集合
- Enumeration 迭代器的用法
- Java 的集合体系

Java 集合类是一种特别有用的工具类，可用于存储数量不等的对象，并可以实现常用的数据结构，如栈、队列等。除此之外，Java 集合还可用于保存具有映射关系的关联数组。Java 集合大致可分为 Set、List、Queue 和 Map 四种体系，其中 Set 代表无序、不可重复的集合；List 代表有序、重复的集合；而 Map 则代表具有映射关系的集合，Java 5 又增加了 Queue 体系集合，代表一种队列集合实现。

Java 集合就像一种容器，可以把多个对象（实际上是对象的引用，但习惯上都称对象）“丢进”该容器中。在 Java 5 之前，Java 集合会丢失容器中所有对象的数据类型，把所有对象都当成 Object 类型处理；从 Java 5 增加了泛型以后，Java 集合可以记住容器中对象的数据类型，从而可以编写出更简洁、健壮的代码。本章不会介绍泛型的知识，本章重点介绍 Java 的 4 种集合体系的功能和用法。本章将详细介绍 Java 的 4 种集合体系的常规功能，深入介绍各集合实现类所提供的独特功能，深入分析各实现类的实现机制，以及用法上的细微差别，并给出不同应用场景选择哪种集合实现类的建议。

8.1 Java 集合概述

在编程时，常常需要集中存放多个数据，例如第 6 章练习题中梭哈游戏里剩下的牌。可以使用数组来保存多个对象，但数组长度不可变化，一旦在初始化数组时指定了数组长度，这个数组长度就是不可变的，如果需要保存数量变化的数据，数组就有点无能为力了；而且数组无法保存具有映射关系的数据，如成绩表：语文—79，数学—80，这种数据看上去像两个数组，但这两个数组的元素之间有一定的关联关系。

为了保存数量不确定的数据，以及保存具有映射关系的数据（也被称为关联数组），Java 提供了集合类。集合类主要负责保存、盛装其他数据，因此集合类也被称为容器类。所有的集合类都位于 java.util 包下，后来为了处理多线程环境下的并发安全问题，Java 5 还在 java.util.concurrent 包下提供了一些多线程支持的集合类。

集合类和数组不一样，数组元素既可以是基本类型的值，也可以是对象（实际上保存的是对象的引用变量）；而集合里只能保存对象（实际上只是保存对象的引用变量，但通常习惯上认为集合里保存的是对象）。

Java 的集合类主要由两个接口派生而出：Collection 和 Map，Collection 和 Map 是 Java 集合框架的根接口，这两个接口又包含了一些子接口或实现类。如图 8.1 所示是 Collection 接口、子接口及其实现类的继承树。

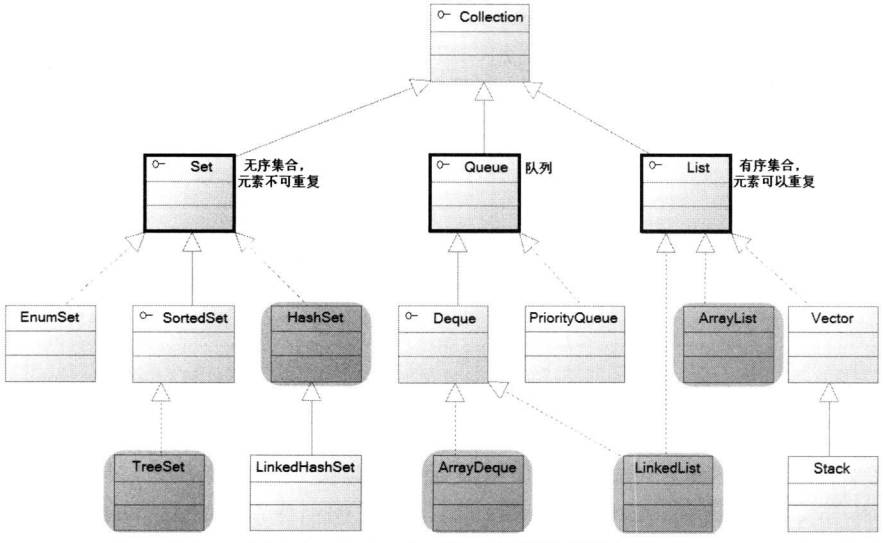


图 8.1 Collection 集合体系的继承树

图 8.1 显示了 Collection 体系里的集合，其中粗线圈出的 Set 和 List 接口是 Collection 接口派生的两个子接口，它们分别代表了无序集合和有序集合；Queue 是 Java 提供的队列实现，有点类似于 List，后面章节还会有更详细的介绍，此处不再赘述。

如图 8.2 所示是 Map 体系的继承树，所有的 Map 实现类用于保存具有映射关系的数据（也就是前面介绍的关联数组）。

图 8.2 显示了 Map 接口的众多实现类，这些实现类在功能、用法上存在一定的差异，但它们都有一个功能特征：Map 保存的每项数据都是 key-value 对，也就是由 key 和 value 两个值组成。就像前面介绍的成绩单：语文—79，数学—80，每项成绩都由两个值组成，即科目名和成绩。对于一张成绩表而言，科目通常不会重复，而成绩是可重复的，通常习惯根据科目来查阅成绩，而不会根据成绩来查阅科目。Map 与此类似，Map 里的 key 是不可重复的，key 用于标识集合里的每项数据，如果需要查阅 Map 中的数据时，总是根据 Map 的 key 来获取。

对于图 8.1 和图 8.2 中粗线标识的 4 个接口，可以把 Java 所有集合分成三大类，其中 Set 集合类似于一个罐子，把一个对象添加到 Set 集合时，Set 集合无法记住添加这个元素的顺序，所以 Set 里的元素不能重复（否则系统无法准确识别这个元素）；List 集合非常像一个数组，它可以记住每次添加元素的顺序、且 List 的长度可变。Map 集合也像一个罐子，只是它里面的每项数据都由两个值组成。图 8.3 显示了这三种集合的示意图。

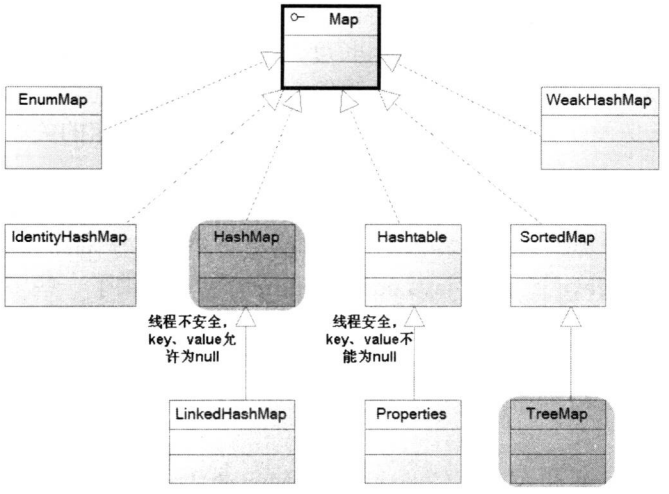


图 8.2　Map 体系的继承树

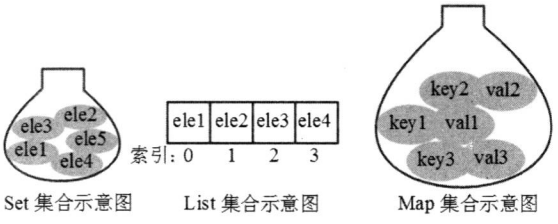


图 8.3　三种集合示意图

从图 8.3 中可以看出，如果访问 List 集合中的元素，可以直接根据元素的索引来访问；如果访问 Map 集合中的元素，可以根据每项元素的 key 来访问其 value；如果访问 Set 集合中的元素，则只能根据元素本身来访问（这也是 Set 集合里元素不允许重复的原因）。

对于 Set、List、Queue 和 Map 四种集合，最常用的实现类在图 8.1、图 8.2 中以灰色背景色覆盖，分别是 HashSet、TreeSet、ArrayList、ArrayDeque、LinkedList 和 HashMap、TreeMap 等实现类。

注意：

本章主要讲解没有涉及并发控制的集合类，对于 Java 5 新增的具有并发控制的集合类，以及 Java 7 新增的 TransferQueue 及其实现类 LinkedTransferQueue，将在第 16 章与多线程一起介绍。

8.2　Collection 和 Iterator 接口

Collection 接口是 List、Set 和 Queue 接口的父接口，该接口里定义的方法既可用于操作 Set 集合，也可用于操作 List 和 Queue 集合。Collection 接口里定义了如下操作集合元素的方法。

- boolean add(Object o)：该方法用于向集合里添加一个元素。如果集合对象被添加操作改变了，则返回 true。
- boolean addAll(Collection c)：该方法把集合 c 里的所有元素添加到指定集合里。如果集合对象被添加操作改变了，则返回 true。
- void clear()：清除集合里的所有元素，将集合长度变为 0。

- boolean contains(Object o)：返回集合里是否包含指定元素。
- boolean containsAll(Collection c)：返回集合里是否包含集合 c 里的所有元素。
- boolean isEmpty()：返回集合是否为空。当集合长度为 0 时返回 true，否则返回 false。
- Iterator iterator()：返回一个 Iterator 对象，用于遍历集合里的元素。
- boolean remove(Object o)：删除集合中的指定元素 o，当集合中包含了一个或多个元素 o 时，该方法只删除第一个符合条件的元素，该方法将返回 true。
- boolean removeAll(Collection c)：从集合中删除集合 c 里包含的所有元素（相当于用调用该方法的集合减集合 c），如果删除了一个或一个以上的元素，则该方法返回 true。
- boolean retainAll(Collection c)：从集合中删除集合 c 里不包含的元素（相当于把调用该方法的集合变成该集合和集合 c 的交集），如果该操作改变了调用该方法的集合，则该方法返回 true。
- int size()：该方法返回集合里元素的个数。
- Object[] toArray()：该方法把集合转换成一个数组，所有的集合元素变成对应的数组元素。

提示：

这些方法完全来自于 Java API 文档，读者可自行参考 API 文档来查阅这些方法的详细信息。实际上，读者无须硬性记忆这些方法，只要牢记一点：集合类就像容器，现实生活中容器的功能，无非就是添加对象、删除对象、清空容器、判断容器是否为空等，集合类就为这些功能提供了对应的方法。

下面程序示范了如何通过上面方法来操作 Collection 集合里的元素。

程序清单：codes\08\8.2\CollectionTest.java

```
public class CollectionTest
{
    public static void main(String[] args)
    {
        Collection c = new ArrayList();
        // 添加元素
        c.add("孙悟空");
        // 虽然集合里不能放基本类型的值，但 Java 支持自动装箱
        c.add(6);
        System.out.println("c 集合的元素个数为:" + c.size()); // 输出 2
        // 删除指定元素
        c.remove(6);
        System.out.println("c 集合的元素个数为:" + c.size()); // 输出 1
        // 判断是否包含指定字符串
        System.out.println("c 集合是否包含\"孙悟空\"字符串:"
            + c.contains("孙悟空")); // 输出 true
        c.add("轻量级 Java EE 企业应用实战");
        System.out.println("c 集合的元素: " + c);
        Collection books = new HashSet();
        books.add("轻量级 Java EE 企业应用实战");
        books.add("疯狂 Java 讲义");
        System.out.println("c 集合是否完全包含 books 集合? "
            + c.containsAll(books)); // 输出 false
        // 用 c 集合减去 books 集合里的元素
        c.removeAll(books);
        System.out.println("c 集合的元素: " + c);
        // 删除 c 集合里的所有元素
        c.clear();
        System.out.println("c 集合的元素: " + c);
        // 控制 books 集合里只剩下 c 集合里也包含的元素
        books.retainAll(c);
        System.out.println("books 集合的元素:" + books);
    }
}
```

上面程序中创建了两个 Collection 对象，一个是 c 集合，一个是 books 集合，其中 c 集合是 ArrayList，而 books 集合是 HashSet。虽然它们使用的实现类不同，但当把它们当成 Collection 来使用时，使用 add、

remove、clear 等方法来操作集合元素时没有任何区别。

编译和运行上面程序，看到如下运行结果：

```
c 集合的元素个数为:2
c 集合的元素个数为:1
c 集合是否包含"孙悟空"字符串:true
c 集合的元素: [孙悟空, 轻量级 Java EE 企业应用实战]
c 集合是否完全包含 books 集合? false
c 集合的元素: [孙悟空]
c 集合的元素: []
books 集合的元素:[]
```

把运行结果和粗体字标识的代码结合在一起看，可以看出 Collection 的用法有：添加元素、删除元素、返回 Collection 集合的元素个数以及清空整个集合等。

提示：

编译上面程序时，系统可能输出一些警告（warning）提示，这些警告提醒用户没有使用泛型（Generic）来限制集合里的元素类型，读者现在暂时不要理会这些警告，第 9 章会详细介绍泛型编程。

当使用 System.out 的 println()方法来输出集合对象时，将输出[ele1,ele2,...]的形式，这显然是因为所有的 Collection 实现类都重写了 toString()方法，该方法可以一次性地输出集合中的所有元素。

如果想依次访问集合里的每一个元素，则需要使用某种方式来遍历集合元素，下面介绍遍历集合元素的两种方法。

注意：

在传统模式下，把一个对象“丢进”集合中后，集合会忘记这个对象的类型——也就是说，系统把所有的集合元素都当成 Object 类型。从 JDK 1.5 以后，这种状态得到了改进：可以使用泛型来限制集合里元素的类型，并让集合记住所有集合元素的类型。关于泛型的介绍，请参考本书第 9 章。

8.2.1 使用 Lambda 表达式遍历集合

Java 8 为 Iterable 接口新增了一个 forEach(Consumer action)默认方法，该方法所需参数的类型是一个函数式接口，而 Iterable 接口是 Collection 接口的父接口，因此 Collection 集合也可直接调用该方法。

当程序调用 Iterable 的 forEach(Consumer action)遍历集合元素时，程序会依次将集合元素传给 Consumer 的 accept(T t)方法（该接口中唯一的抽象方法）。正因为 Consumer 是函数式接口，因此可以使用 Lambda 表达式来遍历集合元素。

如下程序示范了使用 Lambda 表达式来遍历集合元素。

程序清单：codes\08\8.2\CollectionEach.java

```
public class CollectionEach
{
    public static void main(String[] args)
    {
        // 创建一个集合
        Collection books = new HashSet();
        books.add("轻量级 Java EE 企业应用实战");
        books.add("疯狂 Java 讲义");
        books.add("疯狂 Android 讲义");
        // 调用 forEach()方法遍历集合
        books.forEach(obj -> System.out.println("迭代集合元素: " + obj));
    }
}
```

上面程序中粗体字代码调用了 Iterable 的 forEach()默认方法来遍历集合元素，传给该方法的参数是一个 Lambda 表达式，该 Lambda 表达式的目标类型是 Comsumer。forEach()方法会自动将集合元素逐个

地传给 Lambda 表达式的形参，这样 Lambda 表达式的代码体即可遍历到集合元素了。

8.2.2 使用 Java 8 增强的 Iterator 遍历集合元素

Iterator 接口也是 Java 集合框架的成员，但它与 Collection 系列、Map 系列的集合不一样：Collection 系列集合、Map 系列集合主要用于盛装其他对象，而 Iterator 则主要用于遍历（即迭代访问）Collection 集合中的元素，Iterator 对象也被称为迭代器。

Iterator 接口隐藏了各种 Collection 实现类的底层细节，向应用程序提供了遍历 Collection 集合元素的统一编程接口。Iterator 接口里定义了如下 4 个方法。

- boolean hasNext()：如果被迭代的集合元素还没有被遍历完，则返回 true。
- Object next()：返回集合里的下一个元素。
- void remove()：删除集合里上一次 next 方法返回的元素。
- void forEachRemaining(Consumer action)，这是 Java 8 为 Iterator 新增的默认方法，该方法可使用 Lambda 表达式来遍历集合元素。

下面程序示范了通过 Iterator 接口来遍历集合元素。

程序清单：codes\08\8.2\IteratorTest.java

```
public class IteratorTest
{
    public static void main(String[] args)
    {
        // 创建集合、添加元素的代码与前一个程序相同
        ...
        // 获取 books 集合对应的迭代器
        Iterator it = books.iterator();
        while(it.hasNext())
        {
            // it.next()方法返回的数据类型是 Object 类型，因此需要强制类型转换
            String book = (String)it.next();
            System.out.println(book);
            if (book.equals("疯狂 Java 讲义"))
            {
                // 从集合中删除上一次 next()方法返回的元素
                it.remove();
            }
            // 对 book 变量赋值，不会改变集合元素本身
            book = "测试字符串";   //①
        }
        System.out.println(books);
    }
}
```

从上面代码中可以看出，Iterator 仅用于遍历集合，Iterator 本身并不提供盛装对象的能力。如果需要创建 Iterator 对象，则必须有一个被迭代的集合。没有集合的 Iterator 仿佛无本之木，没有存在的价值。

注意：

Iterator 必须依附于 Collection 对象，若有一个 Iterator 对象，则必然有一个与之关联的 Collection 对象。Iterator 提供了两个方法来迭代访问 Collection 集合里的元素，并可通过 remove()方法来删除集合中上一次 next()方法返回的集合元素。

上面程序中①行代码对迭代变量 book 进行赋值，但当再次输出 books 集合时，会看到集合里的元素没有任何改变。这就可以得到一个结论：当使用 Iterator 对集合元素进行迭代时，Iterator 并不是把集合元素本身传给了迭代变量，而是把集合元素的值传给了迭代变量，所以修改迭代变量的值对集合元素本身没有任何影响。

当使用 Iterator 迭代访问 Collection 集合元素时，Collection 集合里的元素不能被改变，只有通过 Iterator 的 remove()方法删除上一次 next()方法返回的集合元素才可以；否则将会引发 java.util.ConcurrentModificationException 异常。下面程序示范了这一点。

程序清单：codes\08\8.2\IteratorErrorTest.java

```
public class IteratorErrorTest
{
    public static void main(String[] args)
    {
        // 创建集合、添加元素的代码与前一个程序相同
        ...
        // 获取 books 集合对应的迭代器
        Iterator it = books.iterator();
        while(it.hasNext())
        {
            String book = (String)it.next();
            System.out.println(book);
            if (book.equals("疯狂 Android 讲义"))
            {
                // 使用 Iterator 迭代过程中，不可修改集合元素，下面代码引发异常
                books.remove(book);
            }
        }
    }
}
```

上面程序中粗体字标识的代码位于 Iterator 迭代块内，也就是在 Iterator 迭代 Collection 集合过程中修改了 Collection 集合，所以程序将在运行时引发异常。

Iterator 迭代器采用的是快速失败（fail-fast）机制，一旦在迭代过程中检测到该集合已经被修改（通常是程序中的其他线程修改），程序立即引发 ConcurrentModificationException 异常，而不是显示修改后的结果，这样可以避免共享资源而引发的潜在问题。

注意：

上面程序如果改为删除“疯狂 Java 讲义”字符串，则不会引发异常，这样可能有些读者会“心存侥幸”地想：在迭代时好像也可以删除集合元素啊。实际上这是一种危险的行为：对于 HashSet 以及后面的 ArrayList 等，迭代时删除元素都会导致异常——只有在删除集合中的某个特定元素时才不会抛出异常，这是由集合类的实现代码决定的，程序员不应该这么做。

8.2.3 使用 Lambda 表达式遍历 Iterator

Java 8 起为 Iterator 新增了一个 forEachRemaining(Consumer action)方法，该方法所需的 Consumer 参数同样也是函数式接口。当程序调用 Iterator 的 forEachRemaining(Consumer action)遍历集合元素时，程序会依次将集合元素传给 Consumer 的 accept(T t)方法（该接口中唯一的抽象方法）。

如下程序示范了使用 Lambda 表达式来遍历集合元素。

程序清单：codes\08\8.2\IteratorEach.java

```
public class IteratorEach
{
    public static void main(String[] args)
    {
        // 创建集合、添加元素的代码与前一个程序相同
        ...
        // 获取 books 集合对应的迭代器
        Iterator it = books.iterator();
        // 使用 Lambda 表达式（目标类型是 Comsumer）来遍历集合元素
        it.forEachRemaining(obj -> System.out.println("迭代集合元素：" + obj));
    }
}
```

上面程序中粗体字代码调用了 Iterator 的 forEachRemaining()方法来遍历集合元素，传给该方法的参数是一个 Lambda 表达式，该 Lambda 表达式的目标类型是 Comsumer，因此上面代码也可用于遍历集合元素。

8.2.4 使用 foreach 循环遍历集合元素

除可以使用 Iterator 接口迭代访问 Collection 集合里的元素之外，使用 Java 5 提供的 foreach 循环迭代访问集合元素更加便捷。如下程序示范了使用 foreach 循环来迭代访问集合元素。

程序清单：codes\08\8.2\ForeachTest.java

```
public class ForeachTest
{
    public static void main(String[] args)
    {
        // 创建集合、添加元素的代码与前一个程序相同
        ...
        for (Object obj : books)
        {
            // 此处的 book 变量也不是集合元素本身
            String book = (String)obj;
            System.out.println(book);
            if (book.equals("疯狂 Android 讲义"))
            {
                // 下面代码会引发 ConcurrentModificationException 异常
                books.remove(book);     //①
            }
        }
        System.out.println(books);
    }
}
```

上面代码使用 foreach 循环来迭代访问 Collection 集合里的元素更加简洁，这正是 JDK 1.5 的 foreach 循环带来的优势。与使用 Iterator 接口迭代访问集合元素类似的是，foreach 循环中的迭代变量也不是集合元素本身，系统只是依次把集合元素的值赋给迭代变量，因此在 foreach 循环中修改迭代变量的值也没有任何实际意义。

同样，当使用 foreach 循环迭代访问集合元素时，该集合也不能被改变，否则将引发 ConcurrentModificationException 异常。所以上面程序中①行代码处将引发该异常。

8.2.5 使用 Java 8 新增的 Predicate 操作集合

Java 8 起为 Collection 集合新增了一个 removeIf(Predicate filter)方法，该方法将会批量删除符合 filter 条件的所有元素。该方法需要一个 Predicate（谓词）对象作为参数，Predicate 也是函数式接口，因此可使用 Lambda 表达式作为参数。

如下程序示范了使用 Predicate 来过滤集合。

程序清单：codes\08\8.2\PredicateTest.java

```
// 创建一个集合
Collection books = new HashSet();
books.add(new String("轻量级 Java EE 企业应用实战"));
books.add(new String("疯狂 Java 讲义"));
books.add(new String("疯狂 iOS 讲义"));
books.add(new String("疯狂 Ajax 讲义"));
books.add(new String("疯狂 Android 讲义"));
// 使用 Lambda 表达式（目标类型是 Predicate）过滤集合
books.removeIf(ele -> ((String)ele).length() < 10);
System.out.println(books);
```

上面程序中粗体字代码调用了 Collection 集合的 removeIf()方法批量删除集合中符合条件的元素，程序传入一个 Lambda 表达式作为过滤条件：所有长度小于 10 的字符串元素都会被删除。编译、运行这段代码，可以看到如下输出：

```
[疯狂 Android 讲义, 轻量级 Java EE 企业应用实战]
```

使用 Predicate 可以充分简化集合的运算，假设依然有上面程序所示的 books 集合，如果程序有如下三个统计需求：

- 统计书名中出现“疯狂”字符串的图书数量。
- 统计书名中出现“Java”字符串的图书数量。
- 统计书名长度大于 10 的图书数量。

此处只是一个假设，实际上还可能有更多的统计需求。如果采用传统的编程方式来完成这些需求，则需要执行三次循环，但采用 Predicate 只需要一个方法即可。如下程序示范了这种用法。

程序清单：codes\08\8.2\PredicateTest2.java

```
public class PredicateTest2
{
    public static void main(String[] args)
    {
        // 创建 books 集合、为 books 集合添加元素的代码与前一个程序相同
        ...
        // 统计书名包含“疯狂”子串的图书数量
        System.out.println(calAll(books , ele->((String)ele).contains("疯狂")));
        // 统计书名包含“Java”子串的图书数量
        System.out.println(calAll(books , ele->((String)ele).contains("Java")));
        // 统计书名字符串长度大于 10 的图书数量
        System.out.println(calAll(books , ele->((String)ele).length() > 10));
    }
    public static int calAll(Collection books , Predicate p)
    {
        int total = 0;
        for (Object obj : books)
        {
            // 使用 Predicate 的 test()方法判断该对象是否满足 Predicate 指定的条件
            if (p.test(obj))
            {
                total ++;
            }
        }
        return total;
    }
}
```

上面程序先定义了一个 calAll()方法，该方法将会使用 Predicate 判断每个集合元素是否符合特定条件——该条件将通过 Predicate 参数动态传入。从上面程序中三行粗体字代码可以看到，程序传入了三个 Lambda 表达式（其目标类型都是 Predicate），这样 calAll()方法就只会统计满足 Predicate 条件的图书。

8.2.6 使用 Java 8 新增的 Stream 操作集合

Java 8 还新增了 Stream、IntStream、LongStream、DoubleStream 等流式 API，这些 API 代表多个支持串行和并行聚集操作的元素。上面 4 个接口中，Stream 是一个通用的流接口，而 IntStream、LongStream、DoubleStream 则代表元素类型为 int、long、double 的流。

Java 8 还为上面每个流式 API 提供了对应的 Builder，例如 Stream.Builder、IntStream.Builder、LongStream.Builder、DoubleStream.Builder，开发者可以通过这些 Builder 来创建对应的流。

独立使用 Stream 的步骤如下：

1. 使用 Stream 或 XxxStream 的 builder()类方法创建该 Stream 对应的 Builder。
2. 重复调用 Builder 的 add()方法向该流中添加多个元素。
3. 调用 Builder 的 build()方法获取对应的 Stream。
4. 调用 Stream 的聚集方法。

在上面 4 个步骤中，第 4 步可以根据具体需求来调用不同的方法，Stream 提供了大量的聚集方法供用户调用，具体可参考 Stream 或 XxxStream 的 API 文档。对于大部分聚集方法而言，每个 Stream 只能执行一次。例如如下程序。

程序清单：codes\08\8.2\IntStreamTest.java

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

```
    {
        IntStream is = IntStream.builder()
            .add(20)
            .add(13)
            .add(-2)
            .add(18)
            .build();
        // 下面调用聚集方法的代码每次只能执行一行
        System.out.println("is所有元素的最大值：" + is.max().getAsInt());
        System.out.println("is所有元素的最小值：" + is.min().getAsInt());
        System.out.println("is所有元素的总和：" + is.sum());
        System.out.println("is所有元素的总数：" + is.count());
        System.out.println("is所有元素的平均值：" + is.average());
        System.out.println("is所有元素的平方是否都大于20:"
            + is.allMatch(ele -> ele * ele > 20));
        System.out.println("is是否包含任何元素的平方大于20:"
            + is.anyMatch(ele -> ele * ele > 20));
        // 将is映射成一个新Stream，新Stream的每个元素是原Stream元素的2倍+1
        IntStream newIs = is.map(ele -> ele * 2 + 1);
        // 使用方法引用的方式来遍历集合元素
        newIs.forEach(System.out::println); // 输出41 27 -3 37
    }
}
```

上面程序先创建了一个IntStream，接下来分别多次调用IntStream的聚集方法执行操作，这样即可获取该流的相关信息。注意：上面粗体字代码每次只能执行一行，因此需要把其他粗体字代码注释掉。

Stream 提供了大量的方法进行聚集操作，这些方法既可以是“中间的”（intermediate），也可以是“末端的”（terminal）。

- 中间方法：中间操作允许流保持打开状态，并允许直接调用后续方法。上面程序中的map()方法就是中间方法。中间方法的返回值是另外一个流。
- 末端方法：末端方法是对流的最终操作。当对某个Stream执行末端方法后，该流将会被“消耗”且不再可用。上面程序中的sum()、count()、average()等方法都是末端方法。

除此之外，关于流的方法还有如下两个特征。

- 有状态的方法：这种方法会给流增加一些新的属性，比如元素的唯一性、元素的最大数量、保证元素以排序的方式被处理等。有状态的方法往往需要更大的性能开销。
- 短路方法：短路方法可以尽早结束对流的操作，不必检查所有的元素。

下面简单介绍一下Stream常用的中间方法。

- filter(Predicate predicate)：过滤Stream中所有不符合predicate的元素。
- mapToXxx(ToXxxFunction mapper)：使用ToXxxFunction对流中的元素执行一对一的转换，该方法返回的新流中包含了ToXxxFunction转换生成的所有元素。
- peek(Consumer action)：依次对每个元素执行一些操作，该方法返回的流与原有流包含相同的元素。该方法主要用于调试。
- distinct()：该方法用于排序流中所有重复的元素（判断元素重复的标准是使用equals()比较返回true）。这是一个有状态的方法。
- sorted()：该方法用于保证流中的元素在后续的访问中处于有序状态。这是一个有状态的方法。
- limit(long maxSize)：该方法用于保证对该流的后续访问中最大允许访问的元素个数。这是一个有状态的、短路方法。

下面简单介绍一下Stream常用的末端方法。

- forEach(Consumer action)：遍历流中所有元素，对每个元素执行action。
- toArray()：将流中所有元素转换为一个数组。
- reduce()：该方法有三个重载的版本，都用于通过某种操作来合并流中的元素。
- min()：返回流中所有元素的最小值。
- max()：返回流中所有元素的最大值。

- count()：返回流中所有元素的数量。
- anyMatch(Predicate predicate)：判断流中是否至少包含一个元素符合 Predicate 条件。
- allMatch(Predicate predicate)：判断流中是否每个元素都符合 Predicate 条件。
- noneMatch(Predicate predicate)：判断流中是否所有元素都不符合 Predicate 条件。
- findFirst()：返回流中的第一个元素。
- findAny()：返回流中的任意一个元素。

除此之外，Java 8 允许使用流式 API 来操作集合，Collection 接口提供了一个 stream()默认方法，该方法可返回该集合对应的流，接下来即可通过流式 API 来操作集合元素。由于 Stream 可以对集合元素进行整体的聚集操作，因此 Stream 极大地丰富了集合的功能。

例如，对于 8.2.5 节介绍的示例程序，该程序需要额外定义一个 calAll()方法来遍历集合元素，然后依次对每个集合元素进行判断——这太麻烦了。如果使用 Stream，即可直接对集合中所有元素进行批量操作。下面使用 Stream 来改写这个程序。

程序清单：codes\08\8.2\CollectionStream.java

```
public class CollectionStream
{
    public static void main(String[] args)
    {
        // 创建 books 集合、为 books 集合添加元素的代码与 8.2.5 节的程序相同
        ...
        // 统计书名包含“疯狂”子串的图书数量
        System.out.println(books.stream()
            .filter(ele->((String)ele).contains("疯狂"))
            .count()); // 输出 4
        // 统计书名包含“Java”子串的图书数量
        System.out.println(books.stream()
            .filter(ele->((String)ele).contains("Java") )
            .count()); // 输出 2
        // 统计书名字符串长度大于 10 的图书数量
        System.out.println(books.stream()
            .filter(ele->((String)ele).length() > 10)
            .count()); // 输出 2
        // 先调用 Collection 对象的 stream()方法将集合转换为 Stream
        // 再调用 Stream 的 mapToInt()方法获取原有的 Stream 对应的 IntStream
        books.stream().mapToInt(ele -> ((String)ele).length())
            // 调用 forEach()方法遍历 IntStream 中每个元素
            .forEach(System.out::println);// 输出 8  11  16  7  8
    }
}
```

从上面程序中粗体字代码可以看出，程序只要调用 Collection 的 stream()方法即可返回该集合对应的 Stream，接下来就可通过 Stream 提供的方法对所有集合元素进行处理，这样大大地简化了集合编程的代码，这也是 Stream 编程带来的优势。

上面程序中最后一段粗体字代码先调用 Collection 对象的 stream()方法将集合转换为 Stream 对象，然后调用 Stream 对象的 mapToInt()方法将其转换为 IntStream——这个 mapToInt()方法就是一个中间方法，因此程序可继续调用 IntStream 的 forEach()方法来遍历流中的元素。

8.3 Set 集合

前面已经介绍过 Set 集合，它类似于一个罐子，程序可以依次把多个对象“丢进”Set 集合，而 Set 集合通常不能记住元素的添加顺序。Set 集合与 Collection 基本相同，没有提供任何额外的方法。实际上 Set 就是 Collection，只是行为略有不同（Set 不允许包含重复元素）。

Set 集合不允许包含相同的元素，如果试图把两个相同的元素加入同一个 Set 集合中，则添加操作失败，add()方法返回 false，且新元素不会被加入。

上面介绍的是 Set 集合的通用知识，因此完全适合后面介绍的 HashSet、TreeSet 和 EnumSet 三个实

现类，只是三个实现类还各有特色。

8.3.1 HashSet 类

HashSet 是 Set 接口的典型实现，大多数时候使用 Set 集合时就是使用这个实现类。HashSet 按 Hash 算法来存储集合中的元素，因此具有很好的存取和查找性能。

HashSet 具有以下特点。

- 不能保证元素的排列顺序，顺序可能与添加顺序不同，顺序也有可能发生变化。
- HashSet 不是同步的，如果多个线程同时访问一个 HashSet，假设有两个或者两个以上线程同时修改了 HashSet 集合时，则必须通过代码来保证其同步。
- 集合元素值可以是 null。

当向 HashSet 集合中存入一个元素时，HashSet 会调用该对象的 hashCode()方法来得到该对象的 hashCode 值，然后根据该 hashCode 值决定该对象在 HashSet 中的存储位置。如果有两个元素通过 equals()方法比较返回 true，但它们的 hashCode()方法返回值不相等，HashSet 将会把它们存储在不同的位置，依然可以添加成功。

也就是说，HashSet 集合判断两个元素相等的标准是两个对象通过 equals()方法比较相等，并且两个对象的 hashCode()方法返回值也相等。

下面程序分别提供了三个类 A、B 和 C，它们分别重写了 equals()、hashCode()两个方法的一个或全部，通过此程序可以让读者看到 HashSet 判断集合元素相同的标准。

程序清单：codes\08\8.3\HashSetTest.java

```
// 类 A 的 equals()方法总是返回 true，但没有重写其 hashCode()方法
class A
{
    public boolean equals(Object obj)
    {
        return true;
    }
}
// 类 B 的 hashCode()方法总是返回 1，但没有重写其 equals()方法
class B
{
    public int hashCode()
    {
        return 1;
    }
}
// 类 C 的 hashCode()方法总是返回 2，且重写其 equals()方法总是返回 true
class C
{
    public int hashCode()
    {
        return 2;
    }
    public boolean equals(Object obj)
    {
        return true;
    }
}
public class HashSetTest
{
    public static void main(String[] args)
    {
        HashSet books = new HashSet();
        // 分别向 books 集合中添加两个 A 对象、两个 B 对象、两个 C 对象
        books.add(new A());
        books.add(new A());
        books.add(new B());
        books.add(new B());
        books.add(new C());
        books.add(new C());
```

```
            System.out.println(books);
        }
    }
```

上面程序中向 books 集合中分别添加了两个 A 对象、两个 B 对象和两个 C 对象，其中 C 类重写了 equals()方法总是返回 true，hashCode()方法总是返回 2，这将导致 HashSet 把两个 C 对象当成同一个对象。运行上面程序，看到如下运行结果：

```
[B@1, B@1, C@2, A@5483cd, A@9931f5]
```

从上面程序可以看出，即使两个 A 对象通过 equals()方法比较返回 true，但 HashSet 依然把它们当成两个对象；即使两个 B 对象的 hashCode()返回相同值（都是 1），但 HashSet 依然把它们当成两个对象。

这里有一个注意点：当把一个对象放入 HashSet 中时，如果需要重写该对象对应类的 equals()方法，则也应该重写其 hashCode()方法。规则是：如果两个对象通过 equals()方法比较返回 true，这两个对象的 hashCode 值也应该相同。

如果两个对象通过 equals()方法比较返回 true，但这两个对象的 hashCode()方法返回不同的 hashCode 值时，这将导致 HashSet 会把这两个对象保存在 Hash 表的不同位置，从而使两个对象都可以添加成功，这就与 Set 集合的规则冲突了。

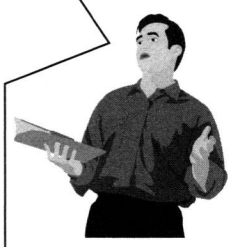
如果两个对象的 hashCode()方法返回的 hashCode 值相同，但它们通过 equals()方法比较返回 false 时将更麻烦：因为两个对象的 hashCode 值相同，HashSet 将试图把它们保存在同一个位置，但又不行（否则将只剩下一个对象），所以实际上会在这个位置用链式结构来保存多个对象；而 HashSet 访问集合元素时也是根据元素的 hashCode 值来快速定位的，如果 HashSet 中两个以上的元素具有相同的 hashCode 值，将会导致性能下降。

注意：

如果需要把某个类的对象保存到 HashSet 集合中，重写这个类的 equals()方法和 hashCode()方法时，应该尽量保证两个对象通过 equals()方法比较返回 true 时，它们的 hashCode()方法返回值也相等。

答：hash（也被翻译为哈希、散列）算法的功能是，它能保证快速查找被检索的对象，hash 算法的价值在于速度。当需要查询集合中某个元素时，hash 算法可以直接根据该元素的 hashCode 值计算出该元素的存储位置，从而快速定位该元素。为了理解这个概念，可以先看数组（数组是所有能存储一组元素里最快的数据结构）。数组可以包含多个元素，每个元素都有索引，如果需要访问某个数组元素，只需提供该元素的索引，接下来即可根据该索引计算该元素在内存里的存储位置。

表面上看起来，HashSet 集合里的元素都没有索引，实际上当程序向 HashSet 集合中添加元素时，HashSet 会根据该元素的 hashCode 值来计算它的存储位置，这样也可快速定位该元素。为什么不直接使用数组、还需要使用 HashSet 呢？因为数组元素的索引是连续的，而且数组的长度是固定的，无法自由增加数组的长度。而 HashSet 就不一样了，HashSet 采用每个元素的 hashCode 值来计算其存储位置，从而可以自由增加 HashSet 的长度，并可以根据元素的 hashCode 值来访问元素。因此，当从 HashSet 中访问元素时，HashSet 先计算该元素的 hashCode 值（也就是调用该对象的 hashCode()方法的返回值），然后直接到该 hashCode 值对应的位置去取出该元素——这就是 HashSet 速度很快的原因。



HashSet 中每个能存储元素的“槽位”（slot）通常称为“桶”（bucket），如果有多个元素的 hashCode 值相同，但它们通过 equals()方法比较返回 false，就需要在一个“桶”里放多个元素，这样会导致性能下降。

前面介绍了 hashCode()方法对于 HashSet 的重要性（实际上，对象的 hashCode 值对于后面的 HashMap 同样重要），下面给出重写 hashCode()方法的基本规则。

- 在程序运行过程中，同一个对象多次调用 hashCode()方法应该返回相同的值。
- 当两个对象通过 equals()方法比较返回 true 时，这两个对象的 hashCode()方法应返回相等的值。
- 对象中用作 equals()方法比较标准的实例变量，都应该用于计算 hashCode 值。

下面给出重写 hashCode()方法的一般步骤。

① 把对象内每个有意义的实例变量（即每个参与 equals()方法比较标准的实例变量）计算出一个 int 类型的 hashCode 值。计算方式如表 8.1 所示。

表 8.1 hashCode 值的计算方式

实例变量类型	计算方式	实例变量类型	计算方式
boolean	hashCode = (f ? 0 : 1);	float	hashCode = Float.floatToIntBits(f);
整数类型（byte、short、char、int）	hashCode = (int)f;	double	long l = Double.doubleToLongBits(f); hashCode = (int)(l ^ (l >>> 32));
long	hashCode = (int)(f ^ (f >>>32));	引用类型	hashCode = f.hashCode();

② 用第 1 步计算出来的多个 hashCode 值组合计算出一个 hashCode 值返回。例如如下代码：

```
return f1.hashCode() + (int)f2;
```

为了避免直接相加产生偶然相等（两个对象的 f1、f2 实例变量并不相等，但它们的 hashCode 的和恰好相等），可以通过为各实例变量的 hashCode 值乘以任意一个质数后再相加。例如如下代码：

```
return f1.hashCode() * 19 + (int)f2 * 31;
```

如果向 HashSet 中添加一个可变对象后，后面程序修改了该可变对象的实例变量，则可能导致它与集合中的其他元素相同（即两个对象通过 equals()方法比较返回 true，两个对象的 hashCode 值也相等），这就有可能导致 HashSet 中包含两个相同的对象。下面程序演示了这种情况。

程序清单：codes\08\8.3\HashSetTest2.java

```
class R
{
    int count;
    public R(int count)
    {
        this.count = count;
    }
    public String toString()
    {
        return "R[count:" + count + "]";
    }
    public boolean equals(Object obj)
    {
        if(this == obj)
            return true;
        if (obj != null && obj.getClass() == R.class)
        {
            R r = (R)obj;
            return this.count == r.count;
        }
        return false;
    }
    public int hashCode()
    {
        return this.count;
    }
}
public class HashSetTest2
```

```
{
    public static void main(String[] args)
    {
        HashSet hs = new HashSet();
        hs.add(new R(5));
        hs.add(new R(-3));
        hs.add(new R(9));
        hs.add(new R(-2));
        // 打印 HashSet 集合，集合元素没有重复
        System.out.println(hs);
        // 取出第一个元素
        Iterator it = hs.iterator();
        R first = (R)it.next();
        // 为第一个元素的 count 实例变量赋值
        first.count = -3;     // ①
        // 再次输出 HashSet 集合，集合元素有重复元素
        System.out.println(hs);
        // 删除 count 为-3 的 R 对象
        hs.remove(new R(-3));    // ②
        // 可以看到被删除了一个 R 元素
        System.out.println(hs);
        System.out.println("hs 是否包含 count 为-3 的 R 对象？"
            + hs.contains(new R(-3))); // 输出 false
        System.out.println("hs 是否包含 count 为-2 的 R 对象？"
            + hs.contains(new R(-2))); // 输出 false
    }
}
```

上面程序中提供了R类，R类重写了equals(Object obj)方法和 hashCode()方法，这两个方法都是根据 R 对象的 count 实例变量来判断的。上面程序的①号粗体字代码处改变了 Set 集合中第一个 R 对象的 count 实例变量的值，这将导致该 R 对象与集合中的其他对象相同。程序运行结果如图 8.4 所示。

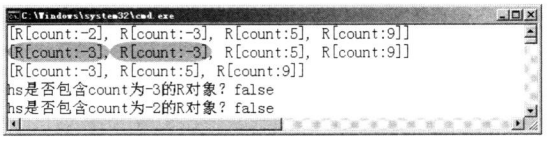


图 8.4　HashSet 集合中出现重复的元素

正如图 8.4 中所见到的，HashSet 集合中的第 1 个元素和第 2 个元素完全相同，这表明两个元素已经重复。此时 HashSet 会比较混乱：当试图删除 count 为-3 的 R 对象时，HashSet 会计算出该对象的 hashCode 值，从而找出该对象在集合中的保存位置，然后把此处的对象与 count 为-3 的 R 对象通过 equals()方法进行比较，如果相等则删除该对象——HashSet 只有第 2 个元素才满足该条件（第 1 个元素实际上保存在 count 为-2 的 R 对象对应的位置），所以第 2 个元素被删除。至于第一个 count 为-3 的 R 对象，它保存在 count 为-2 的 R 对象对应的位置，但使用 equals()方法拿它和 count 为-2 的 R 对象比较时又返回 false——这将导致 HashSet 不可能准确访问该元素。

由此可见，当程序把可变对象添加到 HashSet 中之后，尽量不要去修改该集合元素中参与计算 hashCode()、equals()的实例变量，否则将会导致 HashSet 无法正确操作这些集合元素。

注意：

当向 HashSet 中添加可变对象时，必须十分小心。如果修改 HashSet 集合中的对象，有可能导致该对象与集合中的其他对象相等，从而导致 HashSet 无法准确访问该对象。

8.3.2　LinkedHashSet 类

HashSet 还有一个子类 LinkedHashSet，LinkedHashSet 集合也是根据元素的 hashCode 值来决定元素的存储位置，但它同时使用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。也就是说，当遍历 LinkedHashSet 集合里的元素时，LinkedHashSet 将会按元素的添加顺序来访问集合里的元素。

LinkedHashSet 需要维护元素的插入顺序，因此性能略低于 HashSet 的性能，但在迭代访问 Set 里的全部元素时将有很好的性能，因为它以链表来维护内部顺序。

程序清单：codes\08\8.3\LinkedHashSetTest.java

```
public class LinkedHashSetTest
{
    public static void main(String[] args)
    {
        LinkedHashSet books = new LinkedHashSet();
        books.add("疯狂 Java 讲义");
        books.add("轻量级 Java EE 企业应用实战");
        System.out.println(books);
        // 删除 疯狂 Java 讲义
        books.remove("疯狂 Java 讲义");
        // 重新添加 疯狂 Java 讲义
        books.add("疯狂 Java 讲义");
        System.out.println(books);
    }
}
```

编译、运行上面程序，看到如下输出：

```
[疯狂 Java 讲义, 轻量级 Java EE 企业应用实战]
[轻量级 Java EE 企业应用实战, 疯狂 Java 讲义]
```

输出 LinkedHashSet 集合的元素时，元素的顺序总是与添加顺序一致。

中華工匠

注意：

虽然 LinkedHashSet 使用了链表记录集合元素的添加顺序，但 LinkedHashSet 依然是 HashSet，因此它依然不允许集合元素重复。

8.3.3 TreeSet 类

TreeSet 是 SortedSet 接口的实现类，正如 SortedSet 名字所暗示的，TreeSet 可以确保集合元素处于排序状态。与 HashSet 集合相比，TreeSet 还提供了如下几个额外的方法。

- Comparator comparator()：如果 TreeSet 采用了定制排序，则该方法返回定制排序所使用的 Comparator；如果 TreeSet 采用了自然排序，则返回 null。
- Object first()：返回集合中的第一个元素。
- Object last()：返回集合中的最后一个元素。
- Object lower(Object e)：返回集合中位于指定元素之前的元素（即小于指定元素的最大元素，参考元素不需要是 TreeSet 集合里的元素）。
- Object higher (Object e)：返回集合中位于指定元素之后的元素（即大于指定元素的最小元素，参考元素不需要是 TreeSet 集合里的元素）。
- SortedSet subSet(Object fromElement, Object toElement)：返回此 Set 的子集合，范围从 fromElement（包含）到 toElement（不包含）。
- SortedSet headSet(Object toElement)：返回此 Set 的子集，由小于 toElement 的元素组成。
- SortedSet tailSet(Object fromElement)：返回此 Set 的子集，由大于或等于 fromElement 的元素组成。

提示：

表面上看起来这些方法很多，其实它们很简单：因为 TreeSet 中的元素是有序的，所以增加了访问第一个、前一个、后一个、最后一个元素的方法，并提供了三个从 TreeSet 中截取子 TreeSet 的方法。

下面程序测试了 TreeSet 的通用用法。

程序清单：codes\08\8.3\TreeSetTest.java

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

```
        TreeSet nums = new TreeSet();
        // 向 TreeSet 中添加四个 Integer 对象
        nums.add(5);
        nums.add(2);
        nums.add(10);
        nums.add(-9);
        // 输出集合元素，看到集合元素已经处于排序状态
        System.out.println(nums);
        // 输出集合里的第一个元素
        System.out.println(nums.first()); // 输出-9
        // 输出集合里的最后一个元素
        System.out.println(nums.last());  // 输出 10
        // 返回小于 4 的子集，不包含 4
        System.out.println(nums.headSet(4)); // 输出[-9, 2]
        // 返回大于 5 的子集，如果 Set 中包含 5，子集中还包含 5
        System.out.println(nums.tailSet(5)); // 输出 [5, 10]
        // 返回大于等于-3、小于 4 的子集
        System.out.println(nums.subSet(-3 , 4)); // 输出[2]
    }
}
```

根据上面程序的运行结果即可看出，TreeSet 并不是根据元素的插入顺序进行排序的，而是根据元素实际值的大小来进行排序的。

与 HashSet 集合采用 hash 算法来决定元素的存储位置不同，TreeSet 采用红黑树的数据结构来存储集合元素。那么 TreeSet 进行排序的规则是怎样的呢？TreeSet 支持两种排序方法：自然排序和定制排序。在默认情况下，TreeSet 采用自然排序。

1. 自然排序

TreeSet 会调用集合元素的 compareTo(Object obj)方法来比较元素之间的大小关系，然后将集合元素按升序排列，这种方式就是自然排序。

Java 提供了一个 Comparable 接口，该接口里定义了一个 compareTo(Object obj)方法，该方法返回一个整数值，实现该接口的类必须实现该方法，实现了该接口的类的对象就可以比较大小。当一个对象调用该方法与另一个对象进行比较时，例如 obj1.compareTo(obj2)，如果该方法返回 0，则表明这两个对象相等；如果该方法返回一个正整数，则表明 obj1 大于 obj2；如果该方法返回一个负整数，则表明 obj1 小于 obj2。

Java 的一些常用类已经实现了 Comparable 接口，并提供了比较大小的标准。下面是实现了 Comparable 接口的常用类。

- BigDecimal、BigInteger 以及所有的数值型对应的包装类：按它们对应的数值大小进行比较。
- Character：按字符的 UNICODE 值进行比较。
- Boolean：true 对应的包装类实例大于 false 对应的包装类实例。
- String：按字符串中字符的 UNICODE 值进行比较。
- Date、Time：后面的时间、日期比前面的时间、日期大。

如果试图把一个对象添加到 TreeSet 时，则该对象的类必须实现 Comparable 接口，否则程序将会抛出异常。如下程序示范了这个错误。

程序清单：codes\08\8.3\TreeSetErrorTest.java

```
class Err { }
public class TreeSetErrorTest
{
    public static void main(String[] args)
    {
        TreeSet ts = new TreeSet();
        //向 TreeSet 集合中添加两个 Err 对象
        ts.add(new Err());
    }
}
```

上面程序试图向 TreeSet 集合中添加 Err 对象，在自然排序时，集合元素必须实现 Comparable 接口，

否则将会引发运行时异常：ClassCastException——因此，TreeSet 要求自然排序的集合元素必须都实现该接口。

注意：

Java 9 改进了 TreeSet 实现，如果采用自然排序的 Set 集合的元素没有实现 Comparable 接口，程序就会立即引发 ClassCastException 异常。

还有一点必须指出：大部分类在实现 compareTo(Object obj)方法时，都需要将被比较对象 obj 强制类型转换成相同类型，因为只有相同类的两个实例才会比较大小。当试图把一个对象添加到 TreeSet 集合时，TreeSet 会调用该对象的 compareTo(Object obj)方法与集合中的其他元素进行比较——这就要求集合中的其他元素与该元素是同一个类的实例。也就是说，向 TreeSet 中添加的应该是同一个类的对象，否则也会引发 ClassCastException 异常。如下程序示范了这个错误。

程序清单：codes\08\8.3\TreeSetErrorTest2.java

```
public class TreeSetErrorTest2
{
    public static void main(String[] args)
    {
        TreeSet ts = new TreeSet();
        // 向 TreeSet 集合中添加两个对象
        ts.add(new String("疯狂 Java 讲义"));
        ts.add(new Date());   // ①
    }
}
```

上面程序先向 TreeSet 集合中添加了一个字符串对象，这个操作完全正常。当添加第二个 Date 对象时，TreeSet 就会调用该对象的 compareTo(Object obj)方法与集合中的其他元素进行比较——Date 对象的 compareTo(Object obj)方法无法与字符串对象比较大小，所以上面程序将在①代码处引发异常。

如果向 TreeSet 中添加的对象是程序员自定义类的对象，则可以向 TreeSet 中添加多种类型的对象，前提是用户自定义类实现了 Comparable 接口，且实现 compareTo(Object obj)方法没有进行强制类型转换。但当试图取出 TreeSet 里的集合元素时，不同类型的元素依然会发生 ClassCastException 异常。

总结：如果希望 TreeSet 能正常运作，TreeSet 只能添加同一种类型的对象。

当把一个对象加入 TreeSet 集合中时，TreeSet 调用该对象的 compareTo(Object obj)方法与容器中的其他对象比较大小，然后根据红黑树结构找到它的存储位置。如果两个对象通过 compareTo(Object obj)方法比较相等，新对象将无法添加到 TreeSet 集合中。

对于 TreeSet 集合而言，它判断两个对象是否相等的唯一标准是：两个对象通过 compareTo(Object obj)方法比较是否返回 0——如果通过 compareTo(Object obj)方法比较返回 0，TreeSet 则会认为它们相等；否则就认为它们不相等。

程序清单：codes\08\8.3\TreeSetTest2.java

```
class Z implements Comparable
{
    int age;
    public Z(int age)
    {
        this.age = age;
    }
    // 重写 equals()方法，总是返回 true
    public boolean equals(Object obj)
    {
        return true;
    }
    // 重写了 compareTo(Object obj)方法，总是返回 1
    public int compareTo(Object obj)
    {
        return 1;
    }
```

```
}
public class TreeSetTest2
{
    public static void main(String[] args)
    {
        TreeSet set = new TreeSet();
        Z z1 = new Z(6);
        set.add(z1);
        // 第二次添加同一个对象，输出 true，表明添加成功
        System.out.println(set.add(z1));    //①
        // 下面输出 set 集合，将看到有两个元素
        System.out.println(set);
        // 修改 set 集合的第一个元素的 age 变量
        ((Z)(set.first())).age = 9;
        // 输出 set 集合的最后一个元素的 age 变量，将看到也变成了 9
        System.out.println(((Z)(set.last())).age);
    }
}
```

程序中①代码行把同一个对象再次添加到 TreeSet 集合中，因为 z1 对象的 compareTo(Object obj)方法总是返回 1，虽然它的 equals()方法总是返回 true，但 TreeSet 会认为 z1 对象和它自己也不相等，因此 TreeSet 可以添加两个 z1 对象。图 8.5 显示了 TreeSet 及 Z 对象在内存中的存储示意图。

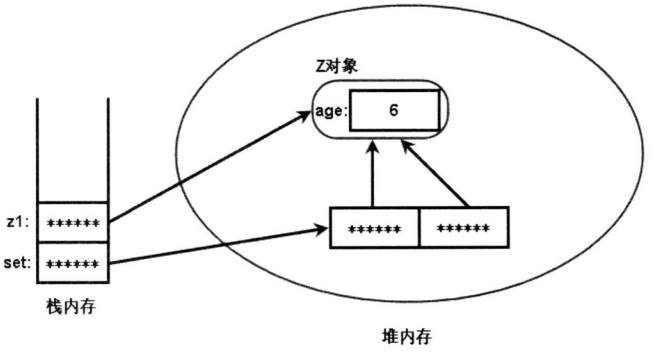


图 8.5　TreeSet 及 Z 对象在内存中的存储示意图

从图 8.5 可以看到 TreeSet 对象保存的两个元素（集合里的元素总是引用，但习惯上把被引用的对象称为集合元素），实际上是同一个元素。所以当修改 TreeSet 集合里第一个元素的 age 变量后，该 TreeSet 集合里最后一个元素的 age 变量也随之改变了。

由此应该注意一个问题：当需要把一个对象放入 TreeSet 中，重写该对象对应类的 equals()方法时，应保证该方法与 compareTo(Object obj)方法有一致的结果，其规则是：如果两个对象通过 equals()方法比较返回 true 时，这两个对象通过 compareTo(Object obj)方法比较应返回 0。

如果两个对象通过 compareTo(Object obj)方法比较返回 0 时，但它们通过 equals()方法比较返回 false 将很麻烦，因为两个对象通过 compareTo(Object obj)方法比较相等，TreeSet 不会让第二个元素添加进去，这就会与 Set 集合的规则产生冲突。

如果向 TreeSet 中添加一个可变对象后，并且后面程序修改了该可变对象的实例变量，这将导致它与其他对象的大小顺序发生了改变，但 TreeSet 不会再次调整它们的顺序，甚至可能导致 TreeSet 中保存的这两个对象通过 compareTo(Object obj)方法比较返回 0。下面程序演示了这种情况。

程序清单：*codes\08\8.3\TreeSetTest3.java*

```
class R implements Comparable
{
    int count;
    public R(int count)
    {
        this.count = count;
    }
    public String toString()
    {
        return "R[count:" + count + "]";
    }
    // 重写 equals()方法，根据 count 来判断是否相等
    public boolean equals(Object obj)
    {
        if (this == obj)
        {
            return true;
```

```
        }
        if(obj != null && obj.getClass() == R.class)
        {
            R r = (R)obj;
            return r.count == this.count;
        }
        return false;
    }
    // 重写 compareTo()方法，根据 count 来比较大小
    public int compareTo(Object obj)
    {
        R r = (R)obj;
        return count > r.count ? 1 :
            count < r.count ? -1 : 0;
    }
}
public class TreeSetTest3
{
    public static void main(String[] args)
    {
        TreeSet ts = new TreeSet();
        ts.add(new R(5));
        ts.add(new R(-3));
        ts.add(new R(9));
        ts.add(new R(-2));
        // 打印 TreeSet 集合，集合元素是有序排列的
        System.out.println(ts);    // ①
        // 取出第一个元素
        R first = (R)ts.first();
        // 对第一个元素的 count 赋值
        first.count = 20;
        // 取出最后一个元素
        R last = (R)ts.last();
        // 对最后一个元素的 count 赋值，与第二个元素的 count 相同
        last.count = -2;
        // 再次输出将看到 TreeSet 里的元素处于无序状态，且有重复元素
        System.out.println(ts);   // ②
        // 删除实例变量被改变的元素，删除失败
        System.out.println(ts.remove(new R(-2)));   // ③
        System.out.println(ts);
        // 删除实例变量没有被改变的元素，删除成功
        System.out.println(ts.remove(new R(5)));    // ④
        System.out.println(ts);
    }
}
```

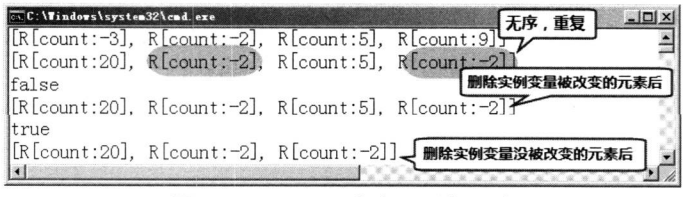


图 8.6 TreeSet 中出现重复元素

上面程序中的R对象对应的类正常重写了equals()方法和compareTo()方法，这两个方法都以R对象的count实例变量作为判断的依据。当程序执行①行代码时，看到程序输出的Set集合元素处于有序状态；因为R类是一个可变类，因此可以改变R对象的count实例变量的值，程序通过粗体字代码行改变了该集合里第一个元素和最后一个元素的count实例变量的值。当程序执行②行代码输出时，将看到该集合处于无序状态，而且集合中包含了重复元素。运行上面程序，看到如图8.6所示的结果。

一旦改变了TreeSet集合里可变元素的实例变量，当再试图删除该对象时，TreeSet也会删除失败（甚至集合中原有的、实例变量没被修改但与修改后元素相等的元素也无法删除），所以在上面程序的③代码处，删除count为-2的R对象时，没有任何元素被删除；程序执行④代码时，可以看到删除了count为5的R对象，这表明TreeSet可以删除没有被修改实例变量、且不与其他被修改实例变量的对象重复的对象。

注意：

当执行了④代码后，TreeSet会对集合中的元素重新索引（不是重新排序），接下来就可以删除TreeSet中的所有元素了，包括那些被修改过实例变量的元素。与HashSet类似的是，如果TreeSet中包含了可变对象，当可变对象的实例变量被修改时，TreeSet在处理这些对象时将非常复杂，而且容易出错。为了让程序更加健壮，推荐不要修改放入HashSet和TreeSet集合中元素的关键实例变量。

2. 定制排序

TreeSet的自然排序是根据集合元素的大小，TreeSet将它们以升序排列。如果需要实现定制排序，例如以降序排列，则可以通过Comparator接口的帮助。该接口里包含一个int compare(T o1, T o2)方法，该方法用于比较o1和o2的大小：如果该方法返回正整数，则表明o1大于o2；如果该方法返回0，则表明o1等于o2；如果该方法返回负整数，则表明o1小于o2。

如果需要实现定制排序，则需要在创建TreeSet集合对象时，提供一个Comparator对象与该TreeSet集合关联，由该Comparator对象负责集合元素的排序逻辑。由于Comparator是一个函数式接口，因此可使用Lambda表达式来代替Comparator对象。

程序清单：codes\08\8.3\TreeSetTest4.java

```
class M
{
    int age;
    public M(int age)
    {
        this.age = age;
    }
    public String toString()
    {
        return "M [age:" + age + "]";
    }
}
public class TreeSetTest4
{
    public static void main(String[] args)
    {
        // 此处Lambda表达式的目标类型是Comparator
        TreeSet ts = new TreeSet((o1 , o2) ->
        {
            M m1 = (M)o1;
            M m2 = (M)o2;
            // 根据M对象的age属性来决定大小，age越大，M对象反而越小
            return m1.age > m2.age ? -1
                : m1.age < m2.age ? 1 : 0;
        });
        ts.add(new M(5));
        ts.add(new M(-3));
        ts.add(new M(9));
        System.out.println(ts);
    }
}
```

上面程序中粗体字部分使用了目标类型为Comparator的Lambda表达式，它负责ts集合的排序。所以当把M对象添加到ts集合中时，无须M类实现Comparable接口，因为此时TreeSet无须通过M对象本身来比较大小，而是由与TreeSet关联的Lambda表达式来负责集合元素的排序。运行程序，看到如下运行结果：

```
[M [age:9], M [age:5], M [age:-3]]
```

注意：

当通过 Comparator 对象（或 Lambda 表达式）来实现 TreeSet 的定制排序时，依然不可以向 TreeSet 中添加类型不同的对象，否则会引发 ClassCastException 异常。使用定制排序时，TreeSet 对集合元素排序不管集合元素本身的大小，而是由 Comparator 对象（或 Lambda 表达式）负责集合元素的排序规则。TreeSet 判断两个集合元素相等的标准是：通过 Comparator（或 Lambda 表达式）比较两个元素返回了 0，这样 TreeSet 不会把第二个元素添加到集合中。

8.3.4 EnumSet 类

EnumSet 是一个专为枚举类设计的集合类，EnumSet 中的所有元素都必须是指定枚举类型的枚举值，该枚举类型在创建 EnumSet 时显式或隐式地指定。EnumSet 的集合元素也是有序的，EnumSet 以枚举值在 Enum 类内的定义顺序来决定集合元素的顺序。

EnumSet 在内部以位向量的形式存储，这种存储形式非常紧凑、高效，因此 EnumSet 对象占用内存很小，而且运行效率很好。尤其是进行批量操作（如调用 containsAll() 和 retainAll()方法）时，如果其参数也是 EnumSet 集合，则该批量操作的执行速度也非常快。

EnumSet 集合不允许加入 null 元素，如果试图插入 null 元素，EnumSet 将抛出 NullPointerException 异常。如果只是想判断 EnumSet 是否包含 null 元素或试图删除 null 元素都不会抛出异常，只是删除操作将返回 false，因为没有任何 null 元素被删除。

EnumSet 类没有暴露任何构造器来创建该类的实例，程序应该通过它提供的类方法来创建 EnumSet 对象。EnumSet 类它提供了如下常用的类方法来创建 EnumSet 对象。

- EnumSet allOf(Class elementType)：创建一个包含指定枚举类里所有枚举值的 EnumSet 集合。
- EnumSet complementOf(EnumSet s)：创建一个其元素类型与指定 EnumSet 里元素类型相同的 EnumSet 集合，新 EnumSet 集合包含原 EnumSet 集合所不包含的、此枚举类剩下的枚举值（即新 EnumSet 集合和原 EnumSet 集合的集合元素加起来就是该枚举类的所有枚举值）。
- EnumSet copyOf(Collection c)：使用一个普通集合来创建 EnumSet 集合。
- EnumSet copyOf(EnumSet s)：创建一个与指定 EnumSet 具有相同元素类型、相同集合元素的 EnumSet 集合。
- EnumSet noneOf(Class elementType)：创建一个元素类型为指定枚举类型的空 EnumSet。
- EnumSet of(E first, E... rest)：创建一个包含一个或多个枚举值的 EnumSet 集合，传入的多个枚举值必须属于同一个枚举类。
- EnumSet range(E from, E to)：创建一个包含从 from 枚举值到 to 枚举值范围内所有枚举值的 EnumSet 集合。

下面程序示范了如何使用 EnumSet 来保存枚举类的多个枚举值。

程序清单：codes\08\8.3\EnumSetTest.java

```
enum Season
{
    SPRING,SUMMER,FALL,WINTER
}
public class EnumSetTest
{
    public static void main(String[] args)
    {
        // 创建一个 EnumSet 集合，集合元素就是 Season 枚举类的全部枚举值
        EnumSet es1 = EnumSet.allOf(Season.class);
        System.out.println(es1); // 输出[SPRING,SUMMER,FALL,WINTER]
        // 创建一个 EnumSet 空集合，指定其集合元素是 Season 类的枚举值
        EnumSet es2 = EnumSet.noneOf(Season.class);
        System.out.println(es2); // 输出[]
        // 手动添加两个元素
        es2.add(Season.WINTER);
```

```
        es2.add(Season.SPRING);
        System.out.println(es2); // 输出[SPRING,WINTER]
        // 以指定枚举值创建 EnumSet 集合
        EnumSet es3 = EnumSet.of(Season.SUMMER , Season.WINTER);
        System.out.println(es3); // 输出[SUMMER,WINTER]
        EnumSet es4 = EnumSet.range(Season.SUMMER , Season.WINTER);
        System.out.println(es4); // 输出[SUMMER,FALL,WINTER]
        // 新创建的 EnumSet 集合元素和 es4 集合元素有相同的类型
        // es5 集合元素 + es4 集合元素 = Season 枚举类的全部枚举值
        EnumSet es5 = EnumSet.complementOf(es4);
        System.out.println(es5); // 输出[SPRING]
    }
}
```

上面程序中粗体字标识的代码示范了 EnumSet 集合的常规用法。除此之外，还可以复制另一个 EnumSet 集合中的所有元素来创建新的 EnumSet 集合，或者复制另一个 Collection 集合中的所有元素来创建新的 EnumSet 集合。当复制 Collection 集合中的所有元素来创建新的 EnumSet 集合时，要求 Collection 集合中的所有元素必须是同一个枚举类的枚举值。下面程序示范了这个用法。

程序清单：codes\08\8.3\EnumSetTest2.java

```
public class EnumSetTest2
{
    public static void main(String[] args)
    {
        Collection c = new HashSet();
        c.clear();
        c.add(Season.FALL);
        c.add(Season.SPRING);
        // 复制 Collection 集合中的所有元素来创建 EnumSet 集合
        EnumSet enumSet = EnumSet.copyOf(c);  // ①
        System.out.println(enumSet); // 输出[SPRING,FALL]
        c.add("疯狂 Java 讲义");
        c.add("轻量级 Java EE 企业应用实战");
        // 下面代码出现异常：因为 c 集合里的元素不是全部都为枚举值
        enumSet = EnumSet.copyOf(c); // ②
    }
}
```

上面程序中两处粗体字标识的代码没有任何区别，只是因为执行②行代码时，c 集合中的元素不全是枚举值，而是包含了两个字符串对象，所以在②行代码处抛出 ClassCastException 异常。

注意：

当试图复制一个 Collection 集合里的元素来创建 EnumSet 集合时，必须保证 Collection 集合里的所有元素都是同一个枚举类的枚举值。

8.3.5 各 Set 实现类的性能分析

HashSet 和 TreeSet 是 Set 的两个典型实现，到底如何选择 HashSet 和 TreeSet 呢？HashSet 的性能总是比 TreeSet 好（特别是最常用的添加、查询元素等操作），因为 TreeSet 需要额外的红黑树算法来维护集合元素的次序。只有当需要一个保持排序的 Set 时，才应该使用 TreeSet，否则都应该使用 HashSet。

HashSet 还有一个子类：LinkedHashSet，对于普通的插入、删除操作，LinkedHashSet 比 HashSet 要略微慢一点，这是由维护链表所带来的额外开销造成的，但由于有了链表，遍历 LinkedHashSet 会更快。

EnumSet 是所有 Set 实现类中性能最好的，但它只能保存同一个枚举类的枚举值作为集合元素。

必须指出的是，Set 的三个实现类 HashSet、TreeSet 和 EnumSet 都是线程不安全的。如果有多个线程同时访问一个 Set 集合，并且有超过一个线程修改了该 Set 集合，则必须手动保证该 Set 集合的同步性。通常可以通过 Collections 工具类的 synchronizedSortedSet 方法来“包装”该 Set 集合。此操作最好在创建时进行，以防止对 Set 集合的意外非同步访问。例如：

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

关于 Collections 工具类的更进一步用法，可以参考 8.8 节的内容。

8.4 List 集合

List 集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List 集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。List 集合默认按元素的添加顺序设置元素的索引，例如第一次添加的元素索引为 0，第二次添加的元素索引为 1……

8.4.1 Java 8 改进的 List 接口和 ListIterator 接口

List 作为 Collection 接口的子接口，当然可以使用 Collection 接口里的全部方法。而且由于 List 是有序集合，因此 List 集合里增加了一些根据索引来操作集合元素的方法。

- void add(int index, Object element)：将元素 element 插入到 List 集合的 index 处。
- boolean addAll(int index, Collection c)：将集合 c 所包含的所有元素都插入到 List 集合的 index 处。
- Object get(int index)：返回集合 index 索引处的元素。
- int indexOf(Object o)：返回对象 o 在 List 集合中第一次出现的位置索引。
- int lastIndexOf(Object o)：返回对象 o 在 List 集合中最后一次出现的位置索引。
- Object remove(int index)：删除并返回 index 索引处的元素。
- Object set(int index, Object element)：将 index 索引处的元素替换成 element 对象，返回被替换的旧元素。
- List subList(int fromIndex, int toIndex)：返回从索引 fromIndex（包含）到索引 toIndex（不包含）处所有集合元素组成的子集合。

所有的 List 实现类都可以调用这些方法来操作集合元素。与 Set 集合相比，List 增加了根据索引来插入、替换和删除集合元素的方法。除此之外，Java 8 还为 List 接口添加了如下两个默认方法。

- void replaceAll(UnaryOperator operator)：根据 operator 指定的计算规则重新设置 List 集合的所有元素。
- void sort(Comparator c)：根据 Comparator 参数对 List 集合的元素排序。

下面程序示范了 List 集合的常规用法。

程序清单：codes\08\8.4\ListTest.java

```
public class ListTest
{
    public static void main(String[] args)
    {
        List books = new ArrayList();
        // 向 books 集合中添加三个元素
        books.add(new String("轻量级 Java EE 企业应用实战"));
        books.add(new String("疯狂 Java 讲义"));
        books.add(new String("疯狂 Android 讲义"));
        System.out.println(books);
        // 将新字符串对象插入在第二个位置
        books.add(1 , new String("疯狂 Ajax 讲义"));
        for (int i = 0 ; i < books.size() ; i++ )
        {
            System.out.println(books.get(i));
        }
        // 删除第三个元素
        books.remove(2);
        System.out.println(books);
        // 判断指定元素在 List 集合中的位置：输出 1，表明位于第二位
        System.out.println(books.indexOf(new String("疯狂 Ajax 讲义"))); //①
        //将第二个元素替换成新的字符串对象
        books.set(1, new String("疯狂 Java 讲义"));
```

```
        System.out.println(books);
        //将 books 集合的第二个元素（包括）
        //到第三个元素（不包括）截取成子集合
        System.out.println(books.subList(1 , 2));
    }
}
```

上面程序中粗体字代码示范了 List 集合的独特用法，List 集合可以根据位置索引来访问集合中的元素，因此 List 增加了一种新的遍历集合元素的方法：使用普通的 for 循环来遍历集合元素。运行上面程序，将看到如下运行结果：

```
[轻量级 Java EE 企业应用实战, 疯狂 Java 讲义, 疯狂 Android 讲义]
轻量级 Java EE 企业应用实战
疯狂 Ajax 讲义
疯狂 Java 讲义
疯狂 Android 讲义
[轻量级 Java EE 企业应用实战, 疯狂 Ajax 讲义, 疯狂 Android 讲义]
1
[轻量级 Java EE 企业应用实战, 疯狂 Java 讲义, 疯狂 Android 讲义]
[疯狂 Java 讲义]
```

从上面运行结果清楚地看出 List 集合的用法。注意①行代码处，程序试图返回新字符串对象在 List 集合中的位置，实际上 List 集合中并未包含该字符串对象。因为 List 集合添加字符串对象时，添加的是通过 new 关键字创建的新字符串对象，①行代码处也是通过 new 关键字创建的新字符串对象，两个字符串显然不是同一个对象，但 List 的 indexOf 方法依然可以返回 1。List 判断两个对象相等的标准是什么呢？List 判断两个对象相等只要通过 equals()方法比较返回 true 即可。看下面程序。

程序清单：codes\08\8.4\ListTest2.java

```
class A
{
    public boolean equals(Object obj)
    {
        return true;
    }
}
public class ListTest2
{
    public static void main(String[] args)
    {
        List books = new ArrayList();
        books.add(new String("轻量级 Java EE 企业应用实战"));
        books.add(new String("疯狂 Java 讲义"));
        books.add(new String("疯狂 Android 讲义"));
        System.out.println(books);
        // 删除集合中的 A 对象，将导致第一个元素被删除
        books.remove(new A());      // ①
        System.out.println(books);
        // 删除集合中的 A 对象，再次删除集合中的第一个元素
        books.remove(new A());      // ②
        System.out.println(books);
    }
}
```

编译、运行上面程序，看到如下运行结果：

```
[轻量级 Java EE 企业应用实战, 疯狂 Java 讲义, 疯狂 Android 讲义]
[疯狂 Java 讲义, 疯狂 Android 讲义]
[疯狂 Android 讲义]
```

从上面运行结果可以看出，执行①行代码时，程序试图删除一个 A 对象，List 将会调用该 A 对象的 equals()方法依次与集合元素进行比较，如果该 equals()方法以某个集合元素作为参数时返回 true，List 将会删除该元素——A 类重写了 equals()方法，该方法总是返回 true。所以每次从 List 集合中删除 A 对象时，总是删除 List 集合中的第一个元素。

注意：

当调用List的set(int index, Object element)方法来改变List集合指定索引处的元素时，指定的索引必须是List集合的有效索引。例如集合长度是4，就不能指定替换索引为4处的元素——也就是说，set(int index, Object element)方法不会改变List集合的长度。

Java 8为List集合增加了sort()和replaceAll()两个常用的默认方法，其中sort()方法需要一个Comparator对象来控制元素排序，程序可使用Lambda表达式来作为参数；而replaceAll()方法则需要一个UnaryOperator来替换所有集合元素，UnaryOperator也是一个函数式接口，因此程序也可使用Lambda表达式作为参数。如下程序示范了List集合的两个默认方法的功能。

程序清单：codes\08\8.4\ListTest3.java

```
public class ListTest3
{
    public static void main(String[] args)
    {
        List books = new ArrayList();
        // 向books集合中添加4个元素
        books.add(new String("轻量级Java EE企业应用实战"));
        books.add(new String("疯狂Java讲义"));
        books.add(new String("疯狂Android讲义"));
        books.add(new String("疯狂iOS讲义"));
        // 使用目标类型为Comparator的Lambda表达式对List集合排序
        books.sort((o1, o2)->((String)o1).length() - ((String)o2).length());
        System.out.println(books);
        // 使用目标类型为UnaryOperator的Lambda表达式来替换集合中所有元素
        // 该Lambda表达式控制使用每个字符串的长度作为新的集合元素
        books.replaceAll(ele->((String)ele).length());
        System.out.println(books); // 输出[7, 8, 11, 16]
    }
}
```

上面程序中第一行粗体字代码控制对List集合进行排序，传给sort()方法的Lambda表达式指定的排序规则是：字符串长度越长，字符串越大，因此执行完第一行粗体字代码之后，List集合中的字符串会按由短到长的顺序排列。

程序中第二行粗体字代码传给replaceAll()方法的Lambda表达式指定了替换集合元素的规则：直接用集合元素（字符串）的长度作为新的集合元素。执行该方法后，集合元素被替换为[7, 8, 11, 16]。

与Set只提供了一个iterator()方法不同，List还额外提供了一个listIterator()方法，该方法返回一个ListIterator对象，ListIterator接口继承了Iterator接口，提供了专门操作List的方法。ListIterator接口在Iterator接口基础上增加了如下方法。

- boolean hasPrevious()：返回该迭代器关联的集合是否还有上一个元素。
- Object previous()：返回该迭代器的上一个元素。
- void add(Object o)：在指定位置插入一个元素。

拿ListIterator与普通的Iterator进行对比，不难发现ListIterator增加了向前迭代的功能（Iterator只能向后迭代），而且ListIterator还可通过add()方法向List集合中添加元素（Iterator只能删除元素）。下面程序示范了ListIterator的用法。

程序清单：codes\08\8.4\ListIteratorTest.java

```
public class ListIteratorTest
{
    public static void main(String[] args)
    {
        String[] books = {
            "疯狂Java讲义", "疯狂iOS讲义",
            "轻量级Java EE企业应用实战"
        };
        List bookList = new ArrayList();
```

```
        for (int i = 0; i < books.length ; i++ )
        {
            bookList.add(books[i]);
        }
        ListIterator lit = bookList.listIterator();
        while (lit.hasNext())
        {
            System.out.println(lit.next());
            lit.add("-------分隔符-------");
        }
        System.out.println("=======下面开始反向迭代=======");
        while(lit.hasPrevious())
        {
            System.out.println(lit.previous());
        }
    }
}
```

从上面程序中可以看出，使用 ListIterator 迭代 List 集合时，开始也需要采用正向迭代，即先使用 next()方法进行迭代，在迭代过程中可以使用 add()方法向上一次迭代元素的后面添加一个新元素。运行上面程序，看到如下结果：

```
疯狂 Java 讲义
疯狂 iOS 讲义
轻量级 Java EE 企业应用实战
=======下面开始反向迭代=======
-------分隔符-------
轻量级 Java EE 企业应用实战
-------分隔符-------
疯狂 iOS 讲义
-------分隔符-------
疯狂 Java 讲义
```

8.4.2 ArrayList 和 Vector 实现类

ArrayList 和 Vector 作为 List 类的两个典型实现，完全支持前面介绍的 List 接口的全部功能。

ArrayList 和 Vector 类都是基于数组实现的 List 类，所以 ArrayList 和 Vector 类封装了一个动态的、允许再分配的 Object[]数组。ArrayList 或 Vector 对象使用 initialCapacity 参数来设置该数组的长度，当向 ArrayList 或 Vector 中添加元素超出了该数组的长度时，它们的 initialCapacity 会自动增加。

对于通常的编程场景，程序员无须关心 ArrayList 或 Vector 的 initialCapacity。但如果向 ArrayList 或 Vector 集合中添加大量元素时，可使用 ensureCapacity(int minCapacity)方法一次性地增加 initialCapacity。这可以减少重分配的次数，从而提高性能。

如果开始就知道 ArrayList 或 Vector 集合需要保存多少个元素，则可以在创建它们时就指定 initialCapacity 大小。如果创建空的 ArrayList 或 Vector 集合时不指定 initialCapacity 参数，则 Object[]数组的长度默认为 10。

除此之外，ArrayList 和 Vector 还提供了如下两个方法来重新分配 Object[]数组。

- void ensureCapacity(int minCapacity)：将 ArrayList 或 Vector 集合的 Object[]数组长度增加大于或等于 minCapacity 值。
- void trimToSize()：调整 ArrayList 或 Vector 集合的 Object[]数组长度为当前元素的个数。调用该方法可减少 ArrayList 或 Vector 集合对象占用的存储空间。

ArrayList 和 Vector 在用法上几乎完全相同，但由于 Vector 是一个古老的集合（从 JDK 1.0 就有了），那时候 Java 还没有提供系统的集合框架，所以 Vector 里提供了一些方法名很长的方法，例如 addElement(Object obj)，实际上这个方法与 add (Object obj)没有任何区别。从 JDK 1.2 以后，Java 提供了系统的集合框架，就将 Vector 改为实现 List 接口，作为 List 的实现之一，从而导致 Vector 里有一些功能重复的方法。

Vector 的系列方法中方法名更短的方法属于后来新增的方法，方法名更长的方法则是 Vector 原有的方法。Java 改写了 Vector 原有的方法，将其方法名缩短是为了简化编程。而 ArrayList 开始就作为 List 的主要实现类，因此没有那些方法名很长的方法。实际上，Vector 具有很多缺点，通常尽量少用 Vector

实现类。

除此之外，ArrayList 和 Vector 的显著区别是：ArrayList 是线程不安全的，当多个线程访问同一个 ArrayList 集合时，如果有超过一个线程修改了 ArrayList 集合，则程序必须手动保证该集合的同步性；但 Vector 集合则是线程安全的，无须程序保证该集合的同步性。因为 Vector 是线程安全的，所以 Vector 的性能比 ArrayList 的性能要低。实际上，即使需要保证 List 集合线程安全，也同样不推荐使用 Vector 实现类。后面会介绍一个 Collections 工具类，它可以将一个 ArrayList 变成线程安全的。

Vector 还提供了一个 Stack 子类，它用于模拟“栈”这种数据结构，“栈”通常是指“后进先出”（LIFO）的容器。最后“push”进栈的元素，将最先被“pop”出栈。与 Java 中的其他集合一样，进栈出栈的都是 Object，因此从栈中取出元素后必须进行类型转换，除非你只是使用 Object 具有的操作。所以 Stack 类里提供了如下几个方法。

- Object peek()：返回“栈”的第一个元素，但并不将该元素“pop”出栈。
- Object pop()：返回“栈”的第一个元素，并将该元素“pop”出栈。
- void push(Object item)：将一个元素“push”进栈，最后一个进“栈”的元素总是位于“栈”顶。

需要指出的是，由于 Stack 继承了 Vector，因此它也是一个非常古老的 Java 集合类，它同样是线程安全的、性能较差的，因此应该尽量少用 Stack 类。如果程序需要使用“栈”这种数据结构，则可以考虑使用后面将要介绍的 ArrayDeque。

提示：

ArrayDeque 也是 List 的实现类，ArrayDeque 既实现了 List 接口，也实现了 Deque 接口，由于实现了 Deque 接口，因此可以作为栈来使用；而且 ArrayDeque 底层也是基于数组的实现，因此性能也很好。本书将在 8.5 节详细介绍 ArrayDeque。

8.4.3 固定长度的 List

前面讲数组时介绍了一个操作数组的工具类：Arrays，该工具类里提供了 asList(Object... a)方法，该方法可以把一个数组或指定个数的对象转换成一个 List 集合，这个 List 集合既不是 ArrayList 实现类的实例，也不是 Vector 实现类的实例，而是 Arrays 的内部类 ArrayList 的实例。

Arrays.ArrayList 是一个固定长度的 List 集合，程序只能遍历访问该集合里的元素，不可增加、删除该集合里的元素。如下程序所示。

程序清单：codes\08\8.4\FixedSizeList.java

```
public class FixedSizeList
{
    public static void main(String[] args)
    {
        List fixedList = Arrays.asList("疯狂 Java 讲义"
            , "轻量级 Java EE 企业应用实战");
        // 获取 fixedList 的实现类，将输出 Arrays$ArrayList
        System.out.println(fixedList.getClass());
        // 使用方法引用遍历集合元素
        fixedList.forEach(System.out::println);
        // 试图增加、删除元素都会引发 UnsupportedOperationException 异常
        fixedList.add("疯狂 Android 讲义");
        fixedList.remove("疯狂 Java 讲义");
    }
}
```

上面程序中粗体字标识的两行代码对于普通的 List 集合完全正常，但如果试图通过这两个方法来增加、删除 Arrays$ArrayList 集合里的元素，将会引发异常。所以上面程序在编译时完全正常，但会在运行第一行粗体字标识的代码行处引发 UnsupportedOperationException 异常。

8.5 Queue 集合

Queue 用于模拟队列这种数据结构，队列通常是指“先进先出”（FIFO）的容器。队列的头部保存

在队列中存放时间最长的元素，队列的尾部保存在队列中存放时间最短的元素。新元素插入（offer）到队列的尾部，访问元素（poll）操作会返回队列头部的元素。通常，队列不允许随机访问队列中的元素。

Queue 接口中定义了如下几个方法。

- void add(Object e)：将指定元素加入此队列的尾部。
- Object element()：获取队列头部的元素，但是不删除该元素。
- boolean offer(Object e)：将指定元素加入此队列的尾部。当使用有容量限制的队列时，此方法通常比 add(Object e)方法更好。
- Object peek()：获取队列头部的元素，但是不删除该元素。如果此队列为空，则返回 null。
- Object poll()：获取队列头部的元素，并删除该元素。如果此队列为空，则返回 null。
- Object remove()：获取队列头部的元素，并删除该元素。

Queue 接口有一个 PriorityQueue 实现类。除此之外，Queue 还有一个 Deque 接口，Deque 代表一个“双端队列”，双端队列可以同时从两端来添加、删除元素，因此 Deque 的实现类既可当成队列使用，也可当成栈使用。Java 为 Deque 提供了 ArrayDeque 和 LinkedList 两个实现类。

8.5.1 PriorityQueue 实现类

PriorityQueue 是一个比较标准的队列实现类。之所以说它是比较标准的队列实现，而不是绝对标准的队列实现，是因为 PriorityQueue 保存队列元素的顺序并不是按加入队列的顺序，而是按队列元素的大小进行重新排序。因此当调用 peek()方法或者 poll()方法取出队列中的元素时，并不是取出最先进入队列的元素，而是取出队列中最小的元素。从这个意义上来看，PriorityQueue 已经违反了队列的最基本规则：先进先出（FIFO）。下面程序示范了 PriorityQueue 队列的用法。

程序清单：codes\08\8.5\PriorityQueueTest.java

```
public class PriorityQueueTest
{
    public static void main(String[] args)
    {
        PriorityQueue pq = new PriorityQueue();
        // 下面代码依次向 pq 中加入四个元素
        pq.offer(6);
        pq.offer(-3);
        pq.offer(20);
        pq.offer(18);
        // 输出 pq 队列，并不是按元素的加入顺序排列
        System.out.println(pq); // 输出[-3, 6, 20, 18]
        // 访问队列的第一个元素，其实就是队列中最小的元素：-3
        System.out.println(pq.poll());
    }
}
```

运行上面程序直接输出 PriorityQueue 集合时，可能看到该队列里的元素并没有很好地按大小进行排序，但这只是受到 PriorityQueue 的 toString()方法的返回值的影响。实际上，程序多次调用 PriorityQueue 集合对象的 poll()方法，即可看到元素按从小到大的顺序“移出队列”。

PriorityQueue 不允许插入 null 元素，它还需要对队列元素进行排序，PriorityQueue 的元素有两种排序方式。

- 自然排序：采用自然顺序的 PriorityQueue 集合中的元素必须实现了 Comparable 接口，而且应该是同一个类的多个实例，否则可能导致 ClassCastException 异常。
- 定制排序：创建 PriorityQueue 队列时，传入一个 Comparator 对象，该对象负责对队列中的所有元素进行排序。采用定制排序时不要求队列元素实现 Comparable 接口。

PriorityQueue 队列对元素的要求与 TreeSet 对元素的要求基本一致，因此关于使用自然排序和定制排序的详细介绍请参考 8.3.3 节。

8.5.2 Deque 接口与 ArrayDeque 实现类

Deque 接口是 Queue 接口的子接口，它代表一个双端队列，Deque 接口里定义了一些双端队列的方

法，这些方法允许从两端来操作队列的元素。

- void addFirst(Object e)：将指定元素插入该双端队列的开头。
- void addLast(Object e)：将指定元素插入该双端队列的末尾。
- Iterator descendingIterator()：返回该双端队列对应的迭代器，该迭代器将以逆向顺序来迭代队列中的元素。
- Object getFirst()：获取但不删除双端队列的第一个元素。
- Object getLast()：获取但不删除双端队列的最后一个元素。
- boolean offerFirst(Object e)：将指定元素插入该双端队列的开头。
- boolean offerLast(Object e)：将指定元素插入该双端队列的末尾。
- Object peekFirst()：获取但不删除该双端队列的第一个元素；如果此双端队列为空，则返回 null。
- Object peekLast()：获取但不删除该双端队列的最后一个元素；如果此双端队列为空，则返回 null。
- Object pollFirst()：获取并删除该双端队列的第一个元素；如果此双端队列为空，则返回 null。
- Object pollLast()：获取并删除该双端队列的最后一个元素；如果此双端队列为空，则返回 null。
- Object pop()（栈方法）：pop 出该双端队列所表示的栈的栈顶元素。相当于 removeFirst()。
- void push(Object e)（栈方法）：将一个元素 push 进该双端队列所表示的栈的栈顶。相当于 addFirst(e)。
- Object removeFirst()：获取并删除该双端队列的第一个元素。
- Object removeFirstOccurrence(Object o)：删除该双端队列的第一次出现的元素 o。
- Object removeLast()：获取并删除该双端队列的最后一个元素。
- boolean removeLastOccurrence(Object o)：删除该双端队列的最后一次出现的元素 o。

从上面方法中可以看出，Deque 不仅可以当成双端队列使用，而且可以被当成栈来使用，因为该类里还包含了 pop（出栈）、push（入栈）两个方法。

Deque 的方法与 Queue 的方法对照表如表 8.2 所示。

表 8.2 Deque 的方法与 Queue 的方法对照表

Queue 的方法	Deque 的方法
add(e)/offer(e)	addLast(e)/offerLast(e)
remove()/poll()	removeFirst()/pollFirst()
element()/peek()	getFirst()/peekFirst()

Deque 的方法与 Stack 的方法对照表如表 8.3 所示。

表 8.3 Deque 的方法与 Stack 的方法对照表

Stack 的方法	Deque 的方法
push(e)	addFirst(e)/offerFirst(e)
pop()	removeFirst()/pollFirst()
peek()	getFirst()/peekFirst()

Deque 接口提供了一个典型的实现类：ArrayDeque，从该名称就可以看出，它是一个基于数组实现的双端队列，创建 Deque 时同样可指定一个 numElements 参数，该参数用于指定 Object[]数组的长度；如果不指定 numElements 参数，Deque 底层数组的长度为 16。

提示：

ArrayList 和 ArrayDeque 两个集合类的实现机制基本相似，它们的底层都采用一个动态的、可重分配的 Object[]数组来存储集合元素，当集合元素超出了该数组的容量时，系统会在底层重新分配一个 Object[]数组来存储集合元素。

下面程序示范了把 ArrayDeque 当成“栈”来使用。

程序清单：codes\08\8.5\ArrayDequeStack.java

```
public class ArrayDequeStack
```

```
{
    public static void main(String[] args)
    {
        ArrayDeque stack = new ArrayDeque();
        // 依次将三个元素 push 入“栈”
        stack.push("疯狂 Java 讲义");
        stack.push("轻量级 Java EE 企业应用实战");
        stack.push("疯狂 Android 讲义");
        // 输出：[疯狂 Android 讲义，轻量级 Java EE 企业应用实战，疯狂 Java 讲义]
        System.out.println(stack);
        // 访问第一个元素，但并不将其 pop 出“栈”，输出：疯狂 Android 讲义
        System.out.println(stack.peek());
        // 依然输出：[疯狂 Android 讲义，疯狂 Java 讲义，轻量级 Java EE 企业应用实战]
        System.out.println(stack);
        // pop 出第一个元素，输出：疯狂 Android 讲义
        System.out.println(stack.pop());
        // 输出：[轻量级 Java EE 企业应用实战，疯狂 Java 讲义]
        System.out.println(stack);
    }
}
```

上面程序的运行结果显示了 ArrayDeque 作为栈的行为，因此当程序中需要使用“栈”这种数据结构时，推荐使用 ArrayDeque，尽量避免使用 Stack——因为 Stack 是古老的集合，性能较差。

当然 ArrayDeque 也可以当成队列使用，此处 ArrayDeque 将按“先进先出”的方式操作集合元素。例如如下程序。

程序清单：codes\08\8.5\ArrayDequeQueue.java

```
public class ArrayDequeQueue
{
    public static void main(String[] args)
    {
        ArrayDeque queue = new ArrayDeque();
        // 依次将三个元素加入队列
        queue.offer("疯狂 Java 讲义");
        queue.offer("轻量级 Java EE 企业应用实战");
        queue.offer("疯狂 Android 讲义");
        // 输出：[疯狂 Java 讲义，轻量级 Java EE 企业应用实战，疯狂 Android 讲义]
        System.out.println(queue);
        // 访问队列头部的元素，但并不将其 poll 出队列“栈”，输出：疯狂 Java 讲义
        System.out.println(queue.peek());
        // 依然输出：[疯狂 Java 讲义，轻量级 Java EE 企业应用实战，疯狂 Android 讲义]
        System.out.println(queue);
        // poll 出第一个元素，输出：疯狂 Java 讲义
        System.out.println(queue.poll());
        // 输出：[轻量级 Java EE 企业应用实战，疯狂 Android 讲义]
        System.out.println(queue);
    }
}
```

上面程序的运行结果显示了 ArrayDeque 作为队列的行为。

通过上面两个程序可以看出，ArrayDeque 不仅可以作为栈使用，也可以作为队列使用。

8.5.3 LinkedList 实现类

LinkedList 类是 List 接口的实现类——这意味着它是一个 List 集合，可以根据索引来随机访问集合中的元素。除此之外，LinkedList 还实现了 Deque 接口，可以被当成双端队列来使用，因此既可以被当成“栈”来使用，也可以当成队列使用。下面程序简单示范了 LinkedList 集合的用法。

程序清单：codes\08\8.5\LinkedListTest.java

```
public class LinkedListTest
{
    public static void main(String[] args)
    {
        LinkedList books = new LinkedList();
```

```
        // 将字符串元素加入队列的尾部
        books.offer("疯狂 Java 讲义");
        // 将一个字符串元素加入栈的顶部
        books.push("轻量级 Java EE 企业应用实战");
        // 将字符串元素添加到队列的头部（相当于栈的顶部）
        books.offerFirst("疯狂 Android 讲义");
        // 以 List 的方式（按索引访问的方式）来遍历集合元素
        for (int i = 0; i < books.size() ; i++ )
        {
            System.out.println("遍历中：" + books.get(i));
        }
        // 访问并不删除栈顶的元素
        System.out.println(books.peekFirst());
        // 访问并不删除队列的最后一个元素
        System.out.println(books.peekLast());
        // 将栈顶的元素弹出“栈”
        System.out.println(books.pop());
        // 下面输出将看到队列中第一个元素被删除
        System.out.println(books);
        // 访问并删除队列的最后一个元素
        System.out.println(books.pollLast());
        // 下面输出：[轻量级 Java EE 企业应用实战]
        System.out.println(books);
    }
}
```

上面程序中粗体字代码分别示范了 LinkedList 作为 List 集合、双端队列、栈的用法。由此可见，LinkedList 是一个功能非常强大的集合类。

LinkedList 与 ArrayList、ArrayDeque 的实现机制完全不同，ArrayList、ArrayDeque 内部以数组的形式来保存集合中的元素，因此随机访问集合元素时有较好的性能；而 LinkedList 内部以链表的形式来保存集合中的元素，因此随机访问集合元素时性能较差，但在插入、删除元素时性能比较出色（只需改变指针所指的地址即可）。需要指出的是，虽然 Vector 也是以数组的形式来存储集合元素的，但因为它实现了线程同步功能（而且实现机制也不好），所以各方面性能都比较差。

注意：

对于所有的内部基于数组的集合实现，例如 ArrayList、ArrayDeque 等，使用随机访问的性能比使用 Iterator 迭代访问的性能要好，因为随机访问会被映射成对数组元素的访问。

8.5.4 各种线性表的性能分析

Java 提供的 List 就是一个线性表接口，而 ArrayList、LinkedList 又是线性表的两种典型实现：基于数组的线性表和基于链的线性表。Queue 代表了队列，Deque 代表了双端队列（既可作为队列使用，也可作为栈使用），接下来对各种实现类的性能进行分析。

初学者可以无须理会 ArrayList 和 LinkedList 之间的性能差异，只需要知道 LinkedList 集合不仅提供了 List 的功能，还提供了双端队列、栈的功能就行。但对于一个成熟的 Java 程序员，在一些性能非常敏感的地方，可能需要慎重选择哪个 List 实现。

一般来说，由于数组以一块连续内存区来保存所有的数组元素，所以数组在随机访问时性能最好，所有的内部以数组作为底层实现的集合在随机访问时性能都比较好；而内部以链表作为底层实现的集合在执行插入、删除操作时有较好的性能。但总体来说，ArrayList 的性能比 LinkedList 的性能要好，因此大部分时候都应该考虑使用 ArrayList。

关于使用 List 集合有如下建议。

- 如果需要遍历 List 集合元素，对于 ArrayList、Vector 集合，应该使用随机访问方法（get）来遍历集合元素，这样性能更好；对于 LinkedList 集合，则应该采用迭代器（Iterator）来遍历集合元素。

- 如果需要经常执行插入、删除操作来改变包含大量数据的 List 集合的大小，可考虑使用 LinkedList 集合。使用 ArrayList、Vector 集合可能需要经常重新分配内部数组的大小，效果可能较差。
- 如果有多个线程需要同时访问 List 集合中的元素，开发者可考虑使用 Collections 将集合包装成线程安全的集合。

8.6 Java 8 增强的 Map 集合

Map 用于保存具有映射关系的数据，因此 Map 集合里保存着两组值，一组值用于保存 Map 里的 key，另外一组值用于保存 Map 里的 value，key 和 value 都可以是任何引用类型的数据。Map 的 key 不允许重复，即同一个 Map 对象的任何两个 key 通过 equals 方法比较总是返回 false。

key 和 value 之间存在单向一对一关系，即通过指定的 key，总能找到唯一的、确定的 value。从 Map 中取出数据时，只要给出指定的 key，就可以取出对应的 value。如果把 Map 的两组值拆开来看，Map 里的数据有如图 8.7 所示的结构。

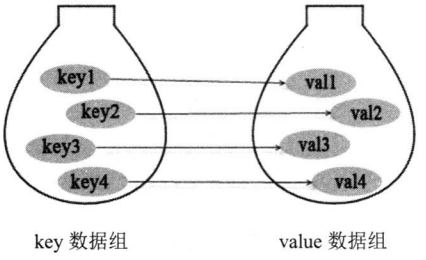


图 8.7　分开看 Map 的 key 组和 value 组

从图 8.7 中可以看出，如果把 Map 里的所有 key 放在一起来看，它们就组成了一个 Set 集合（所有的 key 没有顺序，key 与 key 之间不能重复），实际上 Map 确实包含了一个 keySet()方法，用于返回 Map 里所有 key 组成的 Set 集合。

不仅如此，Map 里 key 集和 Set 集合里元素的存储形式也很像，Map 子类和 Set 子类在名字上也惊人地相似，比如 Set 接口下有 HashSet、LinkedHashSet、SortedSet（接口）、TreeSet、EnumSet 等子接口和实现类，而 Map 接口下则有 HashMap、LinkedHashMap、SortedMap（接口）、TreeMap、EnumMap 等子接口和实现类。正如它们的名字所暗示的，Map 的这些实现类和子接口中 key 集的存储形式和对应 Set 集合中元素的存储形式完全相同。

提示：

Set 与 Map 之间的关系非常密切。虽然 Map 中放的元素是 key-value 对，Set 集合中放的元素是单个对象，但如果把 key-value 对中的 value 当成 key 的附庸：key 在哪里，value 就跟在哪里。这样就可以像对待 Set 一样来对待 Map 了。事实上，Map 提供了一个 Entry 内部类来封装 key-value 对，而计算 Entry 存储时则只考虑 Entry 封装的 key。从 Java 源码来看，Java 是先实现了 Map，然后通过包装一个所有 value 都为 null 的 Map 就实现了 Set 集合。

如果把 Map 里的所有 value 放在一起来看，它们又非常类似于一个 List：元素与元素之间可以重复，每个元素可以根据索引来查找，只是 Map 中的索引不再使用整数值，而是以另一个对象作为索引。如果需要从 List 集合中取出元素，则需要提供该元素的数字索引；如果需要从 Map 中取出元素，则需要提供该元素的 key 索引。因此，Map 有时也被称为字典，或关联数组。Map 接口中定义了如下常用的方法。

- void clear()：删除该 Map 对象中的所有 key-value 对。
- boolean containsKey(Object key)：查询 Map 中是否包含指定的 key，如果包含则返回 true。
- boolean containsValue(Object value)：查询 Map 中是否包含一个或多个 value，如果包含则返回 true。
- Set entrySet()：返回 Map 中包含的 key-value 对所组成的 Set 集合，每个集合元素都是 Map.Entry（Entry 是 Map 的内部类）对象。
- Object get(Object key)：返回指定 key 所对应的 value；如果此 Map 中不包含该 key，则返回 null。
- boolean isEmpty()：查询该 Map 是否为空（即不包含任何 key-value 对），如果为空则返回 true。
- Set keySet()：返回该 Map 中所有 key 组成的 Set 集合。

- Object put(Object key, Object value)：添加一个 key-value 对，如果当前 Map 中已有一个与该 key 相等的 key-value 对，则新的 key-value 对会覆盖原来的 key-value 对。
- void putAll(Map m)：将指定 Map 中的 key-value 对复制到本 Map 中。
- Object remove(Object key)：删除指定 key 所对应的 key-value 对，返回被删除 key 所关联的 value，如果该 key 不存在，则返回 null。
- boolean remove(Object key, Object value)：这是 Java 8 新增的方法，删除指定 key、value 所对应的 key-value 对。如果从该 Map 中成功地删除该 key-value 对，该方法返回 true，否则返回 false。
- int size()：返回该 Map 里的 key-value 对的个数。
- Collection values()：返回该 Map 里所有 value 组成的 Collection。

Map 接口提供了大量的实现类，典型实现如 HashMap 和 Hashtable 等、HashMap 的子类 LinkedHashMap，还有 SortedMap 子接口及该接口的实现类 TreeMap，以及 WeakHashMap、IdentityHashMap 等。下面将详细介绍 Map 接口实现类。

Map 中包括一个内部类 Entry，该类封装了一个 key-value 对。Entry 包含如下三个方法。

- Object getKey()：返回该 Entry 里包含的 key 值。
- Object getValue()：返回该 Entry 里包含的 value 值。
- Object setValue(V value)：设置该 Entry 里包含的 value 值，并返回新设置的 value 值。

Map 集合最典型的用法就是成对地添加、删除 key-value 对，接下来即可判断该 Map 中是否包含指定 key，是否包含指定 value，也可以通过 Map 提供的 keySet()方法获取所有 key 组成的集合，进而遍历 Map 中所有的 key-value 对。下面程序示范了 Map 的基本功能。

程序清单：codes\08\8.6\MapTest.java

```
public class MapTest
{
    public static void main(String[] args)
    {
        Map map = new HashMap();
        // 成对放入多个 key-value 对
        map.put("疯狂 Java 讲义" , 109);
        map.put("疯狂 iOS 讲义" , 10);
        map.put("疯狂 Ajax 讲义" , 79);
        // 多次放入的 key-value 对中 value 可以重复
        map.put("轻量级 Java EE 企业应用实战" , 99);
        // 放入重复的 key 时，新的 value 会覆盖原有的 value
        // 如果新的 value 覆盖了原有的 value，该方法返回被覆盖的 value
        System.out.println(map.put("疯狂 iOS 讲义" , 99)); // 输出 10
        System.out.println(map); // 输出的 Map 集合包含 4 个 key-value 对
        // 判断是否包含指定 key
        System.out.println("是否包含值为 疯狂 iOS 讲义 key: "
            + map.containsKey("疯狂 iOS 讲义")); // 输出 true
        // 判断是否包含指定 value
        System.out.println("是否包含值为 99 value: "
            + map.containsValue(99)); // 输出 true
        // 获取 Map 集合的所有 key 组成的集合，通过遍历 key 来实现遍历所有的 key-value 对
        for (Object key : map.keySet() )
        {
            // map.get(key)方法获取指定 key 对应的 value
            System.out.println(key + "-->" + map.get(key));
        }
        map.remove("疯狂 Ajax 讲义"); // 根据 key 来删除 key-value 对
        System.out.println(map); // 输出结果中不再包含 疯狂 Ajax 讲义=79 的 key-value 对
    }
}
```

上面程序中前 5 行粗体字代码示范了向 Map 中成对地添加 key-value 对。添加 key-value 对时，Map 允许多个 vlaue 重复，但如果添加 key-value 对时 Map 中已有重复的 key，那么新添加的 value 会覆盖该 key 原来对应的 value，该方法将会返回被覆盖的 value。

程序接下来的 2 行粗体字代码分别判断了 Map 集合中是否包含指定 key、指定 value。程序中粗体

字 foreach 循环用于遍历 Map 集合：程序先调用 Map 集合的 keySet()获取所有的 key，然后使用 foreach 循环来遍历 Map 的所有 key，根据 key 即可遍历所有的 value。

HashMap 重写了 toString()方法，实际上所有的 Map 实现类都重写了 toString()方法，调用 Map 对象的 toString()方法总是返回如下格式的字符串：{key1=value1,key2=value2...}。

8.6.1 Java 8 为 Map 新增的方法

Java 8 除为 Map 增加了 remove(Object key , Object value)默认方法之外，还增加了如下方法。

- Object compute(Object key, BiFunction remappingFunction)：该方法使用 remappingFunction 根据原 key-value 对计算一个新 value。只要新 value 不为 null，就使用新 value 覆盖原 value；如果原 value 不为 null，但新 value 为 null，则删除原 key-value 对；如果原 value、新 value 同时为 null，那么该方法不改变任何 key-value 对，直接返回 null。
- Object computeIfAbsent(Object key, Function mappingFunction)：如果传给该方法的 key 参数在 Map 中对应的 value 为 null，则使用 mappingFunction 根据 key 计算一个新的结果，如果计算结果不为 null，则用计算结果覆盖原有的 value。如果原 Map 原来不包括该 key，那么该方法可能会添加一组 key-value 对。
- Object computeIfPresent(Object key, BiFunction remappingFunction)：如果传给该方法的 key 参数在 Map 中对应的 value 不为 null，该方法将使用 remappingFunction 根据原 key、value 计算一个新的结果，如果计算结果不为 null，则使用该结果覆盖原来的 value；如果计算结果为 null，则删除原 key-value 对。
- void forEach(BiConsumer action)：该方法是 Java 8 为 Map 新增的一个遍历 key-value 对的方法，通过该方法可以更简洁地遍历 Map 的 key-value 对。
- Object getOrDefault(Object key, V defaultValue)：获取指定 key 对应的 value。如果该 key 不存在，则返回 defaultValue。
- Object merge(Object key, Object value, BiFunction remappingFunction)：该方法会先根据 key 参数获取该 Map 中对应的 value。如果获取的 value 为 null，则直接用传入的 value 覆盖原有的 value（在这种情况下，可能要添加一组 key-value 对）；如果获取的 value 不为 null，则使用 remappingFunction 函数根据原 value、新 value 计算一个新的结果，并用得到的结果去覆盖原有的 value。
- Object putIfAbsent(Object key, Object value)：该方法会自动检测指定 key 对应的 value 是否为 null，如果该 key 对应的 value 为 null，该方法将会用新 value 代替原来的 null 值。
- Object replace(Object key, Object value)：将 Map 中指定 key 对应的 value 替换成新 value。与传统 put()方法不同的是，该方法不可能添加新的 key-value 对。如果尝试替换的 key 在原 Map 中不存在，该方法不会添加 key-value 对，而是返回 null。
- boolean replace(K key, V oldValue, V newValue)：将 Map 中指定 key-value 对的原 value 替换成新 value。如果在 Map 中找到指定的 key-value 对，则执行替换并返回 true，否则返回 false。
- replaceAll(BiFunction function)：该方法使用 BiFunction 对原 key-value 对执行计算，并将计算结果作为该 key-value 对的 value 值。

下面程序示范了 Map 常用默认方法的功能和用法。

程序清单：codes\08\8.6\MapTest2.java

```
public class MapTest2
{
    public static void main(String[] args)
    {
        Map map = new HashMap();
        // 成对放入多个 key-value 对
        map.put("疯狂 Java 讲义" , 109);
        map.put("疯狂 iOS 讲义" , 99);
        map.put("疯狂 Ajax 讲义" , 79);
        // 尝试替换 key 为"疯狂 XML 讲义"的 value，由于原 Map 中没有对应的 key
```

```
        // 因此 Map 没有改变，不会添加新的 key-value 对
        map.replace("疯狂 XML 讲义" , 66);
        System.out.println(map);
        // 使用原 value 与传入参数计算出来的结果覆盖原有的 value
        map.merge("疯狂 iOS 讲义" , 10 ,
            (oldVal , param) -> (Integer)oldVal + (Integer)param);
        System.out.println(map); // "疯狂 iOS 讲义"的 value 增大了 10
        // 当 key 为"Java"对应的 value 为 null（或不存在）时，使用计算的结果作为新 value
        map.computeIfAbsent("Java" , (key)->((String)key).length());
        System.out.println(map); // map 中添加了 Java=4 这组 key-value 对
        // 当 key 为"Java"对应的 value 存在时，使用计算的结果作为新 value
        map.computeIfPresent("Java",
            (key , value) -> (Integer)value * (Integer)value);
        System.out.println(map); // map 中 Java=4 变成 Java=16
    }
}
```

上面程序中注释已经写得很清楚了，而且给出了每个方法的运行结果，读者可以结合这些方法的介绍文档来阅读该程序，从而掌握 Map 中这些默认方法的功能与用法。

8.6.2 Java 8 改进的 HashMap 和 Hashtable 实现类

HashMap 和 Hashtable 都是 Map 接口的典型实现类，它们之间的关系完全类似于 ArrayList 和 Vector 的关系：Hashtable 是一个古老的 Map 实现类，它从 JDK 1.0 起就已经出现了，当它出现时，Java 还没有提供 Map 接口，所以它包含了两个烦琐的方法，即 elements()（类似于 Map 接口定义的 values()方法）和 keys()（类似于 Map 接口定义的 keySet()方法），现在很少使用这两个方法（关于这两个方法的用法请参考 8.9 节）。

Java 8 改进了 HashMap 的实现，使用 HashMap 存在 key 冲突时依然具有较好的性能。

除此之外，Hashtable 和 HashMap 存在两点典型区别。

- Hashtable 是一个线程安全的 Map 实现，但 HashMap 是线程不安全的实现，所以 HashMap 比 Hashtable 的性能高一点；但如果有多个线程访问同一个 Map 对象时，使用 Hashtable 实现类会更好。
- Hashtable 不允许使用 null 作为 key 和 value，如果试图把 null 值放进 Hashtable 中，将会引发 NullPointerException 异常；但 HashMap 可以使用 null 作为 key 或 value。

由于 HashMap 里的 key 不能重复，所以 HashMap 里最多只有一个 key-value 对的 key 为 null，但可以有无数多个 key-value 对的 value 为 null。下面程序示范了用 null 值作为 HashMap 的 key 和 value 的情形。

程序清单：codes\08\8.6\NullInHashMap.java

```
public class NullInHashMap
{
    public static void main(String[] args)
    {
        HashMap hm = new HashMap();
        // 试图将两个 key 为 null 值的 key-value 对放入 HashMap 中
        hm.put(null , null);
        hm.put(null , null);    // ①
        // 将一个 value 为 null 值的 key-value 对放入 HashMap 中
        hm.put("a" , null);    // ②
        // 输出 Map 对象
        System.out.println(hm);
    }
}
```

上面程序试图向 HashMap 中放入三个 key-value 对，其中①代码处无法将 key-value 对放入，因为 Map 中已经有一个 key-value 对的 key 为 null 值，所以无法再放入 key 为 null 值的 key-value 对。②代码处可以放入该 key-value 对，因为一个 HashMap 中可以有多个 value 为 null 值。编译、运行上面程序，看到如下输出结果：

```
{null=null, a=null}
```

注意：

从Hashtable的类名上就可以看出它是一个古老的类，它的命名甚至没有遵守Java的命名规范：每个单词的首字母都应该大写。也许当初开发Hashtable的工程师也没有注意到这一点，后来大量Java程序中使用了Hashtable类，所以这个类名也就不能改为HashTable了，否则将导致大量程序需要改写。与Vector类似的是，尽量少用Hashtable实现类，即使需要创建线程安全的Map实现类，也无须使用Hashtable实现类，可以通过后面介绍的Collections工具类把HashMap变成线程安全的。

为了成功地在HashMap、Hashtable中存储、获取对象，用作key的对象必须实现hashCode()方法和equals()方法。

与HashSet集合不能保证元素的顺序一样，HashMap、Hashtable也不能保证其中key-value对的顺序。类似于HashSet，HashMap、Hashtable判断两个key相等的标准也是：两个key通过equals()方法比较返回true，两个key的hashCode值也相等。

除此之外，HashMap、Hashtable中还包含一个containsValue()方法，用于判断是否包含指定的value。那么HashMap、Hashtable如何判断两个value相等呢？HashMap、Hashtable判断两个value相等的标准更简单：只要两个对象通过equals()方法比较返回true即可。下面程序示范了Hashtable判断两个key相等的标准和两个value相等的标准。

程序清单：codes\08\8.6\HashtableTest.java

```
class A
{
    int count;
    public A(int count)
    {
        this.count = count;
    }
    // 根据count的值来判断两个对象是否相等
    public boolean equals(Object obj)
    {
        if (obj == this)
            return true;
        if (obj != null && obj.getClass() == A.class)
        {
            A a = (A)obj;
            return this.count == a.count;
        }
        return false;
    }
    // 根据count来计算hashCode值
    public int hashCode()
    {
        return this.count;
    }
}
class B
{
    // 重写equals()方法，B对象与任何对象通过equals()方法比较都返回true
    public boolean equals(Object obj)
    {
        return true;
    }
}
public class HashtableTest
{
    public static void main(String[] args)
    {
        Hashtable ht = new Hashtable();
        ht.put(new A(60000) , "疯狂Java讲义");
        ht.put(new A(87563) , "轻量级Java EE企业应用实战");
        ht.put(new A(1232) , new B());
```

```
        System.out.println(ht);
        // 只要两个对象通过 equals()方法比较返回 true
        // Hashtable 就认为它们是相等的 value
        // 由于 Hashtable 中有一个 B 对象
        // 它与任何对象通过 equals()方法比较都相等，所以下面输出 true
        System.out.println(ht.containsValue("测试字符串")); // ① 输出 true
        // 只要两个 A 对象的 count 相等，它们通过 equals()方法比较返回 true，且 hashCode 值相等
        // Hashtable 即认为它们是相同的 key，所以下面输出 true
        System.out.println(ht.containsKey(new A(87563))); // ② 输出 true
        // 下面语句可以删除最后一个 key-value 对
        ht.remove(new A(1232));    // ③
        System.out.println(ht);
    }
}
```

上面程序定义了 A 类和 B 类，其中 A 类判断两个 A 对象相等的标准是 count 实例变量：只要两个 A 对象的 count 变量相等，则通过 equals()方法比较它们返回 true，它们的 hashCode 值也相等；而 B 对象则可以与任何对象相等。

Hashtable 判断 value 相等的标准是：value 与另外一个对象通过 equals()方法比较返回 true 即可。上面程序中的 ht 对象中包含了一个 B 对象，它与任何对象通过 equals()方法比较总是返回 true，所以在①代码处返回 true。在这种情况下，不管传给 ht 对象的 containtsValue()方法参数是什么，程序总是返回 true。

根据 Hashtable 判断两个 key 相等的标准，程序在②处也将输出 true，因为两个 A 对象虽然不是同一个对象，但它们通过 equals()方法比较返回 true，且 hashCode 值相等，Hashtable 即认为它们是同一个 key。类似的是，程序在③处也可以删除对应的 key-value 对。

注意：

当使用自定义类作为 HashMap、Hashtable 的 key 时，如果重写该类的 equals(Object obj)和 hashCode()方法，则应该保证两个方法的判断标准一致——当两个 key 通过 equals()方法比较返回 true 时，两个 key 的 hashCode()返回值也应该相同。因为 HashMap、Hashtable 保存 key 的方式与 HashSet 保存集合元素的方式完全相同，所以 HashMap、Hashtable 对 key 的要求与 HashSet 对集合元素的要求完全相同。

与 HashSet 类似的是，如果使用可变对象作为 HashMap、Hashtable 的 key，并且程序修改了作为 key 的可变对象，则也可能出现与 HashSet 类似的情形：程序再也无法准确访问到 Map 中被修改过的 key。看下面程序。

程序清单：codes\08\8.6\HashMapErrorTest.java

```
public class HashMapErrorTest
{
    public static void main(String[] args)
    {
        HashMap ht = new HashMap();
        // 此处的 A 类与前一个程序的 A 类是同一个类
        ht.put(new A(60000) , "疯狂 Java 讲义");
        ht.put(new A(87563) , "轻量级 Java EE 企业应用实战");
        // 获得 Hashtable 的 key Set 集合对应的 Iterator 迭代器
        Iterator it = ht.keySet().iterator();
        // 取出 Map 中第一个 key，并修改它的 count 值
        A first = (A)it.next();
        first.count = 87563;   // ①
        // 输出{A@1560b=疯狂 Java 讲义, A@1560b=轻量级 Java EE 企业应用实战}
        System.out.println(ht);
        // 只能删除没有被修改过的 key 所对应的 key-value 对
        ht.remove(new A(87563));
        System.out.println(ht);
        // 无法获取剩下的 value，下面两行代码都将输出 null
        System.out.println(ht.get(new A(87563)));   // ② 输出 null
```

```
        System.out.println(ht.get(new A(60000)));   // ③ 输出 null
    }
}
```

该程序使用了前一个程序定义的 A 类实例作为 key，而 A 对象是可变对象。当程序在①处修改了 A 对象后，实际上修改了 HashMap 集合中元素的 key，这就导致该 key 不能被准确访问。当程序试图删除 count 为 87563 的 A 对象时，只能删除没被修改的 key 所对应的 key-value 对。程序②和③处的代码都不能访问“疯狂 Java 讲义”字符串，这都是因为它对应的 key 被修改过的原因。

注意：

与 HashSet 类似的是，尽量不要使用可变对象作为 HashMap、Hashtable 的 key，如果确实需要使用可变对象作为 HashMap、Hashtable 的 key，则尽量不要在程序中修改作为 key 的可变对象。

8.6.3 LinkedHashMap 实现类

HashSet 有一个 LinkedHashSet 子类，HashMap 也有一个 LinkedHashMap 子类；LinkedHashMap 也使用双向链表来维护 key-value 对的次序（其实只需要考虑 key 的次序），该链表负责维护 Map 的迭代顺序，迭代顺序与 key-value 对的插入顺序保持一致。

LinkedHashMap 可以避免对 HashMap、Hashtable 里的 key-value 对进行排序（只要插入 key-value 对时保持顺序即可），同时又可避免使用 TreeMap 所增加的成本。

LinkedHashMap 需要维护元素的插入顺序，因此性能略低于 HashMap 的性能；但因为它以链表来维护内部顺序，所以在迭代访问 Map 里的全部元素时将有较好的性能。下面程序示范了 LinkedHashMap 的功能：迭代输出 LinkedHashMap 的元素时，将会按添加 key-value 对的顺序输出。

程序清单：codes\08\8.6\LinkedHashMapTest.java

```
public class LinkedHashMapTest
{
    public static void main(String[] args)
    {
        LinkedHashMap scores = new LinkedHashMap();
        scores.put("语文" , 80);
        scores.put("英文" , 82);
        scores.put("数学" , 76);
        // 调用 forEach()方法遍历 scores 里的所有 key-value 对
        scores.forEach((key, value) -> System.out.println(key + "-->" + value));
    }
}
```

上面程序中最后一行代码使用 Java 8 为 Map 新增的 forEach()方法来遍历 Map 集合。编译、运行上面程序，即可看到 LinkedHashMap 的功能：LinkedHashMap 可以记住 key-value 对的添加顺序。

8.6.4 使用 Properties 读写属性文件

Properties 类是 Hashtable 类的子类，正如它的名字所暗示的，该对象在处理属性文件时特别方便（Windows 操作平台上的 ini 文件就是一种属性文件）。Properties 类可以把 Map 对象和属性文件关联起来，从而可以把 Map 对象中的 key-value 对写入属性文件中，也可以把属性文件中的“属性名=属性值”加载到 Map 对象中。由于属性文件里的属性名、属性值只能是字符串类型，所以 Properties 里的 key、value 都是字符串类型。该类提供了如下三个方法来修改 Properties 里的 key、value 值。

提示：

Properties 相当于一个 key、value 都是 String 类型的 Map。

- String getProperty(String key)：获取 Properties 中指定属性名对应的属性值，类似于 Map 的 get(Object key)方法。

- String getProperty(String key, String defaultValue)：该方法与前一个方法基本相似。该方法多一个功能，如果 Properties 中不存在指定的 key 时，则该方法指定默认值。
- Object setProperty(String key, String value)：设置属性值，类似于 Hashtable 的 put()方法。

除此之外，它还提供了两个读写属性文件的方法。

- void load(InputStream inStream)：从属性文件（以输入流表示）中加载 key-value 对，把加载到的 key-value 对追加到 Properties 里（Properties 是 Hashtable 的子类，它不保证 key-value 对之间的次序）。
- void store(OutputStream out, String comments)：将 Properties 中的 key-value 对输出到指定的属性文件（以输出流表示）中。

上面两个方法中使用了 InputStream 类和 OutputStream 类，它们是 Java IO 体系中的两个基类，关于这两个类的详细介绍请参考第 15 章。

程序清单：codes\08\8.6\PropertiesTest.java

```
public class PropertiesTest
{
    public static void main(String[] args)
        throws Exception
    {
        Properties props = new Properties();
        // 向 Properties 中添加属性
        props.setProperty("username" , "yeeku");
        props.setProperty("password" , "123456");
        // 将 Properties 中的 key-value 对保存到 a.ini 文件中
        props.store(new FileOutputStream("a.ini")
            , "comment line");  // ①
        // 新建一个 Properties 对象
        Properties props2 = new Properties();
        // 向 Properties 中添加属性
        props2.setProperty("gender" , "male");
        // 将 a.ini 文件中的 key-value 对追加到 props2 中
        props2.load(new FileInputStream("a.ini") );  // ②
        System.out.println(props2);
    }
}
```

上面程序示范了 Properties 类的用法，其中①代码处将 Properties 对象中的 key-value 对写入 a.ini 文件中；②代码处则从 a.ini 文件中读取 key-value 对，并添加到 props2 对象中。编译、运行上面程序，该程序输出结果如下：

```
{password=123456, gender=male, username=yeeku}
```

上面程序还在当前路径下生成了一个 a.ini 文件，该文件的内容如下：

```
#comment line
#Thu Apr 17 00:40:22 CST 2014
password=123456
username=yeeku
```

Properties 可以把 key-value 对以 XML 文件的形式保存起来，也可以从 XML 文件中加载 key-value 对，用法与此类似，此处不再赘述。

8.6.5 SortedMap 接口和 TreeMap 实现类

正如 Set 接口派生出 SortedSet 子接口，SortedSet 接口有一个 TreeSet 实现类一样，Map 接口也派生出一个 SortedMap 子接口，SortedMap 接口也有一个 TreeMap 实现类。

TreeMap 就是一个红黑树数据结构，每个 key-value 对即作为红黑树的一个节点。TreeMap 存储 key-value 对（节点）时，需要根据 key 对节点进行排序。TreeMap 可以保证所有的 key-value 对处于有序状态。TreeMap 也有两种排序方式。

- 自然排序：TreeMap 的所有 key 必须实现 Comparable 接口，而且所有的 key 应该是同一个类的对象，否则将会抛出 ClassCastException 异常。

- 定制排序：创建 TreeMap 时，传入一个 Comparator 对象，该对象负责对 TreeMap 中的所有 key 进行排序。采用定制排序时不要求 Map 的 key 实现 Comparable 接口。

类似于 TreeSet 中判断两个元素相等的标准，TreeMap 中判断两个 key 相等的标准是：两个 key 通过 compareTo()方法返回 0，TreeMap 即认为这两个 key 是相等的。

如果使用自定义类作为 TreeMap 的 key，且想让 TreeMap 良好地工作，则重写该类的 equals()方法和 compareTo()方法时应保持一致的返回结果：两个 key 通过 equals()方法比较返回 true 时，它们通过 compareTo()方法比较应该返回 0。如果 equals()方法与 compareTo()方法的返回结果不一致， TreeMap 与 Map 接口的规则就会冲突。

注意：

再次强调：Set 和 Map 的关系十分密切，Java 源码就是先实现了 HashMap、TreeMap 等集合，然后通过包装一个所有的 value 都为 null 的 Map 集合实现了 Set 集合类。

与 TreeSet 类似的是，TreeMap 中也提供了一系列根据 key 顺序访问 key-value 对的方法。

- Map.Entry firstEntry()：返回该 Map 中最小 key 所对应的 key-value 对，如果该 Map 为空，则返回 null。
- Object firstKey()：返回该 Map 中的最小 key 值，如果该 Map 为空，则返回 null。
- Map.Entry lastEntry()：返回该 Map 中最大 key 所对应的 key-value 对，如果该 Map 为空或不存在这样的 key-value 对，则都返回 null。
- Object lastKey()：返回该 Map 中的最大 key 值，如果该 Map 为空或不存在这样的 key，则都返回 null。
- Map.Entry higherEntry(Object key)：返回该 Map 中位于 key 后一位的 key-value 对（即大于指定 key 的最小 key 所对应的 key-value 对）。如果该 Map 为空，则返回 null。
- Object higherKey(Object key)：返回该 Map 中位于 key 后一位的 key 值（即大于指定 key 的最小 key 值）。如果该 Map 为空或不存在这样的 key-value 对，则都返回 null。
- Map.Entry lowerEntry(Object key)：返回该 Map 中位于 key 前一位的 key-value 对（即小于指定 key 的最大 key 所对应的 key-value 对）。如果该 Map 为空或不存在这样的 key-value 对，则都返回 null。
- Object lowerKey(Object key)：返回该 Map 中位于 key 前一位的 key 值（即小于指定 key 的最大 key 值）。如果该 Map 为空或不存在这样的 key，则都返回 null。
- NavigableMap subMap(Object fromKey, boolean fromInclusive, Object toKey, boolean toInclusive)：返回该 Map 的子 Map，其 key 的范围是从 fromKey（是否包括取决于第二个参数）到 toKey（是否包括取决于第四个参数）。
- SortedMap subMap(Object fromKey, Object toKey)：返回该 Map 的子 Map，其 key 的范围是从 fromKey（包括）到 toKey（不包括）。
- SortedMap tailMap(Object fromKey)：返回该 Map 的子 Map，其 key 的范围是大于 fromKey（包括）的所有 key。
- NavigableMap tailMap(Object fromKey, boolean inclusive)：返回该 Map 的子 Map，其 key 的范围是大于 fromKey（是否包括取决于第二个参数）的所有 key。
- SortedMap headMap(Object toKey)：返回该 Map 的子 Map，其 key 的范围是小于 toKey（不包括）的所有 key。
- NavigableMap headMap(Object toKey, boolean inclusive)：返回该 Map 的子 Map，其 key 的范围是小于 toKey（是否包括取决于第二个参数）的所有 key。

提示：

表面上看起来这些方法很复杂，其实它们很简单。因为 TreeMap 中的 key-value 对是有序的，所以增加了访问第一个、前一个、后一个、最后一个 key-value 对的方法，并提供了几个从 TreeMap 中截取子 TreeMap 的方法。

下面以自然排序为例，介绍 TreeMap 的基本用法。

程序清单：codes\08\8.6\TreeMapTest.java

```
class R implements Comparable
{
    int count;
    public R(int count)
    {
        this.count = count;
    }
    public String toString()
    {
        return "R[count:" + count + "]";
    }
    // 根据 count 来判断两个对象是否相等
    public boolean equals(Object obj)
    {
        if (this == obj)
            return true;
        if (obj != null    && obj.getClass() == R.class)
        {
            R r = (R)obj;
            return r.count == this.count;
        }
        return false;
    }
    // 根据 count 属性值来判断两个对象的大小
    public int compareTo(Object obj)
    {
        R r = (R)obj;
        return count > r.count ? 1 :
            count < r.count ? -1 : 0;
    }
}
public class TreeMapTest
{
    public static void main(String[] args)
    {
        TreeMap tm = new TreeMap();
        tm.put(new R(3) , "轻量级 Java EE 企业应用实战");
        tm.put(new R(-5) , "疯狂 Java 讲义");
        tm.put(new R(9) , "疯狂 Android 讲义");
        System.out.println(tm);
        // 返回该 TreeMap 的第一个 Entry 对象
        System.out.println(tm.firstEntry());
        // 返回该 TreeMap 的最后一个 key 值
        System.out.println(tm.lastKey());
        // 返回该 TreeMap 的比 new R(2)大的最小 key 值
        System.out.println(tm.higherKey(new R(2)));
        // 返回该 TreeMap 的比 new R(2)小的最大的 key-value 对
        System.out.println(tm.lowerEntry(new R(2)));
        // 返回该 TreeMap 的子 TreeMap
        System.out.println(tm.subMap(new R(-1) , new R(4)));
    }
}
```

上面程序中定义了一个 R 类，该类重写了 equals()方法，并实现了 Comparable 接口，所以可以使用该 R 对象作为 TreeMap 的 key，该 TreeMap 使用自然排序。运行上面程序，看到如下运行结果：

```
{R[count:-5]=疯狂 Java 讲义, R[count:3]=轻量级 Java EE 企业应用实战, R[count:9]=疯狂
Android 讲义}
```

```
R[count:-5]=疯狂 Java 讲义
R[count:9]
R[count:3]
R[count:-5]=疯狂 Java 讲义
{R[count:3]=轻量级 Java EE 企业应用实战}
```

8.6.6 WeakHashMap 实现类

WeakHashMap 与 HashMap 的用法基本相似。与 HashMap 的区别在于，HashMap 的 key 保留了对实际对象的强引用，这意味着只要该 HashMap 对象不被销毁，该 HashMap 的所有 key 所引用的对象就不会被垃圾回收，HashMap 也不会自动删除这些 key 所对应的 key-value 对；但 WeakHashMap 的 key 只保留了对实际对象的弱引用，这意味着如果 WeakHashMap 对象的 key 所引用的对象没有被其他强引用变量所引用，则这些 key 所引用的对象可能被垃圾回收，WeakHashMap 也可能自动删除这些 key 所对应的 key-value 对。

WeakHashMap 中的每个 key 对象只持有对实际对象的弱引用，因此，当垃圾回收了该 key 所对应的实际对象之后，WeakHashMap 会自动删除该 key 对应的 key-value 对。看如下程序。

程序清单：codes\08\8.6\WeakHashMapTest.java

```
public class WeakHashMapTest
{
    public static void main(String[] args)
    {
        WeakHashMap whm = new WeakHashMap();
        // 向 WeakHashMap 中添加三个 key-value 对
        // 三个 key 都是匿名字符串对象（没有其他引用）
        whm.put(new String("语文") , new String("良好"));
        whm.put(new String("数学") , new String("及格"));
        whm.put(new String("英文") , new String("中等"));
        // 向 WeakHashMap 中添加一个 key-value 对
        // 该 key 是一个系统缓存的字符串对象
        whm.put("java" , new String("中等"));    // ①
        // 输出 whm 对象，将看到 4 个 key-value 对
        System.out.println(whm);
        // 通知系统立即进行垃圾回收
        System.gc();
        System.runFinalization();
        // 在通常情况下，将只看到一个 key-value 对
        System.out.println(whm);
    }
}
```

编译、运行上面程序，看到如下运行结果：

```
{英文=中等, java=中等, 数学=及格, 语文=良好}
{java=中等}
```

从上面运行结果可以看出，当系统进行垃圾回收时，删除了 WeakHashMap 对象的前三个 key-value 对。这是因为添加前三个 key-value 对（粗体字部分）时，这三个 key 都是匿名的字符串对象，WeakHashMap 只保留了对它们的弱引用，这样垃圾回收时会自动删除这三个 key-value 对。

WeakHashMap 对象中第 4 个组 key-value 对（①号粗体字代码行）的 key 是一个字符串直接量，（系统会自动保留对该字符串对象的强引用），所以垃圾回收时不会回收它。

注意：

如果需要使用 WeakHashMap 的 key 来保留对象的弱引用，则不要让该 key 所引用的对象具有任何强引用，否则将失去使用 WeakHashMap 的意义。

8.6.7 IdentityHashMap 实现类

这个 Map 实现类的实现机制与 HashMap 基本相似，但它在处理两个 key 相等时比较独特：在 IdentityHashMap 中，当且仅当两个 key 严格相等（key1 == key2）时，IdentityHashMap 才认为两个 key 相等；对于普通的 HashMap 而言，只要 key1 和 key2 通过 equals()方法比较返回 true，且它们的 hashCode 值相等即可。

注意：

IdentityHashMap 是一个特殊的 Map 实现！此类实现 Map 接口时，它有意违反 Map 的通常规范：IdentityHashMap 要求两个 key 严格相等时才认为两个 key 相等。

IdentityHashMap 提供了与 HashMap 基本相似的方法，也允许使用 null 作为 key 和 value。与 HashMap 相似：IdentityHashMap 也不保证 key-value 对之间的顺序，更不能保证它们的顺序随时间的推移保持不变。

程序清单：codes\08\8.6\IdentityHashMapTest.java

```
public class IdentityHashMapTest
{
    public static void main(String[] args)
    {
        IdentityHashMap ihm = new IdentityHashMap();
        // 下面两行代码将会向 IdentityHashMap 对象中添加两个 key-value 对
        ihm.put(new String("语文") , 89);
        ihm.put(new String("语文") , 78);
        // 下面两行代码只会向 IdentityHashMap 对象中添加一个 key-value 对
        ihm.put("java" , 93);
        ihm.put("java" , 98);
        System.out.println(ihm);
    }
}
```

编译、运行上面程序，看到如下运行结果：

```
{java=98, 语文=78, 语文=89}
```

上面程序试图向 IdentityHashMap 对象中添加 4 个 key-value 对，前 2 个 key-value 对中的 key 是新创建的字符串对象，它们通过==比较不相等，所以 IdentityHashMap 会把它们当成 2 个 key 来处理；后 2 个 key-value 对中的 key 都是字符串直接量，而且它们的字符序列完全相同，Java 使用常量池来管理字符串直接量，所以它们通过==比较返回 true，IdentityHashMap 会认为它们是同一个 key，因此只有一次可以添加成功。

8.6.8 EnumMap 实现类

EnumMap 是一个与枚举类一起使用的 Map 实现，EnumMap 中的所有 key 都必须是单个枚举类的枚举值。创建 EnumMap 时必须显式或隐式指定它对应的枚举类。EnumMap 具有如下特征。

- EnumMap 在内部以数组形式保存，所以这种实现形式非常紧凑、高效。
- EnumMap 根据 key 的自然顺序（即枚举值在枚举类中的定义顺序）来维护 key-value 对的顺序。当程序通过 keySet()、entrySet()、values()等方法遍历 EnumMap 时可以看到这种顺序。
- EnumMap 不允许使用 null 作为 key，但允许使用 null 作为 value。如果试图使用 null 作为 key 时将抛出 NullPointerException 异常。如果只是查询是否包含值为 null 的 key，或只是删除值为 null 的 key，都不会抛出异常。

与创建普通的 Map 有所区别的是，创建 EnumMap 时必须指定一个枚举类，从而将该 EnumMap 和指定枚举类关联起来。

下面程序示范了 EnumMap 的用法。

程序清单：codes\08\8.6\EnumMapTest.java

```
enum Season
{
```

```
    SPRING,SUMMER,FALL,WINTER
}
public class EnumMapTest
{
    public static void main(String[] args)
    {
        // 创建 EnumMap 对象，该 EnumMap 的所有 key 都是 Season 枚举类的枚举值
        EnumMap enumMap = new EnumMap(Season.class);
        enumMap.put(Season.SUMMER , "夏日炎炎");
        enumMap.put(Season.SPRING , "春暖花开");
        System.out.println(enumMap);
    }
}
```

上面程序中创建了一个 EnumMap 对象，创建该 EnumMap 对象时指定它的 key 只能是 Season 枚举类的枚举值。如果向该 EnumMap 中添加两个 key-value 对后，这两个 key-value 对将会以 Season 枚举值的自然顺序排序。

编译、运行上面程序，看到如下运行结果：

```
{SPRING=春暖花开, SUMMER=夏日炎炎}
```

8.6.9 各 Map 实现类的性能分析

对于 Map 的常用实现类而言，虽然 HashMap 和 Hashtable 的实现机制几乎一样，但由于 Hashtable 是一个古老的、线程安全的集合，因此 HashMap 通常比 Hashtable 要快。

TreeMap 通常比 HashMap、Hashtable 要慢（尤其在插入、删除 key-value 对时更慢），因为 TreeMap 底层采用红黑树来管理 key-value 对（红黑树的每个节点就是一个 key-value 对）。

使用 TreeMap 有一个好处：TreeMap 中的 key-value 对总是处于有序状态，无须专门进行排序操作。当 TreeMap 被填充之后，就可以调用 keySet()，取得由 key 组成的 Set，然后使用 toArray()方法生成 key 的数组，接下来使用 Arrays 的 binarySearch()方法在已排序的数组中快速地查询对象。

对于一般的应用场景，程序应该多考虑使用 HashMap，因为 HashMap 正是为快速查询设计的（HashMap 底层其实也是采用数组来存储 key-value 对）。但如果程序需要一个总是排好序的 Map 时，则可以考虑使用 TreeMap。

LinkedHashMap 比 HashMap 慢一点，因为它需要维护链表来保持 Map 中 key-value 时的添加顺序。IdentityHashMap 性能没有特别出色之处，因为它采用与 HashMap 基本相似的实现，只是它使用==而不是 equals()方法来判断元素相等。EnumMap 的性能最好，但它只能使用同一个枚举类的枚举值作为 key。

8.7 HashSet 和 HashMap 的性能选项

对于 HashSet 及其子类而言，它们采用 hash 算法来决定集合中元素的存储位置，并通过 hash 算法来控制集合的大小；对于 HashMap、Hashtable 及其子类而言，它们采用 hash 算法来决定 Map 中 key 的存储，并通过 hash 算法来增加 key 集合的大小。

hash 表里可以存储元素的位置被称为“桶(bucket)”，在通常情况下，单个“桶”里存储一个元素，此时有最好的性能：hash 算法可以根据 hashCode 值计算出“桶”的存储位置，接着从“桶”中取出元素。但 hash 表的状态是 open 的：在发生“hash 冲突”的情况下，单个桶会存储多个元素，这些元素以链表形式存储，必须按顺序搜索。如图 8.8 所示是 hash 表保存各元素，且发生“hash 冲突”的示意图。

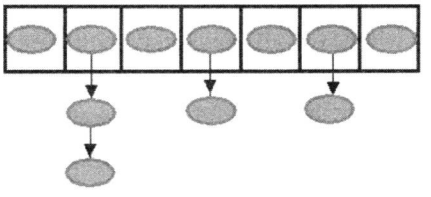


图 8.8 hash 表中存储元素的示意图

因为 HashSet 和 HashMap、Hashtable 都使用 hash 算法来决定其元素（HashMap 则只考虑 key）的存储，因此 HashSet、HashMap 的 hash 表包含如下属性。

- 容量（capacity）：hash 表中桶的数量。

- 初始化容量（initial capacity）：创建 hash 表时桶的数量。HashMap 和 HashSet 都允许在构造器中指定初始化容量。
- 尺寸（size）：当前 hash 表中记录的数量。
- 负载因子（load factor）：负载因子等于“size/capacity”。负载因子为 0，表示空的 hash 表，0.5 表示半满的 hash 表，依此类推。轻负载的 hash 表具有冲突少、适宜插入与查询的特点（但是使用 Iterator 迭代元素时比较慢）。

除此之外，hash 表里还有一个“负载极限”，“负载极限”是一个 0~1 的数值，“负载极限”决定了 hash 表的最大填满程度。当 hash 表中的负载因子达到指定的“负载极限”时，hash 表会自动成倍地增加容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为 rehashing。

HashSet 和 HashMap、Hashtable 的构造器允许指定一个负载极限，HashSet 和 HashMap、Hashtable 默认的“负载极限”为 0.75，这表明当该 hash 表的 3/4 已经被填满时，hash 表会发生 rehashing。

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：较高的“负载极限”可以降低 hash 表所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap 的 get()与 put()方法都要用到查询）；较低的“负载极限”会提高查询数据的性能，但会增加 hash 表所占用的内存开销。程序员可以根据实际情况来调整 HashSet 和 HashMap 的“负载极限”值。

如果开始就知道 HashSet 和 HashMap、Hashtable 会保存很多记录，则可以在创建时就使用较大的初始化容量，如果初始化容量始终大于 HashSet 和 HashMap、Hashtable 所包含的最大记录数除以“负载极限”，就不会发生 rehashing。使用足够大的初始化容量创建 HashSet 和 HashMap、Hashtable 时，可以更高效地增加记录，但将初始化容量设置太高可能会浪费空间，因此通常不要将初始化容量设置得过高。

8.8 操作集合的工具类：Collections

Java 提供了一个操作 Set、List 和 Map 等集合的工具类：Collections，该工具类里提供了大量方法对集合元素进行排序、查询和修改等操作，还提供了将集合对象设置为不可变、对集合对象实现同步控制等方法。

8.8.1 排序操作

Collections 提供了如下常用的类方法用于对 List 集合元素进行排序。

- void reverse(List list)：反转指定 List 集合中元素的顺序。
- void shuffle(List list)：对 List 集合元素进行随机排序（shuffle 方法模拟了“洗牌”动作）。
- void sort(List list)：根据元素的自然顺序对指定 List 集合的元素按升序进行排序。
- void sort(List list, Comparator c)：根据指定 Comparator 产生的顺序对 List 集合元素进行排序。
- void swap(List list, int i, int j)：将指定 List 集合中的 i 处元素和 j 处元素进行交换。
- void rotate(List list , int distance)：当 distance 为正数时，将 list 集合的后 distance 个元素“整体”移到前面；当 distance 为负数时，将 list 集合的前 distance 个元素“整体”移到后面。该方法不会改变集合的长度。

下面程序简单示范了利用 Collections 工具类来操作 List 集合。

程序清单：codes\08\8.8\SortTest.java

```
public class SortTest
{
    public static void main(String[] args)
    {
        ArrayList nums = new ArrayList();
        nums.add(2);
        nums.add(-5);
        nums.add(3);
        nums.add(0);
        System.out.println(nums); // 输出:[2, -5, 3, 0]
        Collections.reverse(nums); // 将 List 集合元素的次序反转
        System.out.println(nums); // 输出:[0, 3, -5, 2]
```

```
        Collections.sort(nums); // 将List集合元素按自然顺序排序
        System.out.println(nums); // 输出:[-5, 0, 2, 3]
        Collections.shuffle(nums); // 将List集合元素按随机顺序排序
        System.out.println(nums); // 每次输出的次序不固定
    }
}
```

上面代码示范了 Collections 类常用的排序操作。下面通过编写一个梭哈游戏来演示 List 集合、Collections 工具类的强大功能。

程序清单：codes\08\8.8\ShowHand.java

```
public class ShowHand
{
    // 定义该游戏最多支持多少个玩家
    private final int PLAY_NUM = 5;
    // 定义扑克牌的所有花色和数值
    private String[] types = {"方块" , "草花" ,"红心" , "黑桃"};
    private String[] values = {"2" , "3" , "4" , "5"
        , "6" , "7" , "8" , "9", "10"
        , "J" , "Q" , "K" , "A"};
    // cards是一局游戏中剩下的扑克牌
    private List<String> cards = new LinkedList<String>();
    // 定义所有的玩家
    private String[] players = new String[PLAY_NUM];
    // 所有玩家手上的扑克牌
    private List<String>[] playersCards = new List[PLAY_NUM];
    /**
     * 初始化扑克牌，放入52张扑克牌
     * 并且使用shuffle方法将它们按随机顺序排列
     */
    public void initCards()
    {
        for (int i = 0 ; i < types.length ; i++ )
        {
            for (int j = 0; j < values.length ; j++ )
            {
                cards.add(types[i] + values[j]);
            }
        }
        // 随机排列
        Collections.shuffle(cards);
    }
    /**
     * 初始化玩家，为每个玩家分派用户名
     */
    public void initPlayer(String... names)
    {
        if (names.length > PLAY_NUM || names.length < 2)
        {
            // 校验玩家数量，此处使用异常机制更合理
            System.out.println("玩家数量不对");
            return ;
        }
        else
        {
            // 初始化玩家用户名
            for (int i = 0; i < names.length ; i++ )
            {
                players[i] = names[i];
            }
        }
    }
    /**
     * 初始化玩家手上的扑克牌，开始游戏时每个玩家手上的扑克牌为空
     * 程序使用一个长度为0的LinkedList来表示
     */
    public void initPlayerCards()
```

```
    {
        for (int i = 0; i < players.length ; i++ )
        {
            if (players[i] != null && !players[i].equals(""))
            {
                playersCards[i] = new LinkedList<String>();
            }
        }
    }
    /**
     * 输出全部扑克牌，该方法没有实际作用，仅用作测试
     */
    public void showAllCards()
    {
        for (String card : cards )
        {
            System.out.println(card);
        }
    }
    /**
     * 派扑克牌
     * @param first 最先派给谁
     */
    public void deliverCard(String first)
    {
        // 调用 ArrayUtils 工具类的 search 方法
        // 查询出指定元素在数组中的索引
        int firstPos = ArrayUtils.search(players , first);
        // 依次给位于该指定玩家之后的每个玩家派扑克牌
        for (int i = firstPos; i < PLAY_NUM ; i ++)
        {
            if (players[i] != null)
            {
                playersCards[i].add(cards.get(0));
                cards.remove(0);
            }
        }
        // 依次给位于该指定玩家之前的每个玩家派扑克牌
        for (int i = 0; i < firstPos ; i ++)
        {
            if (players[i] != null)
            {
                playersCards[i].add(cards.get(0));
                cards.remove(0);
            }
        }
    }
    /**
     * 输出玩家手上的扑克牌
     * 实现该方法时，应该控制每个玩家看不到别人的第一张牌，但此处没有增加该功能
     */
    public void showPlayerCards()
    {
        for (int i = 0; i < PLAY_NUM ; i++ )
        {
            // 当该玩家不为空时
            if (players[i] != null)
            {
                // 输出玩家
                System.out.print(players[i] + " : " );
                // 遍历输出玩家手上的扑克牌
                for (String card : playersCards[i])
                {
                    System.out.print(card + "\t");
                }
            }
            System.out.print("\n");
        }
    }
```

```
    public static void main(String[] args)
    {
        ShowHand sh = new ShowHand();
        sh.initPlayer("电脑玩家" , "孙悟空");
        sh.initCards();
        sh.initPlayerCards();
        // 下面测试所有扑克牌，没有实际作用
        sh.showAllCards();
        System.out.println("---------------");
        // 下面从"孙悟空"开始派牌
        sh.deliverCard("孙悟空");
        sh.showPlayerCards();
        /*
        这个地方需要增加处理：
        1.牌面最大的玩家下注
        2.其他玩家是否跟注
        3.游戏是否只剩一个玩家?如果是，则他胜利了
        4.如果已经是最后一张扑克牌，则需要比较剩下玩家的牌面大小
        */
        // 再次从"电脑玩家"开始派牌
        sh.deliverCard("电脑玩家");
        sh.showPlayerCards();
    }
}
```

与五子棋游戏类似的是，这个程序也没有写完，读者可以参考该程序的思路把这个游戏补充完整。这个程序还使用了另一个工具类：ArrayUtils，这个工具类的代码保存在 codes\08\8.8\ArrayUtils.java 文件中，读者可参考本书光盘中的代码。

运行上面程序，即可看到如图 8.9 所示的界面。

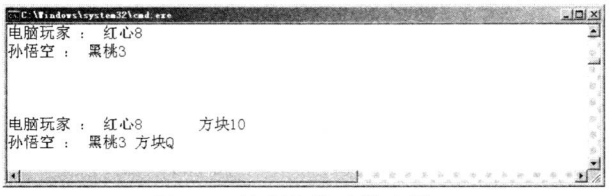


图 8.9　控制台梭哈游戏界面

注意：

上面程序中用到了泛型(Generic)知识，如 List<String>或 LinkedList<String>等写法，它表示在 List 集合中只能放 String 类型的对象。关于泛型的详细介绍，请参考第 9 章知识。

上面程序还有一个很烦琐的难点，就是比较玩家手上牌面值的大小，主要是因为梭哈游戏的规则较多（它分为对、三个、同花、顺子等），所以处理起来比较麻烦，读者可以一点一点地增加这些规则，只要该游戏符合自定义的规则，即表明这个游戏已经接近完成了。

8.8.2　查找、替换操作

Collections 还提供了如下常用的用于查找、替换集合元素的类方法。

- int binarySearch(List list, Object key)：使用二分搜索法搜索指定的 List 集合，以获得指定对象在 List 集合中的索引。如果要使该方法可以正常工作，则必须保证 List 中的元素已经处于有序状态。
- Object max(Collection coll)：根据元素的自然顺序，返回给定集合中的最大元素。
- Object max(Collection coll, Comparator comp)：根据 Comparator 指定的顺序，返回给定集合中的最大元素。
- Object min(Collection coll)：根据元素的自然顺序，返回给定集合中的最小元素。
- Object min(Collection coll, Comparator comp)：根据 Comparator 指定的顺序，返回给定集合中的最小元素。
- void fill(List list, Object obj)：使用指定元素 obj 替换指定 List 集合中的所有元素。
- int frequency(Collection c, Object o)：返回指定集合中指定元素的出现次数。
- int indexOfSubList(List source, List target)：返回子 List 对象在父 List 对象中第一次出现的位置索

引；如果父 List 中没有出现这样的子 List，则返回□-1。

- int lastIndexOfSubList(List source, List target)：返回子 List 对象在父 List 对象中最后一次出现的位置索引；如果父 List 中没有出现这样的子 List，则返回□-1。
- boolean replaceAll(List list, Object oldVal, Object newVal)：使用一个新值 newVal 替换 List 对象的所有旧值 oldVal。

下面程序简单示范了 Collections 工具类的用法。

程序清单：codes\08\8.8\SearchTest.java

```
public class SearchTest
{
    public static void main(String[] args)
    {
        ArrayList nums = new ArrayList();
        nums.add(2);
        nums.add(-5);
        nums.add(3);
        nums.add(0);
        System.out.println(nums); // 输出:[2, -5, 3, 0]
        System.out.println(Collections.max(nums)); // 输出最大元素，将输出 3
        System.out.println(Collections.min(nums)); // 输出最小元素，将输出-5
        Collections.replaceAll(nums , 0 , 1); // 将 nums 中的 0 使用 1 来代替
        System.out.println(nums); // 输出:[2, -5, 3, 1]
        // 判断-5 在 List 集合中出现的次数，返回 1
        System.out.println(Collections.frequency(nums , -5));
        Collections.sort(nums); // 对 nums 集合排序
        System.out.println(nums); // 输出:[-5, 1, 2, 3]
        //只有排序后的 List 集合才可用二分法查询，输出 3
        System.out.println(Collections.binarySearch(nums , 3));
    }
}
```

8.8.3 同步控制

Collections类中提供了多个synchronizedXxx()方法，该方法可以将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题。

Java 中常用的集合框架中的实现类 HashSet、TreeSet、ArrayList、ArrayDeque、LinkedList、HashMap 和 TreeMap 都是线程不安全的。如果有多个线程访问它们，而且有超过一个的线程试图修改它们，则存在线程安全的问题。Collections 提供了多个类方法可以把它们包装成线程同步的集合。

下面的示例程序创建了 4 个线程安全的集合对象。

程序清单：codes\08\8.8\SynchronizedTest.java

```
public class SynchronizedTest
{
    public static void main(String[] args)
    {
        // 下面程序创建了 4 个线程安全的集合对象
        Collection c = Collections
            .synchronizedCollection(new ArrayList());
        List list = Collections.synchronizedList(new ArrayList());
        Set s = Collections.synchronizedSet(new HashSet());
        Map m = Collections.synchronizedMap(new HashMap());
    }
}
```

在上面示例程序中，直接将新创建的集合对象传给了 Collections 的 synchronizedXxx 方法，这样就可以直接获取 List、Set 和 Map 的线程安全实现版本。

8.8.4 设置不可变集合

Collections 提供了如下三类方法来返回一个不可变的集合。

- emptyXxx()：返回一个空的、不可变的集合对象，此处的集合既可以是 List，也可以是 SortedSet、

Set，还可以是 Map、SortedMap 等。

- singletonXxx()：返回一个只包含指定对象（只有一个或一项元素）的、不可变的集合对象，此处的集合既可以是 List，还可以是 Map。
- unmodifiableXxx()：返回指定集合对象的不可变视图，此处的集合既可以是 List，也可以是 Set、SortedSet，还可以是 Map、SorteMap 等。

上面三类方法的参数是原有的集合对象，返回值是该集合的“只读”版本。通过 Collections 提供的三类方法，可以生成“只读”的 Collection 或 Map。看下面程序。

程序清单：codes\08\8.8\UnmodifiableTest.java

```
public class UnmodifiableTest
{
    public static void main(String[] args)
    {
        // 创建一个空的、不可改变的 List 对象
        List unmodifiableList = Collections.emptyList();
        // 创建一个只有一个元素，且不可改变的 Set 对象
        Set unmodifiableSet = Collections.singleton("疯狂 Java 讲义");
        // 创建一个普通的 Map 对象
        Map scores = new HashMap();
        scores.put("语文" , 80);
        scores.put("Java" , 82);
        // 返回普通的 Map 对象对应的不可变版本
        Map unmodifiableMap = Collections.unmodifiableMap(scores);
        // 下面任意一行代码都将引发 UnsupportedOperationException 异常
        unmodifiableList.add("测试元素");   // ①
        unmodifiableSet.add("测试元素");    // ②
        unmodifiableMap.put("语文" , 90);   // ③
    }
}
```

上面程序的三行粗体字代码分别定义了一个空的、不可变的 List 对象，一个只包含一个元素的、不可变的 Set 对象和一个不可变的 Map 对象。不可变的集合对象只能访问集合元素，不可修改集合元素。所以上面程序中①②③处的代码都将引发 UnsupportedOperationException 异常。

8.8.5 Java 9 新增的不可变集合

Java 9 终于增加这个功能了——以前假如要创建一个包含 6 个元素的 Set 集合，程序需要先创建 Set 集合，然后调用 6 次 add()方法向 Set 集合中添加元素。Java 9 对此进行了简化，程序直接调用 Set、List、Map 的 of()方法即可创建包含 *N* 个元素的不可变集合，这样一行代码就可创建包含 *N* 个元素的集合。

不可变意味着程序不能向集合中添加元素，也不能从集合中删除元素。

如下程序示范了如何创建不可变集合。

程序清单：codes\08\8.8\Java9Collection.java

```
public class Java9Collection
{
    public static void main(String[] args)
    {
        // 创建包含 4 个元素的 Set 集合
        Set set = Set.of("Java", "Kotlin", "Go", "Swift");
        System.out.println(set);
        // 不可变集合，下面代码导致运行时错误
//      set.add("Ruby");
        // 创建包含 4 个元素的 List 集合
        List list = List.of(34, -25, 67, 231);
        System.out.println(list);
        // 不可变集合，下面代码导致运行时错误
//      list.remove(1);
        // 创建包含 3 个 key-value 对的 Map 集合
        Map map = Map.of("语文", 89, "数学", 82, "英语", 92);
        System.out.println(map);
```

```
            // 不可变集合，下面代码导致运行时错误
//          map.remove("语文");
            // 使用Map.entry()方法显式构建key-value对
            Map map2 = Map.ofEntries(Map.entry("语文", 89),
                  Map.entry("数学", 82),
                  Map.entry("英语", 92));
            System.out.println(map2);
        }
    }
```

上面粗体字代码示范了如何使用集合元素创建不可变集合，其中 Set、List 比较简单，程序只要为它们的 of()方法传入 *N* 个集合元素即可创建 Set、List 集合。

从上面粗体字代码可以看出，创建不可变的 Map 集合有两个方法：使用 of()方法时只要依次传入多个 key-value 对即可；还可使用 ofEntries()方法，该方法可接受多个 Entry 对象，因此程序显式使用 Map.entry()方法来创建 Map.Entry 对象。

8.9 烦琐的接口：Enumeration

Enumeration 接口是 Iterator 迭代器的“古老版本”，从 JDK 1.0 开始，Enumeration 接口就已经存在了（Iterator 从 JDK 1.2 才出现）。Enumeration 接口只有两个名字很长的方法。

- boolean hasMoreElements( )：如果此迭代器还有剩下的元素，则返回 true。
- Object nextElement( )：返回该迭代器的下一个元素，如果还有的话（否则抛出异常）。

通过这两个方法不难发现，Enumeration 接口中的方法名称冗长，难以记忆，而且没有提供 Iterator 的 remove()方法。如果现在编写 Java 程序，应该尽量采用 Iterator 迭代器，而不是用 Enumeration 迭代器。

Java 之所以保留 Enumeration 接口，主要是为了照顾以前那些“古老”的程序，那些程序里大量使用了 Enumeration 接口，如果新版本的 Java 里直接删除 Enumeration 接口，将会导致那些程序全部出错。在计算机行业有一条规则：加入任何规则都必须慎之又慎，因为以后无法删除规则。

实际上，前面介绍的 Vector（包括其子类 Stack）、Hashtable 两个集合类，以及另一个极少使用的 BitSet，都是从 JDK 1.0 遗留下来的集合类，而 Enumeration 接口可用于遍历这些“古老”的集合类。对于 ArrayList、HashMap 等集合类，不再支持使用 Enumeration 迭代器。

下面程序示范了如何通过 Enumeration 接口来迭代 Vector 和 Hashtable。

程序清单：codes\08\8.9\EnumerationTest.java

```
public class EnumerationTest
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        v.add("疯狂Java讲义");
        v.add("轻量级Java EE企业应用实战");
        Hashtable scores = new Hashtable();
        scores.put("语文" , 78);
        scores.put("数学" , 88);
        Enumeration em = v.elements();
        while (em.hasMoreElements())
        {
            System.out.println(em.nextElement());
        }
        Enumeration keyEm = scores.keys();
        while (keyEm.hasMoreElements())
        {
            Object key = keyEm.nextElement();
            System.out.println(key + "--->" + scores.get(key));
        }
    }
}
```

上面程序使用 Enumeration 迭代器来遍历 Vector 和 Hashtable 集合里的元素，其工作方式与 Iterator 迭代器的工作方式基本相似。但使用 Enumeration 迭代器时方法名更加冗长，而且 Enumeration 迭代器

只能遍历 Vector、Hashtable 这种古老的集合，因此通常不要使用它。除非在某些极端情况下，不得不使用 Enumeration，否则都应该选择 Iterator 迭代器。

8.10 本章小结

本章详细介绍了 Java 集合框架的相关知识。本章从 Java 的集合框架体系开始讲起，概述了 Java 集合框架的 4 个主要体系：Set、List、Queue 和 Map，并简述了集合在编程中的重要性。本章详细介绍了 Java 8 对集合框架的改进，包括使用 Lambda 表达式简化集合编程，以及集合的 Stream 编程等。本章细致地讲述了 Set、List、Queue、Map 接口及各实现类的详细用法，并深入分析了各种实现类实现机制的差异，并给出了选择集合实现类时的原则。本章从原理上剖析了 Map 结构特征，以及 Map 结构和 Set、List 之间的区别及联系。本章最后通过梭哈游戏示范了 Collections 工具类的基本用法。

本章练习

1. 创建一个 Set 集合，并用 Set 集合保存用户通过控制台输入的 20 个字符串。

2. 创建一个 List 集合，并随意添加 10 个元素。然后获取索引为 5 处的元素；再获取其中某 2 个元素的索引；再删除索引为 3 处的元素。

3. 给定["a", "b" , "a" , "b", "c" , "a", "b" , "c" , "b"]字符串数组，然后使用 Map 的 key 来保存数组中字符串元素，value 保存该字符串元素的出现次数，最后统计出各字符串元素的出现次数。

4. 将本章未完成的梭哈游戏补充完整，不断地添加梭哈规则，开发一个控制台的梭哈游戏。

CHAPTER

9

第9章 泛型

本章要点

- 编译时类型检查的重要性
- 使用泛型实现编译时进行类型检查
- 定义泛型接口、泛型类
- 派生泛型接口、泛型类的子类、实现类
- 使用类型通配符
- 设定类型通配符的上限
- 设定类型通配符的下限
- 设定泛型形参的上限
- 在方法签名中定义泛型
- 泛型方法和类型通配符的区别与联系
- 泛型方法与方法重载
- Java 8 改进的类型推断
- 擦除与转换
- 泛型与数组

本章的知识可以与前一章的内容补充阅读，因为 Java 5 增加泛型支持在很大程度上都是为了让集合能记住其元素的数据类型。在没有泛型之前，一旦把一个对象“丢进”Java 集合中，集合就会忘记对象的类型，把所有的对象当成 Object 类型处理。当程序从集合中取出对象后，就需要进行强制类型转换，这种强制类型转换不仅使代码臃肿，而且容易引起 ClassCastExeception 异常。

增加了泛型支持后的集合，完全可以记住集合中元素的类型，并可以在编译时检查集合中元素的类型，如果试图向集合中添加不满足类型要求的对象，编译器就会提示错误。增加泛型后的集合，可以让代码更加简洁，程序更加健壮（Java 泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生 ClassCastException 异常）。除此之外，Java 泛型还增强了枚举类、反射等方面的功能，泛型在反射中的用法，将在第 18 章中介绍。

本章不仅会介绍如何通过泛型来实现编译时检查集合元素的类型，而且会深入介绍 Java 泛型的详细用法，包括定义泛型类、泛型接口，以及类型通配符、泛型方法等知识。

9.1 泛型入门

Java 集合有个缺点——把一个对象“丢进”集合里之后，集合就会“忘记”这个对象的数据类型，当再次取出该对象时，该对象的编译类型就变成了 Object 类型（其运行时类型没变）。

Java 集合之所以被设计成这样，是因为集合的设计者不知道我们会用集合来保存什么类型的对象，所以他们把集合设计成能保存任何类型的对象，只要求具有很好的通用性。但这样做带来如下两个问题：

- 集合对元素类型没有任何限制，这样可能引发一些问题。例如，想创建一个只能保存 Dog 对象的集合，但程序也可以轻易地将 Cat 对象“丢”进去，所以可能引发异常。
- 由于把对象“丢进”集合时，集合丢失了对象的状态信息，集合只知道它盛装的是 Object，因此取出集合元素后通常还需要进行强制类型转换。这种强制类型转换既增加了编程的复杂度，也可能引发 ClassCastException 异常。

下面将深入介绍编译时不检查类型可能引发的异常，以及如何做到在编译时进行类型检查。

9.1.1 编译时不检查类型的异常

下面程序将会看到编译时不检查类型所导致的异常。

程序清单：codes\09\9.1\ListErr.java

```
public class ListErr
{
    public static void main(String[] args)
    {
        // 创建一个只想保存字符串的 List 集合
        List strList = new ArrayList();
        strList.add("疯狂 Java 讲义");
        strList.add("疯狂 Android 讲义");
        //“不小心”把一个 Integer 对象“丢进”了集合
        strList.add(5);     // ①
        strList.forEach(str -> System.out.println(((String)str).length())); // ②
    }
}
```

上面程序创建了一个 List 集合，而且只希望该 List 集合保存字符串对象——但程序不能进行任何限制，如果程序在①处“不小心”把一个 Integer 对象“丢进”了 List 集合中，这将导致程序在②处引发 ClassCastException 异常，因为程序试图把一个 Integer 对象转换为 String 类型。

9.1.2 使用泛型

从 Java 5 以后，Java 引入了“参数化类型（parameterized type）”的概念，允许程序在创建集合时指定集合元素的类型，正如在第 8 章的 ShowHand.java 程序中见到的 List<String>，这表明该 List 只能保存字符串类型的对象。Java 的参数化类型被称为泛型（Generic）。

对于前面的 ListErr.java 程序，可以使用泛型改进这个程序。

程序清单：codes\09\9.1\GenericList.java

```
public class GenericList
{
    public static void main(String[] args)
    {
        // 创建一个只想保存字符串的 List 集合
        List<String> strList = new ArrayList<String>(); // ①
        strList.add("疯狂 Java 讲义");
        strList.add("疯狂 Android 讲义");
        // 下面代码将引起编译错误
        strList.add(5);    // ②
        strList.forEach(str -> System.out.println(str.length())); // ③
    }
}
```

上面程序成功创建了一个特殊的 List 集合：strList，这个 List 集合只能保存字符串对象，不能保存其他类型的对象。创建这种特殊集合的方法是：在集合接口、类后增加尖括号，尖括号里放一个数据类型，即表明这个集合接口、集合类只能保存特定类型的对象。注意①处的类型声明，它指定 strList 不是一个任意的 List，而是一个 String 类型的 List，写作：List<String>。可以称 List 是带一个类型参数的泛型接口，在本例中，类型参数是 String。在创建这个 ArrayList 对象时也指定了一个类型参数。

上面程序将在②处引发编译异常，因为 strList 集合只能添加 String 对象，所以不能将 Integer 对象“丢进”该集合。

而且程序在③处不需要进行强制类型转换，因为 strList 对象可以“记住”它的所有集合元素都是 String 类型。

上面代码不仅更加健壮，程序再也不能“不小心”地把其他对象“丢进”strList 集合中；而且程序更加简洁，集合自动记住所有集合元素的数据类型，从而无须对集合元素进行强制类型转换。这一切，都是因为 Java 5 提供的泛型支持。

9.1.3 Java 9 增强的“菱形”语法

在 Java 7 以前，如果使用带泛型的接口、类定义变量，那么调用构造器创建对象时构造器的后面也必须带泛型，这显得有些多余了。例如如下两条语句：

```
List<String> strList = new ArrayList<String>();
Map<String , Integer> scores = new HashMap<String , Integer>();
```

上面两条语句中的粗体字代码部分完全是多余的，在 Java 7 以前这是必需的，不能省略。从 Java 7 开始，Java 允许在构造器后不需要带完整的泛型信息，只要给出一对尖括号（<>）即可，Java 可以推断尖括号里应该是什么泛型信息。即上面两条语句可以改写为如下形式：

```
List<String> strList = new ArrayList<>();
Map<String , Integer> scores = new HashMap<>();
```

把两个尖括号并排放在一起非常像一个菱形，这种语法也就被称为“菱形”语法。下面程序示范了 Java 7 的菱形语法。

程序清单：codes\09\9.1\DiamondTest.java

```
public class DiamondTest
{
    public static void main(String[] args)
    {
        // Java 自动推断出 ArrayList 的<>里应该是 String
        List<String> books = new ArrayList<>();
        books.add("疯狂 Java 讲义");
        books.add("疯狂 Android 讲义");
        // 遍历 books 集合，集合元素就是 String 类型
        books.forEach(ele -> System.out.println(ele.length()));
        // Java 自动推断出 HashMap 的<>里应该是 String , List<String>
        Map<String , List<String>> schoolsInfo = new HashMap<>();
```

```
        // Java 自动推断出 ArrayList 的<>里应该是 String
        List<String> schools = new ArrayList<>();
        schools.add("斜月三星洞");
        schools.add("西天取经路");
        schoolsInfo.put("孙悟空" , schools);
        // 遍历 Map 时，Map 的 key 是 String 类型，value 是 List<String>类型
        schoolsInfo.forEach((key , value) -> System.out.println(key + "-->" + value));
    }
}
```

上面程序中三行粗体字代码就是“菱形”语法的示例。从该程序不难看出，“菱形”语法对原有的泛型并没有改变，只是更好地简化了泛型编程。

Java 9 再次增强了“菱形”语法，它甚至允许在创建匿名内部类时使用菱形语法，Java 可根据上下文来推断匿名内部类中泛型的类型。下面程序示范了在匿名内部类中使用菱形语法。

程序清单：codes\09\9.1\AnnoymousDiamond.java

```
interface Foo<T>
{
    void test(T t);
}
public class AnnoymousTest
{
    public static void main(String[] args)
    {
        // 指定 Foo 类中泛型为 String
        Foo<String> f = new Foo<>()
        {
            // test()方法的参数类型为 String
            public void test(String t)
            {
                System.out.println("test 方法的 t 参数为：" + t);
            }
        };
        // 使用泛型通配符，此时相当于通配符的上限为 Object
        Foo<?> fo = new Foo<>()
        {
            // test()方法的参数类型为 Object
            public void test(Object t)
            {
                System.out.println("test 方法的 Object 参数为：" + t);
            }
        };
        // 使用泛型通配符，通配符的上限为 Number
        Foo<? extends Number> fn = new Foo<>()
        {
            // 此时 test()方法的参数类型为 Number
            public void test(Number t)
            {
                System.out.println("test 方法的 Number 参数为：" + t);
            }
        };
    }
}
```

上面程序先定义了一个带泛型声明的接口，接下来三行粗体字代码分别示范了在匿名内部类中使用菱形语法。第一行粗体字代码声明变量时明确地将泛型指定为 String 类型，因此在该匿名内部类中 T 类型就代表了 String 类型；第二行粗体字代码声明变量时使用通配符来代表泛型（相当于通配符的上限为 Object），因此系统只能推断出 T 代表 Object，所以在该匿名内部类中 T 类型就代表了 Object 类型；第三行粗体字代码声明变量时使用了带上限（上限是 Number）的通配符，因此系统可以推断出 T 代表 Number 类。

无论哪种方式，Java 9 都允许在使用匿名内部类时使用菱形语法。

9.2 深入泛型

所谓泛型，就是允许在定义类、接口、方法时使用类型形参，这个类型形参（或叫泛型）将在声明变量、创建对象、调用方法时动态地指定（即传入实际的类型参数，也可称为类型实参）。Java 5 改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参，这就是在前面程序中看到的 List<String>和 ArrayList<String>两种类型。

9.2.1 定义泛型接口、类

下面是 Java 5 改写后 List 接口、Iterator 接口、Map 的代码片段。

```
// 定义接口时指定了一个泛型形参，该形参名为 E
public interface List<E>
{
    // 在该接口里，E 可作为类型使用
    // 下面方法可以使用 E 作为参数类型
    void add(E x);
    Iterator<E> iterator();    // ①
    ...
}
// 定义接口时指定了一个泛型形参，该形参名为 E
public interface Iterator<E>
{
    // 在该接口里 E 完全可以作为类型使用
    E next();
    boolean hasNext();
    ...
}
// 定义该接口时指定了两个泛型形参，其形参名为 K、V
public interface Map<K , V>
{
    // 在该接口里 K、V 完全可以作为类型使用
    Set<K> keySet()    // ②
    V put(K key, V value)
    ...
}
```

上面三个接口声明是比较简单的，除了尖括号中的内容——这就是泛型的实质：允许在定义接口、类时声明泛型形参，泛型形参在整个接口、类体内可当成类型使用，几乎所有可使用普通类型的地方都可以使用这种泛型形参。

除此之外，①②处方法声明返回值类型是 Iterator<E>、Set<K>，这表明 Set<K>形式是一种特殊的数据类型，是一种与 Set 不同的数据类型——可以认为是 Set 类型的子类。

例如使用 List 类型时，如果为 E 形参传入 String 类型实参，则产生了一个新的类型：List<String>类型，可以把 List<String>想象成 E 被全部替换成 String 的特殊 List 子接口。

```
// List<String>等同于如下接口
public interface ListString extends List
{
    // 原来的 E 形参全部变成 String 类型实参
    void add(String x);
    Iterator<String> iterator();
    ...
}
```

通过这种方式，就解决了 9.1.2 节中的问题——虽然程序只定义了一个 List<E>接口，但实际使用时可以产生无数多个 List 接口，只要为 E 传入不同的类型实参，系统就会多出一个新的 List 子接口。必须指出：List<String>绝不会被替换成 ListString，系统没有进行源代码复制，二进制代码中没有，磁盘中没有，内存中也没有。

注意：

包含泛型声明的类型可以在定义变量、创建对象时传入一个类型实参，从而可以动态地生成无数多个逻辑上的子类，但这种子类在物理上并不存在。

可以为任何类、接口增加泛型声明（并不是只有集合类才可以使用泛型声明，虽然集合类是泛型的重要使用场所）。下面自定义一个 Apple 类，这个 Apple 类就可以包含一个泛型声明。

程序清单：codes\09\9.2\Apple.java

```
// 定义 Apple 类时使用了泛型声明
public class Apple<T>
{
    // 使用 T 类型定义实例变量
    private T info;
    public Apple(){}
    // 下面方法中使用 T 类型来定义构造器
    public Apple(T info)
    {
        this.info = info;
    }
    public void setInfo(T info)
    {
        this.info = info;
    }
    public T getInfo()
    {
        return this.info;
    }
    public static void main(String[] args)
    {
        // 由于传给 T 形参的是 String，所以构造器参数只能是 String
        Apple<String> a1 = new Apple<>("苹果");
        System.out.println(a1.getInfo());
        // 由于传给 T 形参的是 Double，所以构造器参数只能是 Double 或 double
        Apple<Double> a2 = new Apple<>(5.67);
        System.out.println(a2.getInfo());
    }
}
```

上面程序定义了一个带泛型声明的 Apple<T>类（不要理会这个泛型形参是否具有实际意义），使用 Apple<T>类时就可为 T 形参传入实际类型，这样就可以生成如 Apple<String>、Apple<Double>…形式的多个逻辑子类（物理上并不存在）。这就是 9.1 节可以使用 List<String>、ArrayList<String>等类型的原因——JDK 在定义 List、ArrayList 等接口、类时使用了泛型声明，所以在使用这些类时为之传入了实际的类型参数。

注意：

当创建带泛型声明的自定义类，为该类定义构造器时，构造器名还是原来的类名，不要增加泛型声明。例如，为 Apple<T>类定义构造器，其构造器名依然是 Apple，而不是 Apple<T>！调用该构造器时却可以使用 Apple<T>的形式，当然应该为 T 形参传入实际的类型参数。Java 7 提供了菱形语法，允许省略<>中的类型实参。

9.2.2 从泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或从该父类派生子类，需要指出的是，当使用这些接口、父类时不能再包含泛型形参。例如，下面代码就是错误的。

```
// 定义类 A 继承 Apple 类，Apple 类不能跟泛型形参
public class A extends Apple<T>{ }
```

方法中的形参代表变量、常量、表达式等数据，本书把它们直接称为形参，或者称为数据形参。定义方法时可以声明数据形参，调用方法（使用方法）时必须为这些数据形参传入实际的数据；与此类似的是，定义类、接口、方法时可以声明泛型形参，使用类、接口、方法时应该为泛型形参传入实际的类型。

如果想从 Apple 类派生一个子类，则可以改为如下代码：

```
// 使用 Apple 类时为 T 形参传入 String 类型
public class A extends Apple<String>
```

调用方法时必须为所有的数据形参传入参数值，与调用方法不同的是，使用类、接口时也可以不为泛型形参传入实际的类型参数，即下面代码也是正确的。

```
// 使用 Apple 类时，没有为 T 形参传入实际的类型参数
public class A extends Apple
```

像这种使用 Apple 类时省略泛型的形式被称为原始类型（raw type）。

如果从 Apple<String>类派生子类，则在 Apple 类中所有使用 T 类型的地方都将被替换成 String 类型，即它的子类将会继承到 String getInfo()和 void setInfo(String info)两个方法，如果子类需要重写父类的方法，就必须注意这一点。下面程序示范了这一点。

程序清单：codes\09\9.2\A1.java

```
public class A1 extends Apple<String>
{
    // 正确重写了父类的方法，返回值
    // 与父类 Apple<String>的返回值完全相同
    public String getInfo()
    {
        return "子类" + super.getInfo();
    }
    /*
    // 下面方法是错误的，重写父类方法时返回值类型不一致
    public Object getInfo()
    {
        return "子类";
    }
    */
}
```

如果使用 Apple 类时没有传入实际的类型（即使用原始类型），Java 编译器可能发出警告：使用了未经检查或不安全的操作——这就是泛型检查的警告，读者在前一章中应该多次看到这样的警告。如果希望看到该警告提示的更详细信息，则可以通过为 javac 命令增加-Xlint:unchecked 选项来实现。此时，系统会把 Apple<T>类里的 T 形参当成 Object 类型处理。如下程序所示。

程序清单：codes\09\9.2\A2.java

```
public class A2 extends Apple
{
    // 重写父类的方法
    public String getInfo()
    {
        // super.getInfo()方法返回值是 Object 类型
        // 所以加 toString()才返回 String 类型
        return super.getInfo().toString();
    }
}
```

上面程序都是从带泛型声明的父类来派生子类，创建带泛型声明的接口的实现类与此几乎完全一样，此处不再赘述。

9.2.3 并不存在泛型类

前面提到可以把 ArrayList<String>类当成 ArrayList 的子类，事实上，ArrayList<String>类也确实像

一种特殊的 ArrayList 类：该 ArrayList<String>对象只能添加 String 对象作为集合元素。但实际上，系统并没有为 ArrayList<String>生成新的 class 文件，而且也不会把 ArrayList<String>当成新类来处理。

看下面代码的打印结果是什么？

```
// 分别创建 List<String>对象和 List<Integer>对象
List<String> l1 = new ArrayList<>();
List<Integer> l2 = new ArrayList<>();
// 调用 getClass()方法来比较 l1 和 l2 的类是否相等
System.out.println(l1.getClass() == l2.getClass());
```

运行上面的代码片段，可能有读者认为应该输出 false，但实际输出 true。因为不管泛型的实际类型参数是什么，它们在运行时总有同样的类（class）。

不管为泛型形参传入哪一种类型实参，对于 Java 来说，它们依然被当成同一个类处理，在内存中也只占用一块内存空间，因此在静态方法、静态初始化块或者静态变量的声明和初始化中不允许使用泛型形参。下面程序演示了这种错误。

程序清单：codes\09\9.2\R.java

```
public class R<T>
{
    // 下面代码错误，不能在静态变量声明中使用泛型形参
    static T info;
    T age;
    public void foo(T msg){}
    // 下面代码错误，不能在静态方法声明中使用泛型形参
    public static void bar(T msg){}
}
```

由于系统中并不会真正生成泛型类，所以 instanceof 运算符后不能使用泛型类。例如，下面代码是错误的。

```
java.util.Collection<String> cs = new java.util.ArrayList<>();
// 下面代码编译时引起错误：instanceof 运算符后不能使用泛型
if (cs instanceof java.util.ArrayList<String>){...}
```

9.3 类型通配符

正如前面讲的，当使用一个泛型类时（包括声明变量和创建对象两种情况），都应该为这个泛型类传入一个类型实参。如果没有传入类型实际参数，编译器就会提出泛型警告。假设现在需要定义一个方法，该方法里有一个集合形参，集合形参的元素类型是不确定的，那应该怎样定义呢？

考虑如下代码：

```
public void test(List c)
{
    for (int i = 0; i < c.size(); i++)
    {
        System.out.println(c.get(i));
    }
}
```

上面程序当然没有问题：这是一段最普通的遍历 List 集合的代码。问题是上面程序中 List 是一个有泛型声明的接口，此处使用 List 接口时没有传入实际类型参数，这将引起泛型警告。为此，考虑为 List 接口传入实际的类型参数——因为 List 集合里的元素类型是不确定的，将上面方法改为如下形式：

```
public void test(List<Object> c)
{
    for (int i = 0; i < c.size(); i++)
    {
        System.out.println(c.get(i));
    }
}
```

表面上看起来，上面方法声明没有问题，这个方法声明确实没有任何问题。问题是调用该方法传入的实际参数值时可能不是我们所期望的，例如，下面代码试图调用该方法。

```
// 创建一个 List<String>对象
List<String> strList = new ArrayList<>();
// 将 strList 作为参数来调用前面的 test 方法
test(strList);    //①
```

编译上面程序，将在①处发生如下编译错误：

```
无法将 Test 中的 test(java.util.List<java.lang.Object>)
应用于 (java.util.List<java.lang.String>)
```

上面程序出现了编译错误，这表明 List<String>对象不能被当成 List<Object>对象使用，也就是说，List<String>类并不是 List<Object>类的子类。

注意：

如果 Foo 是 Bar 的一个子类型(子类或者子接口)，而 G 是具有泛型声明的类或接口，G<Foo>并不是 G<Bar>的子类型！这一点非常值得注意，因为它与大部分人的习惯认为是不同的。

与数组进行对比，先看一下数组是如何工作的。在数组中，程序可以直接把一个 Integer[]数组赋给一个 Number[]变量。如果试图把一个 Double 对象保存到该 Number[]数组中，编译可以通过，但在运行时抛出 ArrayStoreException 异常。例如如下程序。

程序清单：codes\09\9.3\ArrayErr.java

```
public class ArrayErr
{
    public static void main(String[] args)
    {
        // 定义一个 Integer 数组
        Integer[] ia = new Integer[5];
        // 可以把一个 Integer[]数组赋给 Number[]变量
        Number[] na = ia;
        // 下面代码编译正常，但运行时会引发 ArrayStoreException 异常
        // 因为 0.5 并不是 Integer
        na[0] = 0.5;   // ①
    }
}
```

上面程序在①号粗体字代码处会引发 ArrayStoreException 运行时异常，这就是一种潜在的风险。

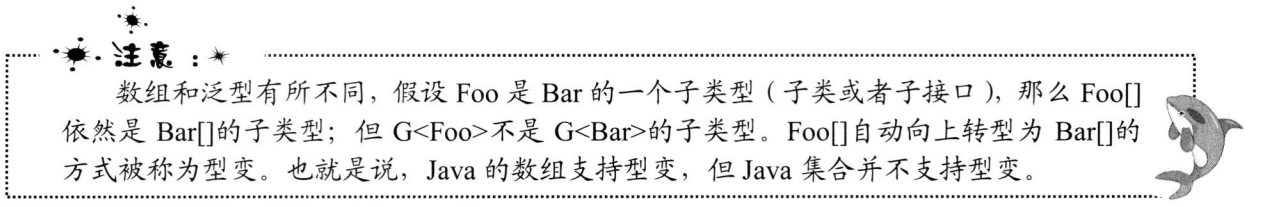
提示：

一门设计优秀的语言，不仅需要提供强大的功能，而且能提供强大的“错误提示”和“出错警告”，这样才能尽量避免开发者犯错。而 Java 允许 Integer[]数组赋值给 Number[]变量显然不是一种安全的设计。

在 Java 的早期设计中，允许 Integer[]数组赋值给 Number[]变量存在缺陷，因此 Java 在泛型设计时进行了改进，它不再允许把 List<Integer>对象赋值给 List<Number>变量。例如，如下代码将会导致编译错误（程序清单同上）。

```
List<Integer> iList = new ArrayList<>();
// 下面代码导致编译错误
List<Number> nList = iList;
```

Java 泛型的设计原则是，只要代码在编译时没有出现警告，就不会遇到运行时 ClassCastException 异常。

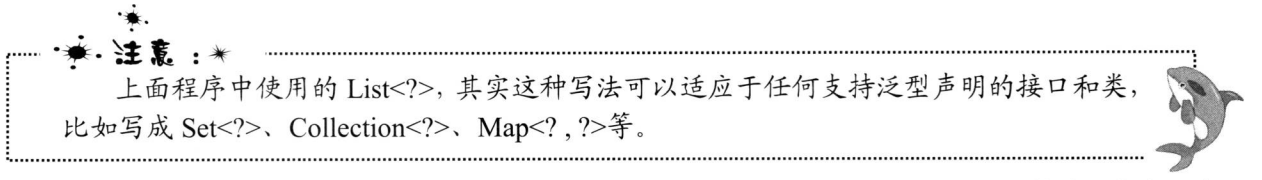


9.3.1 使用类型通配符

为了表示各种泛型 List 的父类，可以使用类型通配符，类型通配符是一个问号（?），将一个问号作为类型实参传给 List 集合，写作：List<?>（意思是元素类型未知的 List）。这个问号（?）被称为通配符，它的元素类型可以匹配任何类型。可以将上面方法改写为如下形式：

```
public void test(List<?> c)
{
    for (int i = 0; i < c.size(); i++)
    {
        System.out.println(c.get(i));
    }
}
```

现在使用任何类型的 List 来调用它，程序依然可以访问集合 c 中的元素，其类型是 Object，这永远是安全的，因为不管 List 的真实类型是什么，它包含的都是 Object。



但这种带通配符的 List 仅表示它是各种泛型 List 的父类，并不能把元素加入到其中。例如，如下代码将会引起编译错误。

```
List<?> c = new ArrayList<String>();
// 下面程序引起编译错误
c.add(new Object());
```

因为程序无法确定 c 集合中元素的类型，所以不能向其中添加对象。根据前面的 List<E>接口定义的代码可以发现：add()方法有类型参数 E 作为集合的元素类型，所以传给 add 的参数必须是 E 类的对象或者其子类的对象。但因为在该例中不知道 E 是什么类型，所以程序无法将任何对象“丢进”该集合。唯一的例外是 null，它是所有引用类型的实例。

另一方面，程序可以调用 get()方法来返回 List<?>集合指定索引处的元素，其返回值是一个未知类型，但可以肯定的是，它总是一个 Object。因此，把 get()的返回值赋值给一个 Object 类型的变量，或者放在任何希望是 Object 类型的地方都可以。

9.3.2 设定类型通配符的上限

当直接使用 List<?>这种形式时，即表明这个 List 集合可以是任何泛型 List 的父类。但还有一种特殊的情形，程序不希望这个 List<?>是任何泛型 List 的父类，只希望它代表某一类泛型 List 的父类。考虑一个简单的绘图程序，下面先定义三个形状类。

程序清单：codes\09\9.3\Shape.java

```
// 定义一个抽象类 Shape
public abstract class Shape
{
    public abstract void draw(Canvas c);
}
```

程序清单：codes\09\9.3\Circle.java

```
// 定义 Shape 的子类 Circle
public class Circle extends Shape
```

```
{
    // 实现画图方法，以打印字符串来模拟画图方法实现
    public void draw(Canvas c)
    {
        System.out.println("在画布" + c + "上画一个圆");
    }
}
```

程序清单：codes\09\9.3\Rectangle.java

```
// 定义 Shape 的子类 Rectangle
public class Rectangle extends Shape
{
    // 实现画图方法，以打印字符串来模拟画图方法实现
    public void draw(Canvas c)
    {
        System.out.println("把一个矩形画在画布" + c + "上");
    }
}
```

上面定义了三个形状类，其中 Shape 是一个抽象父类，该抽象父类有两个子类：Circle 和 Rectangle。接下来定义一个 Canvas 类，该画布类可以画数量不等的形状（Shape 子类的对象），那应该如何定义这个 Canvas 类呢？考虑如下的 Canvas 实现类。

程序清单：codes\09\9.3\Canvas.java

```
public class Canvas
{
    // 同时在画布上绘制多个形状
    public void drawAll(List<Shape> shapes)
    {
        for (Shape s : shapes)
        {
            s.draw(this);
        }
    }
}
```

注意上面的 drawAll()方法的形参类型是 List<Shape>，而 List<Circle>并不是 List<Shape>的子类型，因此，下面代码将引起编译错误。

```
List<Circle> circleList = new ArrayList<>();
Canvas c = new Canvas();
// 不能把 List<Circle>当成 List<Shape>使用，所以下面代码引起编译错误
c.drawAll(circleList);
```

关键在于 List<Circle>并不是 List<Shape>的子类型，所以不能把 List<Circle>对象当成 List<Shape>使用。为了表示 List<Circle>的父类，可以考虑使用 List<?>，但此时从 List<?>集合中取出的元素只能被编译器当成 Object 处理。为了表示 List 集合的所有元素是 Shape 的子类，Java 泛型提供了被限制的泛型通配符。被限制的泛型通配符表示如下：

```
// 它表示泛型形参必须是 Shape 子类的 List
List<? extends Shape>
```

有了这种被限制的泛型通配符，就可以把上面的 Canvas 程序改为如下形式（程序清单同上）：

```
public class Canvas
{
    // 同时在画布上绘制多个形状，使用被限制的泛型通配符
    public void drawAll(List<? extends Shape> shapes)
    {
        for (Shape s : shapes)
        {
            s.draw(this);
        }
    }
}
```

将 Canvas 改为如上形式，就可以把 List<Circle>对象当成 List<? extends Shape>使用。即 List<?

extends Shape>可以表示 List<Circle>、List<Rectangle>的父类——只要 List 后尖括号里的类型是 Shape 的子类型即可。

List<? extends Shape>是受限制通配符的例子，此处的问号（?）代表一个未知的类型，就像前面看到的通配符一样。但是此处的这个未知类型一定是 Shape 的子类型（也可以是 Shape 本身），因此可以把 Shape 称为这个通配符的上限（upper bound）。

类似地，由于程序无法确定这个受限制的通配符的具体类型，所以不能把 Shape 对象或其子类的对象加入这个泛型集合中。例如，下面代码就是错误的。

```
public void addRectangle(List<? extends Shape> shapes)
{
    // 下面代码引起编译错误
    shapes.add(0, new Rectangle());
}
```

与使用普通通配符相似的是，shapes.add()的第二个参数类型是? extends Shape，它表示 Shape 未知的子类，程序无法确定这个类型是什么，所以无法将任何对象添加到这种集合中。

简而言之，这种指定通配符上限的集合，只能从集合中取元素（取出的元素总是上限的类型），不能向集合中添加元素（因为编译器没法确定集合元素实际是哪种子类型）。

对于更广泛的泛型类来说，指定通配符上限就是为了支持类型型变。比如 Foo 是 Bar 的子类，这样 A<Bar>就相当于 A<? extends Foo>的子类，可以将 A<Bar>赋值给 A<? extends Foo>类型的变量，这种型变方式被称为协变。

对于协变的泛型类来说，它只能调用泛型类型作为返回值类型的方法（编译器会将该方法返回值当成通配符上限的类型）；而不能调用泛型类型作为参数的方法。口诀是：协变只出不进！

提示：

对于指定通配符上限的泛型类，相当于通配符上限是 Object。

9.3.3 设定类型通配符的下限

除可以指定通配符的上限之外，Java 也允许指定通配符的下限，通配符的下限用<? super 类型>的方式来指定，通配符下限的作用与通配符上限的作用恰好相反。

指定通配符的下限就是为了支持类型型变。比如 Foo 是 Bar 的子类，当程序需要一个 A<? super Bar>变量时，程序可以将 A<Foo>、A<Object>赋值给 A<? super Bar>类型的变量，这种型变方式被称为逆变。

对于逆变的泛型集合来说，编译器只知道集合元素是下限的父类型，但具体是哪种父类型则不确定。因此，这种逆变的泛型集合能向其中添加元素（因为实际赋值的集合元素总是逆变声明的父类），从集合中取元素时只能被当成 Object 类型处理（编译器无法确定取出的到底是哪个父类的对象）。

假设自己实现一个工具方法：实现将 src 集合中的元素复制到 dest 集合的功能，因为 dest 集合可以保存 src 集合中的所有元素，所以 dest 集合元素的类型应该是 src 集合元素类型的父类。

对于上面的 copy()方法，可以这样理解两个集合参数之间的依赖关系：不管 src 集合元素的类型是什么，只要 dest 集合元素的类型与前者相同或者是前者的父类即可，此时通配符的下限就有了用武之地。下面程序采用通配符下限的方式来实现该 copy()方法。

程序清单：codes\09\9.3\MyUtils.java

```
public class MyUtils
{
    // 下面 dest 集合元素的类型必须与 src 集合元素的类型相同，或者是其父类
    public static <T> T copy(Collection<? super T> dest
        , Collection<T> src)
    {
        T last = null;
        for (T ele  : src)
        {
            last = ele;
            // 逆变的泛型集合添加元素是安全的
```

```
            dest.add(ele);
        }
        return last;
    }
    public static void main(String[] args)
    {
        List<Number> ln = new ArrayList<>();
        List<Integer> li = new ArrayList<>();
        li.add(5);
        // 此处可准确地知道最后一个被复制的元素是 Integer 类型
        // 与 src 集合元素的类型相同
        Integer last = copy(ln , li);    // ①
        System.out.println(ln);
    }
}
```

使用这种语句，就可以保证程序的①处调用后推断出最后一个被复制的元素类型是 Integer，而不是笼统的 Number 类型。

提示：

上面方法用到了泛型方法的语法，就是在方法修饰符和返回值类型之间用<>定义泛型形参。关于泛型方法更详细介绍可参考下一节。

实际上，Java 集合框架中的 TreeSet<E>有一个构造器也用到了这种设定通配符下限的语法，如下所示。

```
// 下面的 E 是定义 TreeSet 类时的泛型形参
TreeSet(Comparator<? super E> c)
```

正如前一章所介绍的，TreeSet 会对集合中的元素按自然顺序或定制顺序进行排序。如果需要 TreeSet 对集合中的所有元素进行定制排序，则要求 TreeSet 对象有一个与之关联的 Comparator 对象。上面构造器中的参数 c 就是进行定制排序的 Comparator 对象。

Comparator 接口也是一个带泛型声明的接口：

```
public interface Comparator<T>
{
    int compare(T fst, T snd);
}
```

通过这种带下限的通配符的语法，可以在创建 TreeSet 对象时灵活地选择合适的 Comparator。假定需要创建一个 TreeSet<String>集合，并传入一个可以比较 String 大小的 Comparator，这个 Comparator 既可以是 Comparator<String>，也可以是 Comparator<Object>——只要尖括号里传入的类型是 String 的父类型（或它本身）即可。

程序清单：codes\09\9.4\TreeSetTest.java

```
public class TreeSetTest
{
    public static void main(String[] args)
    {
        // Comparator 的实际类型是 TreeSet 的元素类型的父类，满足要求
        TreeSet<String> ts1 = new TreeSet<>(
            new Comparator<Object>()
        {
            public int compare(Object fst, Object snd)
            {
                return hashCode() > snd.hashCode() ? 1
                    : hashCode() < snd.hashCode() ? -1 : 0;
            }
        });
        ts1.add("hello");
        ts1.add("wa");
        // Comparator 的实际类型是 TreeSet 元素的类型，满足要求
        TreeSet<String> ts2 = new TreeSet<>(
            new Comparator<String>()
        {
```

```
            public int compare(String first, String second)
            {
                return first.length() > second.length() ? -1
                    : first.length() < second.length() ? 1 : 0;
            }
        });
        ts2.add("hello");
        ts2.add("wa");
        System.out.println(ts1);
        System.out.println(ts2);
    }
}
```

通过使用这种通配符下限的方式来定义 TreeSet 构造器的参数，就可以将所有可用的 Comparator 作为参数传入，从而增加了程序的灵活性。当然，不仅 TreeSet 有这种用法，TreeMap 也有类似的用法，具体的请查阅 Java 的 API 文档。

9.3.4 设定泛型形参的上限

Java 泛型不仅允许在使用通配符形参时设定上限，而且可以在定义泛型形参时设定上限，用于表示传给该泛型形参的实际类型要么是该上限类型，要么是该上限类型的子类。下面程序示范了这种用法。

程序清单：codes\09\9.3\Apple.java

```
public class Apple<T extends Number>
{
    T col;
    public static void main(String[] args)
    {
        Apple<Integer> ai = new Apple<>();
        Apple<Double> ad = new Apple<>();
        // 下面代码将引发编译异常，下面代码试图把 String 类型传给 T 形参
        // 但 String 不是 Number 的子类型，所以引起编译错误
        Apple<String> as = new Apple<>();    // ①
    }
}
```

上面程序定义了一个 Apple 泛型类，该 Apple 类的泛型形参的上限是 Number 类，这表明使用 Apple 类时为 T 形参传入的实际类型参数只能是 Number 或 Number 类的子类。上面程序在①处将引起编译错误：类型 T 的上限是 Number 类型，而此处传入的实际类型是 String 类型，既不是 Number 类型，也不是 Number 类型的子类型，所以将会导致编译错误。

在一种更极端的情况下，程序需要为泛型形参设定多个上限（至多有一个父类上限，可以有多个接口上限），表明该泛型形参必须是其父类的子类（是父类本身也行），并且实现多个上限接口。如下代码所示。

```
// 表明 T 类型必须是 Number 类或其子类，并必须实现 java.io.Serializable 接口
public class Apple<T extends Number & java.io.Serializable>
{
    ...
}
```

与类同时继承父类、实现接口类似的是，为泛型形参指定多个上限时，所有的接口上限必须位于类上限之后。也就是说，如果需要为泛型形参指定类上限，类上限必须位于第一位。

9.4 泛型方法

前面介绍了在定义类、接口时可以使用泛型形参，在该类的方法定义和成员变量定义、接口的方法定义中，这些泛型形参可被当成普通类型来用。在另外一些情况下，定义类、接口时没有使用泛型形参，但定义方法时想自己定义泛型形参，这也是可以的，Java 5 还提供了对泛型方法的支持。

9.4.1 定义泛型方法

假设需要实现这样一个方法——该方法负责将一个 Object 数组的所有元素添加到一个 Collection 集合中。考虑采用如下代码来实现该方法。

```
static void fromArrayToCollection(Object[] a, Collection<Object> c)
{
    for (Object o : a)
    {
        c.add(o);
    }
}
```

上面定义的方法没有任何问题，关键在于方法中的 c 形参，它的数据类型是 Collection<Object>。正如前面所介绍的，Collection<String>不是 Collection<Object>的子类型——所以这个方法的功能非常有限，它只能将 Object[]数组的元素复制到元素为 Object（Object 的子类不行）的 Collection 集合中，即下面代码将引起编译错误。

```
String[] strArr = {"a" , "b"};
List<String> strList = new ArrayList<>();
// Collection<String>对象不能当成Collection<Object>使用，下面代码出现编译错误
fromArrayToCollection(strArr, strList);
```

可见上面方法的参数类型不可以使用 Collection<String>，那使用通配符 Collection<?>是否可行呢？显然也不行，因为 Java 不允许把对象放进一个未知类型的集合中。

为了解决这个问题，可以使用 Java 5 提供的泛型方法（Generic Method）。所谓泛型方法，就是在声明方法时定义一个或多个泛型形参。泛型方法的语法格式如下：

```
修饰符 <T , S> 返回值类型 方法名(形参列表)
{
    // 方法体...
}
```

把上面方法的格式和普通方法的格式进行对比，不难发现泛型方法的方法签名比普通方法的方法签名多了泛型形参声明，泛型形参声明以尖括号括起来，多个泛型形参之间以逗号（,）隔开，所有的泛型形参声明放在方法修饰符和方法返回值类型之间。

采用支持泛型的方法，就可以将上面的 fromArrayToCollection 方法改为如下形式：

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c)
{
    for (T o : a)
    {
        c.add(o);
    }
}
```

下面程序示范了完整的用法。

程序清单：codes\09\9.4\GenericMethodTest.java

```
public class GenericMethodTest
{
    // 声明一个泛型方法，该泛型方法中带一个 T 泛型形参
    static <T> void fromArrayToCollection(T[] a, Collection<T> c)
    {
        for (T o : a)
        {
            c.add(o);
        }
    }
    public static void main(String[] args)
    {
        Object[] oa = new Object[100];
        Collection<Object> co = new ArrayList<>();
        // 下面代码中 T 代表 Object 类型
        fromArrayToCollection(oa, co);
        String[] sa = new String[100];
        Collection<String> cs = new ArrayList<>();
        // 下面代码中 T 代表 String 类型
        fromArrayToCollection(sa, cs);
        // 下面代码中 T 代表 Object 类型
        fromArrayToCollection(sa, co);
        Integer[] ia = new Integer[100];
```

```
        Float[] fa = new Float[100];
        Number[] na = new Number[100];
        Collection<Number> cn = new ArrayList<>();
        // 下面代码中 T 代表 Number 类型
        fromArrayToCollection(ia, cn);
        // 下面代码中 T 代表 Number 类型
        fromArrayToCollection(fa, cn);
        // 下面代码中 T 代表 Number 类型
        fromArrayToCollection(na, cn);
        // 下面代码中 T 代表 Object 类型
        fromArrayToCollection(na, co);
        // 下面代码中 T 代表 String 类型，但 na 是一个 Number 数组
        // 因为 Number 既不是 String 类型
        // 也不是它的子类，所以出现编译错误
//        fromArrayToCollection(na, cs);
    }
}
```

上面程序定义了一个泛型方法，该泛型方法中定义了一个 T 泛型形参，这个 T 类型就可以在该方法内当成普通类型使用。与接口、类声明中定义的泛型不同的是，方法声明中定义的泛型只能在该方法里使用，而接口、类声明中定义的泛型则可以在整个接口、类中使用。

与类、接口中使用泛型参数不同的是，方法中的泛型参数无须显式传入实际类型参数，如上面程序所示，当程序调用 fromArrayToCollection()方法时，无须在调用该方法前传入 String、Object 等类型，但系统依然可以知道为泛型实际传入的类型，因为编译器根据实参推断出泛型所代表的类型，它通常推断出最直接的类型。例如，下面调用代码：

```
fromArrayToCollection(sa, cs);
```

上面代码中 cs 是一个 Collection<String>类型，与方法定义时的 fromArrayToCollection(T[] a, Collection<T> c)进行比较——只比较泛型参数，不难发现该 T 类型代表的实际类型是 String 类型。

对于如下调用代码：

```
fromArrayToCollection(ia, cn);
```

上面的 cn 是 Collection<Number>类型，与此方法的方法签名进行比较——只比较泛型参数，不难发现该 T 类型代表了 Number 类型。

为了让编译器能准确地推断出泛型方法中泛型的类型，不要制造迷惑！系统一旦迷惑了，就是你错了！看如下程序。

程序清单：codes\09\9.4\ErrorTest.java

```
public class ErrorTest
{
    // 声明一个泛型方法，该泛型方法中带一个 T 泛型形参
    static <T> void test(Collection<T> from, Collection<T> to)
    {
        for (T ele : from)
        {
            to.add(ele);
        }
    }
    public static void main(String[] args)
    {
        List<Object> as = new ArrayList<>();
        List<String> ao = new ArrayList<>();
        // 下面代码将产生编译错误
        test(as , ao);
    }
}
```

上面程序中定义了 test()方法，该方法用于将前一个集合里的元素复制到下一个集合中，该方法中的两个形参 from、to 的类型都是 Collection<T>，这要求调用该方法时的两个集合实参中的泛型类型相同，否则编译器无法准确地推断出泛型方法中泛型形参的类型。

上面程序中调用 test 方法传入了两个实际参数，其中 as 的数据类型是 List<String>，而 ao 的数据类

型是 List<Object>，与泛型方法签名进行对比：test(Collection<T> a, Collection<T> c)，编译器无法正确识别 T 所代表的实际类型。为了避免这种错误，可以将该方法改为如下形式：

程序清单：codes\09\9.4\RightTest.java

```
public class RightTest
{
    // 声明一个泛型方法，该泛型方法中带一个 T 形参
    static <T> void test(Collection<? extends T> from , Collection<T> to)
    {
        for (T ele : from)
        {
            to.add(ele);
        }
    }
    public static void main(String[] args)
    {
        List<Object> ao = new ArrayList<>();
        List<String> as = new ArrayList<>();
        // 下面代码完全正常
        test(as , ao);
    }
}
```

上面代码改变了 test()方法签名，将该方法的前一个形参类型改为 Collection<? extends T>，这种采用类型通配符的表示方式，只要 test()方法的前一个 Collection 集合里的元素类型是后一个 Collection 集合里元素类型的子类即可。

那么这里产生了一个问题：到底何时使用泛型方法？何时使用类型通配符呢？接下来详细介绍泛型方法和类型通配符的区别。

9.4.2 泛型方法和类型通配符的区别

大多数时候都可以使用泛型方法来代替类型通配符。例如，对于 Java 的 Collection 接口中两个方法定义：

```
public interface Collection<E>
{
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    ...
}
```

上面集合中两个方法的形参都采用了类型通配符的形式，也可以采用泛型方法的形式，如下所示。

```
public interface Collection<E>
{
    <T> boolean containsAll(Collection<T> c);
    <T extends E> boolean addAll(Collection<T> c);
    ...
}
```

上面方法使用了<T extends E>泛型形式，这时定义泛型形参时设定上限（其中 E 是 Collection 接口里定义的泛型，在该接口里 E 可当成普通类型使用）。

上面两个方法中泛型形参 T 只使用了一次，泛型形参 T 产生的唯一效果是可以在不同的调用点传入不同的实际类型。对于这种情况，应该使用通配符：通配符就是被设计用来支持灵活的子类化的。

泛型方法允许泛型形参被用来表示方法的一个或多个参数之间的类型依赖关系，或者方法返回值与参数之间的类型依赖关系。如果没有这样的类型依赖关系，就不应该使用泛型方法。

提示：

如果某个方法中一个形参（a）的类型或返回值的类型依赖于另一个形参（b）的类型，则形参（b）的类型声明不应该使用通配符——因为形参（a）或返回值的类型依赖于该形参（b）的类型，如果形参（b）的类型无法确定，程序就无法定义形参（a）的类型。在这种情况下，只能考虑使用在方法签名中声明泛型——也就是泛型方法。

如果有需要，也可以同时使用泛型方法和通配符，如 Java 的 Collections.copy()方法。

```
public class Collections
{
    public static <T> void copy(List<T> dest, List<? extends T> src){...}
    ...
}
```

上面 copy 方法中的 dest 和 src 存在明显的依赖关系，从源 List 中复制出来的元素，必须可以“丢进”目标 List 中，所以源 List 集合元素的类型只能是目标集合元素的类型的子类型或者它本身。但 JDK 定义 src 形参类型时使用的是类型通配符，而不是泛型方法。这是因为：该方法无须向 src 集合中添加元素，也无须修改 src 集合里的元素，所以可以使用类型通配符，无须使用泛型方法。

提示：简而言之，指定上限的类型通配符支持协变，因此这种协变的集合可以安全地取出元素（协变只出不进），因此无须使用泛型方法。

当然，也可以将上面的方法签名改为使用泛型方法，不使用类型通配符，如下所示。

```
class Collections
{
    public static <T , S extends T> void copy(List<T> dest, List<S> src){...}
    ...
}
```

这个方法签名可以代替前面的方法签名。但注意上面的泛型形参 S，它仅使用了一次，其他参数的类型、方法返回值的类型都不依赖于它，那泛型形参 S 就没有存在的必要，即可以用通配符来代替 S。使用通配符比使用泛型方法（在方法签名中显式声明泛型形参）更加清晰和准确，因此 Java 设计该方法时采用了通配符，而不是泛型方法。

类型通配符与泛型方法（在方法签名中显式声明泛型形参）还有一个显著的区别：类型通配符既可以在方法签名中定义形参的类型，也可以用于定义变量的类型；但泛型方法中的泛型形参必须在对应方法中显式声明。

9.4.3　Java 7 的“菱形”语法与泛型构造器

正如泛型方法允许在方法签名中声明泛型形参一样，Java 也允许在构造器签名中声明泛型形参，这样就产生了所谓的泛型构造器。

一旦定义了泛型构造器，接下来在调用构造器时，就不仅可以让 Java 根据数据参数的类型来“推断”泛型形参的类型，而且程序员也可以显式地为构造器中的泛型形参指定实际的类型。如下程序所示。

程序清单：codes\09\9.4\GenericConstructor.java

```
class Foo
{
    public <T> Foo(T t)
    {
        System.out.println(t);
    }
}
public class GenericConstructor
{
    public static void main(String[] args)
    {
        // 泛型构造器中的 T 类型为 String
        new Foo("疯狂 Java 讲义");
        // 泛型构造器中的 T 类型为 Integer
        new Foo(200);
        // 显式指定泛型构造器中的 T 类型为 String
        // 传给 Foo 构造器的实参也是 String 对象，完全正确
        new <String> Foo("疯狂 Android 讲义");      // ①
        // 显式指定泛型构造器中的 T 类型为 String，
        // 但传给 Foo 构造器的实参是 Double 对象，下面代码出错
        new <String> Foo(12.3);       // ②
    }
}
```

上面程序中①号代码不仅显式指定了泛型构造器中的泛型形参 T 的类型应该是 String，而且程序传给该构造器的参数值也是 String 类型，因此程序完全正常。但在②号代码处，程序显式指定了泛型构造器中的泛型形参 T 的类型应该是 String，但实际传给该构造器的参数值是 Double 类型，因此这行代码将会出现错误。

前面介绍过 Java 7 新增的“菱形”语法，它允许调用构造器时在构造器后使用一对尖括号来代表泛型信息。但如果程序显式指定了泛型构造器中声明的泛型形参的实际类型，则不可以使用“菱形”语法。如下程序所示。

程序清单：codes\09\9.4\GenericDiamondTest.java

```
class MyClass<E>
{
    public <T> MyClass(T t)
    {
        System.out.println("t 参数的值为：" + t);
    }
}
public class GenericDiamondTest
{
    public static void main(String[] args)
    {
        // MyClass 类声明中的 E 形参是 String 类型
        // 泛型构造器中声明的 T 形参是 Integer 类型
        MyClass<String> mc1 = new MyClass<>(5);
        // 显式指定泛型构造器中声明的 T 形参是 Integer 类型
        MyClass<String> mc2 = new <Integer> MyClass<String>(5);
        // MyClass 类声明中的 E 形参是 String 类型
        // 如果显式指定泛型构造器中声明的 T 形参是 Integer 类型
        // 此时就不能使用“菱形”语法，下面代码是错的
//      MyClass<String> mc3 = new <Integer> MyClass<>(5);
    }
}
```

上面程序中粗体字代码既指定了泛型构造器中的泛型形参是 Integer 类型，又想使用“菱形”语法，所以这行代码无法通过编译。

9.4.4 泛型方法与方法重载

因为泛型既允许设定通配符的上限，也允许设定通配符的下限，从而允许在一个类里包含如下两个方法定义。

```
public class MyUtils
{
    public static <T> void copy(Collection<T> dest , Collection<? extends T> src)
    {...}   // ①
    public static <T> T copy(Collection<? super T> dest , Collection<T> src)
    {...}   // ②
}
```

上面的 MyUtils 类中包含两个 copy()方法，这两个方法的参数列表存在一定的区别，但这种区别不是很明确：这两个方法的两个参数都是 Collection 对象，前一个集合里的集合元素类型是后一个集合里集合元素类型的父类。如果只是在该类中定义这两个方法不会有任何错误，但只要调用这个方法就会引起编译错误。例如，对于如下代码：

```
List<Number> ln = new ArrayList<>();
List<Integer> li = new ArrayList<>();
copy(ln , li);
```

上面程序中粗体字部分调用 copy()方法，但这个 copy()方法既可以匹配①号 copy()方法，此时泛型 T 表示的类型是 Number；也可以匹配②号 copy()方法，此时泛型 T 表示的类型是 Integer。编译器无法确定这行代码想调用哪个 copy()方法，所以这行代码将引起编译错误。

9.4.5 Java 8 改进的类型推断

Java 8 改进了泛型方法的类型推断能力，类型推断主要有如下两方面。

- 可通过调用方法的上下文来推断泛型的目标类型。
- 可在方法调用链中，将推断得到的泛型传递到最后一个方法。

如下程序示范了 Java 8 对泛型方法的类型推断。

程序清单：codes\09\9.4\InferenceTest.java

```
class MyUtil<E>
{
    public static <Z> MyUtil<Z> nil()
    {
        return null;
    }
    public static <Z> MyUtil<Z> cons(Z head, MyUtil<Z> tail)
    {
        return null;
    }
    E head()
    {
        return null;
    }
}
public class InferenceTest
{
    public static void main(String[] args)
    {
        // 可以通过方法赋值的目标参数来推断泛型为 String
        MyUtil<String> ls = MyUtil.nil();
        // 无须使用下面语句在调用 nil()方法时指定泛型的类型
        MyUtil<String> mu = MyUtil.<String>nil();
        // 可调用 cons()方法所需的参数类型来推断泛型为 Integer
        MyUtil.cons(42, MyUtil.nil());
        // 无须使用下面语句在调用 nil()方法时指定泛型的类型
        MyUtil.cons(42, MyUtil.<Integer>nil());
    }
}
```

上面程序中前两行粗体字代码的作用完全相同，但第 1 行粗体字代码无须在调用 MyUtil 类的 nil()方法时显式指定泛型参数为 String，这是因为程序需要将该方法的返回值赋值给 MyUtil<String>类型，因此系统可以自动推断出此处的泛型参数为 String 类型。

上面程序中第 3 行与第 4 行粗体字代码的作用也完全相同，但第 3 行粗体字代码也无须在调用 MyUtil 类的 nil()方法时显式指定泛型参数为 Integer，这是因为程序将 nil()方法的返回值作为了 MyUtil 类的 cons()方法的第二个参数，而程序可以根据 cons()方法的第一个参数（42）推断出此处的泛型参数为 Integer 类型。

需要指出的是，虽然 Java 8 增强了泛型推断的能力，但泛型推断不是万能的，例如如下代码就是错误的。

```
// 希望系统能推断出调用 nil()方法时泛型为 String 类型
// 但实际上 Java 8 依然推断不出来，所以下面代码报错
String s = MyUtil.nil().head();
```

因此，上面这行代码必须显式指定泛型的实际类型，即将代码改为如下形式：

```
String s = MyUtil.<String>nil().head();
```

9.5 擦除和转换

在严格的泛型代码里，带泛型声明的类总应该带着类型参数。但为了与老的 Java 代码保持一致，也允许在使用带泛型声明的类时不指定实际的类型。如果没有为这个泛型类指定实际的类型，此时被称作 raw type（原始类型），默认是声明该泛型形参时指定的第一个上限类型。

当把一个具有泛型信息的对象赋给另一个没有泛型信息的变量时，所有在尖括号之间的类型信息都将被扔掉。比如一个 List<String>类型被转换为 List，则该 List 对集合元素的类型检查变成了泛型参数的上限（即 Object）。下面程序示范了这种擦除。

程序清单：codes\09\9.5\ErasureTest.java

```
class Apple<T extends Number>
{
    T size;
    public Apple()
    {
    }
    public Apple(T size)
    {
        this.size = size;
    }
    public void setSize(T size)
    {
        this.size = size;
    }
    public T getSize()
    {
        return this.size;
    }
}
public class ErasureTest
{
    public static void main(String[] args)
    {
        Apple<Integer> a = new Apple<>(6);    // ①
        // a 的 getSize()方法返回 Integer 对象
        Integer as = a.getSize();
        // 把 a 对象赋给 Apple 变量，丢失尖括号里的类型信息
        Apple b = a;      // ②
        // b 只知道 size 的类型是 Number
        Number size1 = b.getSize();
        // 下面代码引起编译错误
        Integer size2 = b.getSize();  // ③
    }
}
```

上面程序中定义了一个带泛型声明的 Apple 类，其泛型形参的上限是 Number，这个泛型形参用来定义 Apple 类的 size 变量。程序在①处创建了一个 Apple 对象，该 Apple 对象的泛型代表了 Integer 类型，所以调用 a 的 getSize()方法时返回 Integer 类型的值。当把 a 赋给一个不带泛型信息的 b 变量时，编译器就会丢失 a 对象的泛型信息，即所有尖括号里的信息都会丢失——因为 Apple 的泛型形参的上限是 Number 类，所以编译器依然知道 b 的 getSize()方法返回 Number 类型，但具体是 Number 的哪个子类就不清楚了。

从逻辑上来看，List<String>是 List 的子类，如果直接把一个 List 对象赋给一个 List<String>对象应该引起编译错误，但实际上不会。对泛型而言，可以直接把一个 List 对象赋给一个 List<String>对象，编译器仅仅提示“未经检查的转换”，看下面程序。

程序清单：codes\09\9.5\ErasureTest2.java

```
public class ErasureTest2
{
    public static void main(String[] args)
    {
        List<Integer> li = new ArrayList<>();
        li.add(6);
        li.add(9);
        List list = li;
        // 下面代码引起“未经检查的转换”警告，编译、运行时完全正常
        List<String> ls = list;     // ①
```

```
        // 但只要访问 ls 里的元素，如下面代码将引起运行时异常
        System.out.println(ls.get(0));
    }
}
```

上面程序中定义了一个 List<Integer>对象，这个 List<Integer>对象保留了集合元素的类型信息。当把这个 List<Integer>对象赋给一个 List 类型的 list 后，编译器就会丢失前者的泛型信息，即丢失 list 集合里元素的类型信息，这是典型的擦除。Java 又允许直接把 List 对象赋给一个 List<Type>（Type 可以是任何类型）类型的变量，所以程序在①处可以编译通过，只是发出“未经检查的转换”警告。但对 list 变量实际上引用的是 List<Integer>集合，所以当试图把该集合里的元素当成 String 类型的对象取出时，将引发 ClassCastException 异常。

下面代码与上面代码的行为完全相似。

```
public class ErasureTest2
{
    public static void main(String[] args)
    {
        List li = new ArrayList();
        li.add(6);
        li.add(9);
        System.out.println((String)li.get(0));
    }
}
```

程序从 li 中获取一个元素，并且试图通过强制类型转换把它转换成一个 String，将引发运行时异常。前面使用泛型代码时，系统与之存在完全相似的行为，所以引发相同的 ClassCastException 异常。

9.6 泛型与数组

Java 泛型有一个很重要的设计原则——如果一段代码在编译时没有提出“[unchecked] 未经检查的转换”警告，则程序在运行时不会引发 ClassCastException 异常。正是基于这个原因，所以数组元素的类型不能包含泛型变量或泛型形参，除非是无上限的类型通配符。但可以声明元素类型包含泛型变量或泛型形参的数组。也就是说，只能声明 List<String>[]形式的数组，但不能创建 ArrayList<String>[10]这样的数组对象。

假设 Java 支持创建 ArrayList<String>[10]这样的数组对象，则有如下程序：

```
// 下面代码实际上是不允许的
List<String>[] lsa = new ArrayList<String>[10];
// 将 lsa 向上转型为 Object[]类型的变量
Object[] oa = lsa;
List<Integer> li = new ArrayList<>();
li.add(3);
// 将 List<Integer>对象作为 oa 的第二个元素
// 下面代码没有任何警告
oa[1] = li;
// 下面代码也不会有任何警告，但将引发 ClassCastException 异常
String s = lsa[1].get(0);    // ①
```

在上面代码中，如果粗体字代码是合法的，经过中间系列的程序运行，势必在①处引发运行时异常，这就违背了 Java 泛型的设计原则。

如果将程序改为如下形式：

程序清单：codes\09\9.6\GenericAndArray.java

```
// 下面代码编译时有“[unchecked] 未经检查的转换”警告
List<String>[] lsa = new ArrayList[10];
// 将 lsa 向上转型为 Object[]类型的变量
Object[] oa = lsa;
List<Integer> li = new ArrayList<>();
li.add(3);
```

```
oa[1] = li;
// 下面代码引起 ClassCastException 异常
String s = lsa[1].get(0);            // ①
```

上面程序粗体字代码行声明了 List<String>[]类型的数组变量，这是允许的；但不允许创建 List<String>[]类型的对象，所以创建了一个类型为 ArrayList[10]的数组对象，这也是允许的。只是把 ArrayList[10]对象赋给 List<String>[]变量时会有编译警告“[unchecked] 未经检查的转换”，即编译器并不保证这段代码是类型安全的。上面代码同样会在①处引发运行时异常，但因为编译器已经提出了警告，所以完全可能出现这种异常。

Java 允许创建无上限的通配符泛型数组，例如 new ArrayList<?>[10]，因此也可以将第一段代码改为使用无上限的通配符泛型数组，在这种情况下，程序不得不进行强制类型转换。正如前面所介绍的，在进行强制类型转换之前应通过 instanceof 运算符来保证它的数据类型。将上面代码改为如下形式（程序清单同上）：

```
List<?>[] lsa = new ArrayList<?>[10];
Object[] oa = lsa;
List<Integer> li = new ArrayList<>();
li.add(3);
oa[1] = li;
Object target = lsa[1].get(0);
if (target instanceof String)
{
    // 下面代码安全了
    String s = (String) target;
}
```

与此类似的是，创建元素类型是泛型类型的数组对象也将导致编译错误。如下代码所示。

```
<T> T[] makeArray(Collection<T> coll)
{
    // 下面代码导致编译错误
    return new T[coll.size()];
}
```

由于类型变量在运行时并不存在，而编译器无法确定实际类型是什么，因此编译器在粗体字代码处报错。

9.7 本章小结

本章主要介绍了 Java 提供的泛型支持，还介绍了为何需要在编译时检查集合元素的类型，以及如何编程来实现这种检查，从而引出 Java 泛型给程序带来的简洁性和健壮性。本章详细讲解了如何定义泛型接口、泛型类，以及如何从泛型类、泛型接口派生子类或实现类，并深入讲解了泛型类的实质。本章介绍了类型通配符的用法，包括设定类型通配符的上限、下限等；本章重点介绍了泛型方法的知识，包括如何在方法签名时定义泛型形参，以及泛型方法和类型通配符之间的区别与联系。本章最后介绍了 Java 不支持创建泛型数组，并深入分析了原因。

本章练习

1. 改写第 8 章的 8.2 节、8.4 节和 8.6 节的示例程序，为它们添加泛型，使得编译器不再提示警告。

2. 定义一个 BaseDao<T>接口，接口包括 save、update、delete、findAll 方法，这些方法的参数、返回值应该根据接口的泛型动态改变。

3. 定义一个包含 copy 的工具方法，该 copy 方法可以把 List 集合的元素（元素都是 Number 的子类）拷贝到对应的数组中。

CHAPTER

10

第10章 异常处理

本章要点

- 异常的定义和概念
- Java 异常机制的优势
- 使用 try...catch 捕获异常
- 多异常捕获
- Java 异常类的继承体系
- 异常对象的常用方法
- finally 块的作用
- 自动关闭资源的 try 语句
- 异常处理的合理嵌套
- Checked 异常和 Runtime 异常
- 使用 throws 声明异常
- 使用 throw 抛出异常
- 自定义异常
- 异常链和异常转译
- 异常的跟踪栈信息
- 异常的处理规则

异常机制已经成为判断一门编程语言是否成熟的标准，除传统的像C语言没有提供异常机制之外，目前主流的编程语言如Java、C#、Ruby、Python等都提供了成熟的异常机制。异常机制可以使程序中的异常处理代码和正常业务代码分离，保证程序代码更加优雅，并可以提高程序的健壮性。

Java的异常机制主要依赖于try、catch、finally、throw和throws五个关键字，其中try关键字后紧跟一个花括号扩起来的代码块（花括号不可省略），简称try块，它里面放置可能引发异常的代码。catch后对应异常类型和一个代码块，用于表明该catch块用于处理这种类型的代码块。多个catch块后还可以跟一个finally块，finally块用于回收在try块里打开的物理资源，异常机制会保证finally块总被执行。throws关键字主要在方法签名中使用，用于声明该方法可能抛出的异常；而throw用于抛出一个实际的异常，throw可以单独作为语句使用，抛出一个具体的异常对象。

Java 7进一步增强了异常处理机制的功能，包括带资源的try语句、捕获多异常的catch两个新功能，这两个功能可以极好地简化异常处理。

开发者都希望所有的错误都能在编译阶段被发现，就是在试图运行程序之前排除所有错误，但这是不现实的，余下的问题必须在运行期间得到解决。Java将异常分为两种，Checked异常和Runtime异常，Java认为Checked异常都是可以在编译阶段被处理的异常，所以它强制程序处理所有的Checked异常；而Runtime异常则无须处理。Checked异常可以提醒程序员需要处理所有可能发生的异常，但Checked异常也给编程带来一些烦琐之处，所以Checked异常也是Java领域一个备受争论的话题。

10.1 异常概述

异常处理已经成为衡量一门语言是否成熟的标准之一，目前的主流编程语言如C++、C#、Ruby、Python等大都提供了异常处理机制。增加了异常处理机制后的程序有更好的容错性，更加健壮。

与很多图书喜欢把异常处理放在开始部分介绍不一样，本书宁愿把异常处理放在“后面”介绍。因为异常处理是一件很乏味、不能带来成就感的事情，没有人希望自己遇到异常，大家都希望每天都能爱情甜蜜、家庭和睦、风和日丽、春暖花开……但事实上，这不可能！（如果可以这样顺利，上帝也会想做凡人了。）

对于计算机程序而言，情况就更复杂了——没有人能保证自己写的程序永远不会出错！就算程序没有错误，你能保证用户总是按你的意愿来输入？就算用户都是非常“聪明而且配合”的，你能保证运行该程序的操作系统永远稳定？你能保证运行该程序的硬件不会突然坏掉？你能保证网络永远通畅？……太多你无法保证的情况了！

对于一个程序设计人员，需要尽可能地预知所有可能发生的情况，尽可能地保证程序在所有糟糕的情形下都可以运行。考虑前面介绍的五子棋程序：当用户输入下棋坐标时，程序要判断用户输入是否合法，如果保证程序有较好的容错性，将会有如下的伪码。

```
if(用户输入包含除逗号之外的其他非数字字符)
{
    alert 坐标只能是数值
    goto retry
}
else if (用户输入不包含逗号)
{
    alert 应使用逗号分隔两个坐标值
    goto retry
}
else if (用户输入坐标值超出了有效范围)
{
    alert 用户输入坐标应位于棋盘坐标之内
    goto retry
}
else if(用户输入的坐标已有棋子)
{
    alert "只能在没有棋子的地方下棋"
    goto retry
}
```

```
else
{
    // 业务实现代码
    ...
}
```

上面代码还未涉及任何有效处理，只是考虑了4种可能的错误，代码就已经急剧增加了。但实际上，上面考虑的4种情形还远未考虑到所有的可能情形（事实上，世界上的意外是不可穷举的），程序可能发生的异常情况总是大于程序员所能考虑的意外情况。

而且正如前面提到的，高傲的程序员们开发程序时更倾向于认为："对，错误也许会发生，但那是别人造成的，不关我的事"。

如果每次在实现真正的业务逻辑之前，都需要不厌其烦地考虑各种可能出错的情况，针对各种错误情况给出补救措施——这是多么乏味的事情啊。程序员喜欢解决问题，喜欢开发带来的"创造"快感，都不喜欢像一个"堵漏"工人，去堵那些由外在条件造成的"漏洞"。

提示：

对于构造大型、健壮、可维护的应用而言，错误处理是整个应用需要考虑的重要方面，曾经有一个教授告诉我：国内的程序员做开发时，往往只做了"对"的事情！他这句话有很深的遗憾——程序员开发程序的过程，是一个创造的过程，这个过程需要有全面的考虑，仅做"对"的事情是远远不够的。

对于上面的错误处理机制，主要有如下两个缺点。

- 无法穷举所有的异常情况。因为人类知识的限制，异常情况总比可以考虑到的情况多，总有"漏网之鱼"的异常情况，所以程序总是不够健壮。
- 错误处理代码和业务实现代码混杂。这种错误处理和业务实现混杂的代码严重影响程序的可读性，会增加程序维护的难度。

程序员希望有一种强大的机制来解决上面的问题，希望上面程序换成如下伪码。

```
if(用户输入不合法)
{
    alert 输入不合法
    goto retry
}
else
{
    // 业务实现代码
    ...
}
```

上面伪码提供了一个非常强大的"if块"——程序不管输入错误的原因是什么，只要用户输入不满足要求，程序就一次处理所有的错误。这种处理方法的好处是，使得错误处理代码变得更有条理，只需在一个地方处理错误。

现在的问题是"用户输入不合法"这个条件怎么定义？当然，对于这个简单的要求，可以使用正则表达式对用户输入进行匹配，当用户输入与正则表达式不匹配时即可判断"用户输入不合法"。但对于更复杂的情形呢？恐怕就没有这么简单了。使用Java的异常处理机制就可解决这个问题。

10.2 异常处理机制

Java的异常处理机制可以让程序具有极好的容错性，让程序更加健壮。当程序运行出现意外情形时，系统会自动生成一个Exception对象来通知程序，从而实现将"业务功能实现代码"和"错误处理代码"分离，提供更好的可读性。

10.2.1 使用try...catch捕获异常

正如前一节代码所提示的，希望有一种非常强大的"if 块"，可以表示所有的错误情况，让程序可

以一次处理所有的错误，也就是希望将错误集中处理。

出于这种考虑，此处试图把“错误处理代码”从“业务实现代码”中分离出来。将上面最后一段伪码改为如下所示伪码。

```
if(一切正常)
{
    // 业务实现代码
    ...
}
else
{
    alert 输入不合法
    goto retry
}
```

上面代码中的“if 块”依然不可表示——一切正常是很抽象的，无法转换为计算机可识别的代码，在这种情形下，Java 提出了一种假设：如果程序可以顺利完成，那就“一切正常”，把系统的业务实现代码放在 try 块中定义，所有的异常处理逻辑放在 catch 块中进行处理。下面是 Java 异常处理机制的语法结构。

```
try
{
    // 业务实现代码
    ...
}
catch (Exception e)
{
    alert 输入不合法
    goto retry
}
```

如果执行 try 块里的业务逻辑代码时出现异常，系统自动生成一个异常对象，该异常对象被提交给 Java 运行时环境，这个过程被称为抛出（throw）异常。

当 Java 运行时环境收到异常对象时，会寻找能处理该异常对象的 catch 块，如果找到合适的 catch 块，则把该异常对象交给该 catch 块处理，这个过程被称为捕获（catch）异常；如果 Java 运行时环境找不到捕获异常的 catch 块，则运行时环境终止，Java 程序也将退出。

提示：

不管程序代码块是否处于 try 块中，甚至包括 catch 块中的代码，只要执行该代码块时出现了异常，系统总会自动生成一个异常对象。如果程序没有为这段代码定义任何的 catch 块，则 Java 运行时环境无法找到处理该异常的 catch 块，程序就在此退出，这就是前面看到的例子程序在遇到异常时退出的情形。

下面使用异常处理机制来改写前面第 4 章五子棋游戏中用户下棋部分的代码。

程序清单：codes\10\10.2\Gobang.java

```
String inputStr = null;
// br.readLine()：每当在键盘上输入一行内容时按回车键
// 用户刚刚输入的内容将被 br 读取到
while ((inputStr = br.readLine()) != null)
{
    try
    {
        // 将用户输入的字符串以逗号作为分隔符，分解成 2 个字符串
        String[] posStrArr = inputStr.split(",");
        // 将 2 个字符串转换成用户下棋的坐标
        int xPos = Integer.parseInt(posStrArr[0]);
        int yPos = Integer.parseInt(posStrArr[1]);
        // 把对应的数组元素赋为"●"
        if (!gb.board[xPos - 1][yPos - 1].equals("┼"))
        {
            System.out.println("您输入的坐标点已有棋子了，"
```

```
                + "请重新输入");
            continue;
        }
        gb.board[xPos - 1][yPos - 1] = "●";
    }
    catch (Exception e)
    {
        System.out.println("您输入的坐标不合法，请重新输入，"
            + "下棋坐标应以 x,y 的格式");
        continue;
    }
    ...
}
```

上面程序把处理用户输入字符串的代码都放在 try 块里进行，只要用户输入的字符串不是有效的坐标值（包括字母不能正确解析，没有逗号不能正确解析，解析出来的坐标引起数组越界……），系统都将抛出一个异常对象，并把这个异常对象交给对应的 catch 块（也就是上面程序中粗体字代码块）处理，catch 块的处理方式是向用户提示坐标不合法，然后使用 continue 忽略本次循环剩下的代码，开始执行下一次循环，这就保证了该五子棋游戏有足够的容错性——用户可以随意输入，程序不会因为用户输入不合法而突然退出，程序会向用户提示输入不合法，让用户再次输入。

10.2.2　异常类的继承体系

当 Java 运行时环境接收到异常对象时，如何为该异常对象寻找 catch 块呢？注意上面 Gobang 程序中 catch 关键字的形式：(Exception e)，这意味着每个 catch 块都是专门用于处理该异常类及其子类的异常实例。

当 Java 运行时环境接收到异常对象后，会依次判断该异常对象是否是 catch 块后异常类或其子类的实例，如果是，Java 运行时环境将调用该 catch 块来处理该异常；否则再次拿该异常对象和下一个 catch 块里的异常类进行比较。Java 异常捕获流程示意图如图 10.1 所示。

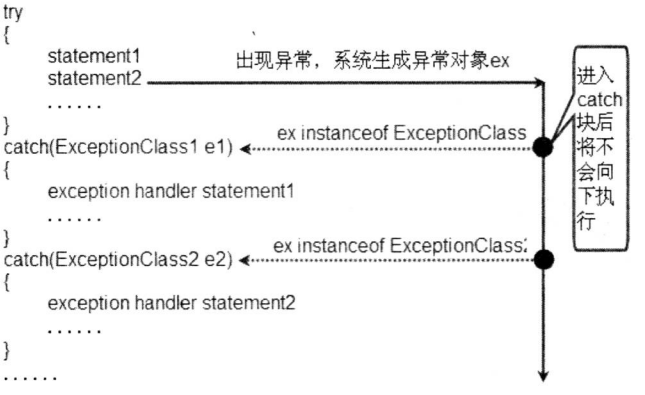


图 10.1　Java 异常捕获流程示意图

当程序进入负责异常处理的 catch 块时，系统生成的异常对象 ex 将会传给 catch 块后的异常形参，从而允许 catch 块通过该对象来获得异常的详细信息。

从图 10.1 中可以看出，try 块后可以有多个 catch 块，这是为了针对不同的异常类提供不同的异常处理方式。当系统发生不同的意外情况时，系统会生成不同的异常对象，Java 运行时就会根据该异常对象所属的异常类来决定使用哪个 catch 块来处理该异常。

通过在 try 块后提供多个 catch 块可以无须在异常处理块中使用 if、switch 判断异常类型，但依然可以针对不同的异常类型提供相应的处理逻辑，从而提供更细致、更有条理的异常处理逻辑。

从图 10.1 中可以看出，在通常情况下，如果 try 块被执行一次，则 try 块后只有一个 catch 块会被执行，绝不可能有多个 catch 块被执行。除非在循环中使用了 continue 开始下一次循环，下一次循环又重新运行了 try 块，这才可能导致多个 catch 块被执行。

注意：

try 块与 if 语句不一样，try 块后的花括号（{...}）不可以省略，即使 try 块里只有一行代码，也不可省略这个花括号。与之类似的是，catch 块后的花括号（{...}）也不可以省略。还有一点需要指出：try 块里声明的变量是代码块内局部变量，它只在 try 块内有效，在 catch 块中不能访问该变量。

Java 提供了丰富的异常类，这些异常类之间有严格的继承关系，图 10.2 显示了 Java 常见的异常类之间的继承关系。

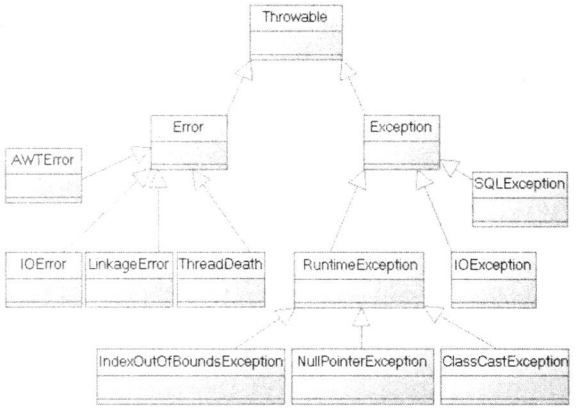


图 10.2　Java 常见的异常类之间的继承关系

从图 10.2 中可以看出，Java 把所有的非正常情况分成两种：异常（Exception）和错误（Error），它们都继承 Throwable 父类。

Error 错误，一般是指与虚拟机相关的问题，如系统崩溃、虚拟机错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断。通常应用程序无法处理这些错误，因此应用程序不应该试图使用 catch 块来捕获 Error 对象。在定义该方法时，也无须在其 throws 子句中声明该方法可能抛出 Error 及其任何子类。下面看几个简单的异常捕获例子。

程序清单：codes\10\10.2\DivTest.java

```
public class DivTest
{
    public static void main(String[] args)
    {
        try
        {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            int c = a / b;
            System.out.println("您输入的两个数相除的结果是：" + c );
        }
        catch (IndexOutOfBoundsException ie)
        {
            System.out.println("数组越界：运行程序时输入的参数个数不够");
        }
        catch (NumberFormatException ne)
        {
            System.out.println("数字格式异常：程序只能接收整数参数");
        }
        catch (ArithmeticException ae)
        {
            System.out.println("算术异常");
        }
        catch (Exception e)
        {
            System.out.println("未知异常");
        }
    }
}
```

上面程序针对 IndexOutOfBoundsException、NumberFormatException、ArithmeticException 类型的异常，提供了专门的异常处理逻辑。Java 运行时的异常处理逻辑可能有如下几种情形。

- 如果运行该程序时输入的参数不够，将会发生数组越界异常，Java 运行时将调用 IndexOutOfBoundsException 对应的 catch 块处理该异常。
- 如果运行该程序时输入的参数不是数字，而是字母，将发生数字格式异常，Java 运行时将调用 NumberFormatException 对应的 catch 块处理该异常。
- 如果运行该程序时输入的第二个参数是 0，将发生除 0 异常，Java 运行时将调用 ArithmeticException 对应的 catch 块处理该异常。
- 如果程序运行时出现其他异常，该异常对象总是 Exception 类或其子类的实例，Java 运行时将调用 Exception 对应的 catch 块处理该异常。

提示：上面程序中的三种异常，都是非常常见的运行时异常，读者应该记住这些异常，并掌握在哪些情况下可能出现这些异常。

程序清单：codes\10\10.2\NullTest.java

```
public class NullTest
{
    public static void main(String[] args)
    {
        Date d = null;
        try
        {
            System.out.println(d.after(new Date()));
        }
        catch (NullPointerException ne)
        {
            System.out.println("空指针异常");
        }
        catch(Exception e)
        {
            System.out.println("未知异常");
        }
    }
}
```

上面程序针对 NullPointerException 异常提供了专门的异常处理块。上面程序调用一个 null 对象的 after()方法，这将引发 NullPointerException 异常（当试图调用一个 null 对象的实例方法或实例变量时，就会引发 NullPointerException 异常），Java 运行时将会调用 NullPointerException 对应的 catch 块来处理该异常；如果程序遇到其他异常，Java 运行时将会调用最后的 catch 块来处理异常。

正如在前面程序所看到的，程序总是把对应 Exception 类的 catch 块放在最后，这是为什么呢？想一下图 10.1 所示的 Java 异常捕获流程，读者可能明白原因：如果把 Exception 类对应的 catch 块排在其他 catch 块的前面，Java 运行时将直接进入该 catch 块（因为所有的异常对象都是 Exception 或其子类的实例），而排在它后面的 catch 块将永远也不会获得执行的机会。

实际上，进行异常捕获时不仅应该把 Exception 类对应的 catch 块放在最后，而且所有父类异常的 catch 块都应该排在子类异常 catch 块的后面（简称：先处理小异常，再处理大异常），否则将出现编译错误。看如下代码片段：

```
try
{
    statements...
}
catch(RuntimeException e)     // ①
{
    System.out.println("运行时异常");
}
catch (NullPointerException ne)     // ②
{
    System.out.println("空指针异常");
}
```

上面代码中有两个 catch 块，前一个 catch 块捕获 RuntimeException 异常，后一个 catch 块捕获 NullPointerException 异常，编译上面代码时将会在②处出现已捕获到异常 java.lang.NullPointerException 的错误，因为①处的 RuntimeException 已经包括了 NullPointerException 异常，所以②处的 catch 块永远也不会获得执行的机会。

注意：

异常捕获时，一定要记住先捕获小异常，再捕获大异常。

10.2.3 Java 7 新增的多异常捕获

在 Java 7 以前，每个 catch 块只能捕获一种类型的异常；但从 Java 7 开始，一个 catch 块可以捕获多种类型的异常。

使用一个 catch 块捕获多种类型的异常时需要注意如下两个地方。

- 捕获多种类型的异常时，多种异常类型之间用竖线（|）隔开。
- 捕获多种类型的异常时，异常变量有隐式的 final 修饰，因此程序不能对异常变量重新赋值。

下面程序示范了 Java 7 提供的多异常捕获。

程序清单：codes\10\10.2\MultiExceptionTest.java

```
public class MultiExceptionTest
{
    public static void main(String[] args)
    {
        try
        {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            int c = a / b;
            System.out.println("您输入的两个数相除的结果是：" + c );
        }
        catch (IndexOutOfBoundsException|NumberFormatException
            |ArithmeticException ie)
        {
            System.out.println("程序发生了数组越界、数字格式异常、算术异常之一");
            // 捕获多异常时，异常变量默认有 final 修饰
            // 所以下面代码有错
            ie = new ArithmeticException("test");  // ①
        }
        catch (Exception e)
        {
            System.out.println("未知异常");
            // 捕获一种类型的异常时，异常变量没有 final 修饰
            // 所以下面代码完全正确
            e = new RuntimeException("test");  // ②
        }
    }
}
```

上面程序中第一行粗体字代码使用了 IndexOutOfBoundsException|NumberFormatException|ArithmeticException 来定义异常类型，这就表明该 catch 块可以同时捕获这三种类型的异常。捕获多种类型的异常时，异常变量使用隐式的 final 修饰，因此上面程序中①号代码将产生编译错误；捕获一种类型的异常时，异常变量没有 final 修饰，因此上面程序中②号代码完全正确。

10.2.4 访问异常信息

如果程序需要在 catch 块中访问异常对象的相关信息，则可以通过访问 catch 块的后异常形参来获得。当 Java 运行时决定调用某个 catch 块来处理该异常对象时，会将异常对象赋给 catch 块后的异常参数，程序即可通过该参数来获得异常的相关信息。

所有的异常对象都包含了如下几个常用方法。

- getMessage()：返回该异常的详细描述字符串。
- printStackTrace()：将该异常的跟踪栈信息输出到标准错误输出。
- printStackTrace(PrintStream s)：将该异常的跟踪栈信息输出到指定输出流。
- getStackTrace()：返回该异常的跟踪栈信息。

下面例子程序演示了程序如何访问异常信息。

程序清单：codes\10\10.2\AccessExceptionMsg.java

```
public class AccessExceptionMsg
{
```

```
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis = new FileInputStream("a.txt");
        }
        catch (IOException ioe)
        {
            System.out.println(ioe.getMessage());
            ioe.printStackTrace();
        }
    }
}
```

上面程序调用了 Exception 对象的 getMessage()方法来得到异常对象的详细信息，也使用了 printStackTrace()方法来打印该异常的跟踪信息。运行上面程序，会看到如图 10.3 所示的界面。

提示：

上面程序中使用的 FileInputStream 是 Java IO 体系中的一个文件输入流，用于读取磁盘文件的内容。关于该类的详细介绍请参考本书第 15 章的内容。

从图 10.3 中可以看到异常的详细描述信息："a.txt (系统找不到指定的文件)"，这就是调用异常的 getMessage()方法返回的字符串。下面更详细的信息是该异常的跟踪栈信息，关于异常的跟踪栈信息后面还有更详细的介绍，此处不再赘述。

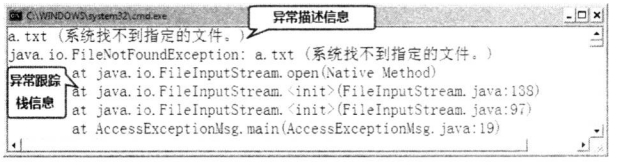


图 10.3 访问异常信息

10.2.5 使用 finally 回收资源

有些时候，程序在 try 块里打开了一些物理资源（例如数据库连接、网络连接和磁盘文件等），这些物理资源都必须显式回收。

提示：

Java 的垃圾回收机制不会回收任何物理资源，垃圾回收机制只能回收堆内存中对象所占用的内存。

在哪里回收这些物理资源呢？在 try 块里回收？还是在 catch 块中进行回收？假设程序在 try 块里进行资源回收，根据图 10.1 所示的异常捕获流程——如果 try 块的某条语句引起了异常，该语句后的其他语句通常不会获得执行的机会，这将导致位于该语句之后的资源回收语句得不到执行。如果在 catch 块里进行资源回收，但 catch 块完全有可能得不到执行，这将导致不能及时回收这些物理资源。

为了保证一定能回收 try 块中打开的物理资源，异常处理机制提供了 finally 块。不管 try 块中的代码是否出现异常，也不管哪一个 catch 块被执行，甚至在 try 块或 catch 块中执行了 return 语句，finally 块总会被执行。完整的 Java 异常处理语法结构如下：

```
try
{
    // 业务实现代码
    ...
}
catch (SubException e)
{
    // 异常处理块 1
    ...
}
catch (SubException2 e)
{
    // 异常处理块 2
    ...
}
...
```

```
finally
{
    // 资源回收块
    ...
}
```

异常处理语法结构中只有try块是必需的，也就是说，如果没有try块，则不能有后面的catch块和finally块；catch块和finally块都是可选的，但catch块和finally块至少出现其中之一，也可以同时出现；可以有多个catch块，捕获父类异常的catch块必须位于捕获子类异常的后面；但不能只有try块，既没有catch块，也没有finally块；多个catch块必须位于try块之后，finally块必须位于所有的catch块之后。看如下程序。

程序清单：codes\10\10.2\FinallyTest.java

```
public class FinallyTest
{
    public static void main(String[] args)
    {
        FileInputStream fis = null;
        try
        {
            fis = new FileInputStream("a.txt");
        }
        catch (IOException ioe)
        {
            System.out.println(ioe.getMessage());
            // return 语句强制方法返回
            return ;        // ①
            // 使用 exit 退出虚拟机
            // System.exit(1);     // ②
        }
        finally
        {
            // 关闭磁盘文件，回收资源
            if (fis != null)
            {
                try
                {
                    fis.close();
                }
                catch (IOException ioe)
                {
                    ioe.printStackTrace();
                }
            }
            System.out.println("执行 finally 块里的资源回收!");
        }
    }
}
```

上面程序的try块后增加了finally块，用于回收在try块中打开的物理资源。注意程序的catch块中①处有一条return语句，该语句强制方法返回。在通常情况下，一旦在方法里执行到return语句的地方，程序将立即结束该方法；现在不会了，虽然return语句也强制方法结束，但一定会先执行finally块里的代码。运行上面程序，看到如下结果：

```
a.txt (系统找不到指定的文件。)
程序已经执行了 finally 里的资源回收!
```

上面运行结果表明方法返回之前还是执行了finally块的代码。将①处的return语句注释掉，取消②处代码的注释，即在异常处理的catch块中使用System.exit(1)语句来退出虚拟机。执行上面代码，看到如下结果：

```
a.txt (系统找不到指定的文件。)
```

上面执行结果表明finally块没有被执行。如果在异常处理代码中使用System.exit(1)语句来退出虚拟机，则finally块将失去执行的机会。

除非在try块、catch块中调用了退出虚拟机的方法，否则不管在try块、catch块中执行怎样的代码，出现怎样的情况，异常处理的finally块总会被执行。

在通常情况下，不要在finally块中使用如return或throw等导致方法终止的语句，（throw语句将在后面介绍），一旦在finally块中使用了return或throw语句，将会导致try块、catch块中的return、throw语句失效。看如下程序。

程序清单：codes\10\10.2\FinallyFlowTest.java

```
public class FinallyFlowTest
{
    public static void main(String[] args)
        throws Exception
    {
        boolean a = test();
        System.out.println(a);
    }
    public static boolean test()
    {
        try
        {
            // 因为finally块中包含了return语句
            // 所以下面的return语句失去作用
            return true;
        }
        finally
        {
            return false;
        }
    }
}
```

上面程序在finally块中定义了一个return false语句，这将导致try块中的return true失去作用。运行上面程序，将打印出false的结果。

当Java程序执行try块、catch块时遇到了return或throw语句，这两个语句都会导致该方法立即结束，但是系统执行这两个语句并不会结束该方法，而是去寻找该异常处理流程中是否包含finally块，如果没有finally块，程序立即执行return或throw语句，方法终止；如果有finally块，系统立即开始执行finally块——只有当finally块执行完成后，系统才会再次跳回来执行try块、catch块里的return或throw语句；如果finally块里也使用了return或throw等导致方法终止的语句，finally块已经终止了方法，系统将不会跳回去执行try块、catch块里的任何代码。

尽量避免在finally块里使用return或throw等导致方法终止的语句，否则可能出现一些很奇怪的情况。

10.2.6　异常处理的嵌套

正如FinallyTest.java程序所示，finally块中也包含了一个完整的异常处理流程，这种在try块、catch块或finally块中包含完整的异常处理流程的情形被称为异常处理的嵌套。

异常处理流程代码可以放在任何能放可执行性代码的地方，因此完整的异常处理流程既可放在try块里，也可放在catch块里，还可放在finally块里。

异常处理嵌套的深度没有很明确的限制，但通常没有必要使用超过两层的嵌套异常处理，层次太深的嵌套异常处理没有太大必要，而且导致程序可读性降低。

10.2.7 Java 9 增强的自动关闭资源的 try 语句

在前面程序中看到，当程序使用 finally 块关闭资源时，程序显得异常臃肿。

```
FileInputStream fis = null;
try
{
    fis = new FileInputStream("a.txt");
}
...
finally
{
    // 关闭磁盘文件，回收资源
    if (fis != null)
    {
        fis.close();
    }
}
```

在 Java 7 以前，上面程序中粗体字代码是不得不写的“臃肿代码”，Java 7 的出现改变了这种局面。Java 7 增强了 try 语句的功能——它允许在 try 关键字后紧跟一对圆括号，圆括号可以声明、初始化一个或多个资源，此处的资源指的是那些必须在程序结束时显式关闭的资源（比如数据库连接、网络连接等），try 语句在该语句结束时自动关闭这些资源。

需要指出的是，为了保证 try 语句可以正常关闭资源，这些资源实现类必须实现 AutoCloseable 或 Closeable 接口，实现这两个接口就必须实现 close()方法。

提示：

Closeable 是 AutoCloseable 的子接口，可以被自动关闭的资源类要么实现 AutoCloseable 接口，要么实现 Closeable 接口。Closeable 接口里的 close()方法声明抛出了 IOException，因此它的实现类在实现 close()方法时只能声明抛出 IOException 或其子类；AutoCloseable 接口里的 close()方法声明抛出了 Exception，因此它的实现类在实现 close()方法时可以声明抛出任何异常。

下面程序示范了如何使用自动关闭资源的 try 语句。

程序清单：codes\10\10.2\AutoCloseTest.java

```
public class AutoCloseTest
{
    public static void main(String[] args)
        throws IOException
    {
        try (
            // 声明、初始化两个可关闭的资源
            // try 语句会自动关闭这两个资源
            BufferedReader br = new BufferedReader(
                new FileReader("AutoCloseTest.java"));
            PrintStream ps = new PrintStream(new
                FileOutputStream("a.txt")))
        {
            // 使用两个资源
            System.out.println(br.readLine());
            ps.println("庄生晓梦迷蝴蝶");
        }
    }
}
```

上面程序中粗体字代码分别声明、初始化了两个 IO 流，由于 BufferedReader、PrintStream 都实现了 Closeable 接口，而且它们放在 try 语句中声明、初始化，所以 try 语句会自动关闭它们。因此上面程序是安全的。

自动关闭资源的 try 语句相当于包含了隐式的 finally 块（这个 finally 块用于关闭资源），因此这个

try 语句可以既没有 catch 块，也没有 finally 块。

提示：

Java 7 几乎把所有的“资源类”（包括文件 IO 的各种类、JDBC 编程的 Connection、Statement 等接口）进行了改写，改写后资源类都实现了 AutoCloseable 或 Closeable 接口。

如果程序需要，自动关闭资源的 try 语句后也可以带多个 catch 块和一个 finally 块。

Java 9 再次增强了这种 try 语句，Java 9 不要求在 try 后的圆括号内声明并创建资源，只需要自动关闭的资源有 final 修饰或者是有效的 final（effectively final），Java 9 允许将资源变量放在 try 后的圆括号内。上面程序在 Java 9 中可改写为如下形式。

程序清单：codes\10\10.2\AutoCloseTest2.java

```
public class AutoCloseTest2
{
    public static void main(String[] args)
        throws IOException
    {
        // 有 final 修饰的资源
        final BufferedReader br = new BufferedReader(
            new FileReader("AutoCloseTest.java"));
        // 没有显式使用 final 修饰，但只要不对该变量重新赋值，该变量就是有效的 final
        PrintStream ps = new PrintStream(new
            FileOutputStream("a.txt"));
        // 只要将两个资源放在 try 后的圆括号内即可
        try (br;ps)
        {
            // 使用两个资源
            System.out.println(br.readLine());
            ps.println("庄生晓梦迷蝴蝶");
        }
    }
}
```

10.3 Checked 异常和 Runtime 异常体系

Java 的异常被分为两大类：Checked 异常和 Runtime 异常（运行时异常）。所有的 RuntimeException 类及其子类的实例被称为 Runtime 异常；不是 RuntimeException 类及其子类的异常实例则被称为 Checked 异常。

只有 Java 语言提供了 Checked 异常，其他语言都没有提供 Checked 异常。Java 认为 Checked 异常都是可以被处理（修复）的异常，所以 Java 程序必须显式处理 Checked 异常。如果程序没有处理 Checked 异常，该程序在编译时就会发生错误，无法通过编译。

Checked 异常体现了 Java 的设计哲学——没有完善错误处理的代码根本就不会被执行！

对于 Checked 异常的处理方式有如下两种。

- 当前方法明确知道如何处理该异常，程序应该使用 try...catch 块来捕获该异常，然后在对应的 catch 块中修复该异常。例如，前面介绍的五子棋游戏中处理用户输入不合法的异常，程序在 catch 块中打印对用户的提示信息，重新开始下一次循环。
- 当前方法不知道如何处理这种异常，应该在定义该方法时声明抛出该异常。

Runtime 异常则更加灵活，Runtime 异常无须显式声明抛出，如果程序需要捕获 Runtime 异常，也可以使用 try...catch 块来实现。

提示：

只有 Java 语言提供了 Checked 异常，Checked 异常体现了 Java 的严谨性，它要求程序员必须注意该异常——要么显式声明抛出，要么显式捕获并处理它，总之不允许对 Checked 异常不闻不问。这是一种非常严谨的设计哲学，可以增加程序的健壮性。问题是：

大部分的方法总是不能明确地知道如何处理异常，因此只能声明抛出该异常，而这种情况又是如此普遍，所以 Checked 异常降低了程序开发的生产率和代码的执行效率。关于 Checked 异常的优劣，在 Java 领域是一个备受争论的问题。

10.3.1 使用 throws 声明抛出异常

使用 throws 声明抛出异常的思路是，当前方法不知道如何处理这种类型的异常，该异常应该由上一级调用者处理；如果 main 方法也不知道如何处理这种类型的异常，也可以使用 throws 声明抛出异常，该异常将交给 JVM 处理。JVM 对异常的处理方法是，打印异常的跟踪栈信息，并中止程序运行，这就是前面程序在遇到异常后自动结束的原因。

前面章节里有些程序已经用到了 throws 声明抛出，throws 声明抛出只能在方法签名中使用，throws 可以声明抛出多个异常类，多个异常类之间以逗号隔开。throws 声明抛出的语法格式如下：

```
throws ExceptionClass1 , ExceptionClass2...
```

上面 throws 声明抛出的语法格式仅跟在方法签名之后，如下例子程序使用了 throws 来声明抛出 IOException 异常，一旦使用 throws 语句声明抛出该异常，程序就无须使用 try...catch 块来捕获该异常了。

程序清单：codes\10\10.3\ThrowsTest.java

```
public class ThrowsTest
{
    public static void main(String[] args)
        throws IOException
    {
        FileInputStream fis = new FileInputStream("a.txt");
    }
}
```

上面程序声明不处理 IOException 异常，将该异常交给 JVM 处理，所以程序一旦遇到该异常，JVM 就会打印该异常的跟踪栈信息，并结束程序。运行上面程序，会看到如图 10.4 所示的运行结果。

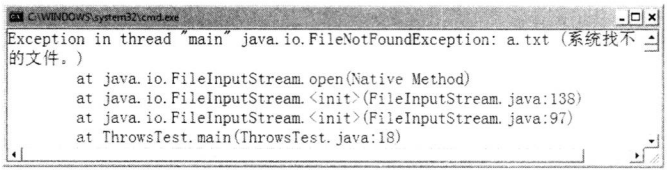


图 10.4 main 方法声明把异常交给 JVM 处理

如果某段代码中调用了一个带 throws 声明的方法，该方法声明抛出了 Checked 异常，则表明该方法希望它的调用者来处理该异常。也就是说，调用该方法时要么放在 try 块中显式捕获该异常，要么放在另一个带 throws 声明抛出的方法中。如下例子程序示范了这种用法。

程序清单：codes\10\10.3/ThrowsTest2.java

```
public class ThrowsTest2
{
    public static void main(String[] args)
        throws Exception
    {
        // 因为 test()方法声明抛出 IOException 异常
        // 所以调用该方法的代码要么处于 try...catch 块中，
        // 要么处于另一个带 throws 声明抛出的方法中
        test();
    }
    public static void test()throws IOException
    {
        // 因为 FileInputStream 的构造器声明抛出 IOException 异常
        // 所以调用 FileInputStream 的代码要么处于 try...catch 块中
        // 要么处于另一个带 throws 声明抛出的方法中
```

```
        FileInputStream fis = new FileInputStream("a.txt");
    }
}
```

10.3.2 方法重写时声明抛出异常的限制

使用 throws 声明抛出异常时有一个限制，就是方法重写时“两小”中的一条规则：子类方法声明抛出的异常类型应该是父类方法声明抛出的异常类型的子类或相同，子类方法声明抛出的异常不允许比父类方法声明抛出的异常多。看如下程序。

程序清单：codes\10\10.3\OverrideThrows.java

```
public class OverrideThrows
{
    public void test()throws IOException
    {
        FileInputStream fis = new FileInputStream("a.txt");
    }
}
class Sub extends OverrideThrows
{
    // 子类方法声明抛出了比父类方法更大的异常
    // 所以下面方法出错
    public void test()throws Exception
    {
    }
}
```

上面程序中 Sub 子类中的 test()方法声明抛出 Exception，该 Exception 是其父类声明抛出异常 IOException 类的父类，这将导致程序无法通过编译。

由此可见，使用 Checked 异常至少存在如下两大不便之处。

- 对于程序中的 Checked 异常，Java 要求必须显式捕获并处理该异常，或者显式声明抛出该异常。这样就增加了编程复杂度。
- 如果在方法中显式声明抛出 Checked 异常，将会导致方法签名与异常耦合，如果该方法是重写父类的方法，则该方法抛出的异常还会受到被重写方法所抛出异常的限制。

在大部分时候推荐使用 Runtime 异常，而不使用 Checked 异常。尤其当程序需要自行抛出异常时(如何自行抛出异常请看下一节)，使用 Runtime 异常将更加简洁。

当使用 Runtime 异常时，程序无须在方法中声明抛出 Checked 异常，一旦发生了自定义错误，程序只管抛出 Runtime 异常即可。

如果程序需要在合适的地方捕获异常并对异常进行处理，则一样可以使用 try...catch 块来捕获 Runtime 异常。

使用 Runtime 异常是比较省事的方式，使用这种方式既可以享受“正常代码和错误处理代码分离”，“保证程序具有较好的健壮性”的优势，又可以避免因为使用 Checked 异常带来的编程烦琐性。因此，C#、Ruby、Python 等语言没有所谓的 Checked 异常，所有的异常都是 Runtime 异常。

但 Checked 异常也有其优势——Checked 异常能在编译时提醒程序员代码可能存在的问题，提醒程序员必须注意处理该异常，或者声明该异常由该方法调用者来处理，从而可以避免程序员因为粗心而忘记处理该异常的错误。

10.4 使用 throw 抛出异常

当程序出现错误时，系统会自动抛出异常；除此之外，Java 也允许程序自行抛出异常，自行抛出异常使用 throw 语句来完成（注意此处的 throw 没有后面的 s，与前面声明抛出的 throws 是有区别的）。

10.4.1 抛出异常

异常是一种很“主观”的说法，以下雨为例，假设大家约好明天去爬山郊游，如果第二天下雨了，这种情况会打破既定计划，就属于一种异常；但对于正在期盼天降甘霖的农民而言，如果第二天下雨了，

他们正好随雨追肥，这就完全正常。

很多时候，系统是否要抛出异常，可能需要根据应用的业务需求来决定，如果程序中的数据、执行与既定的业务需求不符，这就是一种异常。由于与业务需求不符而产生的异常，必须由程序员来决定抛出，系统无法抛出这种异常。

如果需要在程序中自行抛出异常，则应使用 throw 语句，throw 语句可以单独使用，throw 语句抛出的不是异常类，而是一个异常实例，而且每次只能抛出一个异常实例。throw 语句的语法格式如下：

```
throw ExceptionInstance;
```

可以利用 throw 语句再次改写前面五子棋游戏中处理用户输入的代码：

```
try
{
    // 将用户输入的字符串以逗号（,）作为分隔符，分隔成两个字符串
    String[] posStrArr = inputStr.split(",");
    // 将两个字符串转换成用户下棋的坐标
    int xPos = Integer.parseInt(posStrArr[0]);
    int yPos = Integer.parseInt(posStrArr[1]);
    // 如果用户试图下棋的坐标点已经有棋了，程序自行抛出异常
    if (!gb.board[xPos - 1][yPos - 1].equals("╋"))
    {
        throw new Exception("您试图下棋的坐标点已经有棋了");
    }
    // 把对应的数组元素赋为"●"
    gb.board[xPos - 1][yPos - 1] = "●";
}
catch (Exception e)
{
    System.out.println("您输入的坐标不合法，请重新输入，下棋坐标应以 x,y 的格式：");
    continue;
}
```

上面程序中粗体字代码使用 throw 语句来自行抛出异常，程序认为当用户试图向一个已有棋子的坐标点下棋就是异常。当 Java 运行时接收到开发者自行抛出的异常时，同样会中止当前的执行流，跳到该异常对应的 catch 块，由该 catch 块来处理该异常。也就是说，不管是系统自动抛出的异常，还是程序员手动抛出的异常，Java 运行时环境对异常的处理没有任何差别。

如果 throw 语句抛出的异常是 Checked 异常，则该 throw 语句要么处于 try 块里，显式捕获该异常，要么放在一个带 throws 声明抛出的方法中，即把该异常交给该方法的调用者处理；如果 throw 语句抛出的异常是 Runtime 异常，则该语句无须放在 try 块里，也无须放在带 throws 声明抛出的方法中；程序既可以显式使用 try...catch 来捕获并处理该异常，也可以完全不理会该异常，把该异常交给该方法调用者处理。例如下面例子程序。

程序清单：codes\10\10.4\ThrowTest.java

```
public class ThrowTest
{
    public static void main(String[] args)
    {
        try
        {
            // 调用声明抛出 Checked 异常的方法，要么显式捕获该异常
            // 要么在 main 方法中再次声明抛出
            throwChecked(-3);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
        // 调用声明抛出 Runtime 异常的方法既可以显式捕获该异常
        // 也可不理会该异常
        throwRuntime(3);
    }
    public static void throwChecked(int a)throws Exception
    {
```

```
        if (a > 0)
        {
            // 自行抛出 Exception 异常
            // 该代码必须处于 try 块里，或处于带 throws 声明的方法中
            throw new Exception("a 的值大于 0，不符合要求");
        }
    }
    public static void throwRuntime(int a)
    {
        if (a > 0)
        {
            // 自行抛出 RuntimeException 异常，既可以显式捕获该异常
            // 也可完全不理会该异常，把该异常交给该方法调用者处理
            throw new RuntimeException("a 的值大于 0，不符合要求");
        }
    }
}
```

通过上面程序也可以看出，自行抛出 Runtime 异常比自行抛出 Checked 异常的灵活性更好。同样，抛出 Checked 异常则可以让编译器提醒程序员必须处理该异常。

10.4.2 自定义异常类

在通常情况下，程序很少会自行抛出系统异常，因为异常的类名通常也包含了该异常的有用信息。所以在选择抛出异常时，应该选择合适的异常类，从而可以明确地描述该异常情况。在这种情形下，应用程序常常需要抛出自定义异常。

用户自定义异常都应该继承 Exception 基类，如果希望自定义 Runtime 异常，则应该继承 RuntimeException 基类。定义异常类时通常需要提供两个构造器：一个是无参数的构造器；另一个是带一个字符串参数的构造器，这个字符串将作为该异常对象的描述信息（也就是异常对象的 getMessage() 方法的返回值）。

下面例子程序创建了一个自定义异常类。

程序清单：codes\10\10.4\AuctionException.java

```
public class AuctionException extends Exception
{
    // 无参数的构造器
    public AuctionException(){}        // ①
    // 带一个字符串参数的构造器
    public AuctionException(String msg)    // ②
    {
        super(msg);
    }
}
```

上面程序创建了 AuctionException 异常类，并为该异常类提供了两个构造器。尤其是②号粗体字代码部分创建的带一个字符串参数的构造器，其执行体也非常简单，仅通过 super 来调用父类的构造器，正是这行 super 调用可以将此字符串参数传给异常对象的 message 属性，该 message 属性就是该异常对象的详细描述信息。

如果需要自定义 Runtime 异常，只需将 AuctionException.java 程序中的 Exception 基类改为 RuntimeException 基类，其他地方无须修改。

提示：

在大部分情况下，创建自定义异常都可采用与 AuctionException.java 相似的代码完成，只需改变 AuctionException 异常的类名即可，让该异常类的类名可以准确描述该异常。

10.4.3 catch 和 throw 同时使用

前面介绍的异常处理方式有如下两种。

- 在出现异常的方法内捕获并处理异常，该方法的调用者将不能再次捕获该异常。
- 该方法签名中声明抛出该异常，将该异常完全交给方法调用者处理。

在实际应用中往往需要更复杂的处理方式——当一个异常出现时，单靠某个方法无法完全处理该异常，必须由几个方法协作才可完全处理该异常。也就是说，在异常出现的当前方法中，程序只对异常进行部分处理，还有些处理需要在该方法的调用者中才能完成，所以应该再次抛出异常，让该方法的调用者也能捕获到异常。

为了实现这种通过多个方法协作处理同一个异常的情形，可以在catch块中结合throw语句来完成。如下例子程序示范了这种catch和throw同时使用的方法。

程序清单：codes\10\10.4\AuctionTest.java

```
public class AuctionTest
{
    private double initPrice = 30.0;
    // 因为该方法中显式抛出了AuctionException异常
    // 所以此处需要声明抛出AuctionException异常
    public void bid(String bidPrice)
        throws AuctionException
    {
        double d = 0.0;
        try
        {
            d = Double.parseDouble(bidPrice);
        }
        catch (Exception e)
        {
            // 此处完成本方法中可以对异常执行的修复处理
            // 此处仅仅是在控制台打印异常的跟踪栈信息
            e.printStackTrace();
            // 再次抛出自定义异常
            throw new AuctionException("竞拍价必须是数值，"
                + "不能包含其他字符！");
        }
        if (initPrice > d)
        {
            throw new AuctionException("竞拍价比起拍价低，"
                + "不允许竞拍！");
        }
        initPrice = d;
    }
    public static void main(String[] args)
    {
        AuctionTest at = new AuctionTest();
        try
        {
            at.bid("df");
        }
        catch (AuctionException ae)
        {
            // 再次捕获到bid()方法中的异常，并对该异常进行处理
            System.err.println(ae.getMessage());
        }
    }
}
```

上面程序中粗体字代码对应的catch块捕获到异常后，系统打印了该异常的跟踪栈信息，接着抛出一个AuctionException异常，通知该方法的调用者再次处理该AuctionException异常。所以程序中的main方法，也就是bid()方法调用者还可以再次捕获AuctionException异常，并将该异常的详细描述信息输出到标准错误输出。

提示：

这种catch和throw结合使用的情况在大型企业级应用中非常常用。企业级应用对异常的处理通常分成两个部分：① 应用后台需要通过日志来记录异常发生的详细情况；② 应

用还需要根据异常向应用使用者传达某种提示。在这种情形下，所有异常都需要两个方法共同完成，也就必须将 catch 和 throw 结合使用。

10.4.4 Java 7 增强的 throw 语句

对于如下代码：

```
try
{
    new FileOutputStream("a.txt");
}
catch (Exception ex)
{
    ex.printStackTrace();
    throw ex;          // ①
}
```

上面代码片段中的粗体字代码再次抛出了捕获到的异常，但这个 ex 对象的情况比较特殊：程序捕获该异常时，声明该异常的类型为 Exception；但实际上 try 块中可能只调用了 FileOutputStream 构造器，这个构造器声明只是抛出了 FileNotFoundException 异常。

在 Java 7 以前，Java 编译器的处理“简单而粗暴”——由于在捕获该异常时声明 ex 的类型是 Exception，因此 Java 编译器认为这段代码可能抛出 Exception 异常，所以包含这段代码的方法通常需要声明抛出 Exception 异常。例如如下方法。

程序清单：codes\10\10.4\ThrowTest2.java

```
public class ThrowTest2
{
    public static void main(String[] args)
        // Java 6 认为①号代码可能抛出 Exception 异常
        // 所以此处声明抛出 Exception 异常
        throws Exception
    {
        try
        {
            new FileOutputStream("a.txt");
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw ex;          // ①
        }
    }
}
```

从 Java 7 开始，Java 编译器会执行更细致的检查，Java 编译器会检查 throw 语句抛出异常的实际类型，这样编译器知道①号代码处实际上只可能抛出 FileNotFoundException 异常，因此在方法签名中只要声明抛出 FileNotFoundException 异常即可。即可以将代码改为如下形式（程序清单同上）。

```
public class ThrowTest2
{
    public static void main(String[] args)
        // Java 7 会检查①号代码处可能抛出异常的实际类型
        // 因此此处只需声明抛出 FileNotFoundException 异常即可
        throws FileNotFoundException
    {
        try
        {
            new FileOutputStream("a.txt");
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
```

```
            throw ex;        // ①
        }
    }
}
```

10.4.5 异常链

对于真实的企业级应用而言，常常有严格的分层关系，层与层之间有非常清晰的划分，上层功能的实现严格依赖于下层的 API，也不会跨层访问。图 10.5 显示了这种具有分层结构应用的大致示意图。

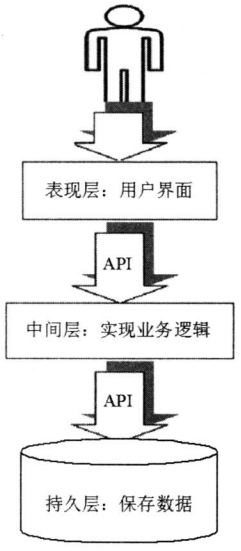


图 10.5 分层结构示意图

对于一个采用图 10.5 所示结构的应用，当业务逻辑层访问持久层出现 SQLException 异常时，程序不应该把底层的 SQLException 异常传到用户界面，有如下两个原因。

- 对于正常用户而言，他们不想看到底层 SQLException 异常，SQLException 异常对他们使用该系统没有任何帮助。
- 对于恶意用户而言，将 SQLException 异常暴露出来不安全。

把底层的原始异常直接传给用户是一种不负责任的表现。通常的做法是：程序先捕获原始异常，然后抛出一个新的业务异常，新的业务异常中包含了对用户的提示信息，这种处理方式被称为异常转译。假设程序需要实现工资计算的方法，则程序应该采用如下结构的代码来实现该方法。

```
public void calSal() throws SalException
{
    try
    {
        // 实现结算工资的业务逻辑
        ...
    }
    catch(SQLException sqle)
    {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 message 就是对用户的提示
        throw new SalException("访问底层数据库出现异常");
    }
    catch(Exception e)
    {
        // 把原始异常记录下来，留给管理员
        ...
        // 下面异常中的 message 就是对用户的提示
        throw new SalException("系统出现未知异常");
    }
}
```

这种把原始异常信息隐藏起来，仅向上提供必要的异常提示信息的处理方式，可以保证底层异常不会扩散到表现层，可以避免向上暴露太多的实现细节，这完全符合面向对象的封装原则。

这种把捕获一个异常然后接着抛出另一个异常，并把原始异常信息保存下来是一种典型的链式处理（23 种设计模式之一：职责链模式），也被称为“异常链”。

在 JDK 1.4 以前，程序员必须自己编写代码来保持原始异常信息。从 JDK 1.4 以后，所有 Throwable 的子类在构造器中都可以接收一个 cause 对象作为参数。这个 cause 就用来表示原始异常，这样可以把原始异常传递给新的异常，使得即使在当前位置创建并抛出了新的异常，你也能通过这个异常链追踪到异常最初发生的位置。例如希望通过上面的 SalException 去追踪到最原始的异常信息，则可以将该方法改写为如下形式。

```
public void calSal() throws SalException
{
    try
    {
```

```
            // 实现结算工资的业务逻辑
            ...
        }
        catch(SQLException sqle)
        {
            // 把原始异常记录下来，留给管理员
            ...
            // 下面异常中的 sqle 就是原始异常
            throw new SalException(sqle);
        }
        catch(Exception e)
        {
            // 把原始异常记录下来，留给管理员
            ...
            // 下面异常中的 e 就是原始异常
            throw new SalException(e);
        }
    }
```

上面程序中粗体字代码创建 SalException 对象时，传入了一个 Exception 对象，而不是传入了一个 String 对象，这就需要 SalException 类有相应的构造器。从 JDK 1.4 以后，Throwable 基类已有了一个可以接收 Exception 参数的方法，所以可以采用如下代码来定义 SalException 类。

程序清单：codes\10\10.4\SalException.java

```
public class SalException extends Exception
{
    public SalException(){}
    public SalException(String msg)
    {
        super(msg);
    }
    // 创建一个可以接收 Throwable 参数的构造器
    public SalException(Throwable t)
    {
        super(t);
    }
}
```

创建了这个 SalException 业务异常类后，就可以用它来封装原始异常，从而实现对异常的链式处理。

10.5 Java 的异常跟踪栈

异常对象的 printStackTrace()方法用于打印异常的跟踪栈信息，根据 printStackTrace()方法的输出结果，开发者可以找到异常的源头，并跟踪到异常一路触发的过程。

看下面用于测试 printStackTrace 的例子程序。

程序清单：codes\10\10.5\PrintStackTraceTest.java

```
class SelfException extends RuntimeException
{
    SelfException(){}
    SelfException(String msg)
    {
        super(msg);
    }
}
public class PrintStackTraceTest
{
    public static void main(String[] args)
    {
        firstMethod();
    }
    public static void firstMethod()
    {
        secondMethod();
    }
```

```
    public static void secondMethod()
    {
        thirdMethod();
    }
    public static void thirdMethod()
    {
        throw new SelfException("自定义异常信息");
    }
}
```

上面程序中 main 方法调用 firstMethod，firstMethod 调用 secondMethod，secondMethod 调用 thirdMethod，thirdMethod 直接抛出一个 SelfException 异常。运行上面程序，会看到如图 10.6 所示的结果。

从图 10.6 中可以看出，异常从 thirdMethod 方法开始触发，传到 secondMethod 方法，再传到 firstMethod 方法，最后传到 main 方法，在 main 方法终止，这个过程就是 Java 的异常跟踪栈。

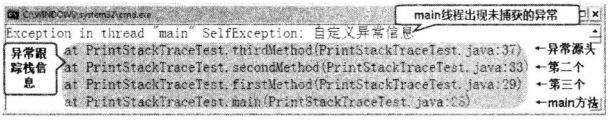


图 10.6 异常的跟踪栈信息

在面向对象的编程中，大多数复杂操作都会被分解成一系列方法调用。这是因为：实现更好的可重用性，将每个可重用的代码单元定义成方法，将复杂任务逐渐分解为更易管理的小型子任务。由于一个大的业务功能需要由多个对象来共同实现，在最终编程模型中，很多对象将通过一系列方法调用来实现通信，执行任务。

所以，面向对象的应用程序运行时，经常会发生一系列方法调用，从而形成“方法调用栈”，异常的传播则相反：只要异常没有被完全捕获（包括异常没有被捕获，或异常被处理后重新抛出了新异常），异常从发生异常的方法逐渐向外传播，首先传给该方法的调用者，该方法调用者再次传给其调用者……直至最后传到 main 方法，如果 main 方法依然没有处理该异常，JVM 会中止该程序，并打印异常的跟踪栈信息。

很多初学者一看到如图 10.6 所示的异常提示信息，就会惊慌失措，其实图 10.6 所示的异常跟踪栈信息非常清晰——它记录了应用程序中执行停止的各个点。

第一行的信息详细显示了异常的类型和异常的详细消息。

接下来跟踪栈记录程序中所有的异常发生点，各行显示被调用方法中执行的停止位置，并标明类、类中的方法名、与故障点对应的文件的行。一行行地往下看，跟踪栈总是最内部的被调用方法逐渐上传，直到最外部业务操作的起点，通常就是程序的入口 main 方法或 Thread 类的 run 方法（多线程的情形）。

下面例子程序示范了多线程程序中发生异常的情形。

程序清单：codes\10\10.5\ThreadExceptionTest.java

```
public class ThreadExceptionTest implements Runnable
{
    public void run()
    {
        firstMethod();
    }
    public void firstMethod()
    {
        secondMethod();
    }
    public void secondMethod()
    {
        int a = 5;
        int b = 0;
        int c = a / b;
    }
    public static void main(String[] args)
    {
        new Thread(new ThreadExceptionTest()).start();
    }
}
```

提示：

关于多线程的知识，请参考本书第 16 章的内容。

运行上面程序，会看到如图 10.7 所示的运行结果。

从图 10.7 中可以看出，程序在 Thread 的 run 方法中出现了 ArithmeticException 异常，这个异常的源头是 ThreadExcetpionTest 的 secondMethod 方法，位于 ThreadExcetpionTest.java 文件的 27 行。这个异常传播到 Thread 类的 run 方法就会结束（如果该异常没有得到处理，将会导致该线程中止运行）。

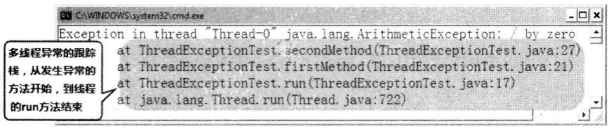


图 10.7　多线程的异常跟踪栈

前面已经讲过，调用 Exception 的 printStackTrace()方法就是打印该异常的跟踪栈信息，也就会看到如图 10.6、图 10.7 所示的信息。当然，如果方法调用的层次很深，将会看到更加复杂的异常跟踪栈。

提示：

虽然 printStackTrace()方法可以很方便地用于追踪异常的发生情况，可以用它来调试程序，但在最后发布的程序中，应该避免使用它；而应该对捕获的异常进行适当的处理，而不是简单地将异常的跟踪栈信息打印出来。

10.6　异常处理规则

前面介绍了使用异常处理的优势、便捷之处，本节将进一步从程序性能优化、结构优化的角度给出异常处理的一般规则。成功的异常处理应该实现如下 4 个目标。

- 使程序代码混乱最小化。
- 捕获并保留诊断信息。
- 通知合适的人员。
- 采用合适的方式结束异常活动。

下面介绍达到这种效果的基本准则。

10.6.1　不要过度使用异常

不可否认，Java 的异常机制确实方便，但滥用异常机制也会带来一些负面影响。过度使用异常主要有两个方面。

- 把异常和普通错误混淆在一起，不再编写任何错误处理代码，而是以简单地抛出异常来代替所有的错误处理。
- 使用异常处理来代替流程控制。

熟悉了异常使用方法后，程序员可能不再愿意编写烦琐的错误处理代码，而是简单地抛出异常。实际上这样做是不对的，对于完全已知的错误，应该编写处理这种错误的代码，增加程序的健壮性；对于普通的错误，应该编写处理这种错误的代码，增加程序的健壮性。只有对外部的、不能确定和预知的运行时错误才使用异常。

对比前面五子棋游戏中，处理用户输入坐标点已有棋子的两种方式。

```
// 如果用户试图下棋的坐标点已有棋子了
if (!gb.board[xPos - 1][yPos - 1].equals("╋"))
{
    System.out.println("您输入的坐标点已有棋子了，请重新输入");
    continue;
}
```

上面这种处理方式检测到用户试图下棋的坐标点已经有棋子了，立即打印一条提示语句，并重新开始下一次循环。这种处理方式简洁明了，逻辑清晰。程序的运行效率也很好——程序进入 if 块后，即结束了本次循环。

如果将上面的处理机制改为如下方式：

```
// 如果用户试图下棋的坐标点已经有棋子了，程序自行抛出异常
if (!gb.board[xPos - 1][yPos - 1].equals("╋"))
{
    throw new Exception("您试图下棋的坐标点已经有棋子了");
}
```

上面的处理方式没有提供有效的错误处理代码，当程序检测到用户试图下棋的坐标点已经有棋子时，并没有提供相应的处理，而是简单地抛出了一个异常。这种处理方式虽然简单，但 Java 运行时接收到这个异常后，还需要进入相应的 catch 块来捕获该异常，所以运行效率要差一些。而且用户下棋重复这个错误完全是预料的，所以程序完全可以针对该错误提供相应的处理，而不是抛出异常。

必须指出：异常处理机制的初衷是将不可预期异常的处理代码和正常的业务逻辑处理代码分离，因此绝不要使用异常处理来代替正常的业务逻辑判断。

另外，异常机制的效率比正常的流程控制效率差，所以不要使用异常处理来代替正常的程序流程控制。例如，对于如下代码：

```
// 定义一个字符串数组
String[] arr = {"Hello" , "Java" , "Spring"};
// 使用异常处理来遍历 arr 数组的每个元素
try
{
    int i = 0;
    while(true)
    {
        System.out.println(arr[i++]);
    }
}
catch(ArrayIndexOutOfBoundsException ae)
{
}
```

运行上面程序确实可以实现遍历 arr 数组元素的功能，但这种写法可读性较差，而且运行效率也不高。程序完全有能力避免产生 ArrayIndexOutOfBoundsException 异常，程序“故意”制造这种异常，然后使用 catch 块去捕获该异常，这是不应该的。将程序改为如下形式肯定要好得多：

```
String[] arr = {"Hello" , "Java" , "Spring"};
for (int i = 0; i < arr.length; i++ )
{
    System.out.println(arr[i]);
}
```

注意：

异常只应该用于处理非正常的情况，不要使用异常处理来代替正常的流程控制。对于一些完全可预知，而且处理方式清楚的错误，程序应该提供相应的错误处理代码，而不是将其笼统地称为异常。

10.6.2 不要使用过于庞大的 try 块

很多初学异常机制的读者喜欢在 try 块里放置大量的代码，在一个 try 块里放置大量的代码看上去“很简单”，但这种“简单”只是一种假象，只是在编写程序时看上去比较简单。但因为 try 块里的代码过于庞大，业务过于复杂，就会造成 try 块中出现异常的可能性大大增加，从而导致分析异常原因的难度也大大增加。

而且当 try 块过于庞大时，就难免在 try 块后紧跟大量的 catch 块才可以针对不同的异常提供不同的处理逻辑。同一个 try 块后紧跟大量的 catch 块则需要分析它们之间的逻辑关系，反而增加了编程复杂度。

正确的做法是，把大块的 try 块分割成多个可能出现异常的程序段落，并把它们放在单独的 try 块中，从而分别捕获并处理异常。

10.6.3 避免使用 Catch All 语句

所谓 Catch All 语句指的是一种异常捕获模块，它可以处理程序发生的所有可能异常。例如，如下代码片段：

```
try
{
    // 可能引发 Checked 异常的代码
}
catch (Throwable t)
{
    // 进行异常处理
    t.printStackTrace();
}
```

不可否认，每个程序员都曾经用过这种异常处理方式；但在编写关键程序时就应避免使用这种异常处理方式。这种处理方式有如下两点不足之处。

- 所有的异常都采用相同的处理方式，这将导致无法对不同的异常分情况处理，如果要分情况处理，则需要在 catch 块中使用分支语句进行控制，这是得不偿失的做法。
- 这种捕获方式可能将程序中的错误、Runtime 异常等可能导致程序终止的情况全部捕获到，从而“压制”了异常。如果出现了一些“关键”异常，那么此异常也会被“静悄悄”地忽略。

实际上，Catch All 语句不过是一种通过避免错误处理而加快编程进度的机制，应尽量避免在实际应用中使用这种语句。

10.6.4 不要忽略捕获到的异常

不要忽略异常！既然已捕获到异常，那 catch 块理应做些有用的事情——处理并修复这个错误。catch 块整个为空，或者仅仅打印出错信息都是不妥的！

catch 块为空就是假装不知道甚至瞒天过海，这是最可怕的事情——程序出了错误，所有的人都看不到任何异常，但整个应用可能已经彻底坏了。仅在 catch 块里打印错误跟踪栈信息稍微好一点，但仅仅比空白多了几行异常信息。通常建议对异常采取适当措施，比如：

- 处理异常。对异常进行合适的修复，然后绕过异常发生的地方继续执行；或者用别的数据进行计算，以代替期望的方法返回值；或者提示用户重新操作……总之，对于 Checked 异常，程序应该尽量修复。
- 重新抛出新异常。把当前运行环境下能做的事情尽量做完，然后进行异常转译，把异常包装成当前层的异常，重新抛出给上层调用者。
- 在合适的层处理异常。如果当前层不清楚如何处理异常，就不要在当前层使用 catch 语句来捕获该异常，直接使用 throws 声明抛出该异常，让上层调用者来负责处理该异常。

10.7 本章小结

本章主要介绍了 Java 异常处理机制的相关知识，Java 的异常处理主要依赖于 try、catch、finally、throw 和 throws 5 个关键字，本章详细讲解了这 5 个关键字的用法。本章还介绍了 Java 异常类之间的继承关系，并介绍了 Checked 异常和 Runtime 异常之间的区别。本章也详细介绍了 Java 7 对异常处理的增强。本章还详细讲解了实际开发中最常用的异常链和异常转译。本章最后从优化程序的角度，给出了实际应用中处理异常的几条基本规则。

本章练习

1. 改写第 4 章的五子棋游戏程序，为该程序增加异常处理机制，让程序更加健壮。
2. 改写第 8 章的梭哈游戏程序，为该程序增加异常处理机制。

CHAPTER

11

第11章 AWT编程

本章要点

- 图形用户界面编程的概念
- AWT 的概念
- AWT 容器和常见布局管理器
- 使用 AWT 基本组件
- 使用对话框
- 使用文件对话框
- Java 的事件机制
- 事件源、事件、事件监听器的关系
- 使用菜单条、菜单、菜单项创建菜单
- 创建并使用右键菜单
- 重写 paint()方法实现绘图
- 使用 Graphics 类
- 使用 BufferedImage 和 ImageIO 处理位图
- 使用剪贴板
- 剪贴板数据风格
- 拖放功能
- 拖放目标与拖放源

本章和下一章的内容会比较“有趣”，因为可以看到非常熟悉的窗口、按钮、动画等效果，而这些图形界面元素不仅会让开发者感到更“有趣”，对最终用户也是一种诱惑，用户总是喜欢功能丰富、操作简单的应用，图形用户界面的程序就可以满足用户的这种渴望。

Java 使用 AWT 和 Swing 类完成图形用户界面编程，其中 AWT 的全称是抽象窗口工具集（Abstract Window Toolkit），它是 Sun 最早提供的 GUI 库，这个 GUI 库提供了一些基本功能，但这个 GUI 库的功能比较有限，所以后来又提供了 Swing 库。通过使用 AWT 和 Swing 提供的图形界面组件库，Java 的图形用户界面编程非常简单，程序只要依次创建所需的图形组件，并以合适的方式将这些组件组织在一起，就可以开发出非常美观的用户界面。

程序以一种“搭积木”的方式将这些图形用户组件组织在一起，就是实际可用的图形用户界面，但这些图形用户界面还不能与用户交互，为了实现图形用户界面与用户交互操作，还应为程序提供事件处理，事件处理负责让程序可以响应用户动作。

通过学习本章，读者应该能开发出简单的图形用户界面应用，并提供相应的事件响应机制。本章也会介绍 Java 中的图形处理、剪贴板操作等知识。

11.1 Java 9 改进的 GUI（图形用户界面）和 AWT

前面介绍的所有程序都是基于命令行的，基于命令行的程序可能只有一些“专业”的计算机人士才会使用。例如前面编写的五子棋、梭哈等程序，恐怕只有程序员自己才愿意玩这么“糟糕”的游戏，很少有最终用户愿意对着黑糊糊的命令行界面敲命令。

相反，如果为程序提供直观的图形用户界面（Graphics User Interface，GUI），最终用户通过鼠标拖动、单击等动作就可以操作整个应用，整个应用程序就会受欢迎得多（实际上，Windows 之所以广为人知，其最初的吸引力就是来自于它所提供的图形用户界面）。作为一个程序设计者，必须优先考虑用户的感受，一定要让用户感到“爽”，程序才会被需要、被使用，这样的程序才有价值。

当 JDK 1.0 发布时，Sun 提供了一套基本的 GUI 类库，这个 GUI 类库希望可以在所有平台下都能运行，这套基本类库被称为“抽象窗口工具集（Abstract Window Toolkit）”，它为 Java 应用程序提供了基本的图形组件。AWT 是窗口框架，它从不同平台的窗口系统中抽取出共同组件，当程序运行时，将这些组件的创建和动作委托给程序所在的运行平台。简而言之，当使用 AWT 编写图形界面应用时，程序仅指定了界面组件的位置和行为，并未提供真正的实现，JVM 调用操作系统本地的图形界面来创建和平台一致的对等体。

使用AWT创建的图形界面应用和所在的运行平台有相同的界面风格，比如在Windows操作系统上，它就表现出 Windows 风格；在 UNIX 操作系统上，它就表现出 UNIX 风格。Sun 希望采用这种方式来实现“Write Once，Run Anywhere”的目标。

但在实际应用中，AWT 出现了如下几个问题。

- 使用 AWT 做出的图形用户界面在所有的平台上都显得很丑陋，功能也非常有限。
- AWT 为了迎合所有主流操作系统的界面设计，AWT 组件只能使用这些操作系统上图形界面组件的交集，所以不能使用特定操作系统上复杂的图形界面组件，最多只能使用 4 种字体。
- AWT 用的是非常笨拙的、非面向对象的编程模式。

1996 年，Netscape 公司开发了一套工作方式完全不同的 GUI 库，简称为 IFC（Internet Foundation Classes），这套 GUI 库的所有图形界面组件，例如文本框、按钮等都是绘制在空白窗口上的，只有窗口本身需要借助于操作系统的窗口实现。IFC 真正实现了各种平台上的界面一致性。不久，Sun 和 Netscape 合作完善了这种方法，并创建了一套新的用户界面库：Swing。AWT、Swing、辅助功能 API、2D API 以及拖放 API 共同组成了 JFC（Java Foundation Classes，Java 基础类库），其中 Swing 组件全面替代了 Java 1.0 中的 AWT 组件，但保留了 Java 1.1 中的 AWT 事件模型。总体上，AWT 是图形用户界面编程的基础，Swing 组件替代了绝大部分 AWT 组件，对 AWT 图形用户界面编程有极好的补充和加强。

Java 9 的 AWT 和 Swing 组件可以自适应高分辨率屏。在 Java 9 之前，如果使用高分辨率屏，由于这种屏幕的像素密度可能是传统显示设备的 2~3 倍（即单位面积里显示像素更多），而 AWT 和 Swing 组件都是基于屏幕像素计算大小的，因此这些组件在高分辨率屏上比较小。

Java 9 对此进行了改进，如果 AWT 或 Swing 组件在高分辨率屏上显示，那么组件的大小可能会以实际屏幕的 2 个或 3 个像素作为“逻辑像素”，这样就可保证 AWT 或 Swing 组件在高分辨率屏上也具有正常大小。另外，Java 9 也支持 OS X 设备的视网膜屏。

简而言之，Java 9 改进后的 AWT 或 Swing 组件完全可以在高分辨率屏、视网膜屏上具有正常大小。

提示：

Swing 并没有完全替代 AWT，而是建立在 AWT 基础之上，Swing 仅提供了能力更强大的用户界面组件，即使是完全采用 Swing 编写的 GUI 程序，也依然需要使用 AWT 的事件处理机制。本章主要介绍 AWT 组件，这些 AWT 组件在 Swing 里将有对应的实现，二者用法基本相似，下一章会有更详细的介绍。

所有和 AWT 编程相关的类都放在 java.awt 包以及它的子包中，AWT 编程中有两个基类：Component 和 MenuComponent。图 11.1 显示了 AWT 图形组件之间的继承关系。

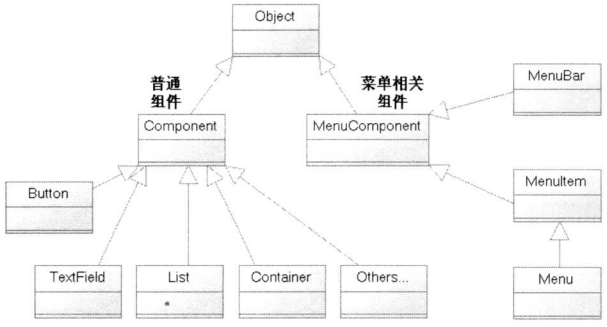


图 11.1　AWT 图形组件之间的继承关系

在 java.awt 包中提供了两种基类表示图形界面元素：Component 和 MenuComponent，其中 Component 代表一个能以图形化方式显示出来，并可与用户交互的对象，例如 Button 代表一个按钮，TextField 代表一个文本框等；而 MenuComponent 则代表图形界面的菜单组件，包括 MenuBar（菜单条）、MenuItem（菜单项）等子类。

除此之外，AWT 图形用户界面编程里还有两个重要的概念：Container 和 LayoutManager，其中 Container 是一种特殊的 Component，它代表一种容器，可以盛装普通的 Component；而 LayoutManager 则是容器管理其他组件布局的方式。

11.2　AWT 容器

如果从程序员的角度来看一个窗口时，这个窗口不是一个整体（有点庖丁解牛的感觉），而是由多个部分组合而成的，如图 11.2 所示。

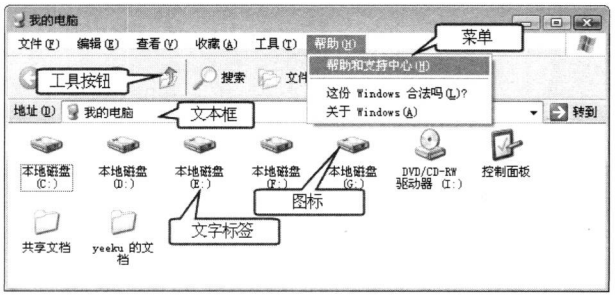


图 11.2　窗口的“分解”

从图 11.2 中可以看出，任何窗口都可被分解成一个空的容器，容器里盛装了大量的基本组件，通过设置这些基本组件的大小、位置等属性，就可以将该空的容器和基本组件组成一个整体的窗口。实际上，图形界面编程非常简单，它非常类似于小朋友玩的拼图游戏，容器类似于拼图的“母板”，而普通组件（如 Button、List 之类）则类似于拼图的图块。创建图形用户界面的过程就是完成拼图的过程。

容器（Container）是 Component 的子类，因此容器对象本身也是一个组件，具有组件的所有性质，可以调用 Component 类的所有方法。Component 类提供了如下几个常用方法来设置组件的大小、位置和可见性等。

- setLocation(int x, int y)：设置组件的位置。
- setSize(int width, int height)：设置组件的大小。
- setBounds(int x, int y, int width, int height)：同时设置组件的位置、大小。

- setVisible(Boolean b)：设置该组件的可见性。

容器还可以盛装其他组件，容器类（Container）提供了如下几个常用方法来访问容器里的组件。

- Component add(Component comp)：向容器中添加其他组件（该组件既可以是普通组件，也可以是容器），并返回被添加的组件。
- Component getComponentAt(int x, int y)：返回指定点的组件。
- int getComponentCount()：返回该容器内组件的数量。
- Component[] getComponents()：返回该容器内的所有组件。

AWT 主要提供了如下两种主要的容器类型。

- Window：可独立存在的顶级窗口。
- Panel：可作为容器容纳其他组件，但不能独立存在，必须被添加到其他容器中（如 Window、Panel 或者 Applet 等）。

AWT 容器的继承关系图如图 11.3 所示。

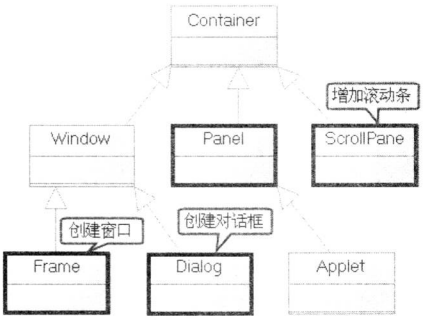


图 11.3　AWT 容器的继承关系

图 11.3 中显示了 AWT 容器之间的继承层次，其中以粗黑线圈出的容器是 AWT 编程中常用的组件。Frame 代表常见的窗口，它是 Window 类的子类，具有如下几个特点。

- Frame 对象有标题，允许通过拖拉来改变窗口的位置、大小。
- 初始化时为不可见，可用 setVisible(true)使其显示出来。
- 默认使用 BorderLayout 作为其布局管理器。

提示：

关于布局管理器的知识，请参考下一节的介绍。

下面的例子程序通过 Frame 创建了一个窗口。

程序清单：codes\11\11.2\FrameTest.java

```
public class FrameTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 设置窗口的大小、位置
        f.setBounds(30, 30 , 250, 200);
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
        f.setVisible(true);
    }
}
```

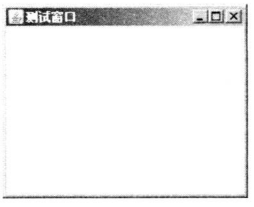


图 11.4　通过 Frame 创建的空白窗口

运行上面程序，会看到如图 11.4 所示的简单窗口。

从图 11.4 所示的窗口中可以看出，该窗口是 Windows 7 窗口风格，这也证明了 AWT 确实是调用程序运行平台的本地 API 创建了该窗口。如果单击图 11.4 所示窗口右上角的“×”按钮，该窗口不会关闭，这是因为还未为该窗口编写任何事件响应。如果想关闭该窗口，可以通过关闭运行该程序的命令行窗口来关闭该窗口。

提示：

正如前面所介绍的，创建图形用户界面的过程类似于拼图游戏，拼图游戏中的母板、图块都需要购买，而 Java 程序中的母板（容器）、图块（普通组件）则无须购买，直接采用 new 关键字创建一个对象即可。

Panel 是 AWT 中另一个典型的容器，它代表不能独立存在、必须放在其他容器中的容器。Panel 外在表现为一个矩形区域，该区域内可盛装其他组件。Panel 容器存在的意义在于为其他组件提供空间，Panel 容器具有如下几个特点。

- 可作为容器来盛装其他组件，为放置组件提供空间。
- 不能单独存在，必须放置到其他容器中。
- 默认使用 FlowLayout 作为其布局管理器。

下面的例子程序使用 Panel 作为容器来盛装一个文本框和一个按钮，并将该 Panel 对象添加到 Frame 对象中。

程序清单：codes\11\11.2\PanelTest.java

```
public class PanelTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 创建一个 Panel 容器
        Panel p = new Panel();
        // 向 Panel 容器中添加两个组件
        p.add(new TextField(20));
        p.add(new Button("单击我"));
        // 将 Panel 容器添加到 Frame 窗口中
        f.add(p);
        // 设置窗口的大小、位置
        f.setBounds(30, 30 , 250, 120);
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
        f.setVisible(true);
    }
}
```

编译、运行上面程序，会看到如图 11.5 所示的运行窗口。

从图 11.5 中可以看出，使用 AWT 创建窗口很简单，程序只需要通过 Frame 创建，然后再创建一些 AWT 组件，把这些组件添加到 Frame 创建的窗口中即可。

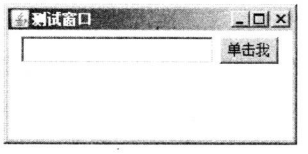


图 11.5 使用 Panel 盛装文本框和按钮

ScrollPane 是一个带滚动条的容器，它也不能独立存在，必须被添加到其他容器中。ScrollPane 容器具有如下几个特点。

- 可作为容器来盛装其他组件，当组件占用空间过大时，ScrollPane 自动产生滚动条。当然也可以通过指定特定的构造器参数来指定默认具有滚动条。
- 不能单独存在，必须放置到其他容器中。
- 默认使用 BorderLayout 作为其布局管理器。ScrollPane 通常用于盛装其他容器，所以通常不允许改变 ScrollPane 的布局管理器。

下面的例子程序使用 ScrollPane 容器来代替 Panel 容器。

程序清单：codes\11\11.2\ScrollPaneTest.java

```
public class ScrollPaneTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 创建一个 ScrollPane 容器，指定总是具有滚动条
        ScrollPane sp = new ScrollPane(
            ScrollPane.SCROLLBARS_ALWAYS);
        // 向 ScrollPane 容器中添加两个组件
        sp.add(new TextField(20));
        sp.add(new Button("单击我"));
        // 将 ScrollPane 容器添加到 Frame 对象中
        f.add(sp);
        // 设置窗口的大小、位置
        f.setBounds(30, 30 , 250, 120);
```

```
            // 将窗口显示出来（Frame 对象默认处于隐藏状态）
            f.setVisible(true);
        }
    }
```

图 11.6　ScrollPane 容器的效果

运行上面程序，会看到如图 11.6 所示的窗口。

图 11.6 所示的窗口中具有水平、垂直滚动条，这符合使用 ScrollPane 后的效果。程序明明向 ScrollPane 容器中添加了一个文本框和一个按钮，但只能看到一个按钮，却看不到文本框，这是为什么呢？这是因为 ScrollPane 使用 BorderLayout 布局管理器的缘故，而 BorderLayout 导致了该容器中只有一个组件被显示出来。下一节将向读者详细介绍布局管理器的知识。

11.3　布局管理器

为了使生成的图形用户界面具有良好的平台无关性，Java 语言提供了布局管理器这个工具来管理组件在容器中的布局，而不使用直接设置组件位置和大小的方式。

例如通过如下语句定义了一个标签（Label）：

```
Label hello = new Label("Hello Java");
```

为了让这个 hello 标签里刚好可以容纳"Hello Java"字符串，也就是实现该标签的最佳大小（既没有冗余空间，也没有内容被遮挡），Windows 可能应该设置为长 100 像素，高 20 像素，但换到 UNIX 上，则可能需要设置为长 120 像素，高 24 像素。当一个应用程序从 Windows 移植到 UNIX 上时，程序需要做大量的工作来调整图形界面。

对于不同的组件而言，它们都有一个最佳大小，这个最佳大小通常是平台相关的，程序在不同平台上运行时，相同内容的大小可能不一样。如果让程序员手动控制每个组件的大小、位置，这将给编程带来巨大的困难，为了解决这个问题，Java 提供了 LayoutManager，LayoutManager 可以根据运行平台来调整组件的大小，程序员要做的，只是为容器选择合适的布局管理器。

所有的 AWT 容器都有默认的布局管理器，如果没有为容器指定布局管理器，则该容器使用默认的布局管理器。为容器指定布局管理器通过调用容器对象的 setLayout(LayoutManager lm)方法来完成。如下代码所示：

```
c.setLayout(new XxxLayout());
```

AWT 提供了 FlowLayout、BorderLayout、GridLayout、GridBagLayout、CardLayout 5 个常用的布局管理器，Swing 还提供了一个 BoxLayout 布局管理器。下面将详细介绍这几个布局管理器。

11.3.1　FlowLayout 布局管理器

在 FlowLayout 布局管理器中，组件像水流一样向某方向流动（排列），遇到障碍（边界）就折回，重头开始排列。在默认情况下，FlowLayout 布局管理器从左向右排列所有组件，遇到边界就会折回下一行重新开始。

提示：

当读者在电脑上输入一篇文章时，所使用的就是 FlowLayout 布局管理器，所有的文字默认从左向右排列，遇到边界就会折回下一行重新开始。AWT 中的 FlowLayout 布局管理器与此完全类似，只是此时排列的是 AWT 组件，而不是文字。

FlowLayout 有如下三个构造器。

- FlowLayout()：使用默认的对齐方式及默认的垂直间距、水平间距创建 FlowLayout 布局管理器。
- FlowLayout(int align)：使用指定的对齐方式及默认的垂直间距、水平间距创建 FlowLayout 布局

管理器。

- FlowLayout(int align,int hgap,int vgap)：使用指定的对齐方式及指定的垂直间距、水平间距创建 FlowLayout 布局管理器。

上面三个构造器的 hgap、vgap 代表水平间距、垂直间距，为这两个参数传入整数值即可。其中 align 表明 FlowLayout 中组件的排列方向（从左向右、从右向左、从中间向两边等），该参数应该使用 FlowLayout 类的静态常量：FlowLayout.LEFT、FlowLayout.CENTER、FlowLayout.RIGHT。

Panel 和 Applet 默认使用 FlowLayout 布局管理器，下面程序将一个 Frame 改为使用 FlowLayout 布局管理器。

程序清单：codes\11\11.3\FlowLayoutTest.java

```
public class FlowLayoutTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 设置 Frame 容器使用 FlowLayout 布局管理器
        f.setLayout(new FlowLayout(FlowLayout.LEFT , 20, 5));
        // 向窗口中添加 10 个按钮
        for (int i = 0; i < 10 ; i++ )
        {
            f.add(new Button("按钮" + i));
        }
        // 设置窗口为最佳大小
        f.pack();
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
        f.setVisible(true);
    }
}
```

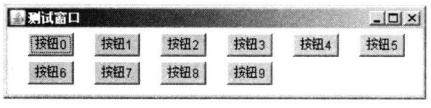


图 11.7 FlowLayout 布局管理器

运行上面程序，会看到如图 11.7 所示的窗口效果。

图 11.7 显示了各组件左对齐、水平间距为 20、垂直间距为 5 的分布效果。

注意：

上面程序中执行了 f.pack()代码，pack()方法是 Window 容器提供的一个方法，该方法用于将窗口调整到最佳大小。通过 Java 编写图形用户界面程序时，很少直接设置窗口的大小，通常都是调用 pack()方法来将窗口调整到最佳大小。

11.3.2 BorderLayout 布局管理器

BorderLayout 将容器分为 EAST、SOUTH、WEST、NORTH、CENTER 五个区域，普通组件可以被放置在这 5 个区域的任意一个中。BorderLayout 布局管理器的布局示意图如图 11.8 所示。

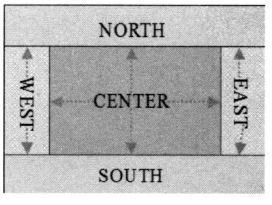


图 11.8 BorderLayout 布局管理器的布局示意图

当改变使用 BorderLayout 的容器大小时，NORTH、SOUTH 和 CENTER 区域水平调整，而 EAST、WEST 和 CENTER 区域垂直调整。使用 BorderLayout 有如下两个注意点。

- 当向使用 BorderLayout 布局管理器的容器中添加组件时，需要指定要添加到哪个区域中。如果没有指定添加到哪个区域中，则默认添加到中间区域中。
- 如果向同一个区域中添加多个组件时，后放入的组件会覆盖先放入的组件。

提示：

第二个注意点就可以解释为什么在 ScrollPaneTest.java 中向 ScrollPane 中添加两个组件后，但运行结果只能看到最后一个按钮，因为最后添加的组件把前面添加的组件覆盖了。

Frame、Dialog、ScrollPane 默认使用 BorderLayout 布局管理器，BorderLayout 有如下两个构造器。

- BorderLayout()：使用默认的水平间距、垂直间距创建 BorderLayout 布局管理器。
- BorderLayout(int hgap,int vgap)：使用指定的水平间距、垂直间距创建 BorderLayout 布局管理器。

当向使用 BorderLayout 布局管理器的容器中添加组件时，应该使用 BorderLayout 类的几个静态常量来指定添加到哪个区域中。BorderLayout 有如下几个静态常量：EAST（东）、NORTH（北）、WEST（西）、SOUTH（南）、CENTER（中）。如下例子程序示范了 BorderLayout 的用法。

程序清单：codes\11\11.3\BorderLayoutTest.java

```
public class BorderLayoutTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 设置 Frame 容器使用 BorderLayout 布局管理器
        f.setLayout(new BorderLayout(30, 5));
        f.add(new Button("南") , SOUTH);
        f.add(new Button("北") , NORTH);
        // 默认添加到中间区域中
        f.add(new Button("中"));
        f.add(new Button("东") , EAST);
        f.add(new Button("西") , WEST);
        // 设置窗口为最佳大小
        f.pack();
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
        f.setVisible(true);
    }
}
```

运行上面程序，会看到如图 11.9 所示的运行窗口。

从图 11.9 中可以看出，当使用 BorderLayout 布局管理器时，每个区域的组件都会尽量去占据整个区域，所以中间的按钮比较大。

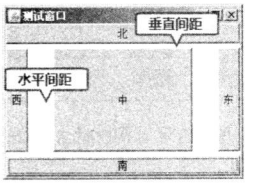
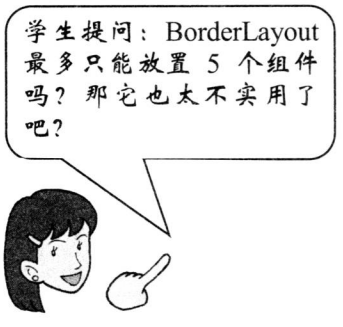
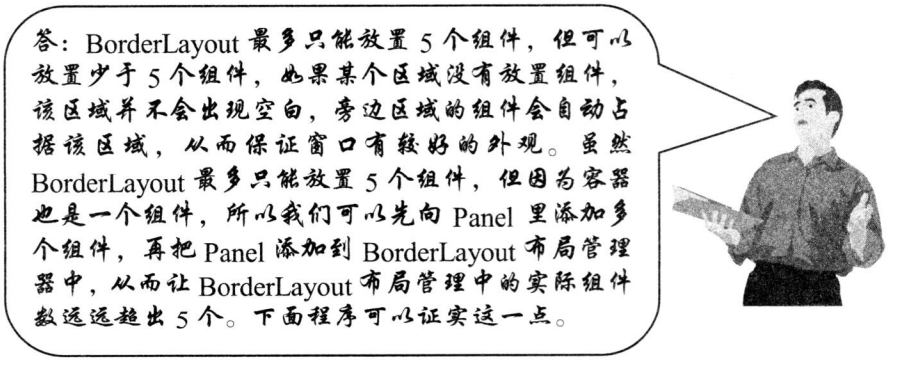


图 11.9　BorderLayout 布局管理器的效果





程序清单：codes\11\11.3\BorderLayoutTest2.java

```
public class BorderLayoutTest2
{
    public static void main(String[] args)
    {
        Frame f = new Frame("测试窗口");
        // 设置 Frame 容器使用 BorderLayout 布局管理器
        f.setLayout(new BorderLayout(30, 5));
        f.add(new Button("南") , SOUTH);
```

```
        f.add(new Button("北") , NORTH);
        // 创建一个 Panel 对象
        Panel p = new Panel();
        // 向 Panel 对象中添加两个组件
        p.add(new TextField(20));
        p.add(new Button("单击我"));
        // 默认添加到中间区域中，向中间区域添加一个 Panel 容器
        f.add(p);
        f.add(new Button("东") , EAST);
        // 设置窗口为最佳大小
        f.pack();
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
        f.setVisible(true);
    }
}
```

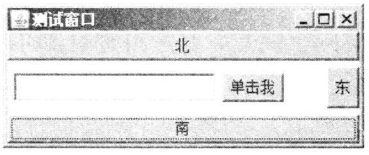


图 11.10 向 BorderLayout 布局管理器中添加 Panel 容器

上面程序没有向 WEST 区域添加组件，但向 CENTER 区域添加了一个 Panel 容器，该 Panel 容器中包含了一个文本框和一个按钮。运行上面程序，会看到如图 11.10 所示的窗口界面。

从图 11.10 中可以看出，虽然程序没有向 WEST 区域添加组件，但窗口中依然有 5 个组件，因为 CENTER 区域添加的是 Panel，而该 Panel 里包含了 2 个组件，所以会看到此界面效果。

11.3.3 GridLayout 布局管理器

GridLayout 布局管理器将容器分割成纵横线分隔的网格，每个网格所占的区域大小相同。当向使用 GridLayout 布局管理器的容器中添加组件时，默认从左向右、从上向下依次添加到每个网格中。与 FlowLayout 不同的是，放置在 GridLayout 布局管理器中的各组件的大小由组件所处的区域来决定（每个组件将自动占满整个区域）。

GridLayout 有如下两个构造器。

- GridLayout(int rows,int cols)：采用指定的行数、列数，以及默认的横向间距、纵向间距将容器分割成多个网格。
- GridLayout(int rows,int cols,int hgap,int vgap)：采用指定的行数、列数，以及指定的横向间距、纵向间距将容器分割成多个网格。

如下程序结合 BorderLayout 和 GridLayout 开发了一个计算器的可视化窗口。

程序清单：codes\11\11.3\GridLayoutTest.java

```
public class GridLayoutTest
{
    public static void main(String[] args)
    {
        Frame f = new Frame("计算器");
        Panel p1 = new Panel();
        p1.add(new TextField(30));
        f.add(p1 , NORTH);
        Panel p2 = new Panel();
        // 设置 Panel 使用 GridLayout 布局管理器
        p2.setLayout(new GridLayout(3, 5 , 4, 4));
        String[] name = {"0" , "1" , "2" , "3"
            , "4" , "5" , "6" , "7" , "8" , "9"
            , "+" , "-" , "*" , "/" , "."};
        // 向 Panel 中依次添加 15 个按钮
        for (int i = 0 ; i < name.length; i++ )
        {
            p2.add(new Button(name[i]));
        }
        // 默认将 Panel 对象添加到 Frame 窗口的中间
        f.add(p2);
        // 设置窗口为最佳大小
        f.pack();
        // 将窗口显示出来（Frame 对象默认处于隐藏状态）
```

```
        f.setVisible(true);
    }
}
```

上面程序的Frame采用默认的BorderLayout布局管理器，程序向BorderLayout中只添加了两个组件：NORTH区域添加了一个文本框，CENTER区域添加了一个Panel容器，该容器采用GridLayout布局管理器，Panel容器中添加了15个按钮。运行上面程序，会看到如图11.11所示的运行窗口。

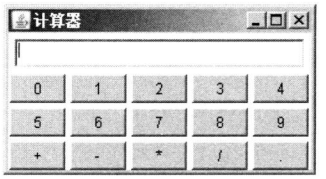


图11.11　使用GridLayout布局管理器的效果

提示：

图11.11所示的效果是结合两种布局管理器的例子：Frame使用BorderLayout布局管理器，CENTER区域的Panel使用GridLayout布局管理器。实际上，大部分应用窗口都不能使用一个布局管理器直接做出来，必须采用这种嵌套的方式。

11.3.4　GridBagLayout布局管理器

GridBagLayout布局管理器的功能最强大，但也最复杂，与GridLayout布局管理器不同的是，在GridBagLayout布局管理器中，一个组件可以跨越一个或多个网格，并可以设置各网格的大小互不相同，从而增加了布局的灵活性。当窗口的大小发生变化时，GridBagLayout布局管理器也可以准确地控制窗口各部分的拉伸。

为了处理GridBagLayout中GUI组件的大小、跨越性，Java提供了GridBagConstraints对象，该对象与特定的GUI组件关联，用于控制该GUI组件的大小、跨越性。

使用GridBagLayout布局管理器的步骤如下。

① 创建GridBagLayout布局管理器，并指定GUI容器使用该布局管理器。

```
GridBagLayout gb = new GridBagLayout();
container.setLayout(gb);
```

② 创建GridBagConstraints对象，并设置该对象的相关属性（用于设置受该对象控制的GUI组件的大小、跨越性等）。

```
gbc.gridx = 2; //设置受该对象控制的GUI组件位于网格的横向索引
gbc.gridy = 1; //设置受该对象控制的GUI组件位于网格的纵向索引
gbc.gridwidth = 2; //设置受该对象控制的GUI组件横向跨越多少网格
gbc.gridheight = 1; //设置受该对象控制的GUI组件纵向跨越多少网格
```

③ 调用GridBagLayout对象的方法来建立GridBagConstraints对象和受控制组件之间的关联。

```
gb.setConstraints(c , gbc); //设置c组件受gbc对象控制
```

④ 添加组件，与采用普通布局管理器添加组件的方法完全一样。

```
container.add(c);
```

如果需要向一个容器中添加多个GUI组件，则需要多次重复步骤2~4。由于GridBagConstraints对象可以多次重用，所以实际上只需要创建一个GridBagConstraints对象，每次添加GUI组件之前先改变GridBagConstraints对象的属性即可。

从上面介绍中可以看出，使用GridBagLayout布局管理器的关键在于GridBagConstraints，它才是精确控制每个GUI组件的核心类，该类具有如下几个属性。

- gridx、gridy：设置受该对象控制的GUI组件左上角所在网格的横向索引、纵向索引（GridBagLayout左上角网格的索引为0、0）。这两个值还可以是GridBagConstraints.RELATIVE（默认值），它表明当前组件紧跟在上一个组件之后。
- gridwidth、gridheight：设置受该对象控制的GUI组件横向、纵向跨越多少个网格，两个属性值的

默认值都是1。如果设置这两个属性值为GridBagConstraints.REMAINDER，这表明受该对象控制的GUI组件是横向、纵向最后一个组件；如果设置这两个属性值为GridBagConstraints. RELATIVE，这表明受该对象控制的GUI组件是横向、纵向倒数第二个组件。

- fill：设置受该对象控制的GUI组件如何占据空白区域。该属性的取值如下。
  - GridBagConstraints.NONE：GUI组件不扩大。
  - GridBagConstraints.HORIZONTAL：GUI组件水平扩大以占据空白区域。
  - GridBagConstraints.VERTICAL：GUI组件垂直扩大以占据空白区域。
  - GridBagConstraints.BOTH：GUI组件水平、垂直同时扩大以占据空白区域。
- ipadx、ipady：设置受该对象控制的GUI组件横向、纵向内部填充的大小，即在该组件最小尺寸的基础上还需要增大多少。如果设置了这两个属性，则组件横向大小为最小宽度再加ipadx*2像素，纵向大小为最小高度再加ipady*2像素。
- insets：设置受该对象控制的GUI组件的外部填充的大小，即该组件边界和显示区域边界之间的距离。
- anchor：设置受该对象控制的GUI组件在其显示区域中的定位方式。定位方式如下。
  - GridBagConstraints.CENTER（中间）
  - GridBagConstraints.NORTH（上中）
  - GridBagConstraints.NORTHWEST（左上角）
  - GridBagConstraints.NORTHEAST（右上角）
  - GridBagConstraints.SOUTH（下中）
  - GridBagConstraints.SOUTHEAST（右下角）
  - GridBagConstraints.SOUTHWEST（左下角）
  - GridBagConstraints.EAST（右中）
  - GridBagConstraints.WEST（左中）
- weightx、weighty：设置受该对象控制的GUI组件占据多余空间的水平、垂直增加比例（也叫权重，即weight的直译），这两个属性的默认值是0，即该组件不占据多余空间。假设某个容器的水平线上包括三个GUI组件，它们的水平增加比例分别是1、2、3，但容器宽度增加60像素时，则第一个组件宽度增加10像素，第二个组件宽度增加20像素，第三个组件宽度增加30像素。如果其增加比例为0，则表示不会增加。

注意：

如果希望某个组件的大小随容器的增大而增大，则必须同时设置控制该组件的GridBagConstraints对象的fill属性和weightx、weighty属性。

下面的例子程序示范了如何使用GridBagLayout布局管理器来管理窗口中的10个按钮。

程序清单：codes\11\11.3\GridBagTest.java

```
public class GridBagTest
{
    private Frame f = new Frame("测试窗口");
    private GridBagLayout gb = new GridBagLayout();
    private GridBagConstraints gbc = new GridBagConstraints();
    private Button[] bs = new Button[10];
    public void init()
    {
        f.setLayout(gb);
        for (int i = 0; i < bs.length ; i++ )
        {
            bs[i] = new Button("按钮" + i);
        }
        // 所有组件都可以在横向、纵向上扩大
```

```
        gbc.fill = GridBagConstraints.BOTH;
        gbc.weightx = 1;
        addButton(bs[0]);
        addButton(bs[1]);
        addButton(bs[2]);
        // 该 GridBagConstraints 控制的 GUI 组件将会成为横向最后一个组件
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        addButton(bs[3]);
        // 该 GridBagConstraints 控制的 GUI 组件将在横向上不会扩大
        gbc.weightx = 0;
        addButton(bs[4]);
        // 该 GridBagConstraints 控制的 GUI 组件将横跨两个网格
        gbc.gridwidth = 2;
        addButton(bs[5]);
        // 该 GridBagConstraints 控制的 GUI 组件将横跨一个网格
        gbc.gridwidth = 1;
        // 该 GridBagConstraints 控制的 GUI 组件将在纵向上跨两个网格
        gbc.gridheight = 2;
        // 该 GridBagConstraints 控制的 GUI 组件将会成为横向最后一个组件
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        addButton(bs[6]);
        // 该 GridBagConstraints 控制的 GUI 组件将横向跨越一个网格，纵向跨越两个网格
        gbc.gridwidth = 1;
        gbc.gridheight = 2;
        // 该 GridBagConstraints 控制的 GUI 组件纵向扩大的权重是 1
        gbc.weighty = 1;
        addButton(bs[7]);
        // 设置下面的按钮在纵向上不会扩大
        gbc.weighty = 0;
        // 该 GridBagConstraints 控制的 GUI 组件将会成为横向最后一个组件
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        // 该 GridBagConstraints 控制的 GUI 组件将在纵向上横跨一个网格
        gbc.gridheight = 1;
        addButton(bs[8]);
        addButton(bs[9]);
        f.pack();
        f.setVisible(true);
    }
    private void addButton(Button button)
    {
        gb.setConstraints(button, gbc);
        f.add(button);
    }
    public static void main(String[] args)
    {
        new GridBagTest().init();
    }
}
```

运行上面程序，会看到如图 11.12 所示的窗口。

从图 11.12 中可以看出，虽然设置了按钮 4、按钮 5 横向上不会扩大，但因为按钮 4、按钮 5 的宽度会受上一行 4 个按钮的影响，所以它们实际上依然会变大；同理，虽然设置了按钮 8、按钮 9 纵向上不会扩大，但因为受按钮 7 的影响，所以按钮 9 纵向上依然会变大（但按钮 8 不会变高）。

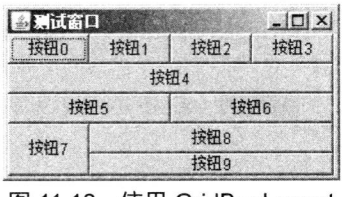


图 11.12　使用 GridBagLayout 布局管理器的效果

提示：

上面程序把需要重复访问的 AWT 组件设置成成员变量，然后使用 init()方法来完成界面的初始化工作，这种做法比前面那种在 main 方法里把 AWT 组件定义成局部变量的方式更好。

11.3.5　CardLayout 布局管理器

CardLayout 布局管理器以时间而非空间来管理它里面的组件，它将加入容器的所有组件看成一叠卡

片，每次只有最上面的那个 Component 才可见。就好像一副扑克牌，它们叠在一起，每次只有最上面的一张扑克牌才可见。CardLayout 提供了如下两个构造器。

- CardLayout()：创建默认的 CardLayout 布局管理器。
- CardLayout(int hgap,int vgap)：通过指定卡片与容器左右边界的间距（hgap）、上下边界（vgap）的间距来创建 CardLayout 布局管理器。

CardLayout 用于控制组件可见的 5 个常用方法如下。

- first(Container target)：显示 target 容器中的第一张卡片。
- last(Container target)：显示 target 容器中的最后一张卡片。
- previous(Container target)：显示 target 容器中的前一张卡片。
- next(Container target)：显示 target 容器中的后一张卡片。
- show(Container target,String name)：显示 target 容器中指定名字的卡片。

如下例子程序示范了 CardLayout 布局管理器的用法。

程序清单：codes\11\11.3\CardLayoutTest.java

```
public class CardLayoutTest
{
    Frame f = new Frame("测试窗口");
    String[] names = {"第一张" , "第二张" , "第三张"
        , "第四张" , "第五张"};
    Panel pl = new Panel();
    public void init()
    {
        final CardLayout c = new CardLayout();
        pl.setLayout(c);
        for (int i = 0 ; i < names.length ; i++)
        {
            pl.add(names[i] , new Button(names[i]));
        }
        Panel p = new Panel();
        ActionListener listener = e ->
        {
            switch(e.getActionCommand())
            {
                case "上一张":
                    c.previous(pl);
                    break;
                case "下一张":
                    c.next(pl);
                    break;
                case "第一张":
                    c.first(pl);
                    break;
                case "最后一张":
                    c.last(pl);
                    break;
                case "第三张":
                    c.show(pl , "第三张");
                    break;
            }
        };
        // 控制显示上一张的按钮
        Button previous = new Button("上一张");
        previous.addActionListener(listener);
        // 控制显示下一张的按钮
        Button next = new Button("下一张");
        next.addActionListener(listener);
        // 控制显示第一张的按钮
        Button first = new Button("第一张");
        first.addActionListener(listener);
        // 控制显示最后一张的按钮
        Button last = new Button("最后一张");
        last.addActionListener(listener);
```

```
        // 控制根据 Card 名显示的按钮
        Button third = new Button("第三张");
        third.addActionListener(listener);
        p.add(previous);
        p.add(next);
        p.add(first);
        p.add(last);
        p.add(third);
        f.add(pl);
        f.add(p , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new CardLayoutTest().init();
    }
}
```

上面程序中通过 Frame 创建了一个窗口，该窗口被分为上下两个部分，其中上面的 Panel 使用 CardLayout 布局管理器，该 Panel 中放置了 5 张卡片，每张卡片里放一个按钮；下面的 Panel 使用 FlowLayout 布局管理器，依次放置了 3 个按钮，用于控制上面 Panel 中卡片的显示。运行上面程序，会看到如图 11.13 所示的运行窗口。

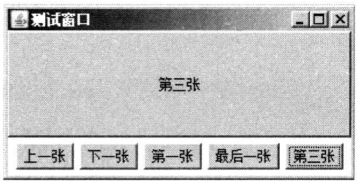


图 11.13　使用 CardLayout 布局管理器的效果

单击图 11.13 中的 5 个按钮，将可以看到上面 Panel 中的 5 张卡片发生改变。

提示：

上面程序使用了 AWT 的事件编程，关于事件编程请参考 11.5 节内容。

11.3.6　绝对定位

很多曾经学习过 VB、Delphi 的读者可能比较怀念那种随意拖动控件的感觉，对 Java 的布局管理器非常不习惯。实际上，Java 也提供了那种拖动控件的方式，即 Java 也可以对 GUI 组件进行绝对定位。在 Java 容器中采用绝对定位的步骤如下。

① 将 Container 的布局管理器设成 null：setLayout(null)。

② 向容器中添加组件时，先调用 setBounds() 或 setSize()方法来设置组件的大小、位置，或者直接创建 GUI 组件时通过构造参数指定该组件的大小、位置，然后将该组件添加到容器中。

下面程序示范了如何使用绝对定位来控制窗口中的 GUI 组件。

程序清单：codes\11\11.3\NullLayoutTest.java

```
public class NullLayoutTest
{
    Frame f = new Frame("测试窗口");
    Button b1 = new Button("第一个按钮");
    Button b2 = new Button("第二个按钮");
    public void init()
    {
        // 设置使用 null 布局管理器
        f.setLayout(null);
        // 下面强制设置每个按钮的大小、位置
        b1.setBounds(20, 30, 90, 28);
        f.add(b1);
        b2.setBounds(50, 45, 120, 35);
        f.add(b2);
        f.setBounds(50, 50, 200, 100);
        f.setVisible(true);
    }
```

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

运行上面程序，会看到如图 11.14 所示的运行窗口。

从图 11.14 中可以看出，使用绝对定位时甚至可以使两个按钮重叠，可见使用绝对定位确实非常灵活，而且很简捷，但这种方式是以丧失跨平台特性作为代价的。

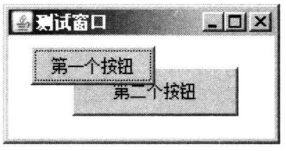


图 11.14 使用绝对定位的效果

注意：

采用绝对定位绝不是最好的方法，它可能导致该 GUI 界面失去跨平台特性。

11.3.7 BoxLayout 布局管理器

GridBagLayout 布局管理器虽然功能强大，但它实在太复杂了，所以 Swing 引入了一个新的布局管理器：BoxLayout，它保留了 GridBagLayout 的很多优点，但是却没那么复杂。BoxLayout 可以在垂直和水平两个方向上摆放 GUI 组件，BoxLayout 提供了如下一个简单的构造器。

- BoxLayout(Container target, int axis)：指定创建基于 target 容器的 BoxLayout 布局管理器，该布局管理器里的组件按 axis 方向排列。其中 axis 有 BoxLayout.X_AXIS（横向）和 BoxLayout.Y_AXIS（纵向）两个方向。

下面程序简单示范了使用 BoxLayout 布局管理器来控制容器中按钮的布局。

中華工匠

程序清单：codes\11\11.3\BoxLayoutTest.java

```
public class BoxLayoutTest
{
    private Frame f = new Frame("测试");
    public void init()
    {
        f.setLayout(new BoxLayout(f , BoxLayout.Y_AXIS));
        // 下面按钮将会垂直排列
        f.add(new Button("第一个按钮"));
        f.add(new Button("按钮二"));
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new BoxLayoutTest().init();
    }
}
```

运行上面程序，会看到如图 11.15 所示的运行窗口。

BoxLayout 通常和 Box 容器结合使用，Box 是一个特殊的容器，它有点像 Panel 容器，但该容器默认使用 BoxLayout 布局管理器。Box 提供了如下两个静态方法来创建 Box 对象。

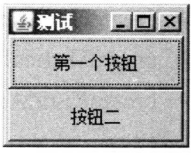


图 11.15 垂直方向的 BoxLayout 布局管理器

- createHorizontalBox()：创建一个水平排列组件的 Box 容器。
- createVerticalBox()：创建一个垂直排列组件的 Box 容器。

一旦获得了 Box 容器之后，就可以使用 Box 来盛装普通的 GUI 组件，然后将这些 Box 组件添加到其他容器中，从而形成整体的窗口布局。下面的例子程序示范了如何使用 Box 容器。

程序清单：codes\11\11.3\BoxTest.java

```
public class BoxTest
{
    private Frame f = new Frame("测试");
    // 定义水平摆放组件的 Box 对象
    private Box horizontal = Box.createHorizontalBox();
    // 定义垂直摆放组件的 Box 对象
    private Box vertical = Box.createVerticalBox();
    public void init()
    {
        horizontal.add(new Button("水平按钮一"));
        horizontal.add(new Button("水平按钮二"));
        vertical.add(new Button("垂直按钮一"));
        vertical.add(new Button("垂直按钮二"));
        f.add(horizontal , BorderLayout.NORTH);
        f.add(vertical);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new BoxTest().init();
    }
}
```

上面程序创建了一个水平摆放组件的 Box 容器和一个垂直摆放组件的 Box 容器，并将这两个 Box 容器添加到 Frame 窗口中。运行该程序会看到如图 11.16 所示的运行窗口。

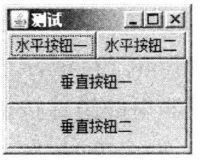
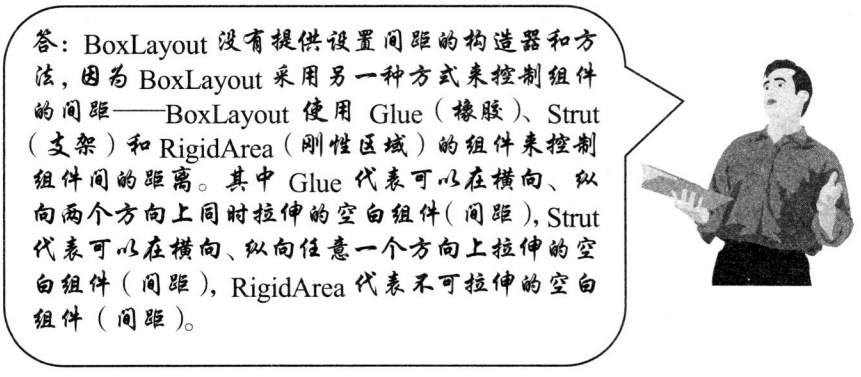


图 11.16　使用 Box 容器的窗口效果



Box 提供了如下 5 个静态方法来创建 Glue、Strut 和 RigidArea。

- createHorizontalGlue()：创建一条水平 Glue（可在两个方向上同时拉伸的间距）。
- createVerticalGlue()：创建一条垂直 Glue（可在两个方向上同时拉伸的间距）。
- createHorizontalStrut(int width)：创建一条指定宽度的水平 Strut（可在垂直方向上拉伸的间距）。
- createVerticalStrut(int height)：创建一条指定高度的垂直 Strut（可在水平方向上拉伸的间距）。
- createRigidArea(Dimension d)：创建指定宽度、高度的 RigidArea（不可拉伸的间距）。

提示：

不管 Glue、Strut、RigidArea 的翻译多么奇怪，这些名称多么古怪，但读者没有必要去纠缠它们的名称，只要知道它们就是代表组件之间的几种间距即可。

上面 5 个方法都返回 Component 对象（代表间距），程序可以将这些分隔 Component 添加到两个普通的 GUI 组件之间，用以控制组件的间距。下面程序使用上面三种间距来分隔 Box 中的按钮。

程序清单：codes\11\11.3\BoxSpaceTest.java

```
public class BoxSpaceTest
{
    private Frame f = new Frame("测试");
    // 定义水平摆放组件的 Box 对象
    private Box horizontal = Box.createHorizontalBox();
    // 定义垂直摆放组件的 Box 对象
    private Box vertical = Box.createVerticalBox();
    public void init()
    {
        horizontal.add(new Button("水平按钮一"));
        horizontal.add(Box.createHorizontalGlue());
        horizontal.add(new Button("水平按钮二"));
        // 水平方向不可拉伸的间距，其宽度为 10px
        horizontal.add(Box.createHorizontalStrut(10));
        horizontal.add(new Button("水平按钮三"));
        vertical.add(new Button("垂直按钮一"));
        vertical.add(Box.createVerticalGlue());
        vertical.add(new Button("垂直按钮二"));
        // 垂直方向不可拉伸的间距，其高度为 10px
        vertical.add(Box.createVerticalStrut(10));
        vertical.add(new Button("垂直按钮三"));
        f.add(horizontal , BorderLayout.NORTH);
        f.add(vertical);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new BoxSpaceTest().init();
    }
}
```

运行上面程序，会看到如图 11.17 所示的运行窗口。

从图 11.17 中可以看出，Glue 可以在两个方向上同时拉伸，但 Strut 只能在一个方向上拉伸，RigidArea 则不可拉伸。

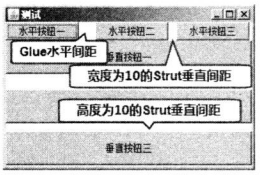


图 11.17 使用间距分隔 Box 器中的按钮效果

提示：

因为 BoxLayout 是 Swing 提供的布局管理器，所以用于管理 Swing 组件将会有更好的表现。

11.4 AWT 常用组件

AWT 组件需要调用运行平台的图形界面来创建和平台一致的对等体，因此 AWT 只能使用所有平台都支持的公共组件，所以 AWT 只提供了一些常用的 GUI 组件。

11.4.1 基本组件

AWT 提供了如下基本组件。

- Button：按钮，可接受单击操作。
- Canvas：用于绘图的画布。
- Checkbox：复选框组件（也可变成单选框组件）。
- CheckboxGroup：用于将多个 Checkbox 组件组合成一组，一组 Checkbox 组件将只有一个可以被选中，即全部变成单选框组件。
- Choice：下拉式选择框组件。
- Frame：窗口，在 GUI 程序里通过该类创建窗口。

- Label：标签类，用于放置提示性文本。
- List：列表框组件，可以添加多项条目。
- Panel：不能单独存在基本容器类，必须放到其他容器中。
- Scrollbar：滑动条组件。如果需要用户输入位于某个范围的值，就可以使用滑动条组件，比如调色板中设置 RGB 的三个值所用的滑动条。当创建一个滑动条时，必须指定它的方向、初始值、滑块的大小、最小值和最大值。
- ScrollPane：带水平及垂直滚动条的容器组件。
- TextArea：多行文本域。
- TextField：单行文本框。

这些 AWT 组件的用法比较简单，读者可以查阅 API 文档来获取它们各自的构造器、方法等详细信息。下面的例子程序示范了它们的基本用法。

程序清单：codes\11\11.4\CommonComponent.java

```
public class CommonComponent
{
    Frame f = new Frame("测试");
    // 定义一个按钮
    Button ok = new Button("确认");
    CheckboxGroup cbg = new CheckboxGroup();
    // 定义一个单选框（处于 cbg 一组），初始处于被选中状态
    Checkbox male = new Checkbox("男" , cbg , true);
    // 定义一个单选框（处于 cbg 一组），初始处于没有选中状态
    Checkbox female = new Checkbox("女" , cbg , false);
    // 定义一个复选框，初始处于没有选中状态
    Checkbox married = new Checkbox("是否已婚？" , false);
    // 定义一个下拉选择框
    Choice colorChooser = new Choice();
    // 定义一个列表选择框
    List colorList = new List(6, true);
    // 定义一个 5 行、20 列的多行文本域
    TextArea ta = new TextArea(5, 20);
    // 定义一个 50 列的单行文本域
    TextField name = new TextField(50);
    public void init()
    {
        colorChooser.add("红色");
        colorChooser.add("绿色");
        colorChooser.add("蓝色");
        colorList.add("红色");
        colorList.add("绿色");
        colorList.add("蓝色");
        // 创建一个装载了文本框、按钮的 Panel
        Panel bottom = new Panel();
        bottom.add(name);
        bottom.add(ok);
        f.add(bottom , BorderLayout.SOUTH);
        // 创建一个装载了下拉选择框、三个 Checkbox 的 Panel
        Panel checkPanel = new Panel();
        checkPanel.add(colorChooser);
        checkPanel.add(male);
        checkPanel.add(female);
        checkPanel.add(married);
        // 创建一个垂直排列组件的 Box，盛装多行文本域、Panel
        Box topLeft = Box.createVerticalBox();
        topLeft.add(ta);
        topLeft.add(checkPanel);
        // 创建一个水平排列组件的 Box，盛装 topLeft、colorList
        Box top = Box.createHorizontalBox();
        top.add(topLeft);
```

```
        top.add(colorList);
        // 将 top Box 容器添加到窗口的中间
        f.add(top);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new CommonComponent().init();
    }
}
```

运行上面程序，会看到如图 11.18 所示的窗口。

提示：

关于 AWT 常用组件的用法，以及布局管理器的用法，读者可以参考 API 文档来逐渐熟悉它们。一旦掌握了它们的用法之后，就可以借助于 IDE 工具来设计 GUI 界面，使用 IDE 工具可以更快地设计出更美观的 GUI 界面。

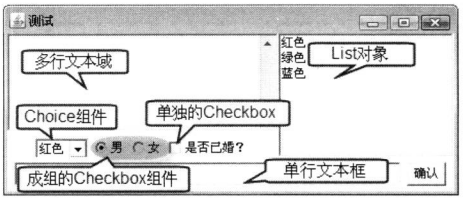


图 11.18 常见的 AWT 组件

11.4.2 对话框（Dialog）

Dialog 是 Window 类的子类，是一个容器类，属于特殊组件。对话框是可以独立存在的顶级窗口，因此用法与普通窗口的用法几乎完全一样。但对话框有如下两点需要注意。

- 对话框通常依赖于其他窗口，就是通常有一个 parent 窗口。
- 对话框有非模式（non-modal）和模式（modal）两种，当某个模式对话框被打开之后，该模式对话框总是位于它依赖的窗口之上；在模式对话框被关闭之前，它依赖的窗口无法获得焦点。

对话框有多个重载的构造器，它的构造器可能有如下三个参数。

- owner：指定该对话框所依赖的窗口，既可以是窗口，也可以是对话框。
- title：指定该对话框的窗口标题。
- modal：指定该对话框是否是模式的，可以是 true 或 false。

下面的例子程序示范了模式对话框和非模式对话框的用法。

程序清单：codes\11\11.4\DialogTest.java

```
public class DialogTest
{
    Frame f = new Frame("测试");
    Dialog d1 = new Dialog(f, "模式对话框" , true);
    Dialog d2 = new Dialog(f, "非模式对话框" , false);
    Button b1 = new Button("打开模式对话框");
    Button b2 = new Button("打开非模式对话框");
    public void init()
    {
        d1.setBounds(20 , 30 , 300, 400);
        d2.setBounds(20 , 30 , 300, 400);
        b1.addActionListener(e -> d1.setVisible(true));
        b2.addActionListener(e -> d2.setVisible(true));
        f.add(b1);
        f.add(b2 , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
```

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

上面程序创建了 d1 和 d2 两个对话框，其中 d1 是一个模式对话框，而 d2 是一个非模式对话框（两个对话框都是空的）。该窗口中还提供了两个按钮，分别用于打开模式对话框和非模式对话框。打开模式对话框后鼠标无法激活原来的“测试窗口”；但打开非模式对话框后还可以激活原来的“测试窗口”。

提示：

上面程序使用了 AWT 的事件处理来打开对话框，关于事件处理介绍请看 11.5 节的内容。

注意：

不管是模式对话框还是非模式对话框，打开后都无法关闭它们，因为程序没有为这两个对话框编写事件监听器。还有，如果主程序需要对话框里接收的输入值，则应该把该对话框设置成模式对话框，因为模式对话框会阻塞该程序；如果把对话框设置成非模式对话框，则可能造成对话框被打开了，但用户并没有操作该对话框，也没有向对话框里进行输入，这就会引起主程序的异常。

Dialog 类还有一个子类：FileDialog，它代表一个文件对话框，用于打开或者保存文件。FileDialog 也提供了几个构造器，可分别支持 parent、title 和 mode 三个构造参数，其中 parent、title 指定文件对话框的所属父窗口和标题；而 mode 指定该窗口用于打开文件或保存文件，该参数支持如下两个参数值：FileDialog.LOAD、FileDialog.SAVE。

提示：

FileDialog 不能指定是模式对话框或非模式对话框，因为 FileDialog 依赖于运行平台的实现，如果运行平台的文件对话框是模式的，那么 FileDialog 也是模式的；否则就是非模式的。

FileDialog 提供了如下两个方法来获取被打开/保存文件的路径。

- getDirectory()：获取 FileDialog 被打开/保存文件的绝对路径。
- getFile()：获取 FileDialog 被打开/保存文件的文件名。

下面程序分别示范了使用 FileDialog 来创建打开/保存文件的对话框。

程序清单：codes\11\11.4\FileDialogTest.java

```
public class FileDialogTest
{
    Frame f = new Frame("测试");
    // 创建两个文件对话框
    FileDialog d1 = new FileDialog(f
        , "选择需要打开文件" , FileDialog.LOAD);
    FileDialog d2 = new FileDialog(f
        , "选择保存文件的路径" , FileDialog.SAVE);
    Button b1 = new Button("打开文件");
    Button b2 = new Button("保存文件");
    public void init()
    {
        b1.addActionListener(e ->
        {
            d1.setVisible(true);
            // 打印出用户选择的文件路径和文件名
            System.out.println(d1.getDirectory()
                + d1.getFile());
        });
```

```
        b2.addActionListener(e ->
        {
            d2.setVisible(true);
            // 打印出用户选择的文件路径和文件名
            System.out.println(d2.getDirectory()
                + d2.getFile());
        });       f.add(b1);
        f.add(b2 , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new FileDialogTest().init();
    }
}
```

运行上面程序，单击主窗口中的“打开文件”按钮，将看到如图 11.19 所示的文件对话框窗口。

从图 11.19 可以看出，这个文件对话框本身就是 Windows（即 Java 程序所在的运行平台）提供的文件对话框，所以当单击其中的图标、按钮等元素时，该对话框都能提供相应的动作。当选中某个文件后，单击“打开”按钮，将看到程序控制台打印出该文件的绝对路径（文件路径+文件名），这就是由 FileDialog 的 getDirectory()和 getFile()方法提供的。

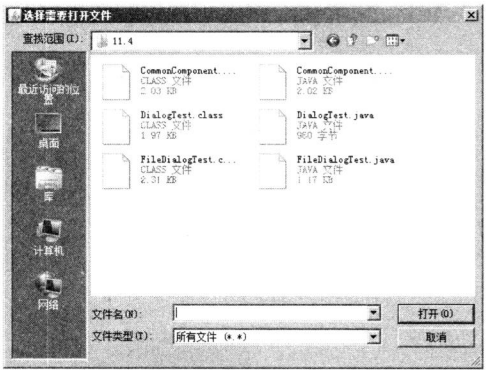


图 11.19 打开文件对话框

11.5 事件处理

前面介绍了如何放置各种组件，从而得到了丰富多彩的图形界面，但这些界面还不能响应用户的任何操作。比如单击前面所有窗口右上角的“×”按钮，但窗口依然不会关闭。因为在 AWT 编程中，所有事件必须由特定对象（事件监听器）来处理，而 Frame 和组件本身并没有事件处理能力。

11.5.1 Java 事件模型的流程

为了使图形界面能够接收用户的操作，必须给各个组件加上事件处理机制。

在事件处理的过程中，主要涉及三类对象。

- Event Source（事件源）：事件发生的场所，通常就是各个组件，例如按钮、窗口、菜单等。
- Event（事件）：事件封装了 GUI 组件上发生的特定事情（通常就是一次用户操作）。如果程序需要获得 GUI 组件上所发生事件的相关信息，都通过 Event 对象来取得。
- Event Listener（事件监听器）：负责监听事件源所发生的事件，并对各种事件做出响应处理。

提示：

有过 JavaScript、VB 等编程经验的读者都知道，事件响应的动作实际上就是一系列的程序语句，通常以方法的形式组织起来。但 Java 是面向对象的编程语言，方法不能独立存在，因此必须以类的形式来组织这些方法，所以事件监听器的核心就是它所包含的方法——这些方法也被称为事件处理器（Event Handler）。当事件源上的事件发生时，事件对象会作为参数传给事件处理器（即事件监听器的实例方法）。

当用户单击一个按钮，或者单击某个菜单项，或者单击窗口右上角的状态按钮时，这些动作就会触发一个相应的事件，该事件由 AWT 封装成相应的 Event 对象，该事件会触发事件源上注册的事件监听器（特殊的 Java 对象），事件监听器调用对应的事件处理器（事件监听器里的实例方法）来做出相应的响应。

AWT 的事件处理机制是一种委派式（Delegation）事件处理方式——普通组件（事件源）将事件的

处理工作委托给特定的对象（事件监听器）；当该事件源发生指定的事件时，就通知所委托的事件监听器，由事件监听器来处理这个事件。

每个组件均可以针对特定的事件指定一个或多个事件监听对象，每个事件监听器也可以监听一个或多个事件源。因为同一个事件源上可能发生多种事件，委派式事件处理方式可以把事件源上可能发生的不同的事件分别授权给不同的事件监听器来处理；同时也可以让一类事件都使用同一个事件监听器来处理。

提示：

委派式事件处理方式明显“抄袭”了人类社会的分工协作，例如某个单位发生了火灾，该单位通常不会自己处理该事件，而是将该事件委派给消防局（事件监听器）处理；如果发生了打架斗殴事件，则委派给公安局（事件监听器）处理；而消防局、公安局也会同时监听多个单位的火灾、打架斗殴事件。这种委派式处理方式将事件源和事件监听器分离，从而提供更好的程序模型，有利于提高程序的可维护性。

图 11.20 显示了 AWT 的事件处理流程示意图。

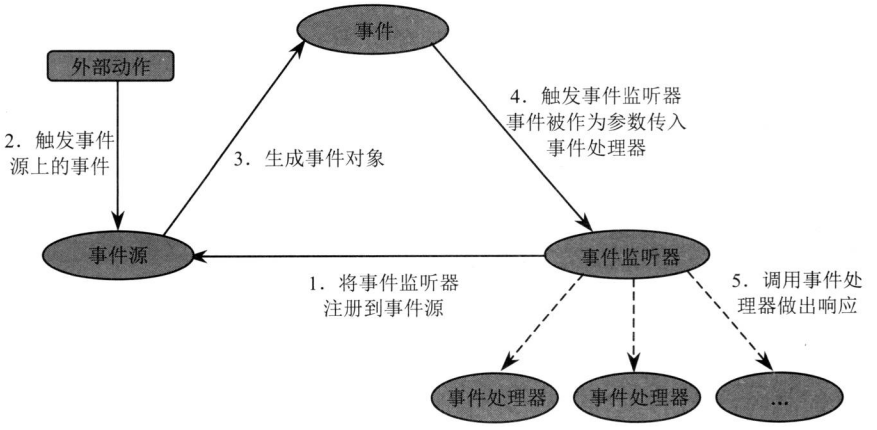


图 11.20　AWT 的事件处理流程示意图

下面以一个简单的 HelloWorld 程序来示范 AWT 事件处理。

程序清单：codes\11\11.5\EventQs.java

```
public class EventQs
{
    private Frame f = new Frame("测试事件");
    private Button ok = new Button("确定");
    private TextField tf = new TextField(30);
    public void init()
    {
        // 注册事件监听器
        ok.addActionListener(new OkListener());   // ①
        f.add(tf);
        f.add(ok , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    // 定义事件监听器类
    class OkListener implements ActionListener   // ②
    {
        // 下面定义的方法就是事件处理器，用于响应特定的事件
        public void actionPerformed(ActionEvent e)      // ③
        {
            System.out.println("用户单击了 ok 按钮");
            tf.setText("Hello World");
        }
    }
    public static void main(String[] args)
    {
```

```
            new EventQs().init();
        }
    }
```

上面程序中粗体字代码用于注册事件监听器，③号粗体字定义的方法就是事件处理器。当程序中的OK按钮被单击时，该处理器被触发，将看到程序中tf文本框内变为"Hello World"，而程序控制台打印出“用户单击了OK按钮”字符串。

从上面程序中可以看出，实现AWT事件处理机制的步骤如下。

1. 实现事件监听器类，该监听器类是一个特殊的Java类，必须实现一个XxxListener接口。
2. 创建普通组件（事件源），创建事件监听器对象。
3. 调用addXxxListener()方法将事件监听器对象注册给普通组件（事件源）。当事件源上发生指定事件时，AWT会触发事件监听器，由事件监听器调用相应的方法（事件处理器）来处理事件，事件源上所发生的事件会作为参数传入事件处理器。

11.5.2 事件和事件监听器

从图11.20中可以看出，当外部动作在AWT组件上进行操作时，系统会自动生成事件对象，这个事件对象是EventObject子类的实例，该事件对象会触发注册到事件源上的事件监听器。

AWT事件机制涉及三个成员：事件源、事件和事件监听器，其中事件源最容易创建，只要通过new来创建一个AWT组件，该组件就是事件源；事件是由系统自动产生的，无须程序员关心。所以，实现事件监听器是整个事件处理的核心。

事件监听器必须实现事件监听器接口，AWT提供了大量的事件监听器接口用于实现不同类型的事件监听器，用于监听不同类型的事件。AWT中提供了丰富的事件类，用于封装不同组件上所发生的特定操作——AWT的事件类都是AWTEvent类的子类，AWTEvent是EventObject的子类。

提示： EventObject类代表更广义的事件对象，包括Swing组件上所触发的事件、数据库连接所触发的事件等。

AWT事件分为两大类：低级事件和高级事件。

1. 低级事件

低级事件是指基于特定动作的事件。比如进入、点击、拖放等动作的鼠标事件，当组件得到焦点、失去焦点时触发焦点事件。

- ComponentEvent：组件事件，当组件尺寸发生变化、位置发生移动、显示/隐藏状态发生改变时触发该事件。
- ContainerEvent：容器事件，当容器里发生添加组件、删除组件时触发该事件。
- WindowEvent：窗口事件，当窗口状态发生改变（如打开、关闭、最大化、最小化）时触发该事件。
- FocusEvent：焦点事件，当组件得到焦点或失去焦点时触发该事件。
- KeyEvent：键盘事件，当按键被按下、松开、单击时触发该事件。
- MouseEvent：鼠标事件，当进行单击、按下、松开、移动鼠标等动作时触发该事件。
- PaintEvent：组件绘制事件，该事件是一个特殊的事件类型，当GUI组件调用update/paint方法来呈现自身时触发该事件，该事件并非专用于事件处理模型。

2. 高级事件（语义事件）

高级事件是基于语义的事件，它可以不和特定的动作相关联，而依赖于触发此事件的类。比如，在TextField中按Enter键会触发ActionEvent事件，在滑动条上移动滑块会触发 AdjustmentEvent事件，选

中项目列表的某一项就会触发 ItemEvent 事件。

- ActionEvent：动作事件，当按钮、菜单项被单击，在 TextField 中按 Enter 键时触发该事件。
- AdjustmentEvent：调节事件，在滑动条上移动滑块以调节数值时触发该事件。
- ItemEvent：选项事件，当用户选中某项，或取消选中某项时触发该事件。
- TextEvent：文本事件，当文本框、文本域里的文本发生改变时触发该事件。

AWT 事件继承层次图如图 11.21 所示。

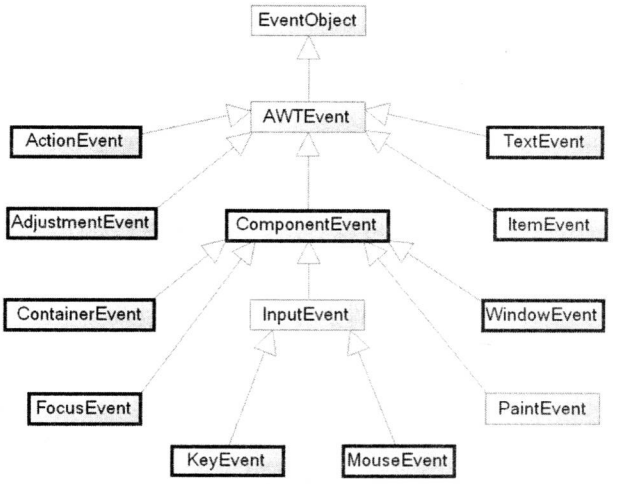


图 11.21　AWT 事件继承层次图

图 11.21 中常用的 AWT 事件使用粗线框圈出；对于没有用粗线框圈出的事件，程序员很少使用它们，它们可能被作为事件基类或作为系统内部实现来使用。

不同的事件需要使用不同的监听器监听，不同的监听器需要实现不同的监听器接口，当指定事件发生后，事件监听器就会调用所包含的事件处理器（实例方法）来处理事件。表 11.1 显示了事件、监听器接口和处理器之间的对应关系。

表 11.1　事件、监听器接口和处理器之间的对应关系

事　件	监听器接口	处理器及触发时机
ActionEvent	ActionListener	actionPerformed：按钮、文本框、菜单项被单击时触发
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged：滑块位置发生改变时触发
ContainerEvent	ContainerListener	componentAdded：向容器中添加组件时触发
		componentRemoved：从容器中删除组件时触发
FocusEvent	FocusListener	focusGained：组件得到焦点时触发
		focusLost：组件失去焦点时触发
ComponentEvent	ComponentListener	componentHidden：组件被隐藏时触发
		componentMoved：组件位置发生改变时触发
		componentResized：组件大小发生改变时触发
		componentShown：组件被显示时触发
KeyEvent	KeyListener	keyPressed：按下某个按键时触发
		keyReleased：松开某个按键时触发
		keyTyped：单击某个按键时触发
MouseEvent	MouseListener	mouseClicked：在某个组件上单击鼠标键时触发
		mouseEntered：鼠标进入某个组件时触发
		mouseExited：鼠标离开某个组件时触发
		mousePressed：在某个组件上按下鼠标键时触发
		mouseReleased：在某个组件上松开鼠标键时触发
	MouseMotionListener	mouseDragged：在某个组件上移动鼠标，且按下鼠标键时触发
		mouseMoved：在某个组件上移动鼠标，且没有按下鼠标键时触发
TextEvent	TextListener	textValueChanged：文本组件里的文本发生改变时触发
ItemEvent	ItemListener	itemStateChanged：某项被选中或取消选中时触发

（续表）

事　件	监听器接口	处理器及触发时机
WindowEvent	WindowListener	windowActivated：窗口被激活时触发
		windowClosed：窗口调用 dispose()即将关闭时触发
		windowClosing：用户单击窗口右上角的“×”按钮时触发
		windowDeactivated：窗口失去激活时触发
		windowDeiconified：窗口被恢复时触发
		windowIconified：窗口最小化时触发
		windowOpened：窗口首次被打开时触发

通过表 11.1 可以大致知道常用组件可能发生哪些事件，以及该事件对应的监听器接口，通过实现该监听器接口就可以实现对应的事件处理器，然后通过 addXxxListener()方法将事件监听器注册给指定的组件（事件源）。当事件源组件上发生特定事件时，被注册到该组件的事件监听器里的对应方法（事件处理器）将被触发。

ActionListener、AdjustmentListener 等事件监听器接口只包含一个抽象方法，这种接口也就是前面介绍的函数式接口，因此可用 Lambda 表达式来创建监听器对象。

提示：

实际上，可以如下理解事件处理模型：当事件源组件上发生事件时，系统将会执行该事件源组件的所有监听器里的对应方法。与前面编程方式不同的是，普通 Java 程序里的方法由程序主动调用，事件处理中的事件处理器方法由系统负责调用。

下面程序示范了一个监听器监听多个组件，一个组件被多个监听器监听的效果。

程序清单：codes\11\11.5\MultiListener.java

```
public class MultiListener
{
    private Frame f = new Frame("测试");
    private TextArea ta = new TextArea(6 , 40);
    private Button b1 = new Button("按钮一");
    private Button b2 = new Button("按钮二");
    public void init()
    {
        // 创建 FirstListener 监听器的实例
        FirstListener fl = new FirstListener();
        // 给 b1 按钮注册两个事件监听器
        b1.addActionListener(fl);
        b1.addActionListener(new SecondListener());
        // 将 f1 事件监听器注册给 b2 按钮
        b2.addActionListener(fl);
        f.add(ta);
        Panel p = new Panel();
        p.add(b1);
        p.add(b2);
        f.add(p, BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    class FirstListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            ta.append("第一个事件监听器被触发,事件源是: "
                + e.getActionCommand() + "\n");
        }
    }
    class SecondListener implements ActionListener
    {
```

```
        public void actionPerformed(ActionEvent e)
        {
            ta.append("单击了“"
                + e.getActionCommand() + "”按钮\n");
        }
    }
    public static void main(String[] args)
    {
        new MultiListener().init();
    }
}
```

上面程序中 b1 按钮增加了两个事件监听器，当用户单击 b1 按钮时，两个监听器的 actionPerform()方法都会被触发；而且 f1 监听器同时监听 b1、b2 两个按钮，当 b1、b2 任意一个按钮被单击时，f1 监听器的 actionPerform()方法都会被触发。

提示：

上面程序中调用了 ActionEvent 对象的 getActionCommand()方法，用于获取被单击按钮上的文本。

运行上面程序，分别单击“按钮一”、“按钮二”一次，将看到如图 11.22 所示的窗口效果。

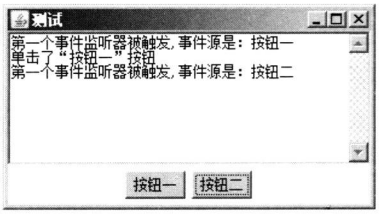


图 11.22 一个按钮被两个监听器监听、一个监听器监听两个按钮

下面程序为窗口添加窗口监听器，从而示范窗口监听器的用法，并允许用户单击窗口右上角的“×”按钮来结束程序。

程序清单：codes\11\11.5\WindowListenerTest.java

```
public class WindowListenerTest
{
    private Frame f = new Frame("测试");
    private TextArea ta = new TextArea(6 , 40);
    public void init()
    {
        // 为窗口添加窗口事件监听器
        f.addWindowListener(new MyListener());
        f.add(ta);
        f.pack();
        f.setVisible(true);
    }
    // 实现一个窗口监听器类
    class MyListener implements WindowListener
    {
        public void windowActivated(WindowEvent e)
        {
            ta.append("窗口被激活！\n");
        }
        public void windowClosed(WindowEvent e)
        {
            ta.append("窗口被成功关闭！\n");
        }
        public void windowClosing(WindowEvent e)
        {
            ta.append("用户关闭窗口！\n");
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent e)
        {
```

```
            ta.append("窗口失去焦点! \n");
        }
        public void windowDeiconified(WindowEvent e)
        {
            ta.append("窗口被恢复! \n");
        }
        public void windowIconified(WindowEvent e)
        {
            ta.append("窗口被最小化! \n");
        }
        public void windowOpened(WindowEvent e)
        {
            ta.append("窗口初次被打开! \n");
        }
    }
    public static void main(String[] args)
    {
        new WindowListenerTest().init();
    }
}
```

上面程序详细监听了窗口的每个动作，当用户单击窗口右上角的每个按钮时，程序都会做出相应的响应，当用户单击窗口中的“×”按钮时，程序将正常退出。

大部分时候，程序无须监听窗口的每个动作，只需要为用户单击窗口中的“×”按钮提供响应即可；无须为每个窗口事件提供响应——即程序只想重写 windowClosing 事件处理器，但因为该监听器实现了 WindowListener 接口，实现该接口就不得不实现该接口里的每个抽象方法，这是非常烦琐的事情。为此，AWT 提供了事件适配器。

11.5.3 事件适配器

事件适配器是监听器接口的空实现——事件适配器实现了监听器接口，并为该接口里的每个方法都提供了实现，这种实现是一种空实现（方法体内没有任何代码的实现）。当需要创建监听器时，可以通过继承事件适配器，而不是实现监听器接口。因为事件适配器已经为监听器接口的每个方法提供了空实现，所以程序自己的监听器无须实现监听器接口里的每个方法，只需要重写自己感兴趣的方法，从而可以简化事件监听器的实现类代码。

注意：

如果某个监听器接口只有一个方法，则该监听器接口就无须提供适配器，因为该接口对应的监听器别无选择，只能重写该方法！如果不重写该方法，就没有必要实现该监听器。

提示：

虽然表 11.2 中只列出了常用的监听器接口对应的事件适配器，实际上，所有包含多个方法的监听器接口都有对应的事件适配器，包括 Swing 中的监听器接口也是如此。

从表 11.2 中可以看出，所有包含多个方法的监听器接口都有一个对应的适配器，但只包含一个方法的监听器接口则没有对应的适配器。

表 11.2 监听器接口和事件适配器对应表

监听器接口	事件适配器	监听器接口	事件适配器
ContainerListener	ContainerAdapter	MouseListener	MouseAdapter
FocusListener	FocusAdapter	MouseMotionListener	MouseMotionAdapter
ComponentListener	ComponentAdapter	WindowListener	WindowAdapter
KeyListener	KeyAdapter		

下面程序通过事件适配器来创建事件监听器。

程序清单：codes\11\11.5\WindowAdapterTest.java

```
public class WindowAdapterTest
{
    private Frame f = new Frame("测试");
    private TextArea ta = new TextArea(6 , 40);
    public void init()
    {
        f.addWindowListener(new MyListener());
        f.add(ta);
        f.pack();
        f.setVisible(true);
    }
    class MyListener extends WindowAdapter
    {
        public void windowClosing(WindowEvent e)
        {
            System.out.println("用户关闭窗口！\n");
            System.exit(0);
        }
    }
    public static void main(String[] args)
    {
        new WindowAdapterTest().init();
    }
}
```

从上面程序中可以看出，窗口监听器继承 WindowAdapter 事件适配器，只需要重写 windowClosing 方法（粗体字方法所示）即可，这个方法才是该程序所关心的——当用户单击“×”按钮时，程序退出。

11.5.4 使用内部类实现监听器

事件监听器是一个特殊的 Java 对象，实现事件监听器对象有如下几种形式。

- 内部类形式：将事件监听器类定义成当前类的内部类。
- 外部类形式：将事件监听器类定义成一个外部类。
- 类本身作为事件监听器类：让当前类本身实现监听器接口或继承事件适配器。
- 匿名内部类形式：使用匿名内部类创建事件监听器对象。

前面示例程序中的所有事件监听器类都是内部类形式，使用内部类可以很好地复用该监听器类，如 MultiListener.java 程序所示；监听器类是外部类的内部类，所以可以自由访问外部类的所有 GUI 组件，这也是内部类的两个优势。

使用内部类来定义事件监听器类的例子可以参考前面的示例程序，此处不再赘述。

11.5.5 使用外部类实现监听器

使用外部类定义事件监听器类的形式比较少见，主要有如下两个原因。

- 事件监听器通常属于特定的 GUI 界面，定义成外部类不利于提高程序的内聚性。
- 外部类形式的事件监听器不能自由访问创建 GUI 界面类中的组件，编程不够简洁。

但如果某个事件监听器确实需要被多个 GUI 界面所共享，而且主要是完成某种业务逻辑的实现，则可以考虑使用外部类形式来定义事件监听器类。下面程序定义了一个外部类作为事件监听器类，该事件监听器实现了发送邮件的功能。

程序清单：codes\11\11.5\MailerListener.java

```
public class MailerListener implements ActionListener
{
    // 该 TextField 文本框用于输入发送邮件的地址
    private TextField mailAddress;
    public MailerListener(){}
    public MailerListener(TextField mailAddress)
    {
        this.mailAddress = mailAddress;
```

```
    }
    public void setMailAddress(TextField mailAddress)
    {
        this.mailAddress = mailAddress;
    }
    // 实现发送邮件
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("程序向“"
            + mailAddress.getText() + "”发送邮件...");
        // 发送邮件的真实实现
    }
}
```

上面的事件监听器类没有与任何 GUI 界面耦合，创建该监听器对象时传入一个 TextField 对象，该文本框里的字符串将被作为收件人地址。下面程序使用了该事件监听器来监听窗口中的按钮。

程序清单：codes\11\11.5\SendMailer.java

```
public class SendMailer
{
    private Frame f = new Frame("测试");
    private TextField tf = new TextField(40);
    private Button send = new Button("发送");
    public void init()
    {
        // 使用 MailerListener 对象作为事件监听器
        send.addActionListener(new MailerListener(tf));
        f.add(tf);
        f.add(send , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SendMailer().init();
    }
}
```

上面程序为“发送”按钮添加事件监听器时，将该窗口中的 TextField 对象传入事件监听器，从而允许事件监听器访问该文本框里的内容。运行上面程序，会看到如图 11.23 所示的运行界面。

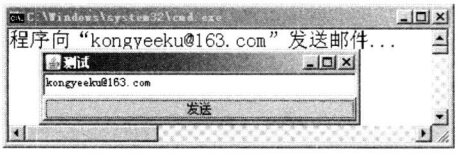


图 11.23 外部类形式的事件监听器类

注意：

实际上并不推荐将业务逻辑实现写在事件监听器中，包含业务逻辑的事件监听器将导致程序的显示逻辑和业务逻辑耦合，从而增加程序后期的维护难度。如果确实有多个事件监听器需要实现相同的业务逻辑功能，则可以考虑使用业务逻辑组件来定义业务逻辑功能，再让事件监听器来调用业务逻辑组件的业务逻辑方法。

11.5.6 类本身作为事件监听器类

类本身作为事件监听器类这种形式使用 GUI 界面类直接作为监听器类，可以直接在 GUI 界面类中定义事件处理器方法。这种形式非常简洁，也是早期 AWT 事件编程里比较喜欢采用的形式。但这种做法有如下两个缺点。

- 这种形式可能造成混乱的程序结构，GUI 界面的职责主要是完成界面初始化工作，但此时还需包含事件处理器方法，从而降低了程序的可读性。
- 如果 GUI 界面类需要继承事件适配器，将会导致该 GUI 界面类不能继承其他父类。

下面程序使用 GUI 界面类作为事件监听器类。

程序清单：codes\11\11.5\SimpleEventHandler.java

```
// GUI 界面类继承 WindowAdapter 作为事件监听器类
public class SimpleEventHandler extends WindowAdapter
{
    private Frame f = new Frame("测试");
    private TextArea ta = new TextArea(6 , 40);
    public void init()
    {
        //将该类的默认对象作为事件监听器对象
        f.addWindowListener(this);
        f.add(ta);
        f.pack();
        f.setVisible(true);
    }
    // GUI 界面类直接包含事件处理器方法
    public void windowClosing(WindowEvent e)
    {
        System.out.println("用户关闭窗口！\n");
        System.exit(0);
    }
    public static void main(String[] args)
    {
        new SimpleEventHandler().init();
    }
}
```

上面程序让 GUI 界面类继承了 WindowAdapter 事件适配器，从而可以在该 GUI 界面类中直接定义事件处理器方法：windowClosing()（如粗体字代码所示）。当为某个组件添加该事件监听器对象时，直接使用 this 作为事件监听器对象即可。

11.5.7 匿名内部类实现监听器

大部分时候，事件处理器都没有复用价值（可复用代码通常会被抽象成业务逻辑方法），因此大部分事件监听器只是临时使用一次，所以使用匿名内部类形式的事件监听器更合适。实际上，这种形式是目前使用最广泛的事件监听器形式。下面程序使用匿名内部类来创建事件监听器。

程序清单：codes\11\11.5\AnonymousEventHandler.java

```
public class AnonymousEventHandler
{
    private Frame f = new Frame("测试");
    private TextArea ta = new TextArea(6 , 40);
    public void init()
    {
        // 以匿名内部类的形式来创建事件监听器对象
        f.addWindowListener(new WindowAdapter()
        {
            // 实现事件处理方法
            public void windowClosing(WindowEvent e)
            {
                System.out.println("用户试图关闭窗口！\n");
                System.exit(0);
            }
        });
        f.add(ta);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new AnonymousEventHandler().init();
    }
}
```

上面程序中的粗体字部分使用匿名内部类创建了一个事件监听器对象，“new 监听器接口”或“new 事件适配器”的形式就是用于创建匿名内部类形式的事件监听器。关于匿名内部类请参考本书 6.7 节内容。

如果事件监听器接口内只包含一个方法，通常会使用 Lambda 表达式代替匿名内部类创建监听器对象，这样就可以避免烦琐的匿名内部类代码。遗憾的是，如果要通过继承事件适配器来创建事件监听器，那就无法使用 Lambda 表达式了。

11.6 AWT 菜单

前面介绍了创建 GUI 界面的方式：将 AWT 组件按某种布局摆放在容器内即可。创建 AWT 菜单的方式与此完全类似：将菜单条、菜单、菜单项组合在一起即可。

11.6.1 菜单条、菜单和菜单项

AWT 中的菜单由如下几个类组合而成。

- MenuBar：菜单条，菜单的容器。
- Menu：菜单组件，菜单项的容器。它也是 MenuItem 的子类，所以可作为菜单项使用。
- PopupMenu：上下文菜单组件（右键菜单组件）。
- MenuItem：菜单项组件。
- CheckboxMenuItem：复选框菜单项组件。
- MenuShortcut：菜单快捷键组件。

图 11.24 显示了 AWT 菜单组件类之间的继承、组合关系。从图中可以看出，MenuBar 和 Menu 都实现了菜单容器接口，所以 MenuBar 可用于盛装 Menu，而 Menu 可用于盛装 MenuItem（包括 Menu 和 CheckboxMenuItem 两个子类对象）。Menu 还有一个子类：PopupMenu，代表上下文菜单，上下文菜单无须使用 MenuBar 盛装。

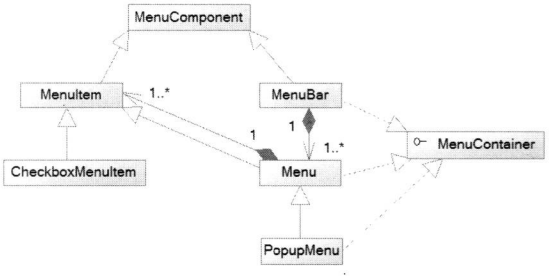


图 11.24 AWT 菜单组件类之间的继承、组合关系

Menu、MenuItem 的构造器都可接收一个字符串参数，该字符串作为其对应菜单、菜单项上的标签文本。除此之外，MenuItem 还可以接收一个 MenuShortcut 对象，该对象用于指定该菜单的快捷键。MenuShortcut 类使用虚拟键代码（而不是字符）来创建快捷键。例如，Ctrl+A（通常都以 Ctrl 键作为快捷键的辅助键）快捷方式通过以下代码创建。

```
MenuShortcut ms = new MenuShortcut(KeyEvent.VK_A);
```

如果该快捷键还需要 Shift 键的辅助，则可使用如下代码。

```
MenuShortcut ms = new MenuShortcut(KeyEvent.VK_A , true);
```

有时候程序还希望对某个菜单进行分组，将功能相似的菜单分成一组，此时需要使用菜单分隔符。AWT 中添加菜单分隔符有如下两种方法。

- 调用 Menu 对象的 addSeparator()方法来添加菜单分隔线。
- 使用添加 new MenuItem("-")的方式来添加菜单分隔线。

创建了 MenuItem、Menu 和 MenuBar 对象之后，调用 Menu 的 add()方法将多个 MenuItem 组合成菜单（也可将另一个 Menu 对象组合进来，从而形成二级菜单），再调用 MenuBar 的 add()方法将多个 Menu 组合成菜单条，最后调用 Frame 对象的 setMenuBar()方法为该窗口添加菜单条。

下面程序示范了为窗口添加菜单的完整程序。

程序清单：codes\11\11.6\SimpleMenu.java

```
public class SimpleMenu
{
```

```
private Frame f = new Frame("测试");
private MenuBar mb = new MenuBar();
Menu file = new Menu("文件");
Menu edit = new Menu("编辑");
MenuItem newItem = new MenuItem("新建");
MenuItem saveItem = new MenuItem("保存");
// 创建 exitItem 菜单项，指定使用“Ctrl+X”快捷键
MenuItem exitItem = new MenuItem("退出"
    , new MenuShortcut(KeyEvent.VK_X));
CheckboxMenuItem autoWrap = new CheckboxMenuItem("自动换行");
MenuItem copyItem = new MenuItem("复制");
MenuItem pasteItem = new MenuItem("粘贴");
Menu format = new Menu("格式");
// 创建 commentItem 菜单项，指定使用“Ctrl+Shift+/”快捷键
    MenuItem commentItem = new MenuItem("注释" ,
new MenuShortcut(KeyEvent.VK_SLASH , true));
MenuItem cancelItem = new MenuItem("取消注释");
private TextArea ta = new TextArea(6 , 40);
public void init()
{
    // 以 Lambda 表达式创建菜单事件监听器
    ActionListener menuListener = e ->
    {
        String cmd = e.getActionCommand();
        ta.append("单击“" + cmd + "”菜单" + "\n");
        if (cmd.equals("退出"))
        {
            System.exit(0);
        }
    };
    // 为 commentItem 菜单项添加事件监听器
    commentItem.addActionListener(menuListener);
    exitItem.addActionListener(menuListener);
    // 为 file 菜单添加菜单项
    file.add(newItem);
    file.add(saveItem);
    file.add(exitItem);
    // 为 edit 菜单添加菜单项
    edit.add(autoWrap);
    // 使用 addSeparator 方法来添加菜单分隔线
    edit.addSeparator();
    edit.add(copyItem);
    edit.add(pasteItem);
    // 为 format 菜单添加菜单项
    format.add(commentItem);
    format.add(cancelItem);
    // 使用添加 new MenuItem("-") 的方式添加菜单分隔线
    edit.add(new MenuItem("-"));
    // 将 format 菜单组合到 edit 菜单中，从而形成二级菜单
    edit.add(format);
    // 将 file、edit 菜单添加到 mb 菜单条中
    mb.add(file);
    mb.add(edit);
    // 为 f 窗口设置菜单条
    f.setMenuBar(mb);
    // 以匿名内部类的形式来创建事件监听器对象
    f.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });
    f.add(ta);
    f.pack();
    f.setVisible(true);
}
```

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

上面程序中的菜单既有复选框菜单项和菜单分隔符，也有二级菜单，并为两个菜单项添加了快捷键，为 commentItem、exitItem 两个菜单项添加了事件监听器。运行该程序，并按“Ctrl+Shift+/”快捷键，将看到如图 11.25 所示的窗口。

提示：

AWT 的菜单组件不能创建图标菜单，如果希望创建带图标的菜单，则应该使用 Swing 的菜单组件：JMenuBar、JMenu、JMenuItem 和 JPopupMenu 组件。Swing 的菜单组件和 AWT 的菜单组件的用法基本相似，读者可参考本程序学习使用 Swing 的菜单组件。

11.6.2 右键菜单

右键菜单使用 PopupMenu 对象表示，创建右键菜单的步骤如下。

① 创建 PopupMenu 的实例。

② 创建多个 MenuItem 的多个实例，依次将这些实例加入 PopupMenu 中。

③ 将 PopupMenu 加入到目标组件中。

④ 为需要出现上下文菜单的组件编写鼠标监听器，当用户释放鼠标右键时弹出右键菜单。

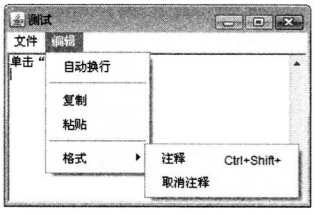


图 11.25 AWT 菜单示例

下面程序创建了一个右键菜单，该右键菜单就是“借用”前面 SimpleMenu 中 edit 菜单下的所有菜单项。

程序清单：codes\11\11.6\PopupMenuTest.java

```
public class PopupMenuTest
{
    private TextArea ta = new TextArea(4 , 30);
    private Frame f = new Frame("测试");
    PopupMenu pop = new PopupMenu();
    CheckboxMenuItem autoWrap =
        new CheckboxMenuItem("自动换行");
    MenuItem copyItem = new MenuItem("复制");
    MenuItem pasteItem = new MenuItem("粘贴");
    Menu format = new Menu("格式");
    // 创建 commentItem 菜单项，指定使用“Ctrl+Shift+/”快捷键
    MenuItem commentItem = new MenuItem("注释" ,
        new MenuShortcut(KeyEvent.VK_SLASH , true));
    MenuItem cancelItem = new MenuItem("取消注释");
    public void init()
    {
        // 以 Lambda 表达式创建菜单事件监听器
        ActionListener menuListener = e ->
        {
            String cmd = e.getActionCommand();
            ta.append("单击“" + cmd + "”菜单" + "\n");
            if (cmd.equals("退出"))
            {
                System.exit(0);
            }
        };
        // 为 commentItem 菜单项添加事件监听
        commentItem.addActionListener(menuListener);
        // 为 pop 菜单添加菜单项
        pop.add(autoWrap);
        // 使用 addSeparator 方法来添加菜单分隔线
        pop.addSeparator();
        pop.add(copyItem);
        pop.add(pasteItem);
```

```
        // 为 format 菜单添加菜单项
        format.add(commentItem);
        format.add(cancelItem);
        // 使用添加 new MenuItem("-")的方式添加菜单分隔线
        pop.add(new MenuItem("-"));
        // 将 format 菜单组合到 pop 菜单中，从而形成二级菜单
        pop.add(format);
        final Panel p = new Panel();
        p.setPreferredSize(new Dimension(300, 160));
        // 向 p 窗口中添加 PopupMenu 对象
        p.add(pop);
        // 添加鼠标事件监听器
        p.addMouseListener(new MouseAdapter()
        {
            public void mouseReleased(MouseEvent e)
            {
                // 如果释放的是鼠标右键
                if (e.isPopupTrigger())
                {
                    pop.show(p , e.getX() , e.getY());
                }
            }
        });
        f.add(p);
        f.add(ta , BorderLayout.NORTH);
        // 以匿名内部类的形式来创建事件监听器对象
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new PopupMenuTest().init();
    }
}
```

运行上面程序，会看到如图 11.26 所示的窗口。

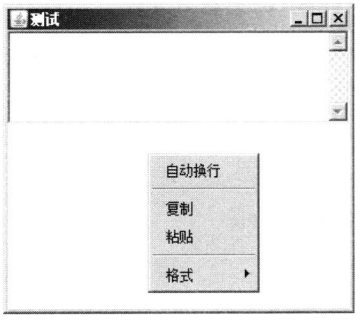
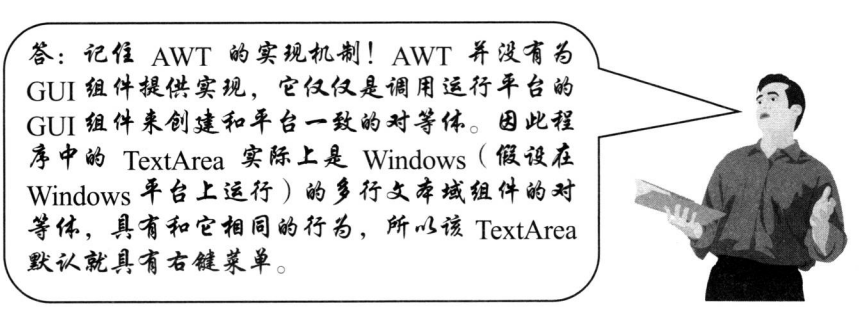


图 11.26　实现右键菜单



11.7 在 AWT 中绘图

很多程序如各种小游戏都需要在窗口中绘制各种图形，除此之外，即使在开发 Java EE 项目时，有时候也必须“动态”地向客户端生成各种图形、图表，比如图形验证码、统计图等，这都需要利用 AWT 的绘图功能。

11.7.1 画图的实现原理

在 Component 类里提供了和绘图有关的三个方法。

- paint(Graphics g)：绘制组件的外观。
- update(Graphics g)：调用 paint()方法，刷新组件外观。
- repaint()：调用 update()方法，刷新组件外观。

上面三个方法的调用关系为：repaint()方法调用 update()方法；update()方法调用 paint()方法。

Container 类中的 update()方法先以组件的背景色填充整个组件区域，然后调用 paint()方法重画组件。Container 类的 update()方法代码如下：

```
public void update(Graphics g) {
    if (isShowing()) {
        //以组件的背景色填充整个组件区域
        if (! (peer instanceof LightweightPeer)) {
            g.clearRect(0, 0, width, height);
        }
        paint(g);
    }
}
```

普通组件的 update()方法则直接调用 paint()方法。

```
public void update(Graphics g) {
    paint(g);
}
```

图 11.27 显示了 paint()、repaint()和 update()三个方法之间的调用关系。

从图 11.27 中可以看出，程序不应该主动调用组件的 paint()和 update()方法，这两个方法都由 AWT 系统负责调用。如果程序希望 AWT 系统重新绘制该组件，则调用该组件的 repaint()方法即可。而 paint()和 update()方法通常被重写。在通常情况下，程序通过重写 paint()方法实现在 AWT 组件上绘图。

重写 update()或 paint()方法时，该方法里包含了一个 Graphics 类型的参数，通过该 Graphics 参数就可以实现绘图功能。

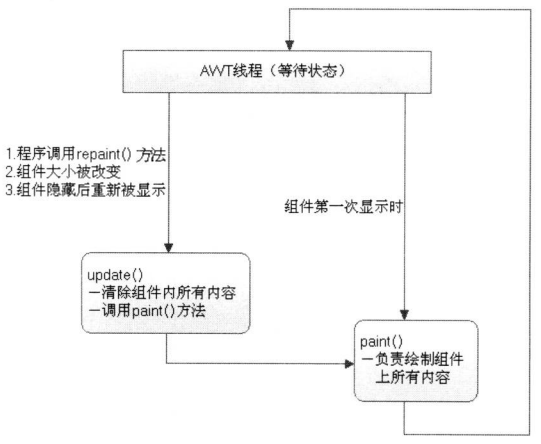


图 11.27 paint()、update()和 repaint()三个方法的调用关系

11.7.2 使用 Graphics 类

Graphics 是一个抽象的画笔对象，Graphics 可以在组件上绘制丰富多彩的几何图形和位图。Graphics 类提供了如下几个方法用于绘制几何图形和位图。

- drawLine()：绘制直线。
- drawString()：绘制字符串。
- drawRect()：绘制矩形。
- drawRoundRect()：绘制圆角矩形。

- drawOval()：绘制椭圆形状。
- drawPolygon()：绘制多边形边框。
- drawArc()：绘制一段圆弧（可能是椭圆的圆弧）。
- drawPolyline()：绘制折线。
- fillRect()：填充一个矩形区域。
- fillRoundRect()：填充一个圆角矩形区域。
- fillOval()：填充椭圆区域。
- fillPolygon()：填充一个多边形区域。
- fillArc()：填充圆弧和圆弧两个端点到中心连线所包围的区域。
- drawImage()：绘制位图。

除此之外，Graphics 还提供了 setColor()和 setFont()两个方法用于设置画笔的颜色和字体（仅当绘制字符串时有效），其中 setColor()方法需要传入一个 Color 参数，它可以使用 RGB、CMYK 等方式设置一个颜色；而 setFont()方法需要传入一个 Font 参数，Font 参数需要指定字体名、字体样式、字体大小三个属性。

提示：

实际上，不仅 Graphics 对象可以使用 setColor()和 setFont()方法来设置画笔的颜色和字体，AWT 普通组件也可以通过 Color()和 Font()方法来改变它的前景色和字体。除此之外，所有组件都有一个 setBackground()方法用于设置组件的背景色。

AWT 专门提供一个 Canvas 类作为绘图的画布，程序可以通过创建 Canvas 的子类，并重写它的 paint()方法来实现绘图。下面程序示范了一个简单的绘图程序。

程序清单：codes\11\11.7\SimpleDraw.java

```
public class SimpleDraw
{
    private final String RECT_SHAPE = "rect";
    private final String OVAL_SHAPE = "oval";
    private Frame f = new Frame("简单绘图");
    private Button rect = new Button("绘制矩形");
    private Button oval = new Button("绘制圆形");
    private MyCanvas drawArea = new MyCanvas();
    // 用于保存需要绘制什么图形的变量
    private String shape = "";
    public void init()
    {
        Panel p = new Panel();
        rect.addActionListener(e ->
        {
            // 设置 shape 变量为 RECT_SHAPE
            shape = RECT_SHAPE;
            // 重画 MyCanvas 对象，即调用它的 repait()方法
            drawArea.repaint();
        });
        oval.addActionListener(e ->
        {
            // 设置 shape 变量为 OVAL_SHAPE
            shape = OVAL_SHAPE;
            // 重画 MyCanvas 对象，即调用它的 repait()方法
            drawArea.repaint();
        });
        p.add(rect);
        p.add(oval);
        drawArea.setPreferredSize(new Dimension(250 , 180));
        f.add(drawArea);
        f.add(p , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
```

```
    public static void main(String[] args)
    {
        new SimpleDraw().init();
    }
    class MyCanvas extends Canvas
    {
        // 重写 Canvas 的 paint()方法，实现绘画
        public void paint(Graphics g)
        {
            Random rand = new Random();
            if (shape.equals(RECT_SHAPE))
            {
                // 设置画笔颜色
                g.setColor(new Color(220, 100, 80));
                // 随机地绘制一个矩形框
                g.drawRect( rand.nextInt(200)
                    , rand.nextInt(120) , 40 , 60);
            }
            if (shape.equals(OVAL_SHAPE))
            {
                // 设置画笔颜色
                g.setColor(new Color(80, 100, 200));
                // 随机地填充一个实心圆形
                g.fillOval( rand.nextInt(200)
                    , rand.nextInt(120) , 50 , 40);
            }
        }
    }
}
```

上面程序定义了一个 MyCanvas 类，它继承了 Canvas 类，重写了 Canvas 类的 paint()方法（上面程序中粗体字代码部分），该方法根据 shape 变量值随机地绘制矩形或填充椭圆区域。窗口中还定义了两个按钮，当用户单击任意一个按钮时，程序调用了 drawArea 对象的 repaint()方法，该方法导致画布重绘（即调用 drawArea 对象的 update()方法，该方法再调用 paint()方法）。

运行上面程序，单击“绘制圆形”按钮，将看到如图 11.28 所示的窗口。

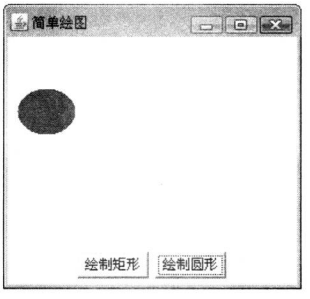


图 11.28 简单绘图

注意：

运行上面程序时，如果改变窗口大小，或者让该窗口隐藏后重新显示都会导致 drawArea 重新绘制形状——这是因为这些动作都会触发组件的 update()方法。

Java 也可用于开发一些动画。所谓动画，就是间隔一定的时间（通常小于 0.1 秒）重新绘制新的图像，两次绘制的图像之间差异较小，肉眼看起来就成了所谓的动画。为了实现间隔一定的时间就重新调用组件的 repaint()方法，可以借助于 Swing 提供的 Timer 类，Timer 类是一个定时器，它有如下一个构造器。

- Timer(int delay, ActionListener listener)：每间隔 delay 毫秒，系统自动触发 ActionListener 监听器里的事件处理器（actionPerformed()方法）。

下面程序示范了一个简单的弹球游戏，其中小球和球拍分别以圆形区域和矩形区域代替，小球开始以随机速度向下运动，遇到边框或球拍时小球反弹；球拍则由用户控制，当用户按下向左、向右键时，球拍将会向左、向右移动。

程序清单：codes\11\11.7\PinBall.java

```
public class PinBall
{
    // 桌面的宽度
    private final int TABLE_WIDTH = 300;
    // 桌面的高度
```

```
private final int TABLE_HEIGHT = 400;
// 球拍的垂直位置
private final int RACKET_Y = 340;
// 下面定义球拍的高度和宽度
private final int RACKET_HEIGHT = 20;
private final int RACKET_WIDTH = 60;
// 小球的大小
private final int BALL_SIZE = 16;
private Frame f = new Frame("弹球游戏");
Random rand = new Random();
// 小球纵向的运行速度
private int ySpeed = 10;
// 返回一个-0.5~0.5的比率，用于控制小球的运行方向
private double xyRate = rand.nextDouble() - 0.5;
// 小球横向的运行速度
private int xSpeed = (int)(ySpeed * xyRate * 2);
// ballX和ballY代表小球的坐标
private int ballX = rand.nextInt(200) + 20;
private int ballY = rand.nextInt(10) + 20;
// racketX代表球拍的水平位置
private int racketX = rand.nextInt(200);
private MyCanvas tableArea = new MyCanvas();
Timer timer;
// 游戏是否结束的旗标
private boolean isLose = false;
public void init()
{
    // 设置桌面区域的最佳大小
    tableArea.setPreferredSize(
        new Dimension(TABLE_WIDTH , TABLE_HEIGHT));
    f.add(tableArea);
    // 定义键盘监听器
    KeyAdapter keyProcessor = new KeyAdapter()
    {
        public void keyPressed(KeyEvent ke)
        {
            // 按下向左、向右键时，球拍水平坐标分别减少、增加
            if (ke.getKeyCode() == KeyEvent.VK_LEFT)
            {
                if (racketX > 0)
                racketX -= 10;
            }
            if (ke.getKeyCode() == KeyEvent.VK_RIGHT)
            {
                if (racketX < TABLE_WIDTH - RACKET_WIDTH)
                racketX += 10;
            }
        }
    };
    // 为窗口和tableArea对象分别添加键盘监听器
    f.addKeyListener(keyProcessor);
    tableArea.addKeyListener(keyProcessor);
    // 定义每0.1秒执行一次的事件监听器
    ActionListener taskPerformer = evt ->
    {
        // 如果小球碰到左边边框
        if (ballX  <= 0 || ballX >= TABLE_WIDTH - BALL_SIZE)
        {
            xSpeed = -xSpeed;
        }
        // 如果小球高度超出了球拍位置，且横向不在球拍范围之内，游戏结束
        if (ballY >= RACKET_Y - BALL_SIZE &&
            (ballX < racketX || ballX > racketX + RACKET_WIDTH))
        {
            timer.stop();
            // 设置游戏是否结束的旗标为true
            isLose = true;
            tableArea.repaint();
```

```
            }
            // 如果小球位于球拍之内，且到达球拍位置，小球反弹
            else if (ballY  <= 0 ||
                (ballY >= RACKET_Y - BALL_SIZE
                    && ballX > racketX && ballX <= racketX + RACKET_WIDTH))
            {
                ySpeed = -ySpeed;
            }
            // 小球坐标增加
            ballY += ySpeed;
            ballX += xSpeed;
            tableArea.repaint();
        };       timer = new Timer(100, taskPerformer);
        timer.start();
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new PinBall().init();
    }
    class MyCanvas extends Canvas
    {
        // 重写 Canvas 的 paint()方法，实现绘画
        public void paint(Graphics g)
        {
            // 如果游戏已经结束
            if (isLose)
            {
                g.setColor(new Color(255, 0, 0));
                g.setFont(new Font("Times" , Font.BOLD, 30));
                g.drawString("游戏已结束！" , 50 ,200);
            }
            // 如果游戏还未结束
            else
            {
                // 设置颜色，并绘制小球
                g.setColor(new Color(240, 240, 80));
                g.fillOval(ballX , ballY , BALL_SIZE, BALL_SIZE);
                // 设置颜色，并绘制球拍
                g.setColor(new Color(80, 80, 200));
                g.fillRect(racketX , RACKET_Y
                    , RACKET_WIDTH , RACKET_HEIGHT);
            }
        }
    }
}
```

运行上面程序，将看到一个简单的弹球游戏，运行效果如图11.29所示。

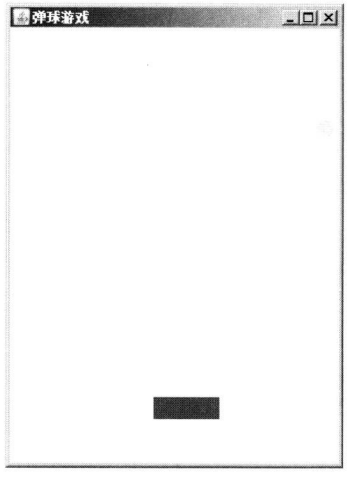


图 11.29 简单的弹球游戏

提示： 上面的弹球游戏还比较简陋，如果为该游戏增加位图背景，使用更逼真的小球位图代替小球，更逼真的球拍位图代替球拍，并在弹球桌面增加一些障碍物，整个弹球游戏将会更有趣味性。细心的读者可能会发现上面的游戏有轻微的闪烁，这是由于AWT组件的绘图没有采用双缓冲技术，当重写paint()方法来绘制图形时，所有的图形都是直接绘制到GUI组件上的，所以多次重新调用paint()方法进行绘制会发生闪烁现象。使用Swing组件就可避免这种闪烁，Swing组件没有提供Canvas对应的组件，使用Swing的Panel组件作为画布即可。

11.8 处理位图

如果仅仅绘制一些简单的几何图形，程序的图形效果依然比较单调。AWT 也允许在组件上绘制位图，Graphics 提供了 drawImage 方法用于绘制位图，该方法需要一个 Image 参数——代表位图，通过该方法就可以绘制出指定的位图。

11.8.1 Image 抽象类和 BufferedImage 实现类

Image 类代表位图，但它是一个抽象类，无法直接创建 Image 对象，为此 Java 为它提供了一个 BufferedImage 子类，这个子类是一个可访问图像数据缓冲区的 Image 实现类。该类提供了一个简单的构造器，用于创建一个 BufferedImage 对象。

- BufferedImage(int width, int height, int imageType)：创建指定大小、指定图像类型的 BufferedImage 对象，其中 imageType 可以是 BufferedImage.TYPE_INT_RGB、BufferedImage. TYPE_BYTE_GRAY 等值。

除此之外，BufferedImage 还提供了一个 getGraphics()方法返回该对象的 Graphics 对象，从而允许通过该 Graphics 对象向 Image 中添加图形。

借助 BufferedImage 可以在 AWT 中实现缓冲技术——当需要向 GUI 组件上绘制图形时，不要直接绘制到该 GUI 组件上，而是先将图形绘制到 BufferedImage 对象中，然后再调用组件的 drawImage 方法一次性地将 BufferedImage 对象绘制到特定组件上。

下面程序通过 BufferedImage 类实现了图形缓冲，并实现了一个简单的手绘程序。

程序清单：codes\11\11.8\HandDraw.java

```
public class HandDraw
{
    // 画图区的宽度
    private final int AREA_WIDTH = 500;
    // 画图区的高度
    private final int AREA_HEIGHT = 400;
    // 下面的 preX、preY 保存了上一次鼠标拖动事件的鼠标坐标
    private int preX = -1;
    private int preY = -1;
    // 定义一个右键菜单用于设置画笔颜色
    PopupMenu pop = new PopupMenu();
    MenuItem redItem = new MenuItem("红色");
    MenuItem greenItem = new MenuItem("绿色");
    MenuItem blueItem = new MenuItem("蓝色");
    // 定义一个 BufferedImage 对象
    BufferedImage image = new BufferedImage(AREA_WIDTH
        , AREA_HEIGHT , BufferedImage.TYPE_INT_RGB);
    // 获取 image 对象的 Graphics
    Graphics g = image.getGraphics();
    private Frame f = new Frame("简单手绘程序");
    private DrawCanvas drawArea = new DrawCanvas();
    // 用于保存画笔颜色
    private Color foreColor = new Color(255, 0 ,0);
    public void init()
    {
        // 定义右键菜单的事件监听器
        ActionListener menuListener = e ->
        {
            if (e.getActionCommand().equals("绿色"))
            {
                foreColor = new Color(0 , 255 , 0);
            }
            if (e.getActionCommand().equals("红色"))
            {
                foreColor = new Color(255 , 0 , 0);
            }
```

```
            if (e.getActionCommand().equals("蓝色"))
            {
                foreColor = new Color(0 , 0 , 255);
            }
        };
        // 为三个菜单添加事件监听器
        redItem.addActionListener(menuListener);
        greenItem.addActionListener(menuListener);
        blueItem.addActionListener(menuListener);
        // 将菜单项组合成右键菜单
        pop.add(redItem);
        pop.add(greenItem);
        pop.add(blueItem);
        // 将右键菜单添加到 drawArea 对象中
        drawArea.add(pop);
        // 将 image 对象的背景色填充成白色
        g.fillRect(0 , 0 ,AREA_WIDTH , AREA_HEIGHT);
        drawArea.setPreferredSize(new Dimension(AREA_WIDTH , AREA_HEIGHT));
        // 监听鼠标移动动作
        drawArea.addMouseMotionListener(new MouseMotionAdapter()
        {
            // 实现按下鼠标键并拖动的事件处理器
            public void mouseDragged(MouseEvent e)
            {
                // 如果 preX 和 preY 大于 0
                if (preX > 0 && preY > 0)
                {
                    // 设置当前颜色
                    g.setColor(foreColor);
                    // 绘制从上一次鼠标拖动事件点到本次鼠标拖动事件点的线段
                    g.drawLine(preX , preY , e.getX() , e.getY());
                }
                // 将当前鼠标事件点的 X、Y 坐标保存起来
                preX = e.getX();
                preY = e.getY();
                // 重绘 drawArea 对象
                drawArea.repaint();
            }
        });
        // 监听鼠标事件
        drawArea.addMouseListener(new MouseAdapter()
        {
            // 实现鼠标键松开的事件处理器
            public void mouseReleased(MouseEvent e)
            {
                // 弹出右键菜单
                if (e.isPopupTrigger())
                {
                    pop.show(drawArea , e.getX() , e.getY());
                }
                // 松开鼠标键时，把上一次鼠标拖动事件的 X、Y 坐标设为-1
                preX = -1;
                preY = -1;
            }
        });
        f.add(drawArea);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new HandDraw().init();
    }
    class DrawCanvas extends Canvas
    {
        // 重写 Canvas 的 paint 方法，实现绘画
        public void paint(Graphics g)
        {
```

```
            // 将 image 绘制到该组件上
            g.drawImage(image , 0 , 0 , null);
        }
    }
}
```

实现手绘功能其实是一种假象：表面上看起来可以随鼠标移动自由画曲线，实际上依然利用 Graphics 的 drawLine()方法画直线，每条直线都是从上一次鼠标拖动事件发生点画到本次鼠标拖动事件发生点。当鼠标拖动时，两次鼠标拖动事件发生点的距离很小，多条极短的直线连接起来，肉眼看起来就是鼠标拖动的轨迹了。上面程序还增加了右键菜单来选择画笔颜色。

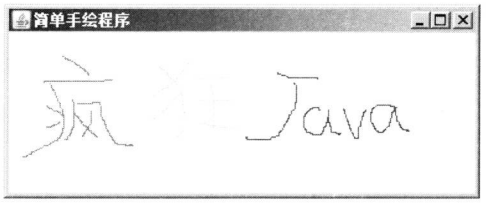


图 11.30　手绘窗口

运行上面程序，出现一个空白窗口，用户可以使用鼠标在该窗口上拖出任意的曲线，如图 11.30 所示。

提示：

上面程序进行手绘时只能选择红、绿、蓝三种颜色，不能调出像 Windows 的颜色选择对话框那种“专业”的颜色选择工具。实际上，Swing 提供了对颜色选择对话框的支持，如果结合 Swing 提供的颜色选择对话框，就可以选择任意的颜色进行画图，并可以提供一些按钮让用户选择绘制直线、折线、多边形等几何图形。如果为该程序分别建立多个 BufferedImage 对象，就可实现多图层效果（每个 BufferedImage 代表一个图层）。

11.8.2　Java 9 增强的 ImageIO

如果希望可以访问磁盘上的位图文件，例如 GIF、JPG 等格式的位图，则需要利用 ImageIO 工具类。ImageIO 利用 ImageReader 和 ImageWriter 读写图形文件，通常程序无须关心该类底层的细节，只需要利用该工具类来读写图形文件即可。

ImageIO 类并不支持读写全部格式的图形文件，程序可以通过 ImageIO 类的如下几个静态方法来访问该类所支持读写的图形文件格式。

- static String[] getReaderFileSuffixes()：返回一个 String 数组，该数组列出 ImageIO 所有能读的图形文件的文件后缀。
- static String[] getReaderFormatNames()：返回一个 String 数组，该数组列出 ImageIO 所有能读的图形文件的非正式格式名称。
- static String[] getWriterFileSuffixes()：返回一个 String 数组，该数组列出 ImageIO 所有能写的图形文件的文件后缀。
- static String[] getWriterFormatNames()：返回一个 String 数组，该数组列出 ImageIO 所有能写的图形文件的非正式格式名称。

下面程序测试了 ImageIO 所支持读写的全部文件格式。

程序清单：codes\11\11.8\ImageIOTest.java

```
public class ImageIOTest
{
    public static void main(String[] args)
    {
        String[] readFormat = ImageIO.getReaderFormatNames();
        System.out.println("-----Image 能读的所有图形文件格式-----");
        for (String tmp : readFormat)
        {
            System.out.println(tmp);
        }
        String[] writeFormat = ImageIO.getWriterFormatNames();
        System.out.println("-----Image 能写的所有图形文件格式-----");
        for (String tmp : writeFormat)
        {
```

```
            System.out.println(tmp);
        }
    }
}
```

运行上面程序就可以看到 Java 所支持的图形文件格式，通过运行结果可以看出，AWT 并不支持 ico 等图标格式。因此，如果需要在 Java 程序中为按钮、菜单等指定图标，也不要使用 ico 格式的图标文件，而应该使用 JPG、GIF 等格式的图形文件。

Java 9 增强了 ImageIO 的功能，ImageIO 可以读写 TIFF（Tag Image File Format）格式的图片。

ImageIO 类包含两个静态方法：read()和 write()，通过这两个方法即可完成对位图文件的读写，调用 wirte()方法输出图形文件时需要指定输出的图形格式，例如 GIF、JPEG 等。下面程序可以将一个原始位图缩小成另一个位图后输出。

程序清单：codes\11\11.8\ZoomImage.java

```
public class ZoomImage
{
    // 下面两个常量设置缩小后图片的大小
    private final int WIDTH = 80;
    private final int HEIGHT = 60;
    // 定义一个 BufferedImage 对象，用于保存缩小后的位图
    BufferedImage image = new BufferedImage(WIDTH , HEIGHT
        , BufferedImage. TYPE_INT_RGB);
    Graphics g = image.getGraphics();
    public void zoom()throws Exception
    {
        // 读取原始位图
        Image srcImage = ImageIO.read(new File("image/board.jpg"));
        // 将原始位图缩小后绘制到 image 对象中
        g.drawImage(srcImage , 0 , 0 , WIDTH , HEIGHT , null);
        // 将 image 对象输出到磁盘文件中
        ImageIO.write(image , "jpeg"
            , new File(System.currentTimeMillis() + ".jpg"));
    }
    public static void main(String[] args)throws Exception
    {
        new ZoomImage().zoom();
    }
}
```

上面程序中第一行粗体字代码从磁盘中读取一个位图文件，第二行粗体字代码则将原始位图按指定大小绘制到 image 对象中，第三行代码再将 image 对象输出，这就完成了位图的缩小（实际上不一定是缩小，程序总是将原始位图缩放到 WIDTH、HEIGHT 常量指定的大小）并输出。

提示：

上面程序总是使用 board.jpg 文件作为原始图片文件，总是缩放到 80 × 60 的尺寸，且总是以当前时间作为文件名来输出该文件，这是为了简化该程序。如果为该程序增加图形界面，允许用户选择需要缩放的原始图片文件和缩放后的目标文件名，并可以设置缩放后的尺寸，该程序将具有更好的实用性。对位图文件进行缩放是非常实用的功能，大部分 Web 应用都允许用户上传的图片，而 Web 应用则需要对用户上传的位图生成相应的缩略图，这就需要对位图进行缩放。

利用 ImageIO 读取磁盘上的位图，然后将这图绘制在 AWT 组件上，就可以做出更加丰富多彩的图形界面程序。

下面程序再次改写第 4 章的五子棋游戏，为该游戏增加图形用户界面，这种改写很简单，只需要改变如下两个地方即可。

- 原来是在控制台打印棋盘和棋子，现在改为使用位图在窗口中绘制棋盘和棋子。
- 原来是靠用户输入下棋坐标，现在改为当用户单击鼠标键时获取下棋坐标，此处需要将鼠标事件的 X、Y 坐标转换为棋盘数组的坐标。

程序清单：codes\11\11.8\Gobang.java

```
public class Gobang
{
    // 下面三个位图分别代表棋盘、黑子、白子
    BufferedImage table;
    BufferedImage black;
    BufferedImage white;
    // 当鼠标移动时的选择框
    BufferedImage selected;
    // 定义棋盘的大小
    private static int BOARD_SIZE = 15;
    // 定义棋盘宽、高多少个像素
    private final int TABLE_WIDTH = 535;
    private final int TABLE_HETGHT = 536;
    // 定义棋盘坐标的像素值和棋盘数组之间的比率
    private final int RATE = TABLE_WIDTH / BOARD_SIZE;
    // 定义棋盘坐标的像素值和棋盘数组之间的偏移距离
    private final int X_OFFSET = 5;
    private final int Y_OFFSET = 6;
    // 定义一个二维数组来充当棋盘
    private String[][] board = new String[BOARD_SIZE][BOARD_SIZE];
    // 五子棋游戏的窗口
    JFrame f = new JFrame("五子棋游戏");
    // 五子棋游戏棋盘对应的Canvas组件
    ChessBoard chessBoard = new ChessBoard();
    // 当前选中点的坐标
    private int selectedX = -1;
    private int selectedY = -1;
    public void init()throws Exception
    {
        table = ImageIO.read(new File("image/board.jpg"));
        black = ImageIO.read(new File("image/black.gif"));
        white = ImageIO.read(new File("image/white.gif"));
        selected = ImageIO.read(new File("image/selected.gif"));
        // 把每个元素赋为"╋"，"╋"代表没有棋子
        for (int i = 0 ; i < BOARD_SIZE ; i++)
        {
            for ( int j = 0 ; j < BOARD_SIZE ; j++)
            {
                board[i][j] = "╋";
            }
        }
        chessBoard.setPreferredSize(new Dimension(
            TABLE_WIDTH , TABLE_HETGHT));
        chessBoard.addMouseListener(new MouseAdapter()
        {
            public void mouseClicked(MouseEvent e)
            {
                // 将用户鼠标事件的坐标转换成棋子数组的坐标
                int xPos = (int)((e.getX() - X_OFFSET) / RATE);
                int yPos = (int)((e.getY() - Y_OFFSET ) / RATE);
                board[xPos][yPos] = "●";
                /*
                电脑随机生成两个整数，作为电脑下棋的坐标，赋给board数组
                还涉及：
                1.如果下棋的点已经有棋子，不能重复下棋
                2.每次下棋后，需要扫描谁赢了
                */
                chessBoard.repaint();
            }
            // 当鼠标退出棋盘区后，复位选中点坐标
            public void mouseExited(MouseEvent e)
            {
                selectedX = -1;
                selectedY = -1;
                chessBoard.repaint();
            }
        });
        chessBoard.addMouseMotionListener(new MouseMotionAdapter()
```

```
        {
            // 当鼠标移动时，改变选中点的坐标
            public void mouseMoved(MouseEvent e)
            {
                selectedX = (e.getX() - X_OFFSET) / RATE;
                selectedY = (e.getY() - Y_OFFSET) / RATE;
                chessBoard.repaint();
            }
        });
        f.add(chessBoard);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)throws Exception
    {
        Gobang gb = new Gobang();
        gb.init();
    }
    class ChessBoard extends JPanel
    {
        // 重写 JPanel 的 paint 方法，实现绘画
        public void paint(Graphics g)
        {
            // 绘制五子棋棋盘
            g.drawImage(table , 0 , 0 , null);
            // 绘制选中点的红框
            if (selectedX >= 0 && selectedY >= 0)
                g.drawImage(selected , selectedX * RATE + X_OFFSET ,
            selectedY * RATE + Y_OFFSET, null);
            // 遍历数组，绘制棋子
            for (int i = 0 ; i < BOARD_SIZE ; i++)
            {
                for ( int j = 0 ; j < BOARD_SIZE ; j++)
                {
                    // 绘制黑棋
                    if (board[i][j].equals("●"))
                    {
                        g.drawImage(black , i * RATE + X_OFFSET
                            , j * RATE + Y_OFFSET, null);
                    }
                    // 绘制白棋
                    if (board[i][j].equals("○"))
                    {
                        g.drawImage(white, i * RATE  + X_OFFSET
                            , j * RATE  + Y_OFFSET, null);
                    }
                }
            }
        }
    }
}
```

上面程序中前面一段粗体字代码负责监听鼠标单击动作，负责把鼠标动作的坐标转换成棋盘数组的坐标，并将对应的数组元素赋值为“●”；后面一段粗体字代码则负责在窗口中绘制棋盘和棋子：先直接绘制棋盘位图，接着遍历棋盘数组，如果数组元素是“●”，则在对应点绘制黑棋，如果数组元素是“○”，则在对应点绘制白棋。

提示： 上面程序为了避免游戏时产生闪烁感，将棋盘所用的画图区改为继承JPanel类，游戏窗口改为使用JFrame类，这两个类都是Swing组件，Swing组件的绘图功能提供了双缓冲技术，可以避免图像闪烁。

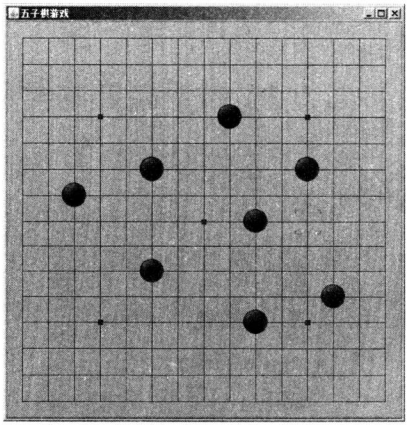


图 11.31 五子棋游戏界面

运行上面程序，会看到如图 11.31 所示的游戏界面。

上面游戏界面中还有一个红色选中框，提示用户鼠标所在的落棋点，这是通过监听鼠标移动事件实现的——当鼠标在游戏界面移动时，程序根据鼠标移动事件发生的坐标来绘制红色选中框。

提示：

上面程序中使用了字符串数组来保存下棋的状态，其实完全可以使用一个 byte[][]数组来保存下棋的状态；数组元素为 0 代表没有棋子；数组元素为 1 代表白棋；数组元素为 2 代表黑棋。上面的游戏程序已经接近完成了，读者只需要按上面思路就可完成这个五子棋游戏，如果能为电脑下棋增加一些智能就更好了。另外，其他小游戏如俄罗斯方块、贪食蛇、连连看、梭哈、斗地主等，只要按这种编程思路来开发都会变得非常简单。实际上，很多程序其实没有想象的那么难，读者只要认真阅读本书，认真完成每章后面的作业，一定可以成为专业的 Java 程序员。

11.9 剪贴板

当进行复制、剪切、粘贴等 Windows 操作时，也许读者从未想过这些操作的实现过程。实际上这是一个看似简单的过程：复制、剪切把一个程序中的数据放置到剪贴板中，而粘贴则读取剪贴板中的数据，并将该数据放入另一个程序中。

剪贴板的复制、剪切和粘贴的过程看似很简单，但实现起来则存在一些具体问题需要处理——假设从一个文字处理程序中复制文本，然后将这段文本复制到另一个文字处理程序中，肯定希望该文字能保持原来的风格，也就是说，剪贴板中必须保留文字原来的格式信息；如果只是将文字复制到纯文本域中，则可以无须包含文字原来的格式信息。除此之外，可能还希望将图像等其他对象复制到剪贴板中。为了处理这种复杂的剪贴板操作，数据提供者（复制、剪切内容的源程序）允许使用多种格式的剪贴板数据，而数据的使用者（粘贴内容的目标程序）则可以从多种格式中选择所需的格式。

提示：

因为 AWT 的实现依赖于底层运行平台的实现，因此 AWT 剪贴板在不同平台上所支持的传输的对象类型并不完全相同。其中 Microsoft、Macintosh 的剪贴板支持传输富格式文本、图像、纯文本等数据，而 X Window 的剪贴板功能则比较有限，它仅仅支持纯文本的剪切和粘贴。读者可以通过查看 JRE 的 jre/lib/flavormap.properties 文件来了解该平台支持哪些类型的对象可以在 Java 程序和系统剪贴板之间传递。

AWT 支持两种剪贴板：本地剪贴板和系统剪贴板。如果在同一个虚拟机的不同窗口之间进行数据传递，则使用 AWT 自己的本地剪贴板就可以了。本地剪贴板则与运行平台无关，可以传输任意格式的数据。如果需要在不同的虚拟机之间传递数据，或者需要在 Java 程序与第三方程序之间传递数据，那就需要使用系统剪贴板了。

11.9.1 数据传递的类和接口

AWT 中剪贴板相关操作的接口和类被放在 java.awt.datatransfer 包下，下面是该包下重要的接口和类的相关说明。

- Clipboard：代表一个剪贴板实例，这个剪贴板既可以是系统剪贴板，也可以是本地剪贴板。
- ClipboardOwner：剪贴板内容的所有者接口，当剪贴板内容的所有权被修改时，系统将会触发该所有者的 lostOwnership 事件处理器。
- Transferable：该接口的实例代表放进剪贴板中的传输对象。
- DataFlavor：用于表述剪贴板中的数据格式。
- StringSelection：Transferable 的实现类，用于传输文本字符串。

- FlavorListener：数据格式监听器接口。
- FlavorEvent：该类的实例封装了数据格式改变的事件。

11.9.2 传递文本

传递文本是最简单的情形，因为 AWT 已经提供了一个 StringSelection 用于传输文本字符串。将一段文本内容（字符串对象）放进剪贴板中的步骤如下。

① 创建一个 Clipboard 实例，既可以创建系统剪贴板，也可以创建本地剪贴板。创建系统剪贴板通过如下代码：

```
Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
```

创建本地剪贴板通过如下代码：

```
Clipboard clipboard = new Clipboard("cb");
```

② 将需要放入剪贴板中的字符串封装成 StringSelection 对象，如下代码所示：

```
StringSelection st = new StringSelection(targetStr);
```

③ 调用剪贴板对象的 setContents()方法将 StringSelection 放进剪贴板中，该方法需要两个参数，第一个参数是 Transferable 对象，代表放进剪贴板中的对象；第二个参数是 ClipboardOwner 对象，代表剪贴板数据的所有者，通常无须关心剪贴板数据的所有者，所以把第二个参数设为 null。

```
clipboard.setContents(st , null);
```

从剪贴板中取出数据则比较简单，调用 Clipboard 对象的 getData(DataFlavor flavor)方法即可取出剪贴板中指定格式的内容，如果指定 flavor 的数据不存在，该方法将引发 UnsupportedFlavorException 异常。为了避免出现异常，可以先调用 Clipboard 对象的 isDataFlavorAvailable(DataFlavor flavor)来判断指定 flavor 的数据是否存在。如下代码所示：

```
if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
{
    String content = (String)clipboard.getData(DataFlavor.stringFlavor);
}
```

下面程序是一个利用系统剪贴板进行复制、粘贴的简单程序。

程序清单：codes\11\11.9\SimpleClipboard.java

```
public class SimpleClipboard
{
    private Frame f = new Frame("简单的剪贴板程序");
    // 获取系统剪贴板
    private Clipboard clipboard = Toolkit
        .getDefaultToolkit().getSystemClipboard();
    // 下面是创建本地剪贴板的代码
    // Clipboard clipboard = new Clipboard("cb");   // ①
    // 用于复制文本的文本框
    private TextArea jtaCopyTo = new TextArea(5,20);
    // 用于粘贴文本的文本框
    private TextArea jtaPaste = new TextArea(5,20);
    private Button btCopy = new Button("复制"); // 复制按钮
    private Button btPaste = new Button("粘贴"); // 粘贴按钮
    public void init()
    {
        Panel p = new Panel();
        p.add(btCopy);
        p.add(btPaste);
        btCopy.addActionListener(event ->
        {
            // 将一个多行文本域里的字符串封装成 StringSelection 对象
            StringSelection contents = new
                StringSelection(jtaCopyTo.getText());
            // 将 StringSelection 对象放入剪贴板
```

```
            clipboard.setContents(contents, null);
        });
        btPaste.addActionListener(event ->
        {
            // 如果剪贴板中包含 stringFlavor 内容
            if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
            {
                try
                {
                    // 取出剪贴板中的 stringFlavor 内容
                    String content = (String)clipboard
                        .getData(DataFlavor.stringFlavor);
                    jtaPaste.append(content);
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        });
        // 创建一个水平排列的 Box 容器
        Box box = new Box(BoxLayout.X_AXIS);
        // 将两个多行文本域放在 Box 容器中
        box.add(jtaCopyTo);
        box.add(jtaPaste);
        // 将按钮所在的 Panel、Box 容器添加到 Frame 窗口中
        f.add(p,BorderLayout.SOUTH);
        f.add(box,BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SimpleClipboard().init();
    }
}
```

上面程序中“复制”按钮的事件监听器负责将第一个文本域的内容复制到系统剪贴板中，“粘贴”按钮的事件监听器则负责取出系统剪贴板中的 stringFlavor 内容，并将其添加到第二个文本域内。运行上面程序，将看到如图 11.32 所示的结果。

因为程序使用的是系统剪贴板，因此可以通过 Windows 的剪贴簿查看器来查看程序放入剪贴板中的内容。在 Windows 的“开始”菜单中运行“clipbrd”程序，将可以看到如图 11.33 所示的窗口。

提示：

Windows 7 系统已经删除了默认的剪贴板查看器，因此读者可以到 Windows XP 的 C:\windows\system32\目录下将 clipbrd.exe 文件复制过来。

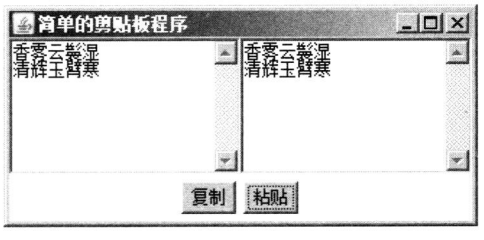


图 11.32 使用剪贴板复制、粘贴文本内容

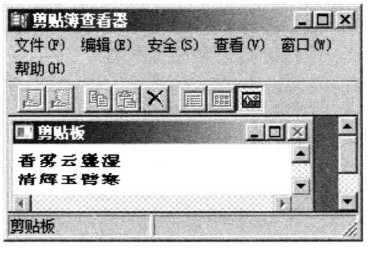


图 11.33 通过剪贴簿查看器查看剪贴板中的内容

11.9.3 使用系统剪贴板传递图像

前面已经介绍了，Transferable 接口代表可以放入剪贴板的传输对象，所以如果希望将图像放入剪贴板内，则必须提供一个 Transferable 接口的实现类，该实现类其实很简单，它封装一个 image 对象，并且向外表现为 imageFlavor 内容。

注意：

JDK为Transferable接口仅提供了一个StringSelection实现类，用于封装字符串内容。但JDK在DataFlavor类中提供了一个imageFlavor常量，用于代表图像格式的DataFlavor，并负责执行所有的复杂操作，以便进行Java图像和剪贴板图像的转换。

下面程序实现了一个ImageSelection类，该类实现了Transferable接口，并实现了该接口所包含的三个方法。

程序清单：codes\11\11.9\ImageSelection.java

```
public class ImageSelection implements Transferable
{
    private Image image;
    // 构造器，负责持有一个 Image 对象
    public ImageSelection(Image image)
    {
        this.image = image;
    }
    // 返回该 Transferable 对象所支持的所有 DataFlavor
    public DataFlavor[] getTransferDataFlavors()
    {
        return new DataFlavor[]{DataFlavor.imageFlavor};
    }
    // 取出该 Transferable 对象里实际的数据
    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException
    {
        if(flavor.equals(DataFlavor.imageFlavor))
        {
            return image;
        }
        else
        {
            throw new UnsupportedFlavorException(flavor);
        }
    }
    // 返回该 Transferable 对象是否支持指定的 DataFlavor
    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(DataFlavor.imageFlavor);
    }
}
```

有了ImageSelection封装类后，程序就可以将指定的Image对象包装成ImageSelection对象放入剪贴板中。下面程序对前面的HandDraw程序进行了改进，改进后的程序允许将用户手绘的图像复制到剪贴板中，也可以把剪贴板里的图像粘贴到该程序中。

程序清单：codes\11\11.9\CopyImage.java

```
public class CopyImage
{
    // 系统剪贴板
    private Clipboard clipboard = Toolkit
        .getDefaultToolkit().getSystemClipboard();
    // 使用 ArrayList 来保存所有粘贴进来的 Image——就是当成图层处理
    java.util.List<Image> imageList = new ArrayList<>();
    // 下面代码与前面 HandDraw 程序中控制绘图的代码一样，省略这部分代码
    ...
        f.add(drawArea);
        Panel p = new Panel();
        Button copy = new Button("复制");
        Button paste = new Button("粘贴");
        copy.addActionListener(event ->
        {
```

```
            // 将 image 对象封装成 ImageSelection 对象
            ImageSelection contents = new ImageSelection(image);
            // 将 ImageSelection 对象放入剪贴板
            clipboard.setContents(contents, null);
        });
        paste.addActionListener(event ->
        {
            // 如果剪贴板中包含 imageFlavor 内容
            if (clipboard.isDataFlavorAvailable(DataFlavor.imageFlavor))
            {
                try
                {
                    // 取出剪贴板中的 imageFlavor 内容，并将其添加到 List 集合中
                    imageList.add((Image)clipboard
                        .getData(DataFlavor.imageFlavor));
                    drawArea.repaint();
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
        });
        p.add(copy);
        p.add(paste);
        f.add(p , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new CopyImage().init();
    }
    class DrawCanvas extends Canvas
    {
        // 重写 Canvas 的 paint 方法，实现绘画
        public void paint(Graphics g)
        {
            // 将 image 绘制到该组件上
            g.drawImage(image , 0 , 0 , null);
            // 将 List 里的所有 Image 对象都绘制出来
            for (Image img : imageList)
            {
                g.drawImage(img , 0 , 0 , null);
            }
        }
    }
}
```

上面程序实现图像复制、粘贴的代码也很简单，就是程序中两段粗体字代码部分：第一段粗体字代码实现了图像复制功能，将 image 对象封装成 ImageSelection 对象，然后调用 Clipboard 的 setContents() 方法将该对象放入剪贴板中；第二段粗体字代码实现了图像粘贴功能，取出剪贴板中的 imageFlavor 内容，返回一个 Image 对象，将该 Image 对象添加到程序的 imageList 集合中。

上面程序中使用了“图层”的概念，使用 imageList 集合来保存所有粘贴到程序中的 Image——每个 Image 就是一个图层，重绘 Canvas 对象时需要绘制 imageList 集合中的每个 image 图像。运行上面程序，当用户在程序中绘制了一些图像后，单击“复制”按钮，将看到程序将该图像复制到了系统剪贴板中，如图 11.34 所示。

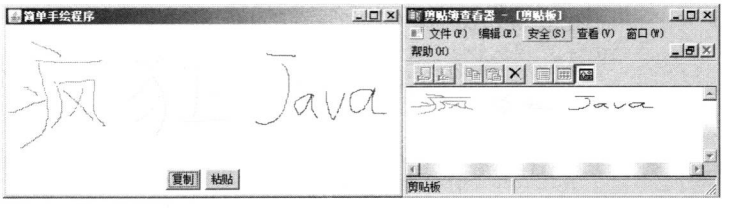


图 11.34 将 Java 程序中的图像放入系统剪贴板中

如果在其他程序中复制一块图像区域（由其他程序负责将图片放入系统剪贴板中），然后单击本程序中的“粘贴”

按钮，就可以将该图像粘贴到本程序中。如图 11.35 所示，将其他程序中的图像复制到 Java 程序中。

图 11.35 将画图程序中的图像复制到 Java 程序中

11.9.4 使用本地剪贴板传递对象引用

本地剪贴板可以保存任何类型的 Java 对象，包括自定义类型的对象。为了将任意类型的 Java 对象保存到剪贴板中，DataFlavor 里提供了一个 javaJVMLocalObjectMimeType 的常量，该常量是一个 MIME 类型字符串：application/x-java-jvm-local-objectref，将 Java 对象放入本地剪贴板中必须使用该 MIME 类型。该 MIME 类型表示仅将对象引用复制到剪贴板中，对象引用只有在同一个虚拟机中才有效，所以只能使用本地剪贴板。创建本地剪贴板的代码如下：

```
Clipboard clipboard = new Clipboard("cp");
```

创建本地剪贴板时需要传入一个字符串，该字符串是剪贴板的名字，通过这种方式允许在一个程序中创建本地剪贴板，就可以实现像 Word 那种多次复制，选择剪贴板粘贴的功能。

注意：

本地剪贴板是 JVM 负责维护的内存区，因此本地剪贴板会随虚拟机的结束而销毁。因此一旦 Java 程序退出，本地剪贴板中的内容将会丢失。

Java 并没有提供封装对象引用的 Transferable 实现类，因此必须自己实现该接口。实现该接口与前面的 ImageSelection 基本相似，一样要实现该接口的三个方法，并持有某个对象的引用。看如下代码。

程序清单：codes\11\11.9\LocalObjectSelection.java

```
public class LocalObjectSelection implements Transferable
{
    // 持有一个对象的引用
    private Object obj;
    public LocalObjectSelection(Object obj)
    {
        this.obj = obj;
    }
    // 返回该 Transferable 对象支持的 DataFlavor
    public DataFlavor[] getTransferDataFlavors()
    {
        DataFlavor[] flavors = new DataFlavor[2];
        //获取被封装对象的类型
        Class clazz = obj.getClass();
        String mimeType = "application/x-java-jvm-local-objectref;"
            + "class=" + clazz.getName();
        try
        {
            flavors[0] = new DataFlavor(mimeType);
            flavors[1] = DataFlavor.stringFlavor;
            return flavors;
        }
```

```
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            return null;
        }
    }
    // 取出该 Transferable 对象封装的数据
    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException
    {
        if(!isDataFlavorSupported(flavor))
        {
            throw new UnsupportedFlavorException(flavor);
        }
        if (flavor.equals(DataFlavor.stringFlavor))
        {
            return obj.toString();
        }
        return obj;
    }
    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(DataFlavor.stringFlavor) ||
            flavor.getPrimaryType().equals("application")
            && flavor.getSubType().equals("x-java-jvm-local-objectref")
           && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
    }
}
```

上面程序创建了一个 DataFlavor 对象，用于表示本地 Person 对象引用的数据格式。创建 DataFlavor 对象可以使用如下构造器。

➢ DataFlavor(String mimeType)：根据 mimeType 字符串构造 DataFlavor。

程序使用上面构造器创建了 MIME 类型为 "application/x-java-jvm-local-objectref;class="+clazz.getName()的 DataFlavor 对象，它表示封装本地对象引用的数据格式。

有了上面的 LocalObjectSelection 封装类后，就可以使用该类来封装某个对象的引用，从而将该对象的引用放入本地剪贴板中。下面程序示范了如何将一个 Person 对象放入本地剪贴板中，以及从本地剪贴板中读取该 Person 对象。

程序清单：codes\11\11.9\CopyPerson.java

```
public class CopyPerson
{
    Frame f = new Frame("复制对象");
    Button copy = new Button("复制");
    Button paste = new Button("粘贴");
    TextField name = new TextField(15);
    TextField age = new TextField(15);
    TextArea ta = new TextArea(3 , 30);
    // 创建本地剪贴板
    Clipboard clipboard = new Clipboard("cp");
    public void init()
    {
        Panel p = new Panel();
        p.add(new Label("姓名"));
        p.add(name);
        p.add(new Label("年龄"));
        p.add(age);
        f.add(p , BorderLayout.NORTH);
        f.add(ta);
        Panel bp = new Panel();
        // 为“复制”按钮添加事件监听器
        copy.addActionListener(e -> copyPerson());
        // 为“粘贴”按钮添加事件监听器
        paste.addActionListener(e ->
        {
            try
```

```
            {
                readPerson();
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
        });       bp.add(copy);
        bp.add(paste);
        f.add(bp , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public void copyPerson()
    {
        // 以 name、age 文本框的内容创建 Person 对象
        Person p = new Person(name.getText()
            , Integer.parseInt(age.getText()));
        // 将 Person 对象封装成 LocalObjectSelection 对象
        LocalObjectSelection ls = new LocalObjectSelection(p);
        // 将 LocalObjectSelection 对象放入本地剪贴板中
        clipboard.setContents(ls , null);
    }
    public void readPerson()throws Exception
    {
        // 创建保存 Person 对象引用的 DataFlavor 对象
        DataFlavor peronFlavor = new DataFlavor(
            "application/x-java-jvm-local-objectref;class=Person");
        // 取出本地剪贴板中的内容
        if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
        {
            Person p = (Person)clipboard.getData(peronFlavor);
            ta.setText(p.toString());
        }
    }
    public static void main(String[] args)
    {
        new CopyPerson().init();
    }
}
```

上面程序中的两段粗体字代码实现了复制、粘贴对象的功能，这两段代码与前面复制、粘贴图像的代码并没有太大的区别，只是前面程序使用了 Java 本身提供的 Data.imageFlavor 数据格式，而此处必须自己创建一个 DataFlavor，用以表示封装 Person 引用的 DataFlavor。运行上面程序，在“姓名”文本框内随意输入一个字符串，在“年龄”文本框内输入年龄数字，然后单击“复制”按钮，就可以将根据两个文本框的内容创建的 Person 对象放入本地剪贴板中；单击“粘贴”按钮，就可以从本地剪贴板中读取刚刚放入的数据，如图 11.36 所示。

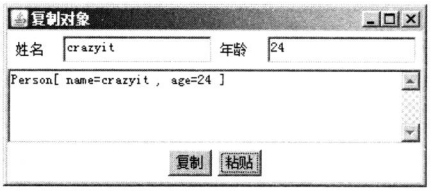


图 11.36 将本地对象复制到本地剪贴板中

上面程序中使用的 Person 类是一个普通的 Java 类，该 Person 类包含了 name 和 age 两个成员变量，并提供了一个包含两个参数的构造器，用于为这两个 Field 成员变量；并重写了 toString()方法，用于返回该 Person 对象的描述性信息。关于 Person 类代码可以参考 codes\11\11.9\CopyPerson.java 文件。

11.9.5 通过系统剪贴板传递 Java 对象

系统剪贴板不仅支持传输文本、图像的基本内容，而且支持传输序列化的 Java 对象和远程对象，复制到剪贴板中的序列化的 Java 对象和远程对象可以使用另一个 Java 程序（不在同一个虚拟机内的程序）来读取。DataFlavor 中提供了 javaSerializedObjectMimeType、javaRemoteObjectMimeType 两个字符串常量来表示序列化的 Java 对象和远程对象的 MIME 类型，这两种 MIME 类型提供了复制对象、读取对象所包含的复杂操作，程序只需创建对应的 Tranferable 实现类即可。

提示：

关于对象序列化请参考本书第 15 章的介绍——如果某个类是可序列化的，则该类的实例可以转换成二进制流，从而可以将该对象通过网络传输或保存到磁盘上。为了保证某个类是可序列化的，只要让该类实现 Serializable 接口即可。

下面程序实现了一个 SerialSelection 类，该类与前面的 ImageSelection、LocalObjectSelection 实现类相似，都需要实现 Tranferable 接口，实现该接口的三个方法，并持有一个可序列化的对象。

程序清单：codes\11\11.9\SerialSelection.java

```
public class SerialSelection implements Transferable
{
    // 持有一个可序列化的对象
    private Serializable obj;
    // 创建该类的对象时传入被持有的对象
    public SerialSelection(Serializable obj)
    {
        this.obj = obj;
    }
    public DataFlavor[] getTransferDataFlavors()
    {
        DataFlavor[] flavors = new DataFlavor[2];
        // 获取被封装对象的类型
        Class clazz = obj.getClass();
        try
        {
            flavors[0] = new DataFlavor(DataFlavor.javaSerializedObjectMimeType
                + ";class=" + clazz.getName());
            flavors[1] = DataFlavor.stringFlavor;
            return flavors;
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
            return null;
        }
    }
    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException
    {
        if(!isDataFlavorSupported(flavor))
        {
            throw new UnsupportedFlavorException(flavor);
        }
        if (flavor.equals(DataFlavor.stringFlavor))
        {
            return obj.toString();
        }
        return obj;
    }
    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(DataFlavor.stringFlavor) ||
            flavor.getPrimaryType().equals("application")
            && flavor.getSubType().equals("x-java-serialized-object")
            && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
    }
}
```

上面程序也创建了一个 DataFlavor 对象，该对象使用的 MIME 类型为"application/x-java-serialized-object;class=" + clazz.getName()，它表示封装可序列化的 Java 对象的数据格式。

有了上面的 SerialSelection 类后，程序就可以把一个可序列化的对象封装成 SerialSelection 对象，并将该对象放入系统剪贴板中，另一个 Java 程序也可以从系统剪贴板中读取该对象。下面复制、读取 Dog 对象的程序与前面的复制、粘贴 Person 对象的程序非常相似，只是该程序使用的是系统剪贴板，

而不是本地剪贴板。

程序清单：codes\11\11.9\CopySerializable.java

```
public class CopySerializable
{
    Frame f = new Frame("复制对象");
    Button copy = new Button("复制");
    Button paste = new Button("粘贴");
    TextField name = new TextField(15);
    TextField age = new TextField(15);
    TextArea ta = new TextArea(3 , 30);
    // 创建系统剪贴板
    Clipboard clipboard = Toolkit.getDefaultToolkit()
        .getSystemClipboard();
    public void init()
    {
        Panel p = new Panel();
        p.add(new Label("姓名"));
        p.add(name);
        p.add(new Label("年龄"));
        p.add(age);
        f.add(p , BorderLayout.NORTH);
        f.add(ta);
        Panel bp = new Panel();
        copy.addActionListener(e -> copyDog());
        paste.addActionListener(e ->
        {
            try
            {
                readDog();
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
        });       bp.add(copy);
        bp.add(paste);
        f.add(bp , BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
    public void copyDog()
    {
        Dog d = new Dog(name.getText()
            , Integer.parseInt(age.getText()));
        // 把 dog 实例封装成 SerialSelection 对象
        SerialSelection ls =new SerialSelection(d);
        // 把 SerialSelection 对象放入系统剪贴板中
        clipboard.setContents(ls , null);
    }
    public void readDog()throws Exception
    {
        DataFlavor peronFlavor = new DataFlavor(DataFlavor
            .javaSerializedObjectMimeType + ";class=Dog");
        if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
        {
            // 从系统剪贴板中读取数据
            Dog d = (Dog)clipboard.getData(peronFlavor);
            ta.setText(d.toString());
        }
    }
    public static void main(String[] args)
    {
        new CopySerializable().init();
    }
}
```

上面程序中的两段粗体字代码实现了复制、粘贴对象的功能，复制时将 Dog 对象封装成

SerialSelection 对象后放入剪贴板中；读取时先创建 application/x-java-serialized-object;class=Dog 类型的 DataFlavor，然后从剪贴板中读取对应格式的内容即可。运行上面程序，在“姓名”文本框内输入字符串，在“年龄”文本框内输入数字，单击“复制”按钮，即可将该 Dog 对象放入系统剪贴板中。

再次运行上面程序（即启动另一个虚拟机），单击窗口中的“粘贴”按钮，将可以看到系统剪贴板中的 Dog 对象被读取出来，启动系统剪贴板也可以看到被放入剪贴板内的 Dog 对象，如图 11.37 所示。

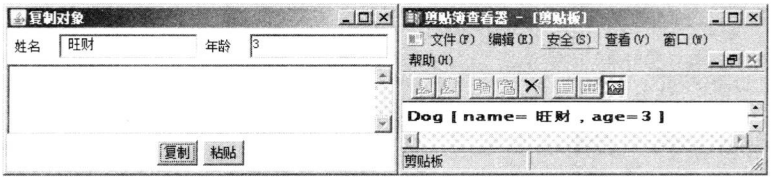


图 11.37 访问系统剪贴板中的 Dog 对象

上面的 Dog 类也非常简单，为了让该类是可序列化的，让该类实现 Serializable 接口即可。读者可以参考 codes\11\11.9\CopySerializable.java 文件来查看 Dog 类的代码。

11.10 拖放功能

拖放是非常常见的操作，人们经常会通过拖放操作来完成复制、剪切功能，但这种复制、剪切操作无须剪贴板支持，程序将数据从拖放源直接传递给拖放目标。这种通过拖放实现的复制、剪切效果也被称为复制、移动。

人们在拖放源中选中一项或多项元素，然后用鼠标将这些元素拖离它们的初始位置，当拖着这些元素在拖放目标上松开鼠标按键时，拖放目标将会查询拖放源，进而访问到这些元素的相关信息，并会相应地启动一些动作。例如，从 Windows 资源管理器中把一个文件图标拖放到 WinPad 图标上，WinPad 将会打开该文件。如果在 Eclipse 中选中一段代码，然后将这段代码拖放到另一个位置，系统将会把这段代码从初始位置删除，并将这段代码放到拖放的目标位置。

除此之外，拖放操作还可以与三种键组合使用，用以完成特殊功能。

- 与 Ctrl 键组合使用：表示该拖放操作完成复制功能。例如，可以在 Eclipse 中通过拖放将一段代码剪切到另一个地方，如果在拖放过程中按住 Ctrl 键，系统将完成代码复制，而不是剪切。
- 与 Shift 键组合使用：表示该拖放操作完成移动功能。有些时候直接拖放默认就是进行复制，例如，从 Windows 资源管理器的一个路径将文件图标拖放到另一个路径，默认就是进行文件复制。此时可以结合 Shift 键来进行拖放操作，用以完成移动功能。
- 与 Ctrl、Shift 键组合使用：表示为目标对象建立快捷方式（在 UNIX 等平台上称为链接）。

在拖放操作中，数据从拖放源直接传递给拖放目标，因此拖放操作主要涉及两个对象：拖放源和拖放目标。AWT 已经提供了对拖放源和拖放目标的支持，分别由 DragSource 和 DropTarget 两个类来表示。下面将具体介绍如何在程序中建立拖放源和拖放目标。

实际上，拖放操作与前面介绍的剪贴板操作有一定的类似之处，它们之间的差别在于：拖放操作将数据从拖放源直接传递给拖放目标，而剪贴板操作则是先将数据传递到剪贴板上，然后再从剪贴板传递给目标。剪贴板操作中被传递的内容使用 Transferable 接口来封装，与此类似的是，拖放操作中被传递的内容也使用 Transferable 来封装；剪贴板操作中被传递的数据格式使用 DataFlavor 来表示，拖放操作中同样使用 DataFlavor 来表示被传递的数据格式。

11.10.1 拖放目标

在 GUI 界面中创建拖放目标非常简单，AWT 提供了 DropTarget 类来表示拖放目标，可以通过该类提供的如下构造器来创建一个拖放目标。

- DropTarget(Component c, int ops, DropTargetListener dtl)：将 c 组件创建成一个拖放目标，该拖放目标默认可接受 ops 值所指定的拖放操作。其中 DropTargetListener 是拖放操作的关键，它负责对拖放操作做出相应的响应。ops 可接受如下几个值。
  - DnDConstants.ACTION_COPY：表示“复制”操作的 int 值。
  - DnDConstants.ACTION_COPY_OR_MOVE：表示“复制”或“移动”操作的 int 值。
  - DnDConstants.ACTION_LINK：表示建立“快捷方式”操作的 int 值。
  - DnDConstants.ACTION_MOVE：表示“移动”操作的 int 值。
  - DnDConstants.ACTION_NONE：表示无任何操作的 int 值。

例如，下面代码将一个 JFrame 对象创建成拖放目标。

```
// 将当前窗口创建成拖放目标
new DropTarget(jf, DnDConstants.ACTION_COPY , new ImageDropTargetListener());
```

正如从上面代码中所看到的，创建拖放目标时需要传入一个 DropTargetListener 监听器，该监听器负责处理用户的拖放动作。该监听器里包含如下 5 个事件处理器。

- dragEnter(DropTargetDragEvent dtde)：当光标进入拖放目标时将触发 DropTargetListener 监听器的该方法。
- dragExit(DropTargetEvent dtde)：当光标移出拖放目标时将触发 DropTargetListener 监听器的该方法。
- dragOver(DropTargetDragEvent dtde)：当光标在拖放目标上移动时将触发 DropTargetListener 监听器的该方法。
- drop(DropTargetDropEvent dtde)：当用户在拖放目标上松开鼠标键，拖放结束时将触发 DropTargetListener 监听器的该方法。
- dropActionChanged(DropTargetDragEvent dtde)：当用户在拖放目标上改变了拖放操作，例如按下或松开了 Ctrl 等辅助键时将触发 DropTargetListener 监听器的该方法。

通常程序不想为上面每个方法提供响应，即不想重写 DropTargetListener 监听器的每个方法，只想重写我们关心的方法，可以通过继承 DropTargetAdapter 适配器来创建拖放监听器。下面程序利用拖放目标创建了一个简单的图片浏览工具，当用户把一个或多个图片文件拖入该窗口时，该窗口将会自动打开每个图片文件。

程序清单：codes\11\11.10\DropTargetTest.java

```
public class DropTargetTest
{
    final int DESKTOP_WIDTH = 480;
    final int DESKTOP_HEIGHT = 360;
    final int FRAME_DISTANCE = 30;
    JFrame jf = new JFrame("测试拖放目标——把图片文件拖入该窗口");
    // 定义一个虚拟桌面
    private JDesktopPane desktop = new JDesktopPane();
    // 保存下一个内部窗口的坐标点
    private int nextFrameX;
    private int nextFrameY;
    // 定义内部窗口为虚拟桌面的 1/2 大小
    private int width = DESKTOP_WIDTH / 2;
    private int height = DESKTOP_HEIGHT / 2;
    public void init()
    {
        desktop.setPreferredSize(new Dimension(DESKTOP_WIDTH
            , DESKTOP_HEIGHT));
        // 将当前窗口创建成拖放目标
        new DropTarget(jf, DnDConstants.ACTION_COPY
            , new ImageDropTargetListener());
        jf.add(desktop);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
```

```
        jf.setVisible(true);
    }
    class ImageDropTargetListener extends DropTargetAdapter
    {
        public void drop(DropTargetDropEvent event)
        {
            // 接受复制操作
            event.acceptDrop(DnDConstants.ACTION_COPY);
            // 获取拖放的内容
            Transferable transferable = event.getTransferable();
            DataFlavor[] flavors = transferable.getTransferDataFlavors();
            // 遍历拖放内容里的所有数据格式
            for (int i = 0; i < flavors.length; i++)
            {
                DataFlavor d = flavors[i];
                try
                {
                    // 如果拖放内容的数据格式是文件列表
                    if (d.equals(DataFlavor.javaFileListFlavor))
                    {
                        // 取出拖放操作里的文件列表
                        List fileList = (List)transferable
                            .getTransferData(d);
                        for (Object f : fileList)
                        {
                            // 显示每个文件
                            showImage((File)f , event);
                        }
                    }
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
                // 强制拖放操作结束，停止阻塞拖放目标
                event.dropComplete(true);    // ①
            }
        }
        // 显示每个文件的工具方法
        private void showImage(File f , DropTargetDropEvent event)
            throws IOException
        {
            Image image = ImageIO.read(f);
            if (image == null)
            {
                // 强制拖放操作结束，停止阻塞拖放目标
                event.dropComplete(true);    // ②
                JOptionPane.showInternalMessageDialog(desktop
                    , "系统不支持这种类型的文件");
                // 方法返回，不会继续操作
                return;
            }
            ImageIcon icon = new ImageIcon(image);
            // 创建内部窗口显示该图片
            JInternalFrame iframe = new JInternalFrame(f.getName()
                , true , true , true , true);
            JLabel imageLabel = new JLabel(icon);
            iframe.add(new JScrollPane(imageLabel));
            desktop.add(iframe);
            // 设置内部窗口的原始位置（内部窗口默认大小是0×0，放在0,0位置）
            iframe.reshape(nextFrameX, nextFrameY, width, height);
            // 使该窗口可见，并尝试选中它
            iframe.show();
            // 计算下一个内部窗口的位置
            nextFrameX += FRAME_DISTANCE;
            nextFrameY += FRAME_DISTANCE;
            if (nextFrameX + width > desktop.getWidth())
                nextFrameX = 0;
```

```
            if (nextFrameY + height > desktop.getHeight())
                nextFrameY = 0;
        }
    }
    public static void main(String[] args)
    {
        new DropTargetTest().init();
    }
}
```

上面程序中粗体字代码部分创建了一个拖放目标，创建拖放目标很简单，关键是需要为该拖放目标编写事件监听器。上面程序中采用 ImageDropTargetListener 对象作为拖放目标的事件监听器，该监听器重写了 drop()方法，即当用户在拖放目标上松开鼠标按键时触发该方法。drop()方法里通过 DropTargetDropEvent 对象的 getTransferable()方法取出被拖放的内容，一旦获得被拖放的内容后，程序就可以对这些内容进行适当处理，本例中只处理被拖放格式是 DataFlavor.javaFileListFlavor（文件列表）的内容，处理方法是把所有的图片文件使用内部窗口显示出来。

运行该程序时，只要用户把图片文件拖入该窗口，程序就会使用内部窗口显示该图片。

注意：

上面程序中①②处的 event.dropComplete(true);代码用于强制结束拖放事件，释放拖放目标的阻塞，如果没有调用该方法，或者在弹出对话框之后调用该方法，将会导致拖放目标被阻塞。在对话框被处理之前，拖放目标窗口也不能获得焦点，这可能不是程序希望的效果，所以程序在弹出内部对话框之前强制结束本次拖放操作（因为文件格式不对），释放拖放目标的阻塞。

上面程序中只处理 DataFlavor.javaFileListFlavor 格式的拖放内容；除此之外，还可以处理文本格式的拖放内容，文本格式的拖放内容使用 DataFlavor.stringFlavor 格式来表示。

更复杂的情况是，可能被拖放的内容是带格式的内容，如 text/html 和 text/rtf 等。为了处理这种内容，需要选择合适的数据格式，如下代码所示：

```
// 如果被拖放的内容是 text/html 格式的输入流
if (d. isMimeTypeEqual("text/html") && d.getRepresentationClass()
    == InputStream.class)
{
    String charset = d.getParameter("charset");
    InputStreamReader reader = new InputStreamReader(
    transferable.getTransferData(d) , charset);
    // 使用 IO 流读取拖放操作的内容
    ...
}
```

关于如何使用 IO 流来处理被拖放的内容，读者需要参考本书第 15 章的内容。

11.10.2 拖放源

前面程序使用 DropTarget 创建了一个拖放目标，直接使用系统资源管理器作为拖放源。下面介绍如何在 Java 程序中创建拖放源，创建拖放源比创建拖放目标要复杂一些，因为程序需要把被拖放内容封装成 Transferable 对象。

创建拖放源的步骤如下。

① 调用 DragSource 的 getDefaultDragSource()方法获得与平台关联的 DragSource 对象。

② 调用 DragSource 对象的 createDefaultDragGestureRecognizer(Component c, int actions, DragGestureListener dgl)方法将指定组件转换成拖放源。其中 actions 用于指定该拖放源可接受哪些拖放操作，而 dgl 是一个拖放监听器，该监听器里只有一个方法：dragGestureRecognized()，当系统检测到用户开始拖放时将会触发该方法。

如下代码将会把一个 JLabel 对象转换为拖放源。

```
// 将 srcLabel 组件转换为拖放源
dragSource.createDefaultDragGestureRecognizer(srcLabel,
    DnDConstants.ACTION_COPY_OR_MOVE, new MyDragGestureListener()
```

③ 为第 2 步中的 DragGestureListener 监听器提供实现类，该实现类需要重写该接口里包含的 dragGestureRecognized()方法，该方法负责把拖放内容封装成 Transferable 对象。

下面程序示范了如何把一个 JLabel 转换成拖放源。

程序清单：codes\11\11.10\DragSourceTest.java

```
public class DragSourceTest
{
    JFrame jf = new JFrame("Swing 的拖放支持");
    JLabel srcLabel = new JLabel("Swing 的拖放支持.\n"
        +"将该文本域的内容拖入其他程序.\n");
    public void init()
    {
        DragSource dragSource = DragSource.getDefaultDragSource();
        // 将 srcLabel 转换成拖放源，它能接受复制、移动两种操作
        dragSource.createDefaultDragGestureRecognizer(srcLabel
            , DnDConstants.ACTION_COPY_OR_MOVE
            , event -> {
            // 将 JLabel 里的文本信息包装成 Transferable 对象
            String txt = srcLabel.getText();
            Transferable transferable = new StringSelection(txt);
            // 继续拖放操作，拖放过程中使用手状光标
            event.startDrag(Cursor.getPredefinedCursor(Cursor
                .HAND_CURSOR), transferable);
        });
        jf.add(new JScrollPane(srcLabel));
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new DragSourceTest().init();
    }
}
```

上面程序中粗体字代码负责把一个 JLabel 组件创建成拖放源，创建拖放源时指定了一个 DragGestureListener 对象，该对象的 dragGestureRecognized()方法负责将 JLabel 上的文本转换成 Transferable 对象后继续拖放。

运行上面程序后，可以把程序窗口中 JLabel 标签的内容直接拖到 Eclipse 编辑窗口中，或者直接拖到 EditPlus 编辑窗口中。

除此之外，如果程序希望能精确监听光标在拖放源上的每个细节，则可以调用 DragGestureEvent 对象的 startDrag(Cursor dragCursor, Transferable transferable, DragSourceListener dsl)方法来继续拖放操作。该方法需要一个 DragSourceListener 监听器对象，该监听器对象里提供了如下几个方法。

- dragDropEnd(DragSourceDropEvent dsde)：当拖放操作已经完成时将会触发该方法。
- dragEnter(DragSourceDragEvent dsde)：当光标进入拖放源组件时将会触发该方法。
- dragExit(DragSourceEvent dse)：当光标离开拖放源组件时将会触发该方法。
- dragOver(DragSourceDragEvent dsde)：当光标在拖放源组件上移动时将会触发该方法。
- dropActionChanged(DragSourceDragEvent dsde)：当用户在拖放源组件上改变了拖放操作，例如按下或松开 Ctrl 等辅助键时将会触发该方法。

掌握了开发拖放源、拖放目标的方法之后，如果接下来在同一个应用程序中既包括拖放源，也包括拖放目标，这样即可在同一个 Java 程序的不同组件之间相互拖动内容。

11.11 本章小结

本章主要介绍了Java AWT编程的基本知识，虽然在实际开发中很少直接使用AWT组件来开发GUI应用，但本章所介绍的知识会作为Swing GUI编程的基础。实际上，AWT编程的布局管理、事件机制、剪贴板内容依然适合Swing GUI编程，所以读者应好好掌握本章内容。

本章介绍了Java GUI界面编程以及AWT的基本概念，详细介绍了AWT容器和布局管理器。本章重点介绍了Java GUI编程的事件机制，详细描述了事件源、事件、事件监听器之间的运行机制，AWT的事件机制也适合Swing的事件处理。除此之外，本章也大致介绍了AWT里的常用组件，如按钮、文本框、对话框、菜单等。本章还介绍了如何在Java程序中绘图，包括绘制各种基本几何图形和绘制位图，并通过简单的弹球游戏介绍了如何在Java程序中实现动画效果。

本章最后介绍了Java剪贴板的用法，通过使用剪贴板，可以让Java程序和操作系统进行数据交换，从而允许把Java程序的数据传入平台中的其他程序，也可以把其他程序中的数据传入Java程序。

➤➤ 本章练习

1. 开发图形界面计算器。
2. 开发桌面弹球游戏。
3. 开发Windows画图程序。
4. 开发图形界面五子棋。

CHAPTER

12

第12章 Swing编程

本章要点

- Swing 编程基础
- Swing 组件的继承层次
- 常见 Swing 组件的用法
- 使用 JToolBar 创建工具条
- 颜色选择对话框和文件浏览对话框
- Swing 提供的特殊容器
- Swing 的简化拖放操作
- 使用 JLayer 装饰组件
- 开发透明的、不规则形状窗口
- 开发进度条
- 开发滑动条
- 使用 JTree 和 TreeModel 开发树
- 使用 JTable 和 TableModel 开发表格
- 使用 JTextPane 组件

使用 Swing 开发图形界面比 AWT 更加优秀，因为 Swing 是一种轻量级组件，它采用 100%的 Java 实现，不再依赖于本地平台的图形界面，所以可以在所有平台上保持相同的运行效果，对跨平台支持比较出色。

除此之外，Swing 提供了比 AWT 更多的图形界面组件，因此可以开发出更美观的图形界面。由于 AWT 需要调用底层平台的 GUI 实现，所以 AWT 只能使用各种平台上 GUI 组件的交集，这大大限制了 AWT 所支持的 GUI 组件。对 Swing 而言，几乎所有组件都采用纯 Java 实现，所以无须考虑底层平台是否支持该组件，因此 Swing 可以提供如 JTabbedPane、JDesktopPane、JInternalFrame 等特殊的容器，也可以提供像 JTree、JTable、JSpinner、JSlider 等特殊的 GUI 组件。

除此之外，Swing 组件都采用 MVC（Model-View-Controller，即模型－视图－控制器）设计模式，从而可以实现 GUI 组件的显示逻辑和数据逻辑的分离，允许程序员自定义 Render 来改变 GUI 组件的显示外观，提供更多的灵活性。

12.1 Swing 概述

前一章已经介绍过 AWT 和 Swing 的关系，因此不难知道：实际使用 Java 开发图形界面程序时，很少使用 AWT 组件，绝大部分时候都是用 Swing 组件开发的。Swing 是由 100%纯 Java 实现的，不再依赖于本地平台的 GUI，因此可以在所有平台上都保持相同的界面外观。独立于本地平台的 Swing 组件被称为轻量级组件；而依赖于本地平台的 AWT 组件被称为重量级组件。

由于 Swing 的所有组件完全采用 Java 实现，不再调用本地平台的 GUI，所以导致 Swing 图形界面的显示速度要比 AWT 图形界面的显示速度慢一些，但相对于快速发展的硬件设施而言，这种微小的速度差别无妨大碍。

使用 Swing 开发图形界面有如下几个优势。

- Swing 组件不再依赖于本地平台的 GUI，无须采用各种平台的 GUI 交集，因此 Swing 提供了大量图形界面组件，远远超出了 AWT 所提供的图形界面组件集。
- Swing 组件不再依赖于本地平台 GUI，因此不会产生与平台相关的 bug。
- Swing 组件在各种平台上运行时可以保证具有相同的图形界面外观。

Swing 提供的这些优势，让 Java 图形界面程序真正实现了“Write Once, Run Anywhere”的目标。

除此之外，Swing 还有如下两个特征。

- Swing 组件采用 MVC（Model-View-Controller，即模型－视图－控制器）设计模式，其中模型（Model）用于维护组件的各种状态，视图（View）是组件的可视化表现，控制器（Controller）用于控制对于各种事件、组件做出怎样的响应。当模型发生改变时，它会通知所有依赖它的视图，视图会根据模型数据来更新自己。Swing 使用 UI 代理来包装视图和控制器，还有另一个模型对象来维护该组件的状态。例如，按钮 JButton 有一个维护其状态信息的模型 ButtonModel 对象。Swing 组件的模型是自动设置的，因此一般都使用 JButton，而无须关心 ButtonModel 对象。因此，Swing 的 MVC 实现也被称为 Model-Delegate（模型－代理）。

提示：对于一些简单的 Swing 组件通常无须关心它对应的 Model 对象，但对于一些高级的 Swing 组件，如 JTree、JTable 等需要维护复杂的数据，这些数据就是由该组件对应的 Model 来维护的。另外，通过创建 Model 类的子类或通过实现适当的接口，可以为组件建立自己的模型，然后用 setModel()方法把模型与组件关联起来。

- Swing 在不同的平台上表现一致，并且有能力提供本地平台不支持的显示外观。由于 Swing 组件采用 MVC 模式来维护各组件，所以当组件的外观被改变时，对组件的状态信息（由模型维护）没有任何影响。因此，Swing 可以使用插拔式外观感觉（Pluggable Look And Feel，PLAF）

来控制组件外观，使得 Swing 图形界面在同一个平台上运行时能拥有不同的外观，用户可以选择自己喜欢的外观。相比之下，在 AWT 图形界面中，由于控制组件外观的对等类与具体平台相关，因此 AWT 组件总是具有与本地平台相同的外观。

Swing 提供了多种独立于各种平台的 LAF（Look And Feel），默认是一种名为 Metal 的 LAF，这种 LAF 吸收了 Macintosh 平台的风格，因此显得比较漂亮。Java 7 则提供了一种名为 Nimbus 的 LAF，这种 LAF 更加漂亮。

为了获取当前 JRE 所支持的 LAF，可以借助于 UIManager 的 getInstalledLookAndFeels()方法，如下程序所示。

程序清单：codes\ 12\12.1\AllLookAndFeel.java

```
public class AllLookAndFeel
{
    public static void main(String[] args)
    {
        System.out.println("当前系统可用的所有 LAF:");
        for (UIManager.LookAndFeelInfo info :
            UIManager.getInstalledLookAndFeels())
        {
            System.out.println(info.getName()
                + "--->" + info);
        }
    }
}
```

提示：

除可以使用 Java 默认提供的数量不多的几种 LAF 之外，还有大量的 Java 爱好者提供了各种开源的 LAF，有兴趣的读者可以自行去下载、体验各种 LAF，使用不同的 LAF 可以让 Swing 应用程序更加美观。

12.2 Swing 基本组件的用法

前面已经提到，Swing 为所有的 AWT 组件提供了对应实现（除 Canvas 组件之外，因为在 Swing 中无须继承 Canvas 组件），通常在 AWT 组件的组件名前添加“J”就变成了对应的 Swing 组件。

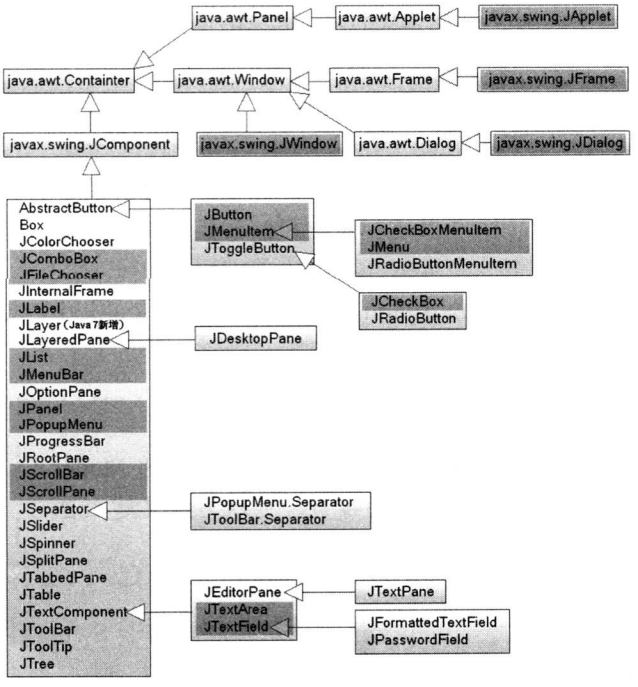


图 12.1 Swing 组件继承层次图

12.2.1 Java 的 Swing 组件层次

大部分Swing组件都是JComponent抽象类的直接或间接子类（并不是全部的 Swing 组件），JComponent 类定义了所有子类组件的通用方法，JComponent 类是 AWT 里 java.awt.Container 类的子类，这也是 AWT 和 Swing 的联系之一。绝大部分 Swing 组件类继承了 Container 类，所以 Swing 组件都可作为容器使用（JFrame 继承了 Frame 类）。图 12.1 显示了 Swing 组件继承层次图。

图 12.1 中绘制了 Swing 所提供的绝大部分组件，其中以灰色区域覆盖的组件可以找到与之对应的 AWT 组件；JWindow 与 AWT 中的 Window 相似，代表没有标题的窗口。读者不难发现这些 Swing 组件的类名和对应 AWT 组件的类型也基本一致，只要在原来的 AWT 组件类型前添加“J”

即可，但有如下几个例外。

- JComboBox：对应于 AWT 里的 Choice 组件，但比 Choice 组件功能更丰富。
- JFileChooser：对应于 AWT 里的 FileDialog 组件。
- JScrollBar：对应于 AWT 里的 Scrollbar 组件，注意两个组件类名中 b 字母的大小写差别。
- JCheckBox：对应于 AWT 里的 Checkbox 组件，注意两个组件类名中 b 字母的大小写差别。
- JCheckBoxMenuItem：对应于 AWT 里的 CheckboxMenuItem 组件，注意两个组件类名中 b 字母的大小写差别。

上面 JCheckBox 和 JCheckBoxMenuItem 与 Checkbox 和 CheckboxMenuItem 的差别主要是由早期 Java 命名不太规范造成的。

注意：

从图 12.1 中可以看出，Swing 中包含了 4 个组件直接继承了 AWT 组件，而不是从 JComponent 派生的，它们分别是：JFrame、JWindow、JDialog 和 JApplet，它们并不是轻量级组件，而是重量级组件（需要部分委托给运行平台上 GUI 组件的对等体）。

将 Swing 组件按功能来分，又可分为如下几类。

- 顶层容器：JFrame、JApplet、JDialog 和 JWindow。
- 中间容器：JPanel、JScrollPane、JSplitPane、JToolBar 等。
- 特殊容器：在用户界面上具有特殊作用的中间容器，如 JInternalFrame、JRootPane、JLayeredPane 和 JDestopPane 等。
- 基本组件：实现人机交互的组件，如 JButton、JComboBox、JList、JMenu、JSlider 等。
- 不可编辑信息的显示组件：向用户显示不可编辑信息的组件，如 JLabel、JProgressBar 和 JToolTip 等。
- 可编辑信息的显示组件：向用户显示能被编辑的格式化信息的组件，如 JTable、JTextArea 和 JTextField 等。
- 特殊对话框组件：可以直接产生特殊对话框的组件，如 JColorChooser 和 JFileChooser 等。

下面将会依次详细介绍各种 Swing 组件的用法。

12.2.2 AWT 组件的 Swing 实现

从图 12.1 中可以看出，Swing 为除 Canvas 之外的所有 AWT 组件提供了相应的实现，Swing 组件比 AWT 组件的功能更加强大。相对于 AWT 组件，Swing 组件具有如下 4 个额外的功能。

- 可以为 Swing 组件设置提示信息。使用 setToolTipText()方法，为组件设置对用户有帮助的提示信息。
- 很多 Swing 组件如按钮、标签、菜单项等，除使用文字外，还可以使用图标修饰自己。为了允许在 Swing 组件中使用图标，Swing 为 Icon 接口提供了一个实现类：ImageIcon，该实现类代表一个图像图标。
- 支持插拔式的外观风格。每个 JComponent 对象都有一个相应的 ComponentUI 对象，为它完成所有的绘画、事件处理、决定尺寸大小等工作。ComponentUI 对象依赖当前使用的 PLAF，使用 UIManager.setLookAndFeel()方法可以改变图形界面的外观风格。
- 支持设置边框。Swing 组件可以设置一个或多个边框。Swing 中提供了各式各样的边框供用户选用，也能建立组合边框或自己设计边框。一种空白边框可以用于增大组件，同时协助布局管理器对容器中的组件进行合理的布局。

每个 Swing 组件都有一个对应的 UI 类，例如 JButton 组件就有一个对应的 ButtonUI 类来作为 UI 代理。每个 Swing 组件的 UI 代理的类名总是将该 Swing 组件类名的 J 去掉，然后在后面添加 UI 后缀。

UI 代理类通常是一个抽象基类，不同的 PLAF 会有不同的 UI 代理实现类。Swing 类库中包含了几套 UI 代理，每套 UI 代理都几乎包含了所有 Swing 组件的 ComponentUI 实现，每套这样的实现都被称为一种 PLAF 实现。以 JButton 为例，其 UI 代理的继承层次如图 12.2 所示。

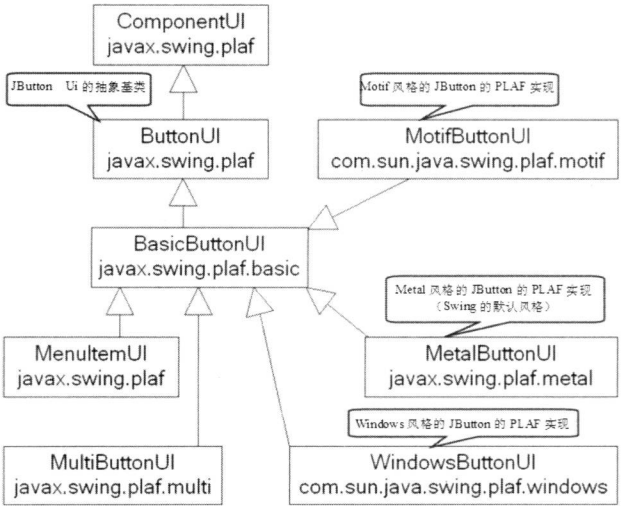


图 12.2　JButton UI 代理的继承层次

如果需要改变程序的外观风格，则可以使用如下代码。

```
try
{
    //设置使用 Windows 风格
    UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    //通过更新 f 容器以及 f 容器里所有组件的 UI
    SwingUtilities.updateComponentTreeUI(f);
}
catch(Exception e)
{
    e.printStackTrace();
}
```

下面程序示范了使用 Swing 组件来创建窗口应用，该窗口里包含了菜单、右键菜单以及基本 AWT 组件的 Swing 实现。

程序清单：codes\12\12.2\SwingComponent.java

```
public class SwingComponent
{
    JFrame f = new JFrame("测试");
    // 定义一个按钮，并为之指定图标
    Icon okIcon = new ImageIcon("ico/ok.png");
    JButton ok = new JButton("确认" , okIcon);
    // 定义一个单选按钮，初始处于选中状态
    JRadioButton male = new JRadioButton("男" , true);
    // 定义一个单选按钮，初始处于没有选中状态
    JRadioButton female = new JRadioButton("女" , false);
    // 定义一个 ButtonGroup，用于将上面两个 JRadioButton 组合在一起
    ButtonGroup bg = new ButtonGroup();
    // 定义一个复选框，初始处于没有选中状态。
    JCheckBox married = new JCheckBox("是否已婚？" , false);
    String[] colors = new String[]{"红色" , "绿色" , "蓝色"};
    // 定义一个下拉选择框
    JComboBox<String> colorChooser = new JComboBox<>(colors);
    // 定义一个列表选择框
    JList<String> colorList = new JList<>(colors);
    // 定义一个 8 行、20 列的多行文本域
    JTextArea ta = new JTextArea(8, 20);
    // 定义一个 40 列的单行文本域
    JTextField name = new JTextField(40);
```

```
JMenuBar mb = new JMenuBar();
JMenu file = new JMenu("文件");
JMenu edit = new JMenu("编辑");
// 创建“新建”菜单项，并为之指定图标
Icon newIcon = new ImageIcon("ico/new.png");
JMenuItem newItem = new JMenuItem("新建" , newIcon);
// 创建“保存”菜单项，并为之指定图标
Icon saveIcon = new ImageIcon("ico/save.png");
JMenuItem saveItem = new JMenuItem("保存" , saveIcon);
// 创建“退出”菜单项，并为之指定图标
Icon exitIcon = new ImageIcon("ico/exit.png");
JMenuItem exitItem = new JMenuItem("退出" , exitIcon);
JCheckBoxMenuItem autoWrap = new JCheckBoxMenuItem("自动换行");
// 创建“复制”菜单项，并为之指定图标
JMenuItem copyItem = new JMenuItem("复制"
    , new ImageIcon("ico/copy.png"));
// 创建“粘贴”菜单项，并为之指定图标
JMenuItem pasteItem = new JMenuItem("粘贴"
    , new ImageIcon("ico/paste.png"));
JMenu format = new JMenu("格式");
JMenuItem commentItem = new JMenuItem("注释");
JMenuItem cancelItem = new JMenuItem("取消注释");
// 定义一个右键菜单用于设置程序风格
JPopupMenu pop = new JPopupMenu();
// 用于组合 3 个风格菜单项的 ButtonGroup
ButtonGroup flavorGroup = new ButtonGroup();
// 创建 5 个单选按钮，用于设定程序的外观风格
JRadioButtonMenuItem metalItem = new JRadioButtonMenuItem("Metal 风格" , true);
JRadioButtonMenuItem nimbusItem = new JRadioButtonMenuItem("Nimbus 风格");
JRadioButtonMenuItem windowsItem = new JRadioButtonMenuItem("Windows 风格");
JRadioButtonMenuItem classicItem = new JRadioButtonMenuItem("Windows 经典风格");
JRadioButtonMenuItem motifItem = new JRadioButtonMenuItem("Motif 风格");
// -----------------用于执行界面初始化的 init 方法---------------------
public void init()
{
    // 创建一个装载了文本框、按钮的 JPanel
    JPanel bottom = new JPanel();
    bottom.add(name);
    bottom.add(ok);
    f.add(bottom , BorderLayout.SOUTH);
    // 创建一个装载了下拉选择框、三个 JCheckBox 的 JPanel
    JPanel checkPanel = new JPanel();
    checkPanel.add(colorChooser);
    bg.add(male);
    bg.add(female);
    checkPanel.add(male);
    checkPanel.add(female);
    checkPanel.add(married);
    // 创建一个垂直排列组件的 Box，盛装多行文本域 JPanel
    Box topLeft = Box.createVerticalBox();
    // 使用 JScrollPane 作为普通组件的 JViewPort
    JScrollPane taJsp = new JScrollPane(ta);     // ⑤
    topLeft.add(taJsp);
    topLeft.add(checkPanel);
    // 创建一个水平排列组件的 Box，盛装 topLeft、colorList
    Box top = Box.createHorizontalBox();
    top.add(topLeft);
    top.add(colorList);
    // 将 top Box 容器添加到窗口的中间
    f.add(top);
    // -----------下面开始组合菜单，并为菜单添加监听器----------
    // 为 newItem 设置快捷键，设置快捷键时要使用大写字母
    newItem.setAccelerator(KeyStroke.getKeyStroke('N'
        , InputEvent.CTRL_MASK));    // ①
    newItem.addActionListener(e -> ta.append("用户单击了“新建”菜单\n"));
    // 为 file 菜单添加菜单项
    file.add(newItem);
    file.add(saveItem);
    file.add(exitItem);
    // 为 edit 菜单添加菜单项
```

```
        edit.add(autoWrap);
        // 使用 addSeparator 方法添加菜单分隔线
        edit.addSeparator();
        edit.add(copyItem);
        edit.add(pasteItem);
        // 为 commentItem 组件添加提示信息
        commentItem.setToolTipText("将程序代码注释起来！");
        // 为 format 菜单添加菜单项
        format.add(commentItem);
        format.add(cancelItem);
        // 使用添加 new JMenuItem("-")的方式不能添加菜单分隔符
        edit.add(new JMenuItem("-"));
        // 将 format 菜单组合到 edit 菜单中，从而形成二级菜单
        edit.add(format);
        // 将 file、edit 菜单添加到 mb 菜单条中
        mb.add(file);
        mb.add(edit);
        // 为 f 窗口设置菜单条
        f.setJMenuBar(mb);
        // ----------下面开始组合右键菜单，并安装右键菜单----------
        flavorGroup.add(metalItem);
        flavorGroup.add(nimbusItem);
        flavorGroup.add(windowsItem);
        flavorGroup.add(classicItem);
        flavorGroup.add(motifItem);
        pop.add(metalItem);
        pop.add(nimbusItem);
        pop.add(windowsItem);
        pop.add(classicItem);
        pop.add(motifItem);
        // 为 5 个风格菜单创建事件监听器
        ActionListener flavorListener = e -> {
            try
            {
                switch(e.getActionCommand())
                {
                    case "Metal 风格":
                        changeFlavor(1);
                        break;
                    case "Nimbus 风格":
                        changeFlavor(2);
                        break;
                    case "Windows 风格":
                        changeFlavor(3);
                        break;
                    case "Windows 经典风格":
                        changeFlavor(4);
                        break;
                    case "Motif 风格":
                        changeFlavor(5);
                        break;
                }
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
        };
        // 为 5 个风格菜单项添加事件监听器
        metalItem.addActionListener(flavorListener);
        nimbusItem.addActionListener(flavorListener);
        windowsItem.addActionListener(flavorListener);
        classicItem.addActionListener(flavorListener);
        motifItem.addActionListener(flavorListener);
        // 调用该方法即可设置右键菜单，无须使用事件机制
        ta.setComponentPopupMenu(pop);     // ④
        // 设置关闭窗口时，退出程序
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.pack();
        f.setVisible(true);
```

```
        }
        // 定义一个方法，用于改变界面风格
        private void changeFlavor(int flavor)throws Exception
        {
            switch (flavor)
            {
                // 设置 Metal 风格
                case 1:
                    UIManager.setLookAndFeel(
                    "javax.swing.plaf.metal.MetalLookAndFeel");
                    break;
                // 设置 Nimbus 风格
                case 2:
                    UIManager.setLookAndFeel(
                    "javax.swing.plaf.nimbus.NimbusLookAndFeel");
                    break;
                // 设置 Windows 风格
                case 3:
                    UIManager.setLookAndFeel(
                    "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
                    break;
                // 设置 Windows 经典风格
                case 4:
                    UIManager.setLookAndFeel(
                    "com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel");
                    break;
                // 设置 Motif 风格
                case 5:
                    UIManager.setLookAndFeel(
                    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                    break;
            }
            // 更新 f 窗口内顶级容器以及内部所有组件的 UI
            SwingUtilities.updateComponentTreeUI(f.getContentPane()); // ②
            // 更新 mb 菜单条以及内部所有组件的 UI
            SwingUtilities.updateComponentTreeUI(mb);
            // 更新 pop 右键菜单以及内部所有组件的 UI
            SwingUtilities.updateComponentTreeUI(pop);
        }
        public static void main(String[] args)
        {
            // 设置 Swing 窗口使用 Java 风格
            // JFrame.setDefaultLookAndFeelDecorated(true);   // ③
            new SwingComponent().init();
        }
    }
```

上面程序在创建按钮、菜单项时传入了一个 ImageIcon 对象，通过这种方式就可以创建带图标的按钮、菜单项。程序的 init 方法中的粗体字代码用于为 comment 菜单项添加提示信息。运行上面程序，并通过右键菜单选择“Nimbus LAF”，可以看到如图 12.3 所示的窗口。

从图 12.3 中可以看出，Swing 菜单不允许使用 add(new JMenuItem("-"))的方式来添加菜单分隔符，只能使用 addSeparator()方法来添加菜单分隔符。

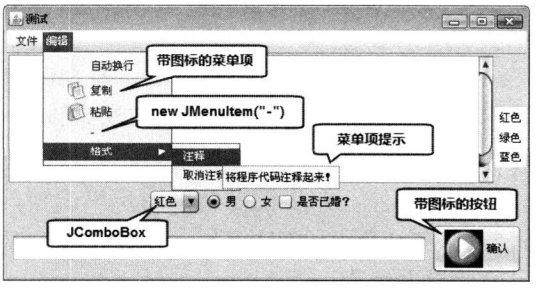


图 12.3 Nimbus 风格的 Swing 图形界面

提示：

Swing 专门为菜单项、工具按钮之间的分隔符提供了一个 JSeparator 类，通常使用 JMenu 或者 JPopupMenu 的 addSeparator()方法来创建并添加 JSeparator 对象，而不是直接使用 JSeparator。实际上，JSeparator 可以用在任何需要使用分隔符的地方。

上面程序为 newItem 菜单项增加了快捷键，为 Swing 菜单项指定快捷键与为 AWT 菜单项指定快捷键的方式有所不同——创建 AWT 菜单对象时可以直接传入 KeyShortcut 对象为其指定快捷键；但为

Swing 菜单项指定快捷键时必须通过 setAccelerator(KeyStroke ks)方法来设置（如①处程序所示），其中 KeyStroke 代表一次击键动作，可以直接通过按键对应字母来指定该击键动作。

提示：

为菜单项指定快捷键时应该使用大写字母来代表按键，例如 KeyStroke.getKeyStroke('N' , InputEvent.CTRL_MASK)代表 “Ctrl+N”，但 KeyStroke.getKeyStroke('n', InputEvent.CTRL_MASK)则不代表 “Ctrl+N”。

除此之外，上面程序中的大段粗体字代码所定义的 changeFlavor()方法用于改变程序外观风格，当用户单击多行文本域里的右键菜单时将会触发该方法，该方法设置 Swing 组件的外观风格后，再次调用 SwingUtilities 类的 updateComponentTreeUI()方法来更新指定容器，以及该容器内所有组件的 UI。注意此处更新的是 JFrame 对象 getContentPane()方法的返回值，而不是直接更新 JFrame 对象本身（如②处程序所示）。这是因为如果直接更新 JFrame 本身，将会导致 JFrame 也被更新，JFrame 是一个特殊的容器，JFrame 依然部分依赖于本地平台的图形组件。尤其是当取消③处代码的注释后，JFrame 将会使用 Java 风格的标题栏、边框，如果强制 JFrame 更新成 Windows 或 Motif 风格，则会导致该窗口失去标题栏和边框。如果通过右键菜单选择程序使用 Motif 风格，将看到如图 12.4 所示的窗口。

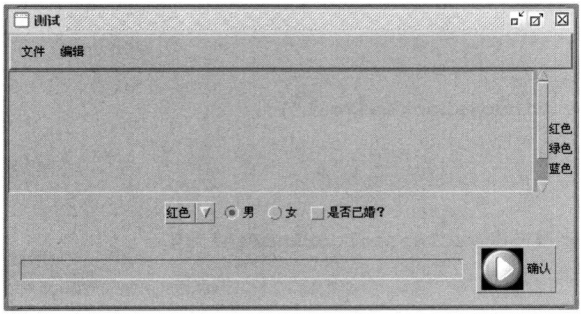


图 12.4　使用 Java 风格窗口标题、边框、Motif 显示风格的窗口

提示：

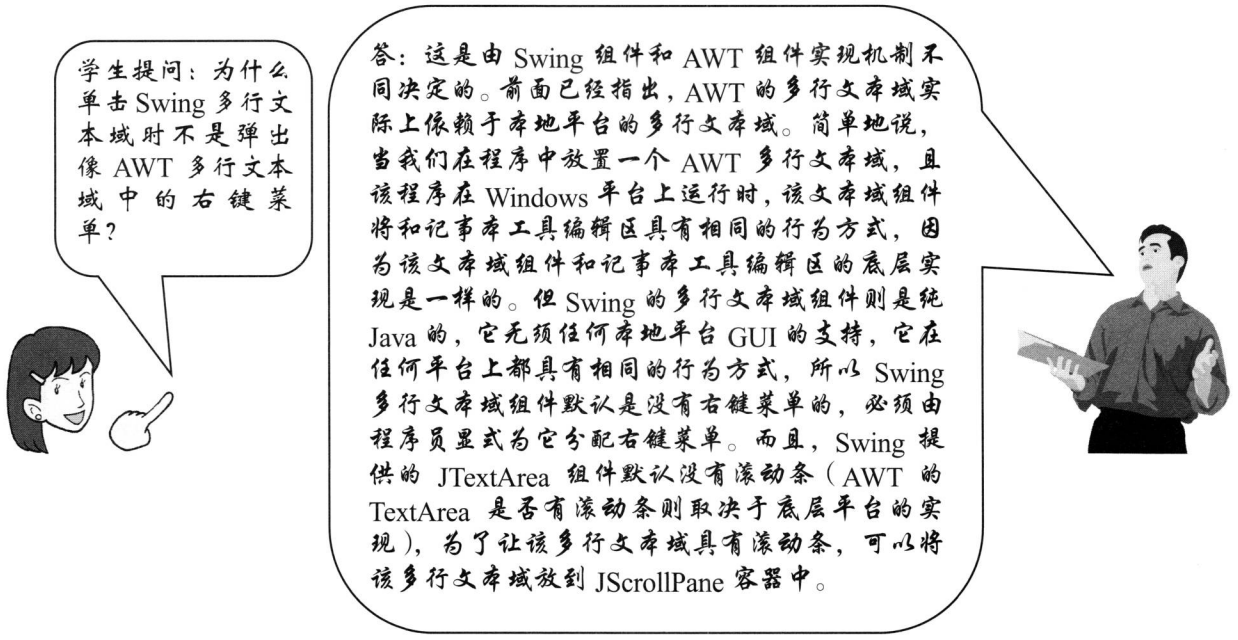
JFrame 提供了一个 getContentPane()方法，这个方法用于返回该 JFrame 的顶级容器（即 JRootPane 对象），这个顶级容器会包含 JFrame 所显示的所有非菜单组件。可以这样理解：所有看似放在 JFrame 中的 Swing 组件，除菜单之外，其实都是放在 JFrame 对应的顶级容器中的，而 JFrame 容器里提供了 getContentPane()方法返回的顶级容器。在 Java 5 以前，Java 甚至不允许直接向 JFrame 中添加组件，必须先调用 JFrame 的 getContentPane()方法获得该窗口的顶级容器，然后将所有组件添加到该顶级容器中。从 Java 5 以后，Java 改写了 JFrame 的 add()和 setLayout()等方法，当程序调用 JFrame 的 add()和 setLayout()等方法时，实际上是对 JFrame 的顶级容器进行操作。

从程序中④处代码可以看出，为 Swing 组件添加右键菜单无须像 AWT 中那样烦琐，只需要简单地调用 setComponentPopupMenu()方法来设置右键菜单即可，无须编写事件监听器。由此可见，使用 Swing 组件编写图形界面程序更加简单。

除此之外，如果程序希望用户单击窗口右上角的 “×” 按钮时，程序退出，也无须使用事件机制，只要调用 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)方法即可，Swing 提供的这种方式也是为了简化界面编程。

JScrollPane 组件是一个特殊的组件，它不同于 JFrame、JPanel 等普通容器，它甚至不能指定自己的布局管理器，它主要用于为其他的 Swing 组件提供滚动条支持，JScrollPane 通常由普通的 Swing 组件，可选的垂直、水平滚动条以及可选的行、列标题组成。

简而言之，如果希望让 JTextArea、JTable 等组件能有滚动条支持，只要将该组件放入 JScrollPane 中，再将该 JScrollPane 容器添加到窗口中即可。关于 JScrollPane 的详细说明，读者可以参考 JScrollPane 的 API 文档。



提示：

JScrollPane对于JTable组件尤其重要，通常需要把JTable放在JScrollPane容器中才可以显示出JTable组件的标题栏。

12.2.3 为组件设置边框

可以调用JComponent提供的setBorder(Border b)方法为Swing组件设置边框，其中Border是Swing提供的一个接口，用于代表组件的边框。该接口有数量众多的实现类，如LineBorder、MatteBorder、BevelBorder等，这些Border实现类都提供了相应的构造器用于创建Border对象，一旦获取了Border对象之后，就可以调用JComponent的setBorder(Border b)方法为指定组件设置边框。

TitledBorder和CompoundBorder比较独特，其中TitledBorder的作用并不是为其他组件添加边框，而是为其他边框设置标题，当创建TitledBorder对象时，需要传入一个已经存在的Border对象，新创建的TitledBorder对象会为原有的Border对象添加标题；而CompoundBorder用于组合两个边框，因此创建CompoundBorder对象时需要传入两个Border对象，一个用作组件的内边框，一个用作组件的外边框。

除此之外，Swing还提供了一个BorderFactory静态工厂类，该类提供了大量的静态工厂方法用于返回Border实例，这些静态方法的参数与各Border实现类的构造器参数基本一致。

提示：

Border不仅提供了上面所提到的一些Border实现类，还提供了MetalBorders.oolBarBorder、MetalBorders.TextFieldBorder等Border实现类，这些实现类用作Swing组件的默认边框，程序中通常无须使用这些系统边框。

为Swing组件添加边框可按如下步骤进行。

1. 使用BorderFactory或者XxxBorder创建XxxBorder实例。
2. 调用Swing组件的setBorder(Border b)方法为该组件设置边框。

图12.5显示了系统可用边框之间的继承层次。

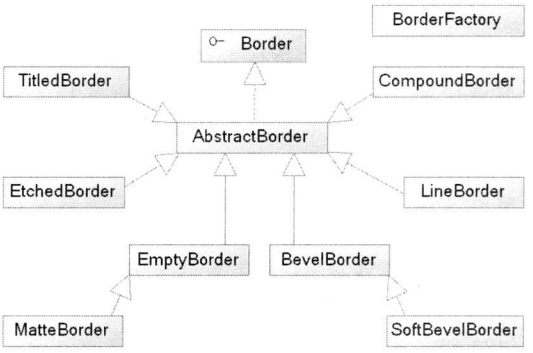


图 12.5 系统可用边框之间的继承层次

下面的例子程序示范了为 Panel 容器分别添加如图 12.5 所示的几种边框。

程序清单：codes\12\12.2\BorderTest.java

```
public class BorderTest
{
    private JFrame jf = new JFrame("测试边框");
    public void init()
    {
        jf.setLayout(new GridLayout(2, 4));
        // 使用静态工厂方法创建 BevelBorder
        Border bb = BorderFactory.createBevelBorder(
            BevelBorder.RAISED , Color.RED, Color.GREEN
            , Color.BLUE, Color.GRAY);
        jf.add(getPanelWithBorder(bb , "BevelBorder"));
        // 使用静态工厂方法创建 LineBorder
        Border lb = BorderFactory.createLineBorder(Color.ORANGE, 10);
        jf.add(getPanelWithBorder(lb , "LineBorder"));
        // 使用静态工厂方法创建 EmptyBorder，EmptyBorder 就是在组件四周留空
        Border eb = BorderFactory.createEmptyBorder(20, 5, 10, 30);
        jf.add(getPanelWithBorder(eb , "EmptyBorder"));
        // 使用静态工厂方法创建 EtchedBorder
        Border etb = BorderFactory.createEtchedBorder(EtchedBorder.RAISED,
        Color.RED, Color.GREEN);
        jf.add(getPanelWithBorder(etb , "EtchedBorder"));
        // 直接创建 TitledBorder，TitledBorder 就是为原有的边框增加标题
        TitledBorder tb = new TitledBorder(lb , "测试标题"
            , TitledBorder.LEFT , TitledBorder.BOTTOM
            , new Font("StSong" , Font.BOLD , 18), Color.BLUE);
        jf.add(getPanelWithBorder(tb , "TitledBorder"));
        // 直接创建 MatteBorder，MatteBorder 是 EmptyBorder 的子类，
        // 它可以指定留空区域的颜色或背景，此处是指定颜色
        MatteBorder mb = new MatteBorder(20, 5, 10, 30, Color.GREEN);
        jf.add(getPanelWithBorder(mb , "MatteBorder"));
        // 直接创建 CompoundBorder，CompoundBorder 将两个边框组合成新边框
        CompoundBorder cb = new CompoundBorder(new LineBorder(
            Color.RED, 8) , tb);
        jf.add(getPanelWithBorder(cb , "CompoundBorder"));
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new BorderTest().init();
    }
    public JPanel getPanelWithBorder(Border b , String BorderName)
    {
        JPanel p = new JPanel();
        p.add(new JLabel(BorderName));
        // 为 Panel 组件设置边框
        p.setBorder(b);
        return p;
    }
}
```

运行上面程序，会看到如图 12.6 所示的效果。

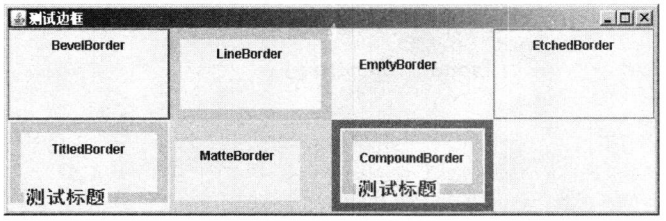


图 12.6 为 Swing 组件设置边框

12.2.4 Swing 组件的双缓冲和键盘驱动

除此之外，Swing 组件还有如下两个功能。

- 所有的 Swing 组件默认启用双缓冲绘图技术。
- 所有的 Swing 组件都提供了简单的键盘驱动。

Swing 组件默认启用双缓冲绘图技术，使用双缓冲技术能改进频繁重绘 GUI 组件的显示效果（避免闪烁现象）。JComponent 组件默认启用双缓冲，无须自己实现双缓冲。如果想关闭双缓冲，可以在组件上调用 setDoubleBuffered(false)方法。前一章介绍五子棋游戏时已经提到 Swing 组件的双缓冲技术，而且可以使用 JPanel 代替前一章所有示例程序中的 Canvas 画布组件，从而可以解决运行那些示例程序时的“闪烁”现象。

JComponent 类提供了 getInputMap()和 getActionMap()两个方法，其中 getInputMap()返回一个 InputMap 对象，该对象用于将 KeyStroke 对象（代表键盘或其他类似输入设备的一次输入事件）和名字关联；getActionMap()返回一个 ActionMap 对象，该对象用于将指定名字和 Action（Action 接口是 ActionListener 接口的子接口，可作为一个事件监听器使用）关联，从而可以允许用户通过键盘操作来替代鼠标驱动 GUI 上的 Swing 组件，相当于为 GUI 组件提供快捷键。典型用法如下：

```
// 把一次键盘事件和一个 aCommand 对象关联
component.getInputMap().put(aKeyStroke, aCommand);
// 将 aCommand 对象和 anAction 事件响应关联
component.getActionMap().put(aCommmand, anAction);
```

下面程序实现这样一个功能：用户在单行文本框内输入内容，当输入完成后，单击后面的“发送”按钮即可将文本框的内容添加到一个多行文本域中；或者输入完成后在文本框内按“Ctrl+Enter”键也可以将文本框的内容添加到一个多行文本域中。

程序清单：codes\12\12.2\BindKeyTest.java

```
public class BindKeyTest
{
    JFrame jf = new JFrame("测试键盘绑定");
    JTextArea jta = new JTextArea(5, 30);
    JButton jb = new JButton("发送");
    JTextField jtf = new JTextField(15);
    public void init()
    {
        jf.add(jta);
        JPanel jp = new JPanel();
        jp.add(jtf);
        jp.add(jb);
        jf.add(jp , BorderLayout.SOUTH);
        // 发送消息的 Action，Action 是 ActionListener 的子接口
        Action sendMsg = new AbstractAction()
        {
            public void actionPerformed(ActionEvent e)
            {
                jta.append(jtf.getText() + "\n");
                jtf.setText("");
            }
        };
        // 添加事件监听器
        jb.addActionListener(sendMsg);
```

```
        // 将 Ctrl+Enter 键和"send"关联
        jtf.getInputMap().put(KeyStroke.getKeyStroke('\n'
            , java.awt.event.InputEvent.CTRL_MASK) , "send");
        // 将"send"和 sendMsg Action 关联
        jtf.getActionMap().put("send", sendMsg);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new BindKeyTest().init();
    }
}
```

上面程序中粗体字代码示范了如何利用键盘事件来驱动 Swing 组件，采用这种键盘事件机制，无须为 Swing 组件绑定键盘监听器，从而可以复用按钮单击事件的事件监听器，程序十分简洁。

12.2.5 使用 JToolBar 创建工具条

Swing 提供了 JToolBar 类来创建工具条，创建 JToolBar 对象时可以指定如下两个参数。

- name：该参数指定该工具条的名称。
- orientation：该参数指定该工具条的方向。

一旦创建了 JToolBar 对象之后，JToolBar 对象还有如下几个常用方法。

- JButton add(Action a)：通过 Action 对象为 JToolBar 添加对应的工具按钮。
- void addSeparator(Dimension size)：向工具条中添加指定大小的分隔符，Java 允许不指定 size 参数，则添加一个默认大小的分隔符。
- void setFloatable(boolean b)：设置该工具条是否可浮动，即该工具条是否可以拖动。
- void setMargin(Insets m)：设置工具条边框和工具按钮之间的页边距。
- void setOrientation(int o)：设置工具条的方向。
- void setRollover(boolean rollover)：设置此工具条的 rollover 状态。

上面的大多数方法都比较容易理解，比较难以理解的是 add(Action a)方法，系统如何为工具条添加 Action 对应的按钮呢？

Action 接口是 ActionListener 接口的子接口，它除包含 ActionListener 接口的 actionPerformed()方法之外，还包含 name 和 icon 两个属性，其中 name 用于指定按钮或菜单项中的文本，而 icon 则用于指定按钮的图标或菜单项中的图标。也就是说，Action 不仅可作为事件监听器使用，而且可被转换成按钮或菜单项。

值得指出的是，Action 本身并不是按钮，也不是菜单项，只是当把 Action 对象添加到某些容器（也可直接使用 Action 来创建按钮），如菜单和工具栏中时，这些容器会为该 Action 对象创建对应的组件（菜单项和按钮）。也就是说，这些容器需要负责完成如下事情。

- 创建一个适用于该容器的组件（例如，在工具栏中创建一个工具按钮）。
- 从 Action 对象中获得对应的属性来设置该组件（例如，通过 name 来设置文本，通过 icon 来设置图标）。
- 检查 Action 对象的初始状态，确定它是否处于激活状态，并根据该 Action 的状态来决定其对应所有组件的行为。只有处于激活状态的 Action 所对应的 Swing 组件才可以响应用户动作。
- 通过 Action 对象为对应组件注册事件监听器，系统将为该 Action 所创建的所有组件注册同一个事件监听器（事件处理器就是 Action 对象里的 actionPerformed()方法）。

例如，程序中有一个菜单项、一个工具按钮，还有一个普通按钮都需要完成某个“复制”动作，程序就可以将该“复制”动作定义成 Action，并为之指定 name 和 icon 属性，然后通过该 Action 来创建菜单项、工具按钮和普通按钮，就可以让这三个组件具有相同的功能。另一个“粘贴”按钮也大致相似，而且“粘贴”组件默认不可用，只有当“复制” 组件被触发后，且剪贴板中有内容时才可用。

程序清单：codes\12\12.2\JToolBarTest.java

```
public class JToolBarTest
{
    JFrame jf = new JFrame("测试工具条");
    JTextArea jta = new JTextArea(6, 35);
```

```
JToolBar jtb = new JToolBar();
JMenuBar jmb = new JMenuBar();
JMenu edit = new JMenu("编辑");
// 获取系统剪贴板
Clipboard clipboard = Toolkit.getDefaultToolkit()
    .getSystemClipboard();
// 创建"粘贴"Action，该Action用于创建菜单项、工具按钮和普通按钮
Action pasteAction = new AbstractAction("粘贴"
    , new ImageIcon("ico/paste.png"))
{
    public void actionPerformed(ActionEvent e)
    {
        // 如果剪贴板中包含stringFlavor内容
        if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
        {
            try
            {
                // 取出剪贴板中的stringFlavor内容
                String content = (String)clipboard.getData
                (DataFlavor.stringFlavor);
                // 将选中内容替换成剪贴板中的内容
                jta.replaceRange(content , jta.getSelectionStart()
                    , jta.getSelectionEnd());
            }
            catch (Exception ee)
            {
                ee.printStackTrace();
            }
        }
    }
};
// 创建"复制"Action
Action copyAction = new AbstractAction("复制"
    , new ImageIcon("ico/copy.png"))
{
    public void actionPerformed(ActionEvent e)
    {
        StringSelection contents = new StringSelection(
            jta.getSelectedText());
        // 将StringSelection对象放入剪贴板中
        clipboard.setContents(contents, null);
        // 如果剪贴板中包含stringFlavor内容
        if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor))
        {
            // 将pasteAction激活
            pasteAction.setEnabled(true);
        }
    }
};
public void init()
{
    // pasteAction默认处于不激活状态
    pasteAction.setEnabled(false);   // ①
    jf.add(new JScrollPane(jta));
    // 以Action创建按钮，并将该按钮添加到Panel中
    JButton copyBn = new JButton(copyAction);
    JButton pasteBn = new JButton(pasteAction);
    JPanel jp = new JPanel();
    jp.add(copyBn);
    jp.add(pasteBn);
    jf.add(jp , BorderLayout.SOUTH);
    // 向工具条中添加Action对象，该对象将会转换成工具按钮
    jtb.add(copyAction);
    jtb.addSeparator();
    jtb.add(pasteAction);
    // 向菜单中添加Action对象，该对象将会转换成菜单项
    edit.add(copyAction);
    edit.add(pasteAction);
    // 将edit菜单添加到菜单条中
    jmb.add(edit);
```

```
        jf.setJMenuBar(jmb);
        // 设置工具条和工具按钮之间的页边距。
        jtb.setMargin(new Insets(20 ,10 , 5 , 30));    // ②
        // 向窗口中添加工具条
        jf.add(jtb , BorderLayout.NORTH);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JToolBarTest().init();
    }
}
```

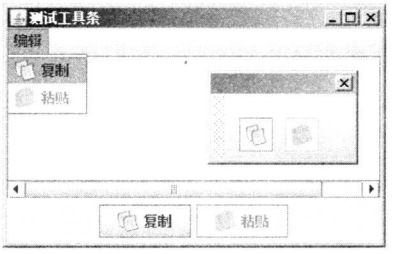


图 12.7　使用 Action 创建按钮、工具按钮和菜单项

上面程序中创建了 pasteAction、copyAction 两个 Action，然后根据这两个 Action 分别创建了按钮、工具按钮、菜单项组件（程序中粗体字代码部分），开始时 pasteAction 处于非激活状态，则该 Action 对应的按钮、工具按钮、菜单项都处于不可用状态。运行上面程序，会看到如图 12.7 所示的界面。

图 12.7 显示了工具条被拖动后的效果，这是因为工具条默认处于浮动状态。除此之外，程序中②号粗体字代码设置了工具条和工具按钮之间的页边距，所以可以看到工具条在工具按钮周围保留了一些空白区域。

12.2.6　使用 JFileChooser 和 Java 7 增强的 JColorChooser

JColorChooser 用于创建颜色选择器对话框，该类的用法非常简单，该类主要提供了如下两个静态方法。

- showDialog(Component component, String title, Color initialColor)：显示一个模式的颜色选择器对话框，该方法返回用户所选颜色。其中 component 指定该对话框的 parent 组件，而 title 指定该对话框的标题，大部分时候都使用该方法来让用户选择颜色。
- createDialog(Component c, String title, boolean modal, JColorChooser chooserPane, ActionListener okListener, ActionListener cancelListener)：该方法返回一个对话框，该对话框内包含指定的颜色选择器，该方法可以指定该对话框是模式的还是非模式的（通过 modal 参数指定），还可以指定该对话框内“确定”按钮的事件监听器（通过 okListener 参数指定）和“取消”按钮的事件监听器（通过 cancelListener 参数指定）。

Java 7 为 JColorChooser 增加了一个 HSV 标签页，允许用户通过 HSV 模式来选择颜色。

下面程序改写了前一章的 HandDraw 程序，改为使用 JPanel 作为绘图组件，而且使用 JColorChooser 来弹出颜色选择器对话框。

程序清单：codes\12\12.2\HandDraw.java

```
public class HandDraw
{
    // 画图区的宽度
    private final int AREA_WIDTH = 500;
    // 画图区的高度
    private final int AREA_HEIGHT = 400;
    // 下面的 preX、preY 保存了上一次鼠标拖动事件的鼠标坐标
    private int preX = -1;
    private int preY = -1;
    // 定义一个右键菜单用于设置画笔颜色
    JPopupMenu pop = new JPopupMenu();
    JMenuItem chooseColor = new JMenuItem("选择颜色");
    // 定义一个 BufferedImage 对象
    BufferedImage image = new BufferedImage(AREA_WIDTH
        , AREA_HEIGHT , BufferedImage.TYPE_INT_RGB);
```

```
// 获取 image 对象的 Graphics
Graphics g = image.getGraphics();
private JFrame f = new JFrame("简单手绘程序");
private DrawCanvas drawArea = new DrawCanvas();
// 用于保存画笔颜色
private Color foreColor = new Color(255, 0 ,0);
public void init()
{
    chooseColor.addActionListener(ae) -> {
        // 下面代码直接弹出一个模式的颜色选择对话框，并返回用户选择的颜色
        // foreColor = JColorChooser.showDialog(f
        //     , "选择画笔颜色" , foreColor);    // ①
        // 下面代码则弹出一个非模式的颜色选择对话框
        // 并可以分别为“确定”按钮、“取消”按钮指定事件监听器
        final JColorChooser colorPane = new JColorChooser(foreColor);
        JDialog jd = JColorChooser.createDialog(f , "选择画笔颜色"
            , false, colorPane, e->foreColor = colorPane.getColor(), null);
        jd.setVisible(true);
    });
    // 将菜单项组合成右键菜单
    pop.add(chooseColor);
    // 将右键菜单添加到 drawArea 对象中
    drawArea.setComponentPopupMenu(pop);
    // 将 image 对象的背景色填充成白色
    g.fillRect(0 , 0 ,AREA_WIDTH , AREA_HEIGHT);
    drawArea.setPreferredSize(new Dimension(AREA_WIDTH , AREA_HEIGHT));
    // 监听鼠标移动动作
    drawArea.addMouseMotionListener(new MouseMotionAdapter()
    {
        // 实现按下鼠标键并拖动的事件处理器
        public void mouseDragged(MouseEvent e)
        {
            // 如果 preX 和 preY 大于 0
            if (preX > 0 && preY > 0)
            {
                // 设置当前颜色
                g.setColor(foreColor);
                // 绘制从上一次鼠标拖动事件点到本次鼠标拖动事件点的线段
                g.drawLine(preX , preY , e.getX() , e.getY());
            }
            // 将当前鼠标事件点的 X、Y 坐标保存起来
            preX = e.getX();
            preY = e.getY();
            // 重绘 drawArea 对象
            drawArea.repaint();
        }
    });
    // 监听鼠标事件
    drawArea.addMouseListener(new MouseAdapter()
    {
        // 实现鼠标松开的事件处理器
        public void mouseReleased(MouseEvent e)
        {
            // 松开鼠标键时，把上一次鼠标拖动事件的 X、Y 坐标设为-1
            preX = -1;
            preY = -1;
        }
    });
    f.add(drawArea);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.pack();
    f.setVisible(true);
}
public static void main(String[] args)
{
    new HandDraw().init();
}
// 让画图区域继承 JPanel 类
class DrawCanvas extends JPanel
{
```

```
        // 重写 JPanel 的 paint 方法，实现绘画
        public void paint(Graphics g)
        {
            // 将 image 绘制到该组件上
            g.drawImage(image , 0 , 0 , null);
        }
    }
}
```

上面程序分别使用了两种方式来弹出颜色选择对话框，其中①号粗体字代码可弹出一个模式的颜色选择对话框，并直接返回用户选择的颜色。这种方式简单明了，编程简单。

如果程序有更多额外的需要，则使用程序下面的粗体字代码，弹出一个非模式的颜色选择对话框（允许程序设定），并为“确定”按钮指定了事件监听器，而“取消”按钮的事件监听器为 null（也可以为该按钮指定事件监听器）。Swing 的颜色选择对话框如图 12.8 所示。

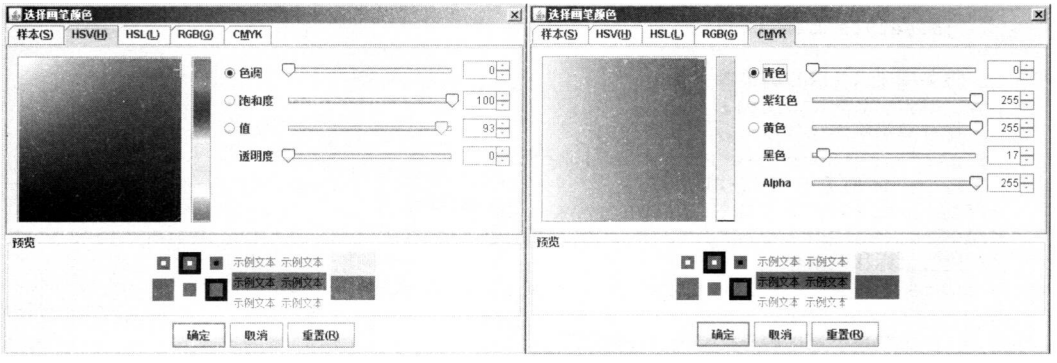


图 12.8　Swing 的颜色选择对话框

从图 12.8 中可以看出，Swing 的颜色选择对话框提供了 5 种方式来选择颜色，图中显示了 HSV 方式、CMYK 方式的颜色选择器，除此之外，该颜色选择器还可以使用 RGB、HSL 方式来选择颜色。

提示：学习过本书第 1 版的读者应该知道，在 Java 6 时，JcolorChooser 只提供了三种颜色选择方式，图 12.8 中看到的 HSV、CMYK 两种颜色选择方式都是新增的。

JFileChooser 的功能与 AWT 中的 FileDialog 基本相似，也是用于生成“打开文件”、“保存文件”对话框；与 FileDialog 不同的是，JFileChooser 无须依赖于本地平台的 GUI，它由 100%纯 Java 实现，在所有平台上具有完全相同的行为，并可以在所有平台上具有相同的外观风格。

为了调用 JFileChooser 来打开一个文件对话框，必须先创建该对话框的实例，JFileChooser 提供了多个构造器来创建 JFileChooser 对象，它的构造器总共包含两个参数。

- currentDirectory：指定所创建文件对话框的当前路径，该参数既可以是一个 String 类型的路径，也可以是一个 File 对象所代表的路径。
- FileSystemView：用于指定基于该文件系统外观来创建文件对话框，如果没有指定该参数，则默认以当前文件系统外观创建文件对话框。

JFileChooser 并不是 JDialog 的子类，所以不能使用 setVisible(true)方法来显示该文件对话框，而是调用 showXxxDialog()方法来显示文件对话框。

使用 JFileChooser 来建立文件对话框并允许用户选择文件的步骤如下。

① 采用构造器创建一个 JFileChooser 对象，该 JFileChooser 对象无须指定 parent 组件，这意味着可以在多个窗口中共用该 JFileChooser 对象。创建 JFileChooser 对象时可以指定初始化路径，如下代码所示。

```
// 以当前路径创建文件选择器
JFileChooser chooser = new JFileChooser(".");
```

② 调用 JFileChooser 的一系列可选的方法对 JFileChooser 执行初始化操作。JFileChooser 大致有如

下几个常用方法。

- setSelectedFile/setSelectedFiles：指定该文件选择器默认选择的文件（也可以默认选择多个文件）。

```
// 默认选择当前路径下的 123.jpg 文件
chooser.setSelectedFile(new File("123.jpg"));
```

- setMultiSelectionEnabled(boolean b)：在默认情况下，该文件选择器只能选择一个文件，通过调用该方法可以设置允许选择多个文件（设置参数值为 true 即可）。
- setFileSelectionMode(int mode)：在默认情况下，该文件选择器只能选择文件，通过调用该方法可以设置允许选择文件、路径、文件与路径，设置参数值为：JFileChooser. FILES_ONLY、JFileChooser.DIRECTORIES_ONLY、JFileChooser.FILES_AND_DIRECTORIES。

```
// 设置既可选择文件，也可选择路径
chooser.setFileSelectionMode (JFileChooser.FILES_AND_DIRECTORIES);
```

提示：

JFileChooser 还提供了一些改变对话框标题、改变按钮标签、改变按钮的提示文本等功能的方法，读者应该查阅 API 文档来了解它们。

3 如果让文件对话框实现文件过滤功能，则需要结合 FileFilter 类来进行文件过滤。JFileChooser 提供了两个方法来安装文件过滤器。

- addChoosableFileFilter(FileFilter filter)：添加文件过滤器。通过该方法允许该文件对话框有多个文件过滤器。

```
// 为文件对话框添加一个文件过滤器
chooser.addChoosableFileFilter(filter);
```

- setFileFilter(FileFilter filter)：设置文件过滤器。一旦调用了该方法，将导致该文件对话框只有一个文件过滤器。

4 如果需要改变文件对话框中文件的视图外观，则可以结合 FileView 类来改变对话框中文件的视图外观。

5 调用 showXxxDialog 方法可以打开文件对话框，通常如下三个方法可用。

- int showDialog(Component parent, String approveButtonText)：弹出文件对话框，该对话框的标题、“同意”按钮的文本（默认是“保存”或“取消”按钮）由 approveButtonText 来指定。
- int showOpenDialog(Component parent)：弹出文件对话框，该对话框具有默认标题，“同意”按钮的文本是“打开”。
- int showSaveDialog(Component parent)：弹出文件对话框，该对话框具有默认标题，“同意”按钮的文本是“保存”。

当用户单击“同意”、“取消”按钮，或者直接关闭文件对话框时才可以关闭该文件对话框，关闭该对话框时返回一个 int 类型的值，分别是：JFileChooser.APPROVE_OPTION、JFileChooser.CANCEL_OPTION 和 JFileChooser. ERROR_OPTION。如果希望获得用户选择的文件，则通常应该先判断对话框的返回值是否为 JFileChooser.APPROVE_OPTION，该选项表明用户单击了“打开”或者“保存”按钮。

6 JFileChooser 提供了如下两个方法来获取用户选择的文件或文件集。

- File getSelectedFile()：返回用户选择的文件。
- File[] getSelectedFiles()：返回用户选择的多个文件。

按上面的步骤，就可以正常地创建一个“打开文件”、“保存文件”对话框，整个过程非常简单。如果要使用 FileFilter 类来进行文件过滤，或者使用 FileView 类来改变文件的视图风格，则有一点麻烦。

先看使用 FileFilter 类来进行文件过滤。Java 在 java.io 包下提供了一个 FileFilter 接口，该接口主要用于作为 File 类的 listFiles(FileFilter)方法的参数，也是一个进行文件过滤的接口。但此处需要使用位于 javax.swing.filechooser 包下的 FileFilter 抽象类，该抽象类包含两个抽象方法。

- boolean accept(File f)：判断该过滤器是否接受给定的文件，只有被该过滤器接受的文件才可以在对应的文件对话框中显示出来。

➢ String getDescription()：返回该过滤器的描述性文本。

如果程序要使用 FileFilter 类进行文件过滤，则通常需要扩展该 FileFilter 类，并重写该类的两个抽象方法，重写 accept()方法时就可以指定自己的业务规则，指定该文件过滤器可以接受哪些文件。例如，如下代码：

```
public boolean accept(File f)
{
    // 如果该文件是路径，则接受该文件
    if (f.isDirectory()) return true;
    // 只接受以.gif 作为后缀的文件
    if (name.endsWith(".gif"))
    {
        return true;
    }
    return false
}
```

在默认情况下，JFileChooser 总会在文件对话框的“文件类型”下拉列表中增加“所有文件”选项，但可以调用 JFileChooser 的 setAcceptAllFileFilterUsed(false)来取消显示该选项。

FileView 类用于改变文件对话框中文件的视图风格，FileView 类也是一个抽象类，通常程序需要扩展该抽象类，并有选择性地重写它所包含的如下几个抽象方法。

➢ String getDescription(File f)：返回指定文件的描述。

➢ Icon getIcon(File f)：返回指定文件在 JFileChooser 对话框中的图标。

➢ String getName(File f)：返回指定文件的文件名。

➢ String getTypeDescription(File f)：返回指定文件所属文件类型的描述。

➢ Boolean isTraversable(File f)：当该文件是目录时，返回该目录是否是可遍历的。

与重写 FileFilter 抽象方法类似的是，重写这些方法实际上就是为文件选择器对话框指定自定义的外观风格。通常可以通过重写 getIcon()方法来改变文件对话框中的文件图标。

下面程序是一个简单的图片查看工具程序，该程序综合使用了上面所介绍的各知识点。

程序清单：codes\12\12.2\ImageViewer.java

```
public class ImageViewer
{
    // 定义图片预览组件的大小
    final int PREVIEW_SIZE = 100;
    JFrame jf = new JFrame("简单图片查看器");
    JMenuBar menuBar = new JMenuBar();
    // 该 label 用于显示图片
    JLabel label = new JLabel();
    // 以当前路径创建文件选择器
    JFileChooser chooser = new JFileChooser(".");
    JLabel accessory = new JLabel();
    // 定义文件过滤器
    ExtensionFileFilter filter = new ExtensionFileFilter();
    public void init()
    {
        // --------下面开始初始化 JFileChooser 的相关属性--------
        //  创建一个 FileFilter
        filter.addExtension("jpg");
        filter.addExtension("jpeg");
        filter.addExtension("gif");
        filter.addExtension("png");
        filter.setDescription("图片文件(*.jpg,*.jpeg,*.gif,*.png)");
        chooser.addChoosableFileFilter(filter);
        // 禁止“文件类型”下拉列表中显示“所有文件”选项
        chooser.setAcceptAllFileFilterUsed(false);   // ①
        // 为文件选择器指定自定义的 FileView 对象
        chooser.setFileView(new FileIconView(filter));
        // 为文件选择器指定一个预览图片的附件
        chooser.setAccessory(accessory);     // ②
        // 设置预览图片组件的大小和边框
```

```
        accessory.setPreferredSize(new Dimension(PREVIEW_SIZE, PREVIEW_SIZE));
        accessory.setBorder(BorderFactory.createEtchedBorder());
        // 用于检测被选择文件的改变事件
        chooser.addPropertyChangeListener(event -> {
            // JFileChooser 的被选文件已经发生了改变
            if (event.getPropertyName() ==
                JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
            {
                // 获取用户选择的新文件
                File f = (File) event.getNewValue();
                if (f == null)
                {
                    accessory.setIcon(null);
                    return;
                }
                // 将所选文件读入 ImageIcon 对象中
                ImageIcon icon = new ImageIcon(f.getPath());
                // 如果图像太大，则缩小它
                if(icon.getIconWidth() > PREVIEW_SIZE)
                {
                    icon = new ImageIcon(icon.getImage().getScaledInstance
                        (PREVIEW_SIZE, -1, Image.SCALE_DEFAULT));
                }
                // 改变 accessory Label 的图标
                accessory.setIcon(icon);
            }
        });
        // ------下面代码开始为该窗口安装菜单------
        JMenu menu = new JMenu("文件");
        menuBar.add(menu);
        JMenuItem openItem = new JMenuItem("打开");
        menu.add(openItem);
        // 单击 openItem 菜单项显示“打开文件”对话框
        openItem.addActionListener(event -> {
            // 设置文件对话框的当前路径
            // chooser.setCurrentDirectory(new File("."));
            // 显示文件对话框
            int result = chooser.showDialog(jf , "打开图片文件");
            // 如果用户选择了 APPROVE（同意）按钮，即打开，保存的等效按钮
            if(result == JFileChooser.APPROVE_OPTION)
            {
                String name = chooser.getSelectedFile().getPath();
                // 显示指定图片
                label.setIcon(new ImageIcon(name));
            }
        });
        JMenuItem exitItem = new JMenuItem("Exit");
        menu.add(exitItem);
        // 为退出菜单绑定事件监听器
        exitItem.addActionListener(event -> System.exit(0));
        jf.setJMenuBar(menuBar);
        // 添加用于显示图片的 JLabel 组件
        jf.add(new JScrollPane(label));
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ImageViewer().init();
    }
}
// 创建 FileFilter 的子类，用以实现文件过滤功能
class ExtensionFileFilter extends FileFilter
{
    private String description;
    private ArrayList<String> extensions = new ArrayList<>();
    // 自定义方法，用于添加文件扩展名
    public void addExtension(String extension)
    {
        if (!extension.startsWith("."))
```

```
        {
            extension = "." + extension;
            extensions.add(extension.toLowerCase());
        }
    }
    // 用于设置该文件过滤器的描述文本
    public void setDescription(String aDescription)
    {
        description = aDescription;
    }
    // 继承 FileFilter 类必须实现的抽象方法，返回该文件过滤器的描述文本
    public String getDescription()
    {
        return description;
    }
    // 继承 FileFilter 类必须实现的抽象方法，判断该文件过滤器是否接受该文件
    public boolean accept(File f)
    {
        // 如果该文件是路径，则接受该文件
        if (f.isDirectory()) return true;
        // 将文件名转为小写（全部转为小写后比较，用于忽略文件名大小写）
        String name = f.getName().toLowerCase();
        // 遍历所有可接受的扩展名，如果扩展名相同，该文件就可接受
        for (String extension : extensions)
        {
            if (name.endsWith(extension))
            {
                return true;
            }
        }
        return false;
    }
}
// 自定义一个 FileView 类，用于为指定类型的文件或文件夹设置图标
class FileIconView extends FileView
{
    private FileFilter filter;
    public FileIconView(FileFilter filter)
    {
        this.filter = filter;
    }
    // 重写该方法，为文件夹、文件设置图标
    public Icon getIcon(File f)
    {
        if (!f.isDirectory() && filter.accept(f))
        {
            return new ImageIcon("ico/pict.png");
        }
        else if (f.isDirectory())
        {
            // 获取所有根路径
            File[] fList = File.listRoots();
            for (File tmp : fList)
            {
                // 如果该路径是根路径
                if (tmp.equals(f))
                {
                    return  new ImageIcon("ico/dsk.png");
                }
            }
            return new ImageIcon("ico/folder.png");
        }
        // 使用默认图标
        else
        {
            return null;
        }
    }
}
```

上面程序中第二段粗体字代码用于为“打开”菜单项指定事件监听器，当用户单击该菜单时，程序打开文件对话框，并将用户打开的图片文件使用 Label 在当前窗口显示出来。

第三段粗体字代码用于重写 FileFilter 类的 accept()方法，该方法根据文件后缀来决定是否接受该文件，其要求是当该文件的后缀等于该文件过滤器的 extensions 集合的某一项元素时，则该文件是可接受的。程序的①处代码禁用了 JFileChooser 中“所有文件”选项，从而让用户只能看到图片文件。

第四段粗体字代码用于重写 FileView 类的 getIcon()方法，该方法决定 JFileChooser 对话框中文件、文件夹的图标——图标文件就返回 pict.png 图标，根文件夹就返回 dsk.png 图标，而普通文件夹则返回 folder.png 图标。

运行上面程序，单击“打开”菜单项，将看到如图 12.9 所示的对话框。

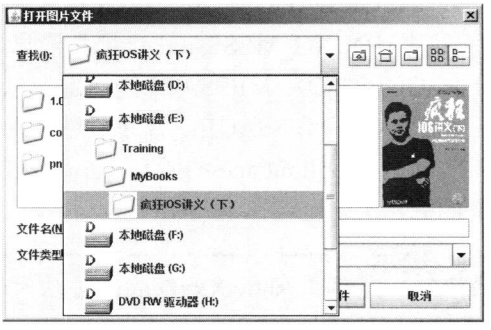


图 12.9　文件对话框

上面程序中的②处粗体字代码还用了 JFileChooser 类的 setAccessory(JComponent newAccessory)方法为该文件对话框指定附件，附件将会被显示在文件对话框的右上角，如图 12.9 所示。该附件可以是任何 Swing 组件（甚至可以使用容器），本程序中使用一个 JLabel 组件作为该附件组件，该 JLabel 用于显示用户所选图片文件的预览图片。该功能的实现很简单——当用户选择的图片发生改变时，以用户所选文件创建 ImageIcon，并将该 ImageIcon 设置成该 Label 的图标即可。

为了实现当用户选择图片发生改变时，附件组件的 icon 随之发生改变的功能，必须为 JFileChooser 添加事件监听器，该事件监听器负责监听该对话框中用户所选择文件的变化。JComponent 类中提供了一个 addPropertyChangeListener 方法，该方法可以为该 JFileChooser 添加一个属性监听器，用于监听用户选择文件的变化。程序中第一段粗体字代码实现了用户选择文件发生改变时的事件处理器。

12.2.7　使用 JOptionPane

通过 JOptionPane 可以非常方便地创建一些简单的对话框，Swing 已经为这些对话框添加了相应的组件，无须程序员手动添加组件。JOptionPane 提供了如下 4 个方法来创建对话框。

- showMessageDialog/showInternalMessageDialog：消息对话框，告知用户某事已发生，用户只能单击“确定”按钮，类似于 JavaScript 的 alert 函数。
- showConfirmDialog/showInternalConfirmDialog：确认对话框，向用户确认某个问题，用户可以选择 yes、no、cancel 等选项。类似于 JavaScript 的 comfirm 函数。该方法返回用户单击了哪个按钮。
- showInputDialog/showInternalInputDialog：输入对话框，提示要求输入某些信息，类似于 JavaScript 的 prompt 函数。该方法返回用户输入的字符串。
- showOptionDialog/showInternalOptionDialog：自定义选项对话框，允许使用自定义选项，可以取代 showConfirmDialog 所产生的对话框，只是用起来更复杂。

JOptionPane 产生的所有对话框都是模式的，在用户完成与对话框的交互之前，showXxxDialog 方法都将一直阻塞当前线程。

JOptionPane 所产生的对话框总是具有如图 12.10 所示的布局。

上面这些方法都提供了相应的 showInternalXxxDialog 版本，这种方法以 InternalFrame 的方式打开对话框。关于什么是 InternalFrame 方式，请参考下一节关于 JInternalFrame 的介绍。

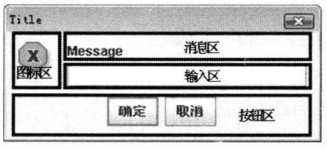


图 12.10　JoptionPane 产生的对话框的布局

下面就图 12.10 中所示的 4 个区域分别进行介绍。

（1）输入区

如果创建的对话框无须接收用户输入，则输入区不存在。

输入区组件可以是普通文本框组件，也可以是下拉列表框组件。

如果调用上面的 showInternalXxxDialog()方法时指定了一个数组类型的 selectionValues 参数，则输入区包含一个下拉列表框组件。

（2）图标区

左上角的图标会随创建的对话框所包含消息类型的不同而不同，JOptionPane 可以提供如下 5 种消息类型。

- ERROR_MESSAGE：错误消息，其图标是一个红色的 X 图标，如图 12.10 所示。
- INFORMATION_MESSAGE：普通消息，其默认图标是蓝色的感叹号。
- WARNING_MESSAGE：警告消息，其默认图标是黄色感叹号。
- QUESTION_MESSAGE：问题消息，其默认图标是绿色问号。
- PLAIN_MESSAGE：普通消息，没有默认图标。

实际上，JoptionPane 的所有 showXxxDialog()方法都可以提供一个可选的 icon 参数，用于指定该对话框的图标。

提示：

调用 showXxxDialog 方法时还可以指定一个可选的 title 参数，该参数指定所创建对话框的标题。

（3）消息区

不管是哪种对话框，其消息区总是存在的，消息区的内容通过 message 参数来指定，根据 message 参数的类型不同，消息区显示的内容也是不同的。该 message 参数可以是如下几种类型。

- String 类型：系统将该字符串对象包装成 JLabel 对象，然后显示在对话框中。
- Icon：该 Icon 被包装成 JLabel 后作为对话框的消息。
- Component：将该 Component 在对话框的消息区中显示出来。
- Object[]：对象数组被解释为在纵向排列的一系列 message 对象，每个 message 对象根据其实际类型又可以是字符串、图标、组件、对象数组等。
- 其他类型：系统调用该对象的 toString()方法返回一个字符串，并将该字符串对象包装成 JLabel 对象，然后显示在对话框中。

大部分时候对话框的消息区都是普通字符串，但使用 Component 作为消息区组件则更加灵活，因为该 Component 参数几乎可以是任何对象，从而可以让对话框的消息区包含任何内容。

提示：

如果用户希望消息区的普通字符串能换行，则可以使用“\n”字符来实现换行。

（4）按钮区

对话框底部的按钮区也是一定存在的，但所包含的按钮则会随对话框的类型、选项类型而改变。对于调用 showInputDialog()和 showMessageDialog()方法得到的对话框，底部总是包含“确定”和“取消”两个标准按钮。

对于 showConfirmDialog()所打开的确认对话框，则可以指定一个整数类型的 optionType 参数，该参数可以取如下几个值。

- DEFAULT_OPTION：按钮区只包含一个“确定”按钮。
- YES_NO_OPTION：按钮区包含“是”、“否”两个按钮。
- YES_NO_CANCEL_OPTION：按钮区包含“是”、“否”、“取消”三个按钮。
- OK_CANCEL_OPTION：按钮区包含“确定”、“取消”两个按钮。

如果使用 showOptionDialog 方法来创建选项对话框，则可以通过指定一个 Object[]类型的 options 参数来设置按钮区能使用的选项按钮。与前面的 message 参数类似的是，options 数组的数组元素可以是

如下几种类型。

- String 类型：使用该字符串来创建一个 JButton，并将其显示在按钮区。
- Icon：使用该 Icon 来创建一个 JButton，并将其显示在按钮区。
- Component：直接将该组件显示在按钮区。
- 其他类型：系统调用该对象的 toString()方法返回一个字符串，并使用该字符串来创建一个 JButton，并将其显示在按钮区。

当用户与对话框交互结束后，不同类型对话框的返回值如下。

- showMessageDialog：无返回值。
- showInputDialog：返回用户输入或选择的字符串。
- showConfirmDialog：返回一个整数代表用户选择的选项。
- showOptionDialog：返回一个整数代表用户选择的选项，如果用户选择第一项，则返回 0；如果选择第二项，则返回 1……依此类推。

对 showConfirmDialog 所产生的对话框，有如下几个返回值。

- YES_OPTION：用户单击了“是”按钮后返回。
- NO_OPTION：用户单击了“否”按钮后返回。
- CANCEL_OPTION：用户单击了“取消”按钮后返回。
- OK_OPTION：用户单击了“确定”按钮后返回。
- CLOSED_OPTION：用户单击了对话框右上角的“×”按钮后返回。

提示： 对于 showOptionDialog 方法所产生的对话框，也可能返回一个 CLOSED_OPTION 值，当用户单击了对话框右上角的“×”按钮后将返回该值。

下面程序允许使用 JOptionPane 来弹出各种对话框。

程序清单：codes\12\12.2\JOptionPaneTest.java

```
public class JOptionPaneTest
{
    JFrame jf = new JFrame("测试 JOptionPane");
    // 定义 6 个面板，分别用于定义对话框的几种选项
    private ButtonPanel messagePanel;
    private ButtonPanel messageTypePanel;
    private ButtonPanel msgPanel;
    private ButtonPanel confirmPanel;
    private ButtonPanel optionsPanel;
    private ButtonPanel inputPanel;
    private String messageString = "消息区内容";
    private Icon messageIcon = new ImageIcon("ico/heart.png");
    private Object messageObject = new Date();
    private Component messageComponent = new JButton("组件消息");
    private JButton msgBn = new JButton("消息对话框");
    private JButton confrimBn = new JButton("确认对话框");
    private JButton inputBn = new JButton("输入对话框");
    private JButton optionBn = new JButton("选项对话框");
    public void init()
    {
        JPanel top = new JPanel();
        top.setBorder(new TitledBorder(new EtchedBorder()
            , "对话框的通用选项" , TitledBorder.CENTER ,TitledBorder.TOP));
        top.setLayout(new GridLayout(1 , 2));
        // 消息类型 Panel，该 Panel 中的选项决定对话框的图标
        messageTypePanel = new ButtonPanel("选择消息的类型",
            new String[]{"ERROR_MESSAGE", "INFORMATION_MESSAGE"
                , "WARNING_MESSAGE", "QUESTION_MESSAGE",    "PLAIN_MESSAGE" });
        // 消息内容类型 Panel，该 Panel 中的选项决定对话框消息区的内容
        messagePanel = new ButtonPanel("选择消息内容的类型",
        new String[]{"字符串消息", "图标消息", "组件消息"
```

```
            , "普通对象消息" , "Object[]消息"});
        top.add(messageTypePanel);
        top.add(messagePanel);
        JPanel bottom = new JPanel();
        bottom.setBorder(new TitledBorder(new EtchedBorder()
            , "弹出不同的对话框" , TitledBorder.CENTER ,TitledBorder.TOP));
        bottom.setLayout(new GridLayout(1 , 4));
        // 创建用于弹出消息对话框的 Panel
        msgPanel = new ButtonPanel("消息对话框", null);
        msgBn.addActionListener(new ShowAction());
        msgPanel.add(msgBn);
        // 创建用于弹出确认对话框的 Panel
        confirmPanel = new ButtonPanel("确认对话框",
            new String[]{"DEFAULT_OPTION", "YES_NO_OPTION"
                , "YES_NO_CANCEL_OPTION","OK_CANCEL_OPTION"});
        confrimBn.addActionListener(new ShowAction());
        confirmPanel.add(confrimBn);
        // 创建用于弹出输入对话框的 Panel
        inputPanel = new ButtonPanel("输入对话框"
            , new String[]{"单行文本框","下拉列表选择框"});
        inputBn.addActionListener(new ShowAction());
        inputPanel.add(inputBn);
        // 创建用于弹出选项对话框的 Panel
        optionsPanel = new ButtonPanel("选项对话框"
            , new String[]{"字符串选项", "图标选项", "对象选项"});
        optionBn.addActionListener(new ShowAction());
        optionsPanel.add(optionBn);
        bottom.add(msgPanel);
        bottom.add(confirmPanel);
        bottom.add(inputPanel);
        bottom.add(optionsPanel);
        Box box = new Box(BoxLayout.Y_AXIS);
        box.add(top);
        box.add(bottom);
        jf.add(box);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    // 根据用户选择返回选项类型
    private int getOptionType()
    {
        switch(confirmPanel.getSelection())
        {
            case "DEFAULT_OPTION":
                return JOptionPane.DEFAULT_OPTION;
            case "YES_NO_OPTION":
                return JOptionPane.YES_NO_OPTION;
            case "YES_NO_CANCEL_OPTION":
                return JOptionPane.YES_NO_CANCEL_OPTION;
            default:
                return JOptionPane.OK_CANCEL_OPTION;
        }
    }
    // 根据用户选择返回消息
    private Object getMessage()
    {
        switch(messagePanel.getSelection())
        {
            case "字符串消息":
                return messageString;
            case "图标消息":
                return messageIcon;
            case "组件消息":
                return messageComponent;
            case "普通对象消息":
                return messageObject;
            default:
                return new Object[]{messageString , messageIcon
                    , messageObject , messageComponent};
```

```
    }
}
// 根据用户选择返回消息类型（决定图标区的图标）
private int getDialogType()
{
    switch(messageTypePanel.getSelection())
    {
        case "ERROR_MESSAGE":
            return JOptionPane.ERROR_MESSAGE;
        case "INFORMATION_MESSAGE":
            return JOptionPane.INFORMATION_MESSAGE;
        case "WARNING_MESSAGE":
            return JOptionPane.WARNING_MESSAGE;
        case "QUESTION_MESSAGE":
            return JOptionPane.QUESTION_MESSAGE;
        default:
            return JOptionPane.PLAIN_MESSAGE;
    }
}
private Object[] getOptions()
{
    switch(optionsPanel.getSelection())
    {
        case "字符串选项":
            return new String[]{"a" , "b" , "c" , "d"};
        case "图标选项":
            return new Icon[]{new ImageIcon("ico/1.gif")
                , new ImageIcon("ico/2.gif")
                , new ImageIcon("ico/3.gif")
                , new ImageIcon("ico/4.gif")};
        default:
            return new Object[]{new Date() ,new Date() , new Date()};
    }
}
// 为各按钮定义事件监听器
private class ShowAction implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        switch(event.getActionCommand())
        {
            case "确认对话框":
                JOptionPane.showConfirmDialog(jf , getMessage()
                    ,"确认对话框", getOptionType(), getDialogType());
                break;
            case "输入对话框":
                if (inputPanel.getSelection().equals("单行文本框"))
                {
                    JOptionPane.showInputDialog(jf, getMessage()
                        , "输入对话框", getDialogType());
                }
                else
                {
                    JOptionPane.showInputDialog(jf, getMessage()
                        , "输入对话框", getDialogType() , null
                        , new String[]{"轻量级 Java EE 企业应用实战"
                        , "疯狂 Java 讲义"}, "疯狂 Java 讲义");
                }
                break;
            case "消息对话框":
                JOptionPane.showMessageDialog(jf,getMessage()
                    ,"消息对话框", getDialogType());
                break;
            case "选项对话框":
                JOptionPane.showOptionDialog(jf , getMessage()
                    , "选项对话框", getOptionType() , getDialogType()
                    , null,getOptions(), "a");
                break;
        }
    }
```

```
    }
    public static void main(String[] args)
    {
        new JOptionPaneTest().init();
    }
}
// 定义一个 JPanel 类扩展类，该类的对象包含多个纵向排列的
// JRadioButton 控件，且 Panel 扩展类可以指定一个字符串作为 TitledBorder
class ButtonPanel extends JPanel
{
    private ButtonGroup group;
    public ButtonPanel(String title, String[] options)
    {
        setBorder(BorderFactory.createTitledBorder(BorderFactory
            .createEtchedBorder(), title));
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        group = new ButtonGroup();
        for (int i = 0; options!= null && i < options.length; i++)
        {
            JRadioButton b = new JRadioButton(options[i]);
            b.setActionCommand(options[i]);
            add(b);
            group.add(b);
            b.setSelected(i == 0);
        }
    }
    // 定义一个方法，用于返回用户选择的选项
    public String getSelection()
    {
        return group.getSelection().getActionCommand();
    }
}
```

运行上面程序，会看到如图 12.11 所示的窗口。

图 12.11 已经非常清楚地显示了 JOptionPane 所支持的 4 种对话框，以及所有对话框的通用选项、每个对话框的特定选项。如果用户选择“INFORMATION_MESSAGE”、“图标消息”，然后打开“下拉列表选择框”的输入对话框，将打开如图 12.12 所示的对话框。

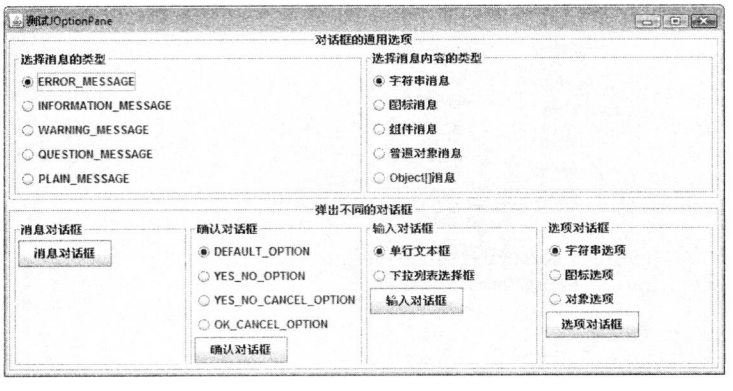


图 12.11 测试对话框的窗口

图 12.12 对话框实例

读者可以通过运行上面程序来查看 JOptionPane 所创建的各种对话框。

12.3 Swing 中的特殊容器

Swing 提供了一些具有特殊功能的容器，这些特殊容器可以用于创建一些更复杂的用户界面。下面将依次介绍这些特殊容器。

12.3.1 使用 JSplitPane

JSplitPane 用于创建一个分割面板，它可以将一个组件（通常是一个容器）分割成两个部分，并提

供一个分割条，用户可以拖动该分割条来调整两个部分的大小。图 12.13 显示了分割面板效果，图中所示的窗口先被分成左右两块，其中左边一块又被分为上下两块。

从图 12.13 中可以看出，分割面板的实质是一个特殊容器，该容器只能容纳两个组件，而且分割面板又分为上下分割、左右分割两种情形，所以创建分割面板的代码非常简单，如下代码所示。

```
new JSplitPane(方向, 左/上组件, 右/下组件)
```

除此之外，创建分割面板时可以指定一个 newContinuousLayout 参数，该参数指定该分割面板是否支持“连续布局”，如果分割面板支持连续布局，则用户拖动分割条时两边组件将会不断调整大小；如果不支持连续布局，则拖动分割条时两边组件不会调整大小，而是只看到一条虚拟的分割条在移动，如图 12.14 所示。

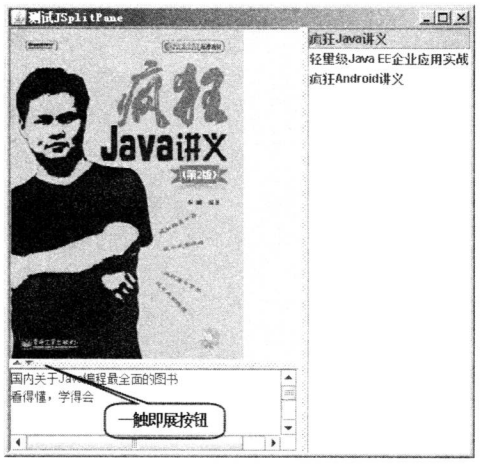


图 12.13　分割面板效果

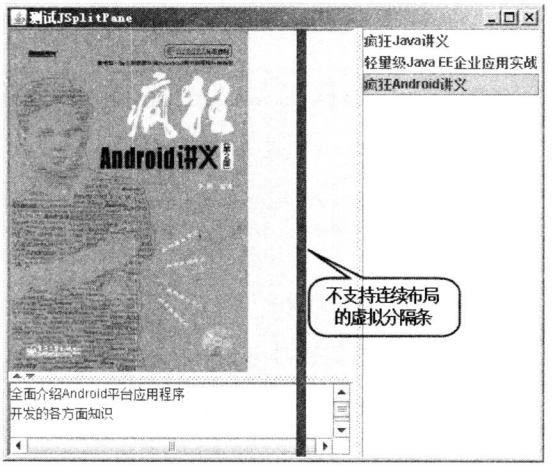


图 12.14　不支持连续布局的虚拟分割条

JSplitPane 默认关闭连续布局特性，因为使用连续布局需要不断重绘两边的组件，因此运行效率很低。如果需要打开指定 JSplitPane 面板的连续布局特性，则可以使用如下代码：

```
// 打开 JSplitPane 的连续布局特性
jsp.setContinuousLayout(true);
```

除此之外，正如图 12.13 中看到的，上下分割面板的分割条中还有两个三角箭头，这两个箭头被称为“一触即展”键，当用户单击某个三角箭头时，将看到箭头所指的组件慢慢缩小到没有，而另一个组件则扩大到占据整个面板。如果需要打开“一触即展”特性，使用如下代码即可：

```
// 打开“一触即展”特性
jsp.setOneTouchExpandable(true);
```

JSplitPane 分割面板还有如下几个可用方法来设置该面板的相关特性。

- setDividerLocation(double proportionalLocation)：设置分隔条的位置为 JSplitPane 的某个百分比。
- setDividerLocation(int location)：通过像素值设置分隔条的位置。
- setDividerSize(int newSize)：通过像素值设置分隔条的大小。
- setLeftComponent(Component comp)/setTopComponent(Component comp)：将指定组件放置到分割面板的左边或者上面。
- setRightComponent(Component comp)/setBottomComponent(Component comp)：将指定组件放置到分割面板的右边或者下面。

下面程序简单示范了 JSplitPane 的用法。

程序清单：codes\12\12.3\SplitPaneTest.java

```
public class SplitPaneTest
{
    Book[] books = new Book[]{
        new Book("疯狂 Java 讲义" , new ImageIcon("ico/java.png")
            , "国内关于 Java 编程最全面的图书\n 看得懂，学得会")
        , new Book("轻量级 Java EE 企业应用实战" , new ImageIcon("ico/ee.png")
            , "SSH 整合开发的经典图书，值得拥有")
```

```
        , new Book("疯狂 Android 讲义" , new ImageIcon("ico/android.png")
            , "全面介绍 Android 平台应用程序\n 开发的各方面知识")
    };
    JFrame jf = new JFrame("测试 JSplitPane");
    JList<Book> bookList = new JList<>(books);
    JLabel bookCover = new JLabel();
    JTextArea bookDesc = new JTextArea();
    public void init()
    {
        // 为三个组件设置最佳大小
        bookList.setPreferredSize(new Dimension(150, 300));
        bookCover.setPreferredSize(new Dimension(300, 150));
        bookDesc.setPreferredSize(new Dimension(300, 150));
        // 为下拉列表添加事件监听器
        bookList.addListSelectionListener(event {
            Book book = (Book)bookList.getSelectedValue();
            bookCover.setIcon(book.getIco());
            bookDesc.setText(book.getDesc());
        });
        // 创建一个垂直的分割面板，
        // 将 bookCover 放在上面，将 bookDesc 放在下面，支持连续布局
        JSplitPane left = new JSplitPane(JSplitPane.VERTICAL_SPLIT
            , true , bookCover, new JScrollPane(bookDesc));
        // 打开“一触即展”特性
        left.setOneTouchExpandable(true);
        // 下面代码设置分割条的大小
        // left.setDividerSize(50);
        // 设置该分割面板根据所包含组件的最佳大小来调整布局
        left.resetToPreferredSizes();
        // 创建一个水平的分割面板
        // 将 left 组件放在左边，将 bookList 组件放在右边
        JSplitPane content = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT
            , left, bookList);
        jf.add(content);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SplitPaneTest().init();
    }
}
```

上面代码中粗体字代码创建了两个 JSplitPane，其中一个支持连续布局，另一个不支持连续布局。运行上面程序，将可看到如图 12.13 所示的界面。

12.3.2 使用 JTabbedPane

JTabbedPane 可以很方便地在窗口上放置多个标签页，每个标签页相当于获得了一个与外部容器具有相同大小的组件摆放区域。通过这种方式，就可以在一个容器里放置更多的组件，例如右击桌面上的“我的电脑”图标，在弹出的快捷菜单里单击“属性”菜单项，就可以看到一个“系统属性”对话框，这个对话框里包含了 7 个标签页。

如果需要使用 JTabbedPane 在窗口上创建标签页，则可以按如下步骤进行。

① 创建一个 JTabbedPane 对象，JTabbedPane 提供了几个重载的构造器，这些构造器里一共包含如下两个参数。

- tabPlacement：该参数指定标签页标题的放置位置，例如前面介绍的“系统属性”对话框里标签页的标题放在窗口顶部。Swing 支持将标签页标题放在窗口的 4 个方位：TOP（顶部）、LEFT（左边）、BOTTOM（下部）和 RIGHT（右边）。
- tabLayoutPolicy：指定标签页标题的布局策略。当窗口不足以在同一行摆放所有的标签页标题时，Swing 有两种处理方式——将标签页标题换行（JTabbedPane.WRAP_TAB_LAYOUT）排列，或者使用滚动条来控制标签页标题的显示（SCROLL_TAB_LAYOUT）。

提示：

即使创建 JTabbedPane 时没有指定这两个参数，程序也可以在后面改变 JTabbedPane 的这两个属性。例如，通过 setTabLayoutPolicy()方法改变标签页标题的布局策略；使用 setTabPlacement()方法设置标签页标题的放置位置。

例如，下面代码创建一个 JTabbedPane 对象，该 JTabbedPane 的标签页标题位于窗口左侧，当窗口的一行不能摆放所有的标签页标题时，JTabbedPane 将采用换行方式来排列标签页标题。

```
JTabbedPane tabPane = new JTabbedPane(JTabbedPane.LEFT
    , JTabbedPane.WRAP_TAB_LAYOUT);
```

② 调用 JTabbedPane 对象的 addTab()、insertTab()、setComponentAt()、removeTabAt()方法来增加、插入、修改和删除标签页。其中 addTab()方法总是在最前面增加标签页，而 insertTab()、setComponentAt()、removeTabAt()方法都可以使用一个 index 参数，表示在指定位置插入标签页，修改指定位置的标签页，删除指定位置的标签页。

添加标签页时可以指定该标签页的标题（title）、图标（icon），以及该 Tab 页面的组件（component）及提示信息（tip），这 4 个参数都可以是 null；如果某个参数是 null，则对应的内容为空。

不管使用增加、插入、修改哪种操作来改变 JTabbedPane 中的标签页，都是传入一个 Component 组件作为标签页。也就是说，如果希望在某个标签页内放置更多的组件，则必须先将这些组件放置到一个容器（例如 JPanel）里，然后将该容器设置为 JTabbedPane 指定位置的组件。

注意：

不要使用 JTabbedPane 的 add()方法来添加组件，该方法是 JTabbedPane 重写 Containner 容器中的 add()方法，如果使用该 add()方法来添加 Tab 页面，每次添加的标签页会直接覆盖原有的标签页。

③ 如果需要让某个标签页显示出来，则可以通过调用 JTabbedPane 的 setSelectedIndex()方法来实现。例如如下代码：

```
// 设置第三个 Tab 页面处于显示状态
tabPane.setSelectedIndex(2);
// 设置最后一个 Tab 页面处于显示状态
tabPane.setSelectedIndex(tabPanel.getTabCount() - 1);
```

④ 正如上面代码见到的，程序还可通过 JTabbedPane 提供的一系列方法来操作 JTabbedPane 的相关属性。例如，有如下几个常用方法。

- setDisabledIconAt(int index, Icon disabledIcon)：将指定位置的禁用图标设置为 icon，该图标也可以是 null，表示不使用禁用图标。
- setEnabledAt(int index, boolean enabled)：设置指定位置的标签页是否启用。
- setForegroundAt(int index, Color foreground)：设置指定位置标签页的前景色为 foreground。该颜色可以是 null，这时将使用该 JTabbedPane 的前景色作为此标签页的前景色。
- setIconAt(int index, Icon icon)：设置指定位置标签页的图标。
- setTitleAt(int index, String title)：设置指定位置标签页的标题为 title，该 title 可以是 null，这表明设置该标签页的标题为空。
- setToolTipTextAt(int index, String toolTipText)：设置指定位置标签页的提示文本。

实际上，Swing 也为这些 setter 方法提供了对应的 getter 方法，用于返回这些属性。

⑤ 如果程序需要监听用户单击标签页的事件，例如，当用户单击某个标签页时才载入该标签页的内容，则可以使用 ChangeListener 监听器来监听 JTabbedPane 对象。例如如下代码：

```
tabPane.addChangeListener(listener);
```

当用户单击标签页时，系统将把该事件封装成 ChangeEvent 对象，并作为参数来触发 ChangeListener 里的 stateChanged 事件处理器方法。

下面程序定义了具有5个标签页的JTabbedPane面板，该程序可以让用户选择标签布局策略、标签位置。

程序清单：codes\12\12.3\JTabbedPaneTest.java

```
public class JTabbedPaneTest
{
    JFrame jf = new JFrame("测试 Tab 页面");
    // 创建一个 Tab 页面的标签放在左边，采用换行布局策略的 JTabbedPane
    JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.LEFT
        , JTabbedPane.WRAP_TAB_LAYOUT);
    ImageIcon icon = new ImageIcon("ico/close.gif");
    String[] layouts = {"换行布局" , "滚动条布局"};
    String[] positions = {"左边" , "顶部" , "右边" , "底部"};
    Map<String , String> books = new LinkedHashMap<>();
    public void init()
    {
        books.put("疯狂 Java 讲义" , "java.png");
        books.put("轻量级 Java EE 企业应用实战" , "ee.png");
        books.put("疯狂 Ajax 讲义" , "ajax.png");
        books.put("疯狂 Android 讲义" , "android.png");
        books.put("经典 Java EE 企业应用实战" , "classic.png");
        String tip = "可看到本书的封面照片";
        // 向 JTabbedPane 中添加 5 个标签页，指定了标题、图标和提示
        // 但该标签页的组件为 null
        for (String bookName : books.keySet())
        {
            tabbedPane.addTab(bookName, icon, null , tip);
        }
        jf.add(tabbedPane, BorderLayout.CENTER);
        // 为 JTabbedPane 添加事件监听器
        tabbedPane.addChangeListener(event -> {
            // 如果被选择的组件依然是空
            if (tabbedPane.getSelectedComponent() == null)
            {
                // 获取所选标签页
                int n = tabbedPane.getSelectedIndex();
                // 为指定标签页加载内容
                loadTab(n);
            }
        });
        // 系统默认选择第一页，加载第一页内容
        loadTab(0);
        tabbedPane.setPreferredSize(new Dimension(500 , 300));
        // 增加控制标签布局、标签位置的单选按钮
        JPanel buttonPanel = new JPanel();
        ChangeAction action = new ChangeAction();
        buttonPanel.add(new ButtonPanel(action
            , "选择标签布局策略" , layouts));
        buttonPanel.add (new ButtonPanel(action
            , "选择标签位置" , positions));
        jf.add(buttonPanel, BorderLayout.SOUTH);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    // 为指定标签页加载内容
    private void loadTab(int n)
    {
        String title = tabbedPane.getTitleAt(n);
        // 根据标签页的标题获取对应的图书封面
        ImageIcon bookImage = new ImageIcon("ico/"
            + books.get(title));
        tabbedPane.setComponentAt(n , new JLabel(bookImage));
        // 改变标签页的图标
        tabbedPane.setIconAt(n, new ImageIcon("ico/open.gif"));
    }
    // 定义改变标签页的布局策略、放置位置的监听器
    class ChangeAction implements ActionListener
    {
```

```
        public void actionPerformed(ActionEvent event)
        {
            JRadioButton source = (JRadioButton)event.getSource();
            String selection = source.getActionCommand();
            // 设置标签页的标题布局策略
            if (selection.equals(layouts[0]))
            {
                tabbedPane.setTabLayoutPolicy(
                    JTabbedPane.WRAP_TAB_LAYOUT);
            }
            else if (selection.equals(layouts[1]))
            {
                tabbedPane.setTabLayoutPolicy(
                    JTabbedPane.SCROLL_TAB_LAYOUT);
            }
            // 设置标签页的标题放置位置
            else if (selection.equals(positions[0]))
            {
                tabbedPane.setTabPlacement(JTabbedPane.LEFT);
            }
            else if (selection.equals(positions[1]))
            {
                tabbedPane.setTabPlacement(JTabbedPane.TOP);
            }
            else if (selection.equals(positions[2]))
            {
                tabbedPane.setTabPlacement(JTabbedPane.RIGHT);
            }
            else if (selection.equals(positions[3]))
            {
                tabbedPane.setTabPlacement(JTabbedPane.BOTTOM);
            }
        }
    }
    public static void main(String[] args)
    {
        new JTabbedPaneTest().init();
    }
}
// 定义一个 JPanel 类扩展类，该类的对象包含多个纵向排列的 JRadioButton 控件
// 且 JPanel 扩展类可以指定一个字符串作为 TitledBorder
class ButtonPanel extends JPanel
{
    private ButtonGroup group;
    public ButtonPanel(JTabbedPaneTest.ChangeAction action
        , String title, String[] labels)
    {
        setBorder(BorderFactory.createTitledBorder(BorderFactory
            .createEtchedBorder(), title));
        setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
        group = new ButtonGroup();
        for (int i = 0; labels!= null && i < labels.length; i++)
        {
            JRadioButton b = new JRadioButton(labels[i]);
            b.setActionCommand(labels[i]);
            add(b);
            // 添加事件监听器
            b.addActionListener(action);
            group.add(b);
            b.setSelected(i == 0);
        }
    }
}
```

上面程序中的粗体字代码是操作 JTabbedPane 各种属性的代码，这些代码完成了向 JTabbedPane 中添加标签页、改变标签页图标等操作。程序运行后会看到如图 12.15 所示的标签页效果。

如果选择滚动条布局，并选择将标签放在底部，将看到如图 12.16 所示的标签页效果。

图 12.15　标签页效果一

图 12.16　标签页效果二

12.3.3　使用 JLayeredPane、JDesktopPane 和 JInternalFrame

JLayeredPane 是一个代表有层次深度的容器，它允许组件在需要时互相重叠。当向 JLayeredPane 容器中添加组件时，需要为该组件指定一个深度索引，其中层次索引较高的层里的组件位于其他层的组件之上。

JLayeredPane 还将容器的层次深度分成几个默认层，程序只是将组件放入相应的层，从而可以更容易地确保组件的正确重叠，无须为组件指定具体的深度索引。JLayeredPane 提供了如下几个默认层。

- DEFAULT_LAYER：大多数组件位于的标准层。这是最底层。
- PALETTE_LAYER：调色板层位于默认层之上。该层对于浮动工具栏和调色板很有用，因此可以位于其他组件之上。
- MODAL_LAYER：该层用于显示模式对话框。它们将出现在容器中所有工具栏、调色板或标准组件的上面。
- POPUP_LAYER：该层用于显示右键菜单，与对话框、工具提示和普通组件关联的弹出式窗口将出现在对应的对话框、工具提示和普通组件之上。
- DRAG_LAYER：该层用于放置拖放过程中的组件（关于拖放操作请看下一节内容），拖放操作中的组件位于所有组件之上。一旦拖放操作结束后，该组件将重新分配到其所属的正常层。

注意：

每一层都是一个不同的整数。可以在调用 add() 的过程中通过 Integer 参数指定该组件所在的层。也可以传入上面几个静态常量，它们分别等于 0，100，200，300，400 等值。

除此之外，也可以使用 JLayeredPane 的 moveToFront()、moveToBack()和 setPosition()方法在组件所在层中对其进行重定位，还可以使用 setLayer()方法更改该组件所属的层。

下面程序简单示范了 JLayeredPane 容器的用法。

程序清单：codes\12\12.3\JLayeredPaneTest.java

```
public class JLayeredPaneTest
{
    JFrame jf = new JFrame("测试 JLayeredPane");
    JLayeredPane layeredPane = new JLayeredPane();
    public void init()
    {
        // 向 layeredPane 中添加 3 个组件
        layeredPane.add(new ContentPanel(10 , 20 , "疯狂 Java 讲义"
            , "ico/java.png"), JLayeredPane.MODAL_LAYER);
        layeredPane.add(new ContentPanel(100 , 60 , "疯狂 Android 讲义"
            , "ico/android.png"), JLayeredPane.DEFAULT_LAYER);
        layeredPane.add(new ContentPanel(190 , 100
            , "轻量级 Java EE 企业应用实战", "ico/ee.png"), 4);
```

```
        layeredPane.setPreferredSize(new Dimension(400, 300));
        layeredPane.setVisible(true);
        jf.add(layeredPane);
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JLayeredPaneTest().init();
    }
}
// 扩展了 JPanel 类，可以直接创建一个放在指定位置
// 且有指定标题、放置指定图标的 JPanel 对象
class ContentPanel extends JPanel
{
    public ContentPanel(int xPos , int yPos
        , String title , String ico)
    {
        setBorder(BorderFactory.createTitledBorder(
            BorderFactory.createEtchedBorder(), title));
        JLabel label = new JLabel(new ImageIcon(ico));
        add(label);
        setBounds(xPos , yPos , 160, 220);    // ①
    }
}
```

上面程序中粗体字代码向 JLayeredPane 中添加了三个 Panel 组件，每个 Panel 组件都必须显式设置大小和位置（程序中①处代码设置了 Panel 组件的大小和位置），否则该组件不能被显示出来。

运行上面程序，会看到如图 12.17 所示的运行效果。

图 12.17 使用 JLayeredPane 的效果

注意：

向 JLayeredPane 中添加组件时，必须显式设置该组件的大小和位置，否则该组件不能显示出来。

JLayeredPane的子类JDesktopPane容器更加常用——很多应用程序都需要启动多个内部窗口来显示信息（典型的如 Eclipse、EditPlus 都使用了这种内部窗口来分别显示每个 Java 源文件），这些内部窗口都属于同一个外部窗口，当外部窗口最小化时，这些内部窗口都被隐藏起来。在 Windows 环境中，这种用户界面被称为多文档界面（Multiple Document Interface，MDI）。

使用 Swing 可以非常简单地创建出这种 MDI 界面，通常，内部窗口有自己的标题栏、标题、图标、三个窗口按钮，并允许拖动改变内部窗口的大小和位置，但内部窗口不能拖出外部窗口。

提示：

内部窗口与外部窗口表现方式上的唯一区别在于：外部窗口的桌面是实际运行平台的桌面，而内部窗口以外部窗口的指定容器作为桌面。就其实现机制来看，外部窗口和内部窗口则完全不同，外部窗口需要部分依赖于本地平台的 GUI 组件，属于重量级组件；而内部窗口则采用 100%的 Java 实现，属于轻量级组件。

JDesktopPane 需要和 JInternalFrame 结合使用，其中 JDesktopPane 代表一个虚拟桌面，而 JInternalFrame 则用于创建内部窗口。使用 JDesktopPane 和 JInternalFrame 创建内部窗口按如下步骤进行即可。

① 创建一个 JDesktopPane 对象。JDesktopPane 类仅提供了一个无参数的构造器，通过该构造器创建 JDesktopPane 对象，该对象代表一个虚拟桌面。

② 使用 JInternalFrame 创建一个内部窗口。创建内部窗口与创建 JFrame 窗口有一些区别，创建 JInternalFrame 对象时除可以传入一个字符串作为该内部窗口的标题之外，还可以传入 4 个 boolean 值，用于指定该内部窗口是否允许改变窗口大小、关闭窗口、最大化窗口、最小化窗口。例如，下面代码可以创建一个内部窗口。

```
// 创建内部窗口
final JInternalFrame iframe = new JInternalFrame("新文档",
    true, // 可改变大小
    true, // 可关闭
    true, // 可最大化
    true); // 可最小化
```

③ 一旦获得了内部窗口之后，该窗口的用法和普通窗口的用法基本相似，一样可以指定该窗口的布局管理器，一样可以向窗口内添加组件、改变窗口图标等。关于操作内部窗口具体存在哪些方法，请参阅 JInternalFrame 类的 API 文档。

④ 将该内部窗口以合适大小、在合适位置显示出来。与普通窗口类似的是，该窗口默认大小是 0×0 像素，位于 0,0 位置（虚拟桌面的左上角处），并且默认处于隐藏状态，程序可以通过如下代码将内部窗口显示出来。

```
// 同时设置窗口的大小和位置
iframe.reshape(20, 20, 300, 400);
// 使该窗口可见，并尝试选中它
iframe.show();
```

⑤ 将内部窗口添加到 JDesktopPane 容器中，再将 JDesktopPane 容器添加到其他容器中。

注意：

外部窗口的 show()方法已经过时了，不再推荐使用。但内部窗口的 show()方法没有过时，该方法不仅可以让内部窗口显示出来，而且可以让该窗口处于选中状态。

注意：

JDesktopPane 不能独立存在，必须将 JDesktopPane 添加到其他顶级容器中才可以正常使用。

下面程序示范了如何使用 JDesktopPane 和 JInternalFrame 来创建 MDI 界面。

程序清单：codes\12\12.3\JInternalFrameTest.java

```
public class JInternalFrameTest
{
    final int DESKTOP_WIDTH = 480;
    final int DESKTOP_HEIGHT = 360;
    final int FRAME_DISTANCE = 30;
    JFrame jf = new JFrame("MDI 界面");
    // 定义一个虚拟桌面
    private MyJDesktopPane desktop = new MyJDesktopPane();
    // 保存下一个内部窗口的坐标点
    private int nextFrameX;
    private int nextFrameY;
    // 定义内部窗口为虚拟桌面的 1/2 大小
    private int width = DESKTOP_WIDTH / 2;
    private int height = DESKTOP_HEIGHT / 2;
    // 为主窗口定义两个菜单
    JMenu fileMenu = new JMenu("文件");
    JMenu windowMenu = new JMenu("窗口");
    // 定义 newAction 用于创建菜单和工具按钮
    Action newAction = new AbstractAction("新建"
        , new ImageIcon("ico/new.png"))
```

```
    {
        public void actionPerformed(ActionEvent event)
        {
            // 创建内部窗口
            final JInternalFrame iframe = new JInternalFrame("新文档",
                true, // 可改变大小
                true, // 可关闭
                true, // 可最大化
                true); // 可最小化
            iframe.add(new JScrollPane(new JTextArea(8, 40)));
            // 将内部窗口添加到虚拟桌面中
            desktop.add(iframe);
            // 设置内部窗口的原始位置（内部窗口默认大小是 0×0，放在 0,0 位置）
            iframe.reshape(nextFrameX, nextFrameY, width, height);
            // 使该窗口可见，并尝试选中它
            iframe.show();
            // 计算下一个内部窗口的位置
            nextFrameX += FRAME_DISTANCE;
            nextFrameY += FRAME_DISTANCE;
            if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
            if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
        }
    };
    // 定义 exitAction 用于创建菜单和工具按钮
    Action exitAction = new AbstractAction("退出"
        , new ImageIcon("ico/exit.png"))
    {
        public void actionPerformed(ActionEvent event)
        {
            System.exit(0);
        }
    };
    public void init()
    {
        // 为窗口安装菜单条和工具条
        JMenuBar menuBar = new JMenuBar();
        JToolBar toolBar = new JToolBar();
        jf.setJMenuBar(menuBar);
        menuBar.add(fileMenu);
        fileMenu.add(newAction);
        fileMenu.add(exitAction);
        toolBar.add(newAction);
        toolBar.add(exitAction);
        menuBar.add(windowMenu);
        JMenuItem nextItem = new JMenuItem("下一个");
        nextItem.addActionListener(event -> desktop.selectNextWindow());
        windowMenu.add(nextItem);
        JMenuItem cascadeItem = new JMenuItem("级联");
        cascadeItem.addActionListener(event ->
            // 级联显示窗口，内部窗口的大小是外部窗口的 0.75 倍
            desktop.cascadeWindows(FRAME_DISTANCE , 0.75));
        windowMenu.add(cascadeItem);
        JMenuItem tileItem = new JMenuItem("平铺");
        // 平铺显示所有内部窗口
        tileItem.addActionListener(event -> desktop.tileWindows());
        windowMenu.add(tileItem);
        final JCheckBoxMenuItem dragOutlineItem = new
            JCheckBoxMenuItem("仅显示拖动窗口的轮廓");
        dragOutlineItem.addActionListener(event ->
            // 根据该菜单项是否选择来决定采用哪种拖动模式
            desktop.setDragMode(dragOutlineItem.isSelected()
                ? JDesktopPane.OUTLINE_DRAG_MODE
                : JDesktopPane.LIVE_DRAG_MODE));    // ①
        windowMenu.add(dragOutlineItem);
        desktop.setPreferredSize(new Dimension(480, 360));
        // 将虚拟桌面添加到顶级 JFrame 容器中
        jf.add(desktop);
        jf.add(toolBar , BorderLayout.NORTH);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
```

```
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JInternalFrameTest().init();
    }
}
class MyJDesktopPane extends JDesktopPane
{
    // 将所有的窗口以级联方式显示
    // 其中 offset 是两个窗口的位移距离
    // scale 是内部窗口与 JDesktopPane 的大小比例
    public void cascadeWindows(int offset , double scale)
    {
        // 定义级联显示窗口时内部窗口的大小
        int width = (int)(getWidth() * scale);
        int height = (int)(getHeight() * scale);
        // 用于保存级联窗口时每个窗口的位置
        int x = 0;
        int y = 0;
        for (JInternalFrame frame : getAllFrames())
        {
            try
            {
                // 取消内部窗口的最大化、最小化
                frame.setMaximum(false);
                frame.setIcon(false);
                // 把窗口重新放置在指定位置
                frame.reshape(x, y, width, height);
                x += offset;
                y += offset;
                // 如果到了虚拟桌面边界
                if (x + width > getWidth()) x = 0;
                if (y + height > getHeight()) y = 0;
            }
            catch (PropertyVetoException e)
            {}
        }
    }
    // 将所有窗口以平铺方式显示
    public void tileWindows()
    {
        // 统计所有窗口
        int frameCount = 0;
        for (JInternalFrame frame : getAllFrames())
        {
            frameCount++;
        }
        // 计算需要多少行、多少列才可以平铺所有窗口
        int rows = (int) Math.sqrt(frameCount);
        int cols = frameCount / rows;
        // 需要额外增加到其他列中的窗口
        int extra = frameCount % rows;
        // 计算平铺时内部窗口的大小
        int width = getWidth() / cols;
        int height = getHeight() / rows;
        // 用于保存平铺窗口时每个窗口在横向、纵向上的索引
        int x = 0;
        int y = 0;
        for (JInternalFrame frame : getAllFrames())
        {
            try
            {
                // 取消内部窗口的最大化、最小化
                frame.setMaximum(false);
                frame.setIcon(false);
                // 将窗口放在指定位置
                frame.reshape(x * width, y * height, width, height);
                y++;
                // 每排完一列窗口
```

```
                if (y == rows)
                {
                    // 开始排放下一列窗口
                    y = 0;
                    x++;
                    // 如果额外多出的窗口与剩下的列数相等
                    // 则后面所有列都需要多排列一个窗口
                    if (extra == cols - x)
                    {
                        rows++;
                        height = getHeight() / rows;
                    }
                }
            }
            catch (PropertyVetoException e)
            {}
        }
    }
    // 选中下一个非图标窗口
    public void selectNextWindow()
    {
        JInternalFrame[] frames = getAllFrames();
        for (int i = 0; i < frames.length; i++)
        {
            if (frames[i].isSelected())
            {
                // 找出下一个非最小化的窗口，尝试选中它
                // 如果选中失败，则继续尝试选中下一个窗口
                int next = (i + 1) % frames.length;
                while (next != i)
                {
                    // 如果该窗口不是处于最小化状态
                    if (!frames[next].isIcon())
                    {
                        try
                        {
                            frames[next].setSelected(true);
                            frames[next].toFront();
                            frames[i].toBack();
                            return;
                        }
                        catch (PropertyVetoException e)
                        {}
                    }
                    next = (next + 1) % frames.length;
                }
            }
        }
    }
}
```

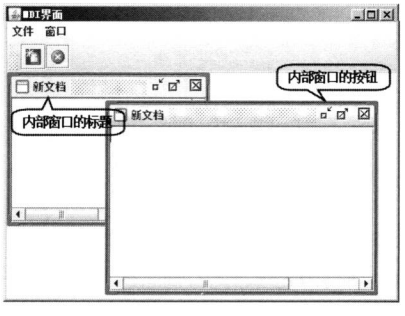


图 12.18 内部窗口效果

上面程序中粗体字代码示范了创建JDesktopPane虚拟桌面，创建 JInternatFrame 内部窗口，并将内部窗口添加到虚拟桌面中，最后将虚拟桌面添加到顶级 JFrame 容器中的过程。

运行上面程序，会看到如图 12.18 所示的内部窗口效果。

在默认情况下，当用户拖动窗口时，内部窗口会紧紧跟随用户鼠标的移动，这种操作会导致系统不断重绘虚拟桌面的内部窗口，从而引起性能下降。为了改变这种拖动模式，可以设置当用户拖动内部窗口时，虚拟桌面上仅绘出该内部窗口的轮廓。可以通过调用 JDesktopPane 的 setDragMode()方法来改变内部窗口的拖动模式，该方法接收如下两个参数值。

- JDesktopPane.OUTLINE_DRAG_MODE：拖动过程中仅显示内部窗口的轮廓。

➢ JDesktopPane.LIVE_DRAG_MODE：拖动过程中显示完整窗口，这是默认选项。

上面程序中①处代码允许用户根据 JCheckBoxMenuItem 的状态来决定窗口采用哪种拖动模式。

读者可能会发现，程序创建虚拟桌面时并不是直接创建 JDesktopPane 对象，而是先扩展了 JDesktopPane 类，为该类增加了如下三个方法。

➢ cascadeWindows()：级联显示所有的内部窗口。

➢ tileWindows()：平铺显示所有的内部窗口。

➢ selectNextWindow()：选中当前窗口的下一个窗口。

JDesktopPane 没有提供这三个方法，但这三个方法在 MDI 应用里又是如此常用，以至于开发者总需要自己来扩展 JDesktopPane 类，而不是直接使用该类。这是一个非常有趣的地方——Oracle 似乎认为这些方法太过简单，不屑为之，于是开发者只能自己实现，这给编程带来一些麻烦。

级联显示窗口其实很简单，先根据内部窗口与 JDesktopPane 的大小比例计算出每个内部窗口的大小，然后以此重新排列每个窗口，重排之前让相邻两个窗口在横向、纵向上产生一定的位移即可。

平铺显示窗口相对复杂一点，程序先计算需要几行、几列可以显示所有的窗口，如果还剩下多余（不能整除）的窗口，则依次分布到最后几列中。图 12.19 显示了平铺窗口的效果。

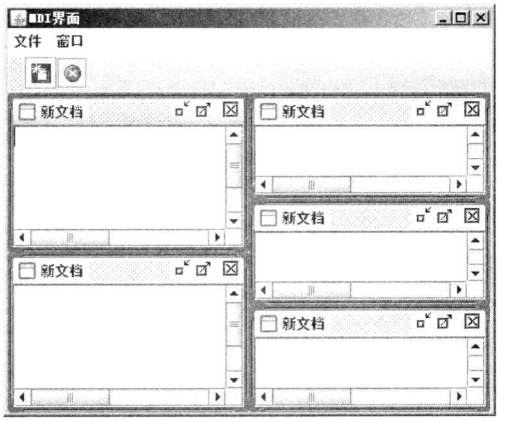


图 12.19　平铺显示所有的窗口效果

前面介绍 JOptionPane 时提到该类包含了多个重载的 showInternalXxxDialog()方法，这些方法用于弹出内部对话框，当使用该方法来弹出内部对话框时通常需要指定一个父组件，这个父组件既可以是虚拟桌面（JDesktopPane 对象），也可以是内部窗口（JInternalFrame 对象）。下面程序示范了如何弹出内部对话框。

程序清单：codes\12\12.3\InternalDialogTest.java

```
public class InternalDialogTest
{
    private JFrame jf = new JFrame("测试内部对话框");
    private JDesktopPane desktop = new JDesktopPane();
    private JButton internalBn = new JButton("内部窗口的对话框");
    private JButton deskBn = new JButton("虚拟桌面的对话框");
    // 定义一个内部窗口，该窗口可拖动，但不可最大化、最小化、关闭
    private JInternalFrame iframe = new JInternalFrame("内部窗口");
    public void init()
    {
        // 向内部窗口中添加组件
        iframe.add(new JScrollPane(new JTextArea(8, 40)));
        desktop.setPreferredSize(new Dimension(400, 300));
        // 把虚拟桌面添加到 JFrame 窗口中
        jf.add(desktop);
        // 设置内部窗口的大小、位置
        iframe.reshape(0 , 0 , 300 , 200);
        // 显示并选中内部窗口
        iframe.show();
        desktop.add(iframe);
        JPanel jp = new JPanel();
        deskBn.addActionListener(event ->
            // 弹出内部对话框，以虚拟桌面作为父组件
            JOptionPane.showInternalMessageDialog(desktop
                , "属于虚拟桌面的对话框"));
        internalBn.addActionListener(event ->
            // 弹出内部对话框，以内部窗口作为父组件
            JOptionPane.showInternalMessageDialog(iframe
                , "属于内部窗口的对话框"));
```

```
        jp.add(deskBn);
        jp.add(internalBn);
        jf.add(jp , BorderLayout.SOUTH);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new InternalDialogTest().init();
    }
}
```

上面程序中两行粗体字弹出两个内部对话框，这两个对话框一个以虚拟桌面作为父窗口，一个以内部窗口作为父组件。运行上面程序会看到如图 12.20 所示的内部窗口的对话框。

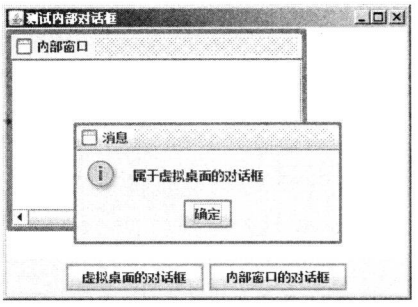


图 12.20 内部窗口的对话框

12.4 Swing 简化的拖放功能

从 JDK 1.4 开始，Swing 的部分组件已经提供了默认的拖放支持，从而能以更简单的方式进行拖放操作。Swing 中支持拖放操作的组件如表 12.1 所示。

表 12.1 支持拖放操作的 Swing 组件

Swing 组件	作为拖放源导出	作为拖放目标接收
JColorChooser	导出颜色对象的本地引用	可接收任何颜色
JFileChooser	导出文件列表	无
JList	导出所选择节点的 HTML 描述	无
JTable	导出所选中的行	无
JTree	导出所选择节点的 HTML 描述	无
JTextComponent	导出所选文本	接收文本，其子类 JTextArea 还可接收文件列表，负责将文件打开

在默认情况下，表 12.1 中的这些 Swing 组件都没有启动拖放支持，可以调用这些组件的 setDragEnabled (true)方法来启动拖放支持。下面程序示范了 Swing 提供的拖放支持。

程序清单：codes\12\12.4\SwingDndSupport.java

```
public class SwingDndSupport
{
    JFrame jf = new JFrame("Swing 的拖放支持");
    JTextArea srcTxt = new JTextArea(8 , 30);
    JTextField jtf = new JTextField(34);
    public void init()
    {
        srcTxt.append("Swing 的拖放支持.\n");
        srcTxt.append("将该文本域的内容拖入其他程序.\n");
        // 启动文本域和单行文本框的拖放支持
        srcTxt.setDragEnabled(true);
        jtf.setDragEnabled(true);
        jf.add(new JScrollPane(srcTxt));
        jf.add(jtf , BorderLayout.SOUTH);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SwingDndSupport().init();
    }
}
```

上面程序中的两行粗体字代码负责开始多行文本域和单行文本框的拖放支持。运行上面程序，会看到如图 12.21 所示的界面。

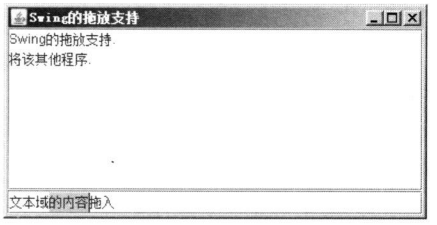


图 12.21　启用 Swing 组件的拖放功能

除此之外，Swing 还提供了一种非常特殊的类：TransferHandler，它可以直接将某个组件的指定属性设置成拖放目标，前提是该组件具有该属性的 setter 方法。例如，JTextArea 类提供了一个 setForeground(Color) 方法，这样即可利用 TransferHandler 将 foreground 定义成拖放目标。代码如下：

```
// 允许直接将一个 Color 对象拖入该 JTextArea 对象，并赋给它的 foreground 属性
txt.setTransferHandler(new TransferHandler("foreground"));
```

下面程序可以直接把颜色选择器面板中的颜色拖放到指定文本域中，用以改变指定文本域的前景色。

程序清单：codes\12\12.4\TransferHandlerTest.java

```
public class TransferHandlerTest
{
    private JFrame jf = new JFrame("测试 TransferHandler");
    JColorChooser chooser = new JColorChooser();
    JTextArea txt = new JTextArea("测试 TransferHandler\n"
        + "直接将上面颜色拖入以改变文本颜色");
    public void init()
    {
        // 启动颜色选择器面板和文本域的拖放功能
        chooser.setDragEnabled(true);
        txt.setDragEnabled(true);
        jf.add(chooser, BorderLayout.SOUTH);
        // 允许直接将一个 Color 对象拖入该 JTextArea 对象
        // 并赋给它的 foreground 属性
        txt.setTransferHandler(new TransferHandler("foreground"));
        jf.add(new JScrollPane(txt));
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.pack();
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TransferHandlerTest().init();
    }
}
```

上面程序中的粗体字代码将 JTextArea 的 foreground 属性转换成拖放目标，它可以接收任何 Color 对象。而 JColorChooser 启动拖放功能后可以导出颜色对象的本地引用，从而可以直接将该颜色对象拖给 JTextArea 的 foreground 属性。运行上面程序，会看到如图 12.22 所示的界面。

从图 12.22 中可以看出，当用户把颜色选择器面板中预览区的颜色拖到上面多行文本域后，多行文本域的颜色也随之发生改变。

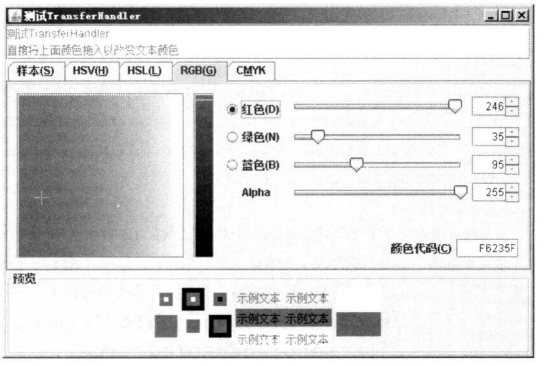


图 12.22　通过拖放操作改变文本域的前景色

12.5　Java 7 新增的 Swing 功能

Java 7 提供的重大更新就包括了对 Swing 的更新，对 Swing 的更新除前面介绍的 Nimbus 外观、改进的 JColorChooser 组件之外，还有两个很有用的更新——JLayer 和创建不规则窗口。下面将会详细介绍这两个知识点。

12.5.1　使用 JLayer 装饰组件

JLayer 的功能是在指定组件上额外地添加一个装饰层，开发者可以在这个装饰层上进行任意绘制

（直接重写 paint(Graphics g, JComponent c)方法），这样就可以为指定组件添加任意装饰。

JLayer 一般总是要和 LayerUI 一起使用，而 LayerUI 用于被扩展，扩展 LayerUI 时重写它的 paint(Graphics g, JComponent c)方法，在该方法中绘制的内容会对指定组件进行装饰。

实际上，使用 JLayer 很简单，只要如下两行代码即可。

```
// 创建 LayerUI 对象
LayerUI<JComponent> layerUI = new XxxLayerUI();
// 使用 layerUI 来装饰指定的 JPanel 组件
JLayer<JComponent> layer = new JLayer<JComponent>(panel, layerUI);
```

上面程序中的 XxxLayerUI 就是开发者自己扩展的子类，这个子类会重写 paint(Graphics g, JComponent c)方法，重写该方法来完成“装饰层”的绘制。

上面第二行代码中的 panel 组件就是被装饰的组件，接下来把 layer 对象（layer 对象包含了被装饰对象和 LayerUi 对象）添加到指定容器中即可。

下面程序示范了使用 JLayer 为窗口添加一层“蒙版”的效果。

程序清单：codes\12\12.5\JLayerTest.java

```
class FirstLayerUI extends LayerUI<JComponent>
{
    public void paint(Graphics g, JComponent c)
    {
        super.paint(g, c);
        Graphics2D g2 = (Graphics2D) g.create();
        // 设置透明效果
        g2.setComposite(AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER, .5f));
        // 使用渐变画笔绘图
        g2.setPaint(new GradientPaint(0 , 0 , Color.RED
            , 0 , c.getHeight() , Color.BLUE));
        // 绘制一个与被装饰组件具有相同大小的组件
        g2.fillRect(0, 0, c.getWidth(), c.getHeight());      // ①
        g2.dispose();
    }
}
public class JLayerTest
{
    public void init()
    {
        JFrame f = new JFrame("JLayer 测试");
        JPanel p = new JPanel();
        ButtonGroup group = new ButtonGroup();
        JRadioButton radioButton;
        // 创建 3 个 RadioButton，并将它们添加成一组
        p.add(radioButton = new JRadioButton("网购购买", true));
        group.add(radioButton);
        p.add(radioButton = new JRadioButton("书店购买"));
        group.add(radioButton);
        p.add(radioButton = new JRadioButton("图书馆借阅"));
        group.add(radioButton);
        // 添加 3 个 JCheckBox
        p.add(new JCheckBox("疯狂 Java 讲义"));
        p.add(new JCheckBox("疯狂 Android 讲义"));
        p.add(new JCheckBox("疯狂 Ajax 讲义"));
        p.add(new JCheckBox("轻量级 Java EE 企业应用"));
        JButton orderButton = new JButton("投票");
        p.add(orderButton);
        // 创建 LayerUI 对象
        LayerUI<JComponent> layerUI = new FirstLayerUI();      // ②
        // 使用 layerUI 来装饰指定的 JPanel 组件
        JLayer<JComponent> layer = new JLayer<JComponent>(p, layerUI);
        // 将装饰后的 JPanel 组件添加到容器中
        f.add(layer);
        f.setSize(300, 170);
        f.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        f.setVisible (true);
```

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

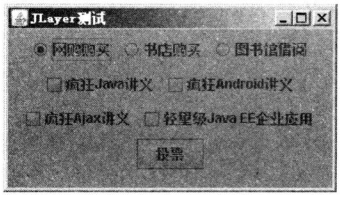


图 12.23　被装饰的 JPanel

上面程序中开发了一个 FirstLayerUI，它扩展了 LayerUI，重写 paint(Graphics g, JComponent c)方法时绘制了一个半透明的、与被装饰组件具有相同大小的矩形。接下来在 main 方法中使用这个 LayerUI 来装饰指定的 JPanel 组件，并把 JLayer 添加到 JFrame 容器中，这就达到了对 JPanel 进行包装的效果。运行该程序，可以看到如图 12.23 所示的效果。

由于开发者可以重写 paint(Graphics g, JComponent c)方法，因此获得对被装饰层的全部控制权——想怎么绘制，就怎么绘制！因此开发者可以“随心所欲”地对指定组件进行装饰。例如，下面提供的 LayerUI 则可以为被装饰组件增加“模糊”效果。程序如下（程序清单同上）。

```
class BlurLayerUI extends LayerUI<JComponent>
{
    private BufferedImage screenBlurImage;
    private BufferedImageOp operation;
    public BlurLayerUI()
    {
        float ninth = 1.0f / 9.0f;
        // 定义模糊参数
        float[] blurKernel = {
            ninth, ninth, ninth,
            ninth, ninth, ninth,
            ninth, ninth, ninth
        };
        // ConvolveOp 代表一个模糊处理，它将原图片的每一个像素与周围
        // 像素的颜色进行混合，从而计算出当前像素的颜色值
        operation = new ConvolveOp(
            new Kernel(3, 3, blurKernel),
            ConvolveOp.EDGE_NO_OP, null);
    }
    public void paint(Graphics g, JComponent c)
    {
        int w = c.getWidth();
        int h = c.getHeight();
        // 如果被装饰窗口大小为 0×0，直接返回
        if (w == 0 || h == 0)
            return;
        // 如果 screenBlurImage 没有初始化，或它的尺寸不对
        if (screenBlurImage == null
            || screenBlurImage.getWidth() != w
            || screenBlurImage.getHeight() != h)
        {
            // 重新创建新的 BufferdImage
            screenBlurImage = new BufferedImage(w
                , h , BufferedImage.TYPE_INT_RGB);
        }
        Graphics2D ig2 = screenBlurImage.createGraphics();
        // 把被装饰组件的界面绘制到当前 screenBlurImage 上
        ig2.setClip(g.getClip());
        super.paint(ig2, c);
        ig2.dispose();
        Graphics2D g2 = (Graphics2D)g;
        // 对 JLayer 装饰的组件进行模糊处理
        g2.drawImage(screenBlurImage, operation, 0, 0);
    }
}
```

上面程序扩展了 LayerUI，重写了 paint(Graphics g, JComponent c)方法，重写该方法时也是绘制

了一个与被装饰组件具有相同大小的矩形，只是这种绘制添加了模糊效果。

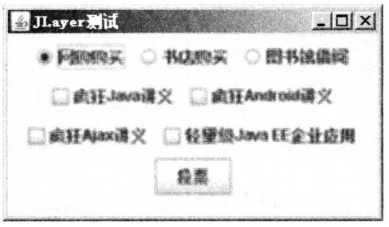


图 12.24　使用 JLayer 装饰的“毛玻璃”窗口

将 JLayerTest.java 中的 ②号粗体字代码改为使用 BlurLayerUI，再次运行该程序，将可以看到如图 12.24 所示的“毛玻璃”窗口。

除此之外，开发者自定义的 LayerUI 还可以增加事件机制，这种事件机制能让装饰层响应用户动作，随着用户动作动态地改变 LayerUI 上的绘制效果。比如下面的 LayerUI 示例，程序通过响应鼠标事件，可以在窗口上增加“探照灯”效果。程序如下（程序清单同上）。

```
class SpotlightLayerUI extends LayerUI<JComponent>
{
    private boolean active;
    private int cx, cy;
    public void installUI(JComponent c)
    {
        super.installUI(c);
        JLayer layer = (JLayer)c;
        // 设置 JLayer 可以响应鼠标事件和鼠标动作事件
        layer.setLayerEventMask(AWTEvent.MOUSE_EVENT_MASK
            | AWTEvent.MOUSE_MOTION_EVENT_MASK);      // ①
    }
    public void uninstallUI(JComponent c)
    {
        JLayer layer = (JLayer)c;
        // 设置 JLayer 不响应任何事件
        layer.setLayerEventMask(0);
        super.uninstallUI(c);
    }
    public void paint(Graphics g, JComponent c)
    {
        Graphics2D g2 = (Graphics2D)g.create();
        super.paint (g2, c);
        // 如果处于激活状态
        if (active)
        {
            // 定义一个 cx、cy 位置的点
            Point2D center = new Point2D.Float(cx, cy);
            float radius = 72;
            float[] dist = {0.0f, 1.0f};
            Color[] colors = {Color.YELLOW , Color.BLACK};
            // 以 center 为中心、colors 为颜色数组创建环形渐变
            RadialGradientPaint p = new RadialGradientPaint(center
                , radius , dist , colors);
            g2.setPaint(p);
            // 设置渐变效果
            g2.setComposite(AlphaComposite.getInstance(
                AlphaComposite.SRC_OVER, .6f));
            // 绘制矩形
            g2.fillRect(0, 0, c.getWidth(), c.getHeight());
        }
        g2.dispose();
    }
    // 处理鼠标事件的方法
    public void processMouseEvent(MouseEvent e, JLayer layer)
    {
        if (e.getID() == MouseEvent.MOUSE_ENTERED)
            active = true;
        if (e.getID() == MouseEvent.MOUSE_EXITED)
            active = false;
        layer.repaint();
    }
    // 处理鼠标动作事件的方法
    public void processMouseMotionEvent(MouseEvent e, JLayer layer)
    {
        Point p = SwingUtilities.convertPoint(
```

```
            e.getComponent(), e.getPoint(), layer);
        // 获取鼠标动作事件发生点的坐标
        cx = p.x;
        cy = p.y;
        layer.repaint();
    }
}
```

上面程序中重写了 LayerUI 的 installUI(JComponent c)方法，重写该方法时控制该组件能响应鼠标事件和鼠标动作事件，如粗体字代码所示。接下来程序重写了 processMouseMotionEvent()方法，该方法负责为 LayerUI 上的鼠标事件提供响应——当鼠标在界面上移动时，程序会改变 cx、cy 的坐标值，重写 paint(Graphics g, JComponent c)方法时会在 cx、cy 对应的点绘制一个环形渐变，这就可以充当“探照灯”效果了。将 JLayerTest.java 中的②号粗体字代码改为使用 SpotlightLayerUI，再次运行该程序，即可看到如图 12.25 所示的效果。

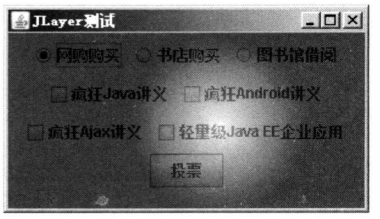


图 12.25　窗口上的“探照灯”效果

既然可以让 LayerUI 上的绘制效果响应鼠标动作，当然也可以在 LayerUI 上绘制“动画”——所谓动画，就是通过定时器控制 LayerUI 上绘制的图形动态地改变即可。

接下来重写的 LayerUI 使用了 Timer 来定时地改变 LayerUI 上的绘制，程序绘制了一个旋转中的“齿轮”，这个旋转的齿轮可以提醒用户“程序正在处理中”。

下面程序重写 LayerUI 时绘制了 12 条辐射状的线条，并通过 Timer 来不断地改变这 12 条线条的排列角度，这样就可以形成“转动的齿轮”了。程序提供的 WaitingLayerUI 类代码如下。

程序清单：codes\12\12.5\WaitingJLayerTest.java

```
class WaitingLayerUI extends LayerUI<JComponent>
{
    private boolean isRunning;
    private Timer timer;
    // 记录转过的角度
    private int angle;      // ①
    public void paint(Graphics g, JComponent c)
    {
        super.paint(g, c);
        int w = c.getWidth();
        int h = c.getHeight();
        // 已经停止运行，直接返回
        if (!isRunning)
            return;
        Graphics2D g2 = (Graphics2D)g.create();
        Composite urComposite = g2.getComposite();
        g2.setComposite(AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER, .5f));
        // 填充矩形
        g2.fillRect(0, 0, w, h);
        g2.setComposite(urComposite);
        // -----下面代码开始绘制转动中的“齿轮”----
        // 计算得到宽、高中较小值的 1/5
        int s = Math.min(w , h) / 5;
        int cx = w / 2;
        int cy = h / 2;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING
            , RenderingHints.VALUE_ANTIALIAS_ON);
        // 设置笔触
        g2.setStroke( new BasicStroke(s / 2
            , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND));
        g2.setPaint(Color.BLUE);
        // 画笔绕被装饰组件的中心转过 angle 度
        g2.rotate(Math.PI * angle / 180, cx, cy);       // ②
        // 循环绘制 12 条线条，形成“齿轮”
        for (int i = 0; i < 12; i++)
        {
```

```
            float scale = (11.0f - (float)i) / 11.0f;
            g2.drawLine(cx + s, cy, cx + s * 2, cy);
            g2.rotate(-Math.PI / 6, cx, cy);
            g2.setComposite(AlphaComposite.getInstance(
                AlphaComposite.SRC_OVER, scale));
        }
        g2.dispose();
    }
    // 控制等待（齿轮开始转动）的方法
    public void start()
    {
        // 如果已经在运行中，直接返回
        if (isRunning)
            return;
        isRunning = true;
        // 每隔 0.1 秒重绘一次
        timer = new Timer(100, e -> {
            if (isRunning)
            {
                // 触发 applyPropertyChange()方法，让 JLayer 重绘
                // 在这行代码中，后面两个参数没有意义
                firePropertyChange("crazyitFlag", 0 , 1);
                // 角度加 6
                angle += 6;     // ③
                // 到达 360 角度后再从 0 开始
                if (angle >= 360)
                    angle = 0;
            }
        });
        timer.start();
    }
    // 控制停止等待（齿轮停止转动）的方法
    public void stop()
    {
        isRunning = false;
        // 最后通知 JLayer 重绘一次，清除曾经绘制的图形
        firePropertyChange("crazyitFlag", 0 , 1);
        timer.stop();
    }
    public void applyPropertyChange(PropertyChangeEvent pce
        , JLayer layer)
    {
        // 控制 JLayer 重绘
        if (pce.getPropertyName().equals("crazyitFlag"))
        {
            layer.repaint();
        }
    }
}
```

上面程序中的①号粗体字代码定义了一个 angle 变量，它负责控制 12 条线条的旋转角度。程序使用 Timer 定时地改变 angle 变量的值（每隔 0.1 秒 angle 加 6），如③号粗体字代码所示。控制了 angle 角度之后，程序根据该 angle 角度绘制 12 条线条，如②号粗体字代码所示。

提供了 WaitingLayerUI 之后，接下来使用该 WaitingLayerUI 与使用前面的 UI 没有任何区别。不过程序需要通过特定事件来显示 WaitingLayerUI 的绘制（就是调用它的 start()方法），下面程序为按钮添加了事件监听器——当用户单击该按钮时，程序会调用 WaitingLayerUI 对象的 start()方法（程序清单同上）。

```
// 为 orderButton 绑定事件监听器：单击该按钮时，调用 layerUI 的 start()方法
orderButton.addActionListener(ae -> {
    layerUI.start();
    // 如果 stopper 定时器已停止，则启动它
    if (!stopper.isRunning())
    {
        stopper.start();
    }
});
```

除此之外，上面代码中还用到了 stopper 计时器，它会控制在一段时间（比如 4 秒）之后停止绘制 WaitingLayerUI，因此程序还通过如下代码进行控制（程序清单同上）。

```
// 设置 4 秒之后执行指定动作：调用 layerUI 的 stop()方法
final Timer stopper = new Timer(4000, ae -> layerUI.stop());
// 设置 stopper 定时器只触发一次
stopper.setRepeats(false);
```

再次运行该程序，可以看到如图 12.26 所示的“动画装饰”效果。

通过上面几个例子可以看出，Swing 提供的 JLayer 为窗口美化提供了无限可能性。只要你想做的，比如希望用户完成输入之后，立即在后面显示一个简单的提示按钮（钩表示输入正确，叉表示输入错误）……都可以通过 JLayer 绘制。

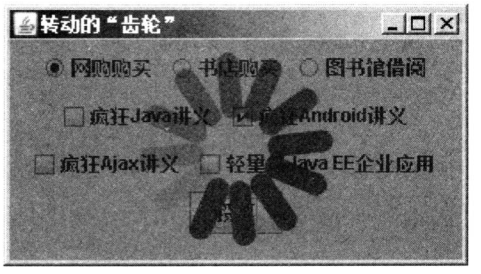


图 12.26　JLayer 生成的“动画装饰”效果

12.5.2　创建透明、不规则形状窗口

Java 7 为 Frame 提供了如下两个方法。

- setShape(Shape shape)：设置窗口的形状，可以将窗口设置成任意不规则的形状。
- setOpacity(float opacity)：设置窗口的透明度，可以将窗口设置成半透明的。当 opacity 为 1.0f 时，该窗口完全不透明。

这两个方法简单、易用，可以直接改变窗口的形状和透明度。除此之外，如果希望开发出渐变透明的窗口，则可以考虑使用一个渐变透明的 JPanel 来代替 JFrame 的 ContentPane；按照这种思路，还可以开发出有图片背景的窗口。

下面程序示范了如何开发出透明、不规则的窗口。

程序清单：codes\12\12.5\NonRegularWindow.java

```
public class NonRegularWindow extends JFrame
    implements ActionListener
{
    // 定义 3 个窗口
    JFrame transWin = new JFrame("透明窗口");
    JFrame gradientWin = new JFrame("渐变透明窗口");
    JFrame bgWin = new JFrame("背景图片窗口");
    JFrame shapeWin = new JFrame("椭圆窗口");
    public NonRegularWindow()
    {
        super("不规则窗口测试");
        setLayout(new FlowLayout());
        JButton transBn = new JButton("透明窗口");
        JButton gradientBn = new JButton("渐变透明窗口");
        JButton bgBn = new JButton("背景图片窗口");
        JButton shapeBn = new JButton("椭圆窗口");
        // 为 3 个按钮添加事件监听器
        transBn.addActionListener(this);
        gradientBn.addActionListener(this);
        bgBn.addActionListener(this);
        shapeBn.addActionListener(this);
        add(transBn);
        add(gradientBn);
        add(bgBn);
        add(shapeBn);
        //-------设置透明窗口-------
        transWin.setLayout(new GridBagLayout());
        transWin.setSize(300,200);
        transWin.add(new JButton("透明窗口里的简单按钮"));
        // 设置透明度为 0.65f，透明度为 1f 时完全不透明
        transWin.setOpacity(0.65f);
        //-------设置渐变透明的窗口-------
        gradientWin.setBackground(new Color(0,0,0,0));
        gradientWin.setSize(new Dimension(300,200));
```

```
// 使用一个 JPanel 对象作为渐变透明的背景
JPanel panel = new JPanel()
{
    protected void paintComponent(Graphics g)
    {
        if (g instanceof Graphics2D)
        {
            final int R = 240;
            final int G = 240;
            final int B = 240;
            // 创建一个渐变画笔
            Paint p = new GradientPaint(0.0f, 0.0f
                , new Color(R, G, B, 0)
                , 0.0f, getHeight()
                , new Color(R, G, B, 255) , true);
            Graphics2D g2d = (Graphics2D)g;
            g2d.setPaint(p);
            g2d.fillRect(0, 0, getWidth(), getHeight());
        }
    }
};
// 使用 JPanel 对象作为 JFrame 的 contentPane
gradientWin.setContentPane(panel);
panel.setLayout(new GridBagLayout());
gradientWin.add(new JButton("渐变透明窗口里的简单按钮"));
//-------设置有背景图片的窗口-------
bgWin.setBackground(new Color(0,0,0,0));
bgWin.setSize(new Dimension(300,200));
// 使用一个 JPanel 对象作为背景图片
JPanel bgPanel = new JPanel()
{
    protected void paintComponent(Graphics g)
    {
        try
        {
            Image bg = ImageIO.read(new File("images/java.png"));
            // 绘制一张图片作为背景
            g.drawImage(bg , 0 , 0 , getWidth() , getHeight() ,  null);
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
};
// 使用 JPanel 对象作为 JFrame 的 contentPane
bgWin.setContentPane(bgPanel);
bgPanel.setLayout(new GridBagLayout());
bgWin.add(new JButton("有背景图片窗口里的简单按钮"));
//-------设置椭圆形窗口-------
shapeWin.setLayout(new GridBagLayout());
shapeWin.setUndecorated(true);
shapeWin.setOpacity(0.7f);
// 通过为 shapeWin 添加监听器来设置窗口的形状
// 当 shapeWin 窗口的大小被改变时，程序动态设置该窗口的形状
shapeWin.addComponentListener(new ComponentAdapter()
{
    // 当窗口大小被改变时，椭圆的大小也会相应地改变
    public void componentResized(ComponentEvent e)
    {
        // 设置窗口的形状
        shapeWin.setShape(new Ellipse2D.Double(0 , 0
            , shapeWin.getWidth() , shapeWin.getHeight())); // ①
    }
});
shapeWin.setSize(300,200);
shapeWin.add(new JButton("椭圆形窗口里的简单按钮"));
//-------设置主程序的窗口-------
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
pack();
```

```
            setVisible(true);
        }
        public void actionPerformed(ActionEvent event)
        {
            switch(event.getActionCommand())
            {
                case "透明窗口":
                    transWin.setVisible(true);
                    break;
                case "渐变透明窗口":
                    gradientWin.setVisible(true);
                    break;
                case "背景图片窗口":
                    bgWin.setVisible(true);
                    break;
                case "椭圆窗口":
                    shapeWin.setVisible(true);
                    break;
            }
        }
        public static void main(String[] args)
        {
            JFrame.setDefaultLookAndFeelDecorated(true);
            new NonRegularWindow();
        }
    }
```

上面程序中的粗体字代码就是设置透明窗口、渐变透明窗口、有背景图片窗口的关键代码；当需要开发不规则形状的窗口时，程序往往会为该窗口实现一个 ComponentListener，该监听器负责监听窗口大小发生改变的事件——当窗口大小发生改变时，程序调用窗口的 setShape()方法来控制窗口的形状，如 ①号粗体字代码所示。

运行上面程序，打开透明窗口和渐变透明窗口，效果如图 12.27 所示。

打开有背景图片窗口和椭圆窗口，效果如图 12.28 所示。

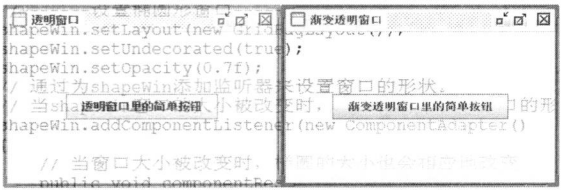


图 12.27　透明窗口和渐变透明窗口

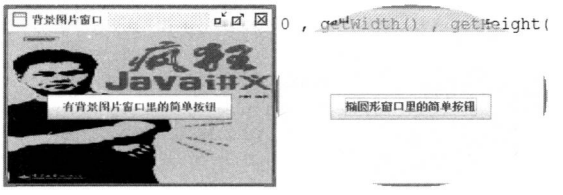


图 12.28　有背景图片窗口和椭圆窗口

12.6　使用 JProgressBar、ProgressMonitor 和 BoundedRangeModel 创建进度条

进度条是图形界面中广泛使用的 GUI 组件，当复制一个较大的文件时，操作系统会显示一个进度条，用于标识复制操作完成的比例；当启动 Eclipse 等程序时，因为需要加载较多的资源，故而启动速度较慢，程序也会在启动过程中显示一个进度条，用以表示该软件启动完成的比例……

12.6.1　创建进度条

使用 JProgressBar 可以非常方便地创建进度条，使用 JProgressBar 创建进度条可按如下步骤进行。

① 创建一个 JProgressBar 对象，创建该对象时可以指定三个参数，用于设置进度条的排列方向（竖直和水平）、进度条的最大值和最小值。也可以在创建该对象时不传入任何参数，而是在后面程序中修改这三个属性。例如，如下代码创建了 JProgressBar 对象。

```
// 创建一条垂直进度条
JProgressBar bar = new JProgressBar(JProgressBar.VERTICAL );
```

② 调用该对象的常用方法设置进度条的普通属性。JProgressBar 除提供设置排列方向、最大值、最小值的 setter 和 getter 方法之外，还提供了如下三个方法：

- setBorderPainted(boolean b)：设置该进度条是否使用边框。
- setIndeterminate(boolean newValue)：设置该进度条是否是进度不确定的进度条，如果指定一个进度条的进度不确定，将看到一个滑块在进度条中左右移动。
- setStringPainted(boolean newValue)：设置是否在进度条中显示完成百分比。

当然，JProgressBar 也为上面三个属性提供了 getter 方法，但这三个 getter 方法通常没有太大作用。

③ 当程序中工作进度改变时，调用 JProgressBar 对象的 setValue()方法。当进度条的完成进度发生改变时，程序还可以调用进度条对象的如下两个方法。

- double getPercentComplete()：返回进度条的完成百分比。
- String getString()：返回进度字符串的当前值。

下面程序示范了使用进度条的简单例子。

程序清单：codes\12\12.6\JProgressBarTest.java

```
public class JProgressBarTest
{
    JFrame frame = new JFrame("测试进度条");
    // 创建一条垂直进度条
    JProgressBar bar = new JProgressBar(JProgressBar.VERTICAL );
    JCheckBox indeterminate = new JCheckBox("不确定进度");
    JCheckBox noBorder = new JCheckBox("不绘制边框");
    public void init()
    {
        Box box = new Box(BoxLayout.Y_AXIS);
        box.add(indeterminate);
        box.add(noBorder);
        frame.setLayout(new FlowLayout());
        frame.add(box);
        // 把进度条添加到 JFrame 窗口中
        frame.add(bar);
        // 设置进度条的最大值和最小值
        bar.setMinimum(0);
        bar.setMaximum(100);
        // 设置在进度条中绘制完成百分比
        bar.setStringPainted(true);
        // 根据该选择框决定是否绘制进度条的边框
        noBorder.addActionListener(event ->
            bar.setBorderPainted(!noBorder.isSelected()));
        indeterminate.addActionListener(event -> {
            // 设置该进度条的进度是否确定
            bar.setIndeterminate(indeterminate.isSelected());
            bar.setStringPainted(!indeterminate.isSelected());
        });       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
        // 采用循环方式来不断改变进度条的完成进度
        for (int i = 0 ; i <= 100 ; i++)
        {
            // 改变进度条的完成进度
            bar.setValue(i);
            try
            {
                // 程序暂停 0.1 秒
                Thread.sleep(100);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args)
```

```
    {
        new JProgressBarTest().init();
    }
}
```

上面程序中的粗体字代码创建了一个垂直进度条，并通过方法来设置进度条的外观形式——是否包含边框，是否在进度条中显示完成百分比，并通过一个循环来不断改变进度条的value属性，该value将会自动转换成进度条的完成百分比。

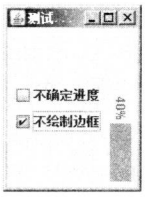


图12.29　使用进度条

运行该程序，将看到如图12.29所示的效果。

在上面程序中，在主程序中使用循环来改变进度条的value属性，即修改进度条的完成百分比，这是没有任何意义的事情。通常会希望用进度条去检测其他任务的完成情况，而不是在其他任务的执行过程中主动修改进度条的value属性，因为其他任务可能根本不知道进度条的存在。此时可以使用一个计时器来不断取得目标任务的完成情况，并根据其完成情况来修改进度条的value属性。下面程序改写了上面程序，使用一个SimulatedTarget来模拟一个耗时的任务。

程序清单：codes\12\12.6\JProgressBarTest2.java

```
public class JProgressBarTest2
{
    JFrame frame = new JFrame("测试进度条");
    // 创建一条垂直进度条
    JProgressBar bar = new JProgressBar(JProgressBar.VERTICAL);
    JCheckBox indeterminate = new JCheckBox("不确定进度");
    JCheckBox noBorder = new JCheckBox("不绘制边框");
    public void init()
    {
        Box box = new Box(BoxLayout.Y_AXIS);
        box.add(indeterminate);
        box.add(noBorder);
        frame.setLayout(new FlowLayout());
        frame.add(box);
        // 把进度条添加到 JFrame 窗口中
        frame.add(bar);
        // 设置在进度条中绘制完成百分比
        bar.setStringPainted(true);
        // 根据该选择框决定是否绘制进度条的边框
        noBorder.addActionListener(event ->
            bar.setBorderPainted(!noBorder.isSelected()));
        final SimulatedActivity target = new SimulatedActivity(1000);
        // 以启动一条线程的方式来执行一个耗时的任务
        new Thread(target).start();
        // 设置进度条的最大值和最小值
        bar.setMinimum(0);
        // 以总任务量作为进度条的最大值
        bar.setMaximum(target.getAmount());
        Timer timer = new Timer(300 , e -> bar.setValue(target.getCurrent()));
        timer.start();
        indeterminate.addActionListener(event -> {
            // 设置该进度条的进度是否确定
            bar.setIndeterminate(indeterminate.isSelected());
            bar.setStringPainted(!indeterminate.isSelected());
        });
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JProgressBarTest2().init();
    }
}
// 模拟一个耗时的任务
class SimulatedActivity implements Runnable
{
    // 任务的当前完成量
```

```
    private volatile int current;
    // 总任务量
    private int amount;
    public SimulatedActivity(int amount)
    {
        current = 0;
        this.amount = amount;
    }
    public int getAmount()
    {
        return amount;
    }
    public int getCurrent()
    {
        return current;
    }
    // run 方法代表不断完成任务的过程
    public void run()
    {
        while (current < amount)
        {
            try
            {
                Thread.sleep(50);
            }
            catch(InterruptedException e)
            {
            }
            current++;
        }
    }
}
```

上面程序的运行效果与前一个程序的运行效果大致相同，但这个程序中的JProgressBar就实用多了，它可以检测并显示 SimulatedTarget 的完成进度。

提示：

SimulatedActivity 类实现了 Runnable 接口，这是一个特殊的接口，实现该接口可以实现多线程功能。关于多线程的介绍请参考本书第 16 章内容。

Swing 组件大都将外观显示和内部数据分离，JProgressBar 也不例外，JProgressBar 组件有一个用于保存其状态数据的 Model 对象，这个对象由 BoundedRangeModel 对象表示，程序调用 JProgressBar 对象的 setValue()方法时，实际上是设置 BoundedRangeModel 对象的 value 属性。

程序可以修改 BoundedRangeModel 对象的 minimum 属性和 maximum 属性，当该 Model 对象的这两个属性被修改后，它所对应的 JProgressBar 对象的这两个属性也会随之修改，因为 JProgressBar 对象的所有状态数据都是保存在该 Model 对象中的。

程序监听 JProgressBar 完成比例的变化，也是通过为 BoundedRangeModel 提供监听器来实现的。BoundedRangeModel 提供了如下一个方法来添加监听器。

- addChangeListener(ChangeListener x)：用于监听 JProgressBar 完成比例的变化，每当 JProgressBar 的 value 属性被改变时，系统都会触发 ChangeListener 监听器的 stateChanged()方法。例如，下面代码为进度条的状态变化添加了一个监听器。

```
// JProgressBar 的完成比例发生变化时会触发该方法
bar.getModel().addChangeListener(ce -> {
    // 对进度变化进行合适处理
    ...
});
```

12.6.2 创建进度对话框

ProgressMonitor 的用法与 JProgressBar 的用法基本相似，只是 ProgressMonitor 可以直接创建一个进度对话框。ProgressMonitor 提供了如下构造器。

- ProgressMonitor(Component parentComponent, Object message, String note, int min, int max)：该构造器中的 parentComponent 参数用于设置该进度对话框的父组件，message 用于设置该进度对话框的描述信息，note 用于设置该进度对话框的提示文本，min 和 max 用于设置该对话框所包含进度条的最小值和最大值。

例如，如下代码创建了一个进度对话框。

```
final ProgressMonitor dialog = new ProgressMonitor(null ,"等待任务完成" ,
    "已完成：" , 0 , target.getAmount());
```

使用上面代码创建的进度对话框如图 12.30 所示。

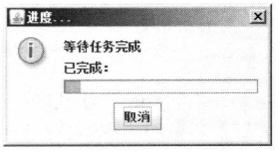


图 12.30 进度对话框

如图 12.30 所示，该对话框中包含了一个“取销”按钮，如果程序希望判断用户是否单击了该按钮，则可以通过 ProgressMonitor 的 isCanceled()方法进行判断。

使用 ProgressMonitor 创建的对话框里包含的进度条是非常固定的，程序甚至不能设置该进度条是否包含边框（总是包含边框），不能设置进度不确定，不能改变进度条的方向（总是水平方向）。

与普通进度条类似的是，进度对话框也不能自动监视目标任务的完成进度，程序通过调用进度对话框的 setProgress()方法来改变进度条的完成比例（该方法类似于 JProgressBar 的 setValue()方法）。

下面程序同样采用前面的 SimulatedTarget 来模拟一个耗时的任务，并创建了一个进度对话框来监测该任务的完成百分比。

程序清单：codes\12\12.6\ProgressMonitorTest.java

```
public class ProgressMonitorTest
{
    Timer timer;
    public void init()
    {
        final SimulatedActivity target = new SimulatedActivity(1000);
        // 以启动一条线程的方式来执行一个耗时的任务
        final Thread targetThread = new Thread(target);
        targetThread.start();
        final ProgressMonitor dialog = new ProgressMonitor(null
            , "等待任务完成" , "已完成：" , 0 , target.getAmount());
        timer = new Timer(300 , e -> {
            // 以任务的当前完成量设置进度对话框的完成比例
            dialog.setProgress(target.getCurrent());
            // 如果用户单击了进度对话框中的“取消”按钮
            if (dialog.isCanceled())
            {
                // 停止计时器
                timer.stop();
                // 中断任务的执行线程
                targetThread.interrupt();    // ①
                // 系统退出
                System.exit(0);
            }
        });
        timer.start();
    }
    public static void main(String[] args)
    {
        new ProgressMonitorTest().init();
    }
}
```

上面程序中的粗体字代码创建了一个进度对话框，并创建了一个 Timer 计时器不断询问 SimulatedTarget 任务的完成比例，进而设置进度对话框里进度条的完成比例。而且该计时器还负责监听用户是否单击了进度对话框中的“取销”按钮，如果用户单击了该按钮，则中止执行 SimulatedTarget 任务的线程，并停止计时器，同时退出该程序。运行该程序，会看到如图 12.30 所示的对话框。

提示：
程序中①处代码用于中止线程的执行，读者可以参考第 16 章内容来理解这行代码。

12.7 使用 JSlider 和 BoundedRangeModel 创建滑动条

JSlider 的用法和 JProgressBar 的用法非常相似，这一点可以从它们共享同一个 Model 类看出来。使用 JSlider 可以创建一个滑动条，这个滑动条同样有最小值、最大值和当前值等属性。JSlider 与 JprogressBar 的主要区别如下。

- JSlider 不是采用填充颜色的方式来表示该组件的当前值，而是采用滑块的位置来表示该组件的当前值。
- JSlider 允许用户手动改变滑动条的当前值。
- JSlider 允许为滑动条指定刻度值，这系列的刻度值既可以是连续的数字，也可以是自定义的刻度值，甚至可以是图标。

使用 JSlider 创建滑动条的步骤如下。

① 使用 JSlider 的构造器创建一个 JSlider 对象，JSlider 有多个重载的构造器，但这些构造器总共可以接收如下 4 个参数。

- orientation：指定该滑动条的摆放方向，默认是水平摆放。可以接收 JSlider.VERTICAL 和 JSlider.HORIZONTAL 两个值。
- min：指定该滑动条的最小值，该属性值默认为 0。
- max：指定该滑动条的最大值，该属性值默认是为 100。
- value：指定该滑动条的当前值，该属性值默认是为 50。

中華工匠

② 调用 JSlider 的如下方法来设置滑动条的外观样式。

- setExtent(int extent)：设置滑动条上的保留区，用户拖动滑块时不能超过保留区。例如，最大值为 100 的滑动条，如果设置保留区为 20，则滑块最大只能拖动到 80。
- setInverted(boolean b)：设置是否需要反转滑动条，滑动条的滑轨上刻度值默认从小到大、从左到右排列。如果该方法设置为 true，则排列方向会反转过来。
- setLabelTable(Dictionary labels)：为该滑动条指定刻度标签。该方法的参数是 Dictionary 类型，它是一个古老的、抽象集合类，其子类是 Hashtable。传入的 Hashtable 集合对象的 key-value 对为{ Integer value, java.swing.JComponent label }格式，刻度标签可以是任何组件。
- setMajorTickSpacing(int n)：设置主刻度标记的间隔。
- setMinorTickSpacing(int n)：设置次刻度标记的间隔。
- setPaintLabels(boolean b)：设置是否在滑块上绘制刻度标签。如果没有为该滑动条指定刻度标签，则默认绘制将刻度值的数值作为标签。
- setPaintTicks(boolean b)： 设置是否在滑块上绘制刻度标记。
- setPaintTrack(boolean b)：设置是否为滑块绘制滑轨。
- setSnapToTicks(boolean b)：设置滑块是否必须停在滑道的有刻度处。如果设置为 true，则滑块只能停在有刻度处；如果用户没有将滑块拖到有刻度处，则系统自动将滑块定位到最近的刻度处。

③ 如果程序需要在用户拖动滑块时做出相应处理，则应为该 JSlider 对象添加事件监听器。JSlider 提供了 addChangeListener()方法来添加事件监听器，该监听器负责监听滑动值的变化。

④ 将 JSlider 对象添加到其他容器中显示出来。

下面程序示范了如何使用 JSlider 来创建滑动条。

程序清单：codes\12\12.7\JSliderTest.java

```
public class JSliderTest
```

```
{
    JFrame mainWin = new JFrame("滑动条示范");
    Box sliderBox = new Box(BoxLayout.Y_AXIS);
    JTextField showVal = new JTextField();
    ChangeListener listener;
    public void init()
    {
        // 定义一个监听器，用于监听所有的滑动条
        listener = event -> {
            // 取出滑动条的值，并在文本中显示出来
            JSlider source = (JSlider) event.getSource();
            showVal.setText("当前滑动条的值为："
                + source.getValue());
        };
        // -----------添加一个普通滑动条-----------
        JSlider slider = new JSlider();
        addSlider(slider, "普通滑动条");
        // -----------添加保留区为 30 的滑动条-----------
        slider = new JSlider();
        slider.setExtent(30);
        addSlider(slider, "保留区为 30");
        // ---添加带主、次刻度的滑动条，并设置其最大值、最小值---
        slider = new JSlider(30 , 200);
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        addSlider(slider, "有刻度");
        // -----------添加滑块必须停在刻度处的滑动条-----------
        slider = new JSlider();
        // 设置滑块必须停在刻度处
        slider.setSnapToTicks(true);
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        addSlider(slider, "滑块停在刻度处");
        // -----------添加没有滑轨的滑动条-----------
        slider = new JSlider();
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        // 设置不绘制滑轨
        slider.setPaintTrack(false);
        addSlider(slider, "无滑轨");
        // -----------添加方向反转的滑动条-----------
        slider = new JSlider();
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        // 设置方向反转
        slider.setInverted(true);
        addSlider(slider, "方向反转");
        // --------添加绘制默认刻度标签的滑动条--------
        slider = new JSlider();
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        // 设置绘制刻度标签，默认绘制数值刻度标签
        slider.setPaintLabels(true);
        addSlider(slider, "数值刻度标签");
        // ------添加绘制 Label 类型的刻度标签的滑动条------
```

```
        slider = new JSlider();
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        // 设置绘制刻度标签
        slider.setPaintLabels(true);
        Dictionary<Integer, Component> labelTable = new Hashtable<>();
        labelTable.put(0, new JLabel("A"));
        labelTable.put(20, new JLabel("B"));
        labelTable.put(40, new JLabel("C"));
        labelTable.put(60, new JLabel("D"));
        labelTable.put(80, new JLabel("E"));
        labelTable.put(100, new JLabel("F"));
        // 指定刻度标签，标签是 JLabel
        slider.setLabelTable(labelTable);
        addSlider(slider, "JLable 标签");
        // ------添加绘制 Label 类型的刻度标签的滑动条------
        slider = new JSlider();
        // 设置绘制刻度
        slider.setPaintTicks(true);
        // 设置主、次刻度的间距
        slider.setMajorTickSpacing(20);
        slider.setMinorTickSpacing(5);
        // 设置绘制刻度标签
        slider.setPaintLabels(true);
        labelTable = new Hashtable<Integer, Component>();
        labelTable.put(0, new JLabel(new ImageIcon("ico/0.GIF")));
        labelTable.put(20, new JLabel(new ImageIcon("ico/2.GIF")));
        labelTable.put(40, new JLabel(new ImageIcon("ico/4.GIF")));
        labelTable.put(60, new JLabel(new ImageIcon("ico/6.GIF")));
        labelTable.put(80, new JLabel(new ImageIcon("ico/8.GIF")));
        // 指定刻度标签，标签是 ImageIcon
        slider.setLabelTable(labelTable);
        addSlider(slider, "Icon 标签");
        mainWin.add(sliderBox, BorderLayout.CENTER);
        mainWin.add(showVal, BorderLayout.SOUTH);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    // 定义一个方法，用于将滑动条添加到容器中
    public void addSlider(JSlider slider, String description)
    {
        slider.addChangeListener(listener);
        Box box = new Box(BoxLayout.X_AXIS);
        box.add(new JLabel(description + ": "));
        box.add(slider);
        sliderBox.add(box);
    }
    public static void main(String[] args)
    {
        new JSliderTest().init();
    }
}
```

上面程序向窗口中添加了多个滑动条，程序通过粗体字代码来控制不同滑动条的不同外观。运行上面程序，会看到如图 12.31 所示的各种滑动条的效果。

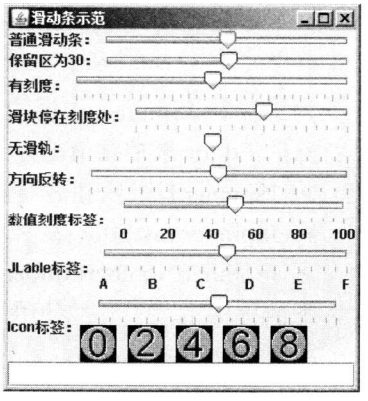


图 12.31 各种滑动条的效果

JSlider 也使用 BoundedRangeModel 作为保存其状态数据的 Model 对象，程序可以直接修改 Model 对象来改变滑动条的状态，但大部分时候程序无须使用该 Model 对象。JSlider 也提供了 addChangeListener()方法来为滑动条添加监听器，无须像 JProgressBar 那样监听它所对应的 Model 对象。

12.8　使用 JSpinner 和 SpinnerModel 创建微调控制器

JSpinner 组件是一个带有两个小箭头的文本框，这个文本框只能接收满足要求的数据，用户既可以通过两个小箭头调整该微调控制器的值，也可以直接在文本框内输入内容作为该微调控制器的值。当用户在该文本框内输入时，如果输入的内容不满足要求，系统将会拒绝用户输入。典型的 JSpinner 组件如图 12.32 所示。

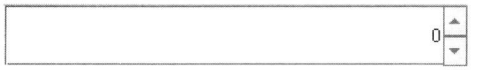


图 12.32　微调控制器组件

JSpinner 组件常常需要和 SpinnerModel 结合使用，其中 JSpinner 组件控制该组件的外观表现，而 SpinnerModel 则控制该组件内部的状态数据。

JSpinner 组件的值可以是数值、日期和 List 中的值，Swing 为这三种类型的值提供了 SpinnerNumberModel、SpinnerDateModel 和 SpinnerListModel 三个 SpinnerModel 实现类；除此之外，JSpinner 组件的值还可以是任意序列，只要这个序列可以通过 previous()、next()获取值即可。在这种情况下，用户必须自行提供 SpinnerModel 实现类。

使用 JSpinner 组件非常简单，JSpinner 提供了如下两个构造器。

- JSpinner()：创建一个默认的微调控制器。
- JSpinner(SpinnerModel model)：使用指定的 SpinnerModel 来创建微调控制器。

采用第一个构造器创建的默认微调控制器只接收整数值，初始值是 0，最大值和最小值没有任何限制。每单击向下箭头或者向上箭头一次，该组件里的值分别减 1 或加 1。

使用 JSpinner 关键在于使用它对应的三个 SpinnerModel，下面依次介绍这三个 SpinnerModel。

- SpinnerNumberModel：这是最简单的 SpinnerModel，创建该 SpinnerModel 时可以指定 4 个参数：最大值、最小值、初始值、步长，其中步长控制单击上、下箭头时相邻两个值之间的差。这 4 个参数既可以是整数，也可以是浮点数。
- SpinnerDateModel：创建该 SpinnerModel 时可以指定 4 个参数：起始时间、结束时间、初始时间和时间差，其中时间差控制单击上、下箭头时相邻两个时间之间的差值。
- SpinnerListModel：创建该 SpinnerModel 只需要传入一个 List 或者一个数组作为序列值即可。该 List 的集合元素和数组元素可以是任意类型的对象，但由于 JSpinner 组件的文本框只能显示字符串，所以 JSpinner 显示每个对象 toString()方法的返回值。

提示：

从图 12.32 中可以看出，JSpinner 创建的微调控制器和 ComboBox 有点像（由 Swing 的 JComboBox 提供，ComboBox 既允许通过下拉列表框进行选择，也允许直接输入），区别在于 ComboBox 可以产生一个下拉列表框供用户选择，而 JSpinner 组件只能通过上、下箭头逐项选择。使用 ComboBox 通常必须明确指定下拉列表框中每一项的值，但使用 JSpinner 则只需给定一个范围，并指定步长即可；当然，使用 JSpinner 也可以明确给出每一项的值（就是对应使用 SpinnerListModel）。

为了控制 JSpinner 中值的显示格式，JSpinner 还提供了一个 setEditor()方法。Swing 提供了如下 3 个特殊的 Editor 来控制值的显示格式。

- JSpinner.DateEditor：控制 JSpinner 中日期值的显示格式。
- JSpinner.ListEditor：控制 JSpinner 中 List 项的显示格式。
- JSpinner.NumberEditor：控制 JSpinner 中数值的显示格式。

下面程序示范了几种使用 JSpinner 的情形。

程序清单：codes\12\12.8\JSpinnerTest.java

```
public class JSpinnerTest
{
```

```
final int SPINNER_NUM = 6;
JFrame mainWin = new JFrame("微调控制器示范");
Box spinnerBox = new Box(BoxLayout.Y_AXIS);
JSpinner[] spinners = new JSpinner[SPINNER_NUM];
JLabel[] valLabels = new JLabel[SPINNER_NUM];
JButton okBn = new JButton("确定");
public void init()
{
    for (int i = 0 ; i < SPINNER_NUM ; i++ )
    {
        valLabels[i] = new JLabel();
    }
    // -----------普通 JSpinner-----------
    spinners[0] = new JSpinner();
    addSpinner(spinners[0], "普通" , valLabels[0]);
    // -----------指定最小值、最大值、步长的 JSpinner-----------
    // 创建一个 SpinnerNumberModel 对象，指定最小值、最大值和步长
    SpinnerNumberModel numModel = new SpinnerNumberModel(
        3.4 , -1.1 , 4.3 , 0.1);
    spinners[1] = new JSpinner(numModel);
    addSpinner(spinners[1], "数值范围" , valLabels[1]);
    // -----------使用 SpinnerListModel 的 JSpinner------------
    String[] books = new String[]
    {
        "轻量级 Java EE 企业应用实战"
        , "疯狂 Java 讲义"
        , "疯狂 Ajax 讲义"
    };
    // 使用字符串数组创建 SpinnerListModel 对象
    SpinnerListModel bookModel = new SpinnerListModel(books);
    // 使用 SpinnerListModel 对象创建 JSpinner 对象
    spinners[2] = new JSpinner(bookModel);
    addSpinner(spinners[2], "字符串序列值" , valLabels[2]);
    // -----------使用序列值是 ImageIcon 的 JSpinner------------
    ArrayList<ImageIcon> icons = new ArrayList<>();
    icons.add(new ImageIcon("a.gif"));
    icons.add(new ImageIcon("b.gif"));
    // 使用 ImageIcon 数组创建 SpinnerListModel 对象
    SpinnerListModel iconModel = new SpinnerListModel(icons);
    // 使用 SpinnerListModel 对象创建 JSpinner 对象
    spinners[3] = new JSpinner(iconModel);
    addSpinner(spinners[3], "图标序列值" , valLabels[3]);
    // -----------使用 SpinnerDateModel 的 JSpinner------------
    // 分别获取起始时间、结束时间、初时时间
    Calendar cal = Calendar.getInstance();
    Date init = cal.getTime();
    cal.add(Calendar.DAY_OF_MONTH , -3);
    Date start = cal.getTime();
    cal.add(Calendar.DAY_OF_MONTH , 8);
    Date end = cal.getTime();
    // 创建一个 SpinnerDateModel 对象，指定最小时间、最大时间和初始时间
    SpinnerDateModel dateModel = new SpinnerDateModel(init
        , start , end , Calendar.HOUR_OF_DAY);
    // 以 SpinnerDateModel 对象创建 JSpinner
    spinners[4] = new JSpinner(dateModel);
    addSpinner(spinners[4], "时间范围" , valLabels[4]);
    // -----------使用 DateEditor 来格式化 JSpinner------------
    dateModel = new SpinnerDateModel();
    spinners[5] = new JSpinner(dateModel);
    // 创建一个 JSpinner.DateEditor 对象，用于对指定的 Spinner 进行格式化
    JSpinner.DateEditor editor = new JSpinner.DateEditor(
        spinners[5] , "公元 yyyy 年 MM 月 dd 日 HH 时");
    // 设置使用 JSpinner.DateEditor 对象进行格式化
    spinners[5].setEditor(editor);
    addSpinner(spinners[5], "使用 DateEditor" , valLabels[5]);
    // 为“确定”按钮添加一个事件监听器
    okBn.addActionListener(evt -> {
        // 取出每个微调控制器的值，并将该值用后面的 Label 标签显示出来
        for (int i = 0 ; i < SPINNER_NUM ; i++)
        {
```

```
                // 将微调控制器的值通过指定的 JLabel 显示出来
                valLabels[i].setText(spinners[i].getValue().toString());
            }
        });
        JPanel bnPanel = new JPanel();
        bnPanel.add(okBn);
        mainWin.add(spinnerBox, BorderLayout.CENTER);
        mainWin.add(bnPanel, BorderLayout.SOUTH);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    // 定义一个方法，用于将滑动条添加到容器中
    public void addSpinner(JSpinner spinner
        , String description , JLabel valLabel)
    {
        Box box = new Box(BoxLayout.X_AXIS);
        JLabel desc = new JLabel(description + ": ");
        desc.setPreferredSize(new Dimension(100 , 30));
        box.add(desc);
        box.add(spinner);
        valLabel.setPreferredSize(new Dimension(180 , 30));
        box.add(valLabel);
        spinnerBox.add(box);
    }
    public static void main(String[] args)
    {
        new JSpinnerTest().init();
    }
}
```

上面程序创建了 6 个 JSpinner 对象，并将它们添加到窗口中显示出来，程序中的粗体字代码用于控制每个微调控制器的具体行为。

第一个 JSpinner 组件是一个默认的微调控制器，其初始值是 0，步长是 1，只能接收整数值。

第二个 JSpinner 通过 SpinnerNumberModel 来创建，指定了 JSpinner 的最小值为–1.1、最大值为 4.3、初始值为 3.4、步长为 0.1，所以用户单击该微调控制器的上、下箭头时，微调控制器的值之间的差值是 0.1，并只能处于–1.1~4.3 之间。

第三个 JSpinner 通过 SpinnerListModel 来创建的，创建 SpinnerListModel 对象时指定字符串数组作为多个序列值，所以当用户单击该微调控制器的上、下箭头时，微调控制器的值总是在该字符串数组之间选择。

第四个 JSpinner 也是通过 SpinnerListModel 来创建的，虽然传给 SpinnerListModel 对象的构造参数是集合元素为 ImageIcon 的 List 对象，但 JSpinner 只能显示字符串内容，所以它会把每个 ImageIcon 对象的 toString()方法返回值当成微调控制器的多个序列值。

第五个 JSpinner 通过 SpinnerDateModel 来创建，而且指定了最小时间、最大时间和初始时间，所以用户单击该微调控制器的上、下箭头时，微调控制器里的时间只能处于指定时间范围之间。这里需要注意的是，SpinnerDateModel 的第 4 个参数没有太大的作用，它不能控制两个相邻时间之间的差。当用户在 JSpinner 组件内选中该时间的指定时间域时，例如年份，则两个相邻时间的时间差就是 1 年。

第六个 JSpinner 使用 JSpinner.DateEditor 来控制时间微调控制器里日期、时间的显示格式，创建 JSpinner.DateEditor 对象时需要传入一个日期时间格式字符串（dateFormatPattern），该参数用于控制日期、时间的显示格式，关于这个格式字符串的定义方式可以参考 SimpleDateFormat 类的介绍。本例程序中使用"公元 yyyy 年 MM 月 dd 日 HH 时"作为格式字符串。

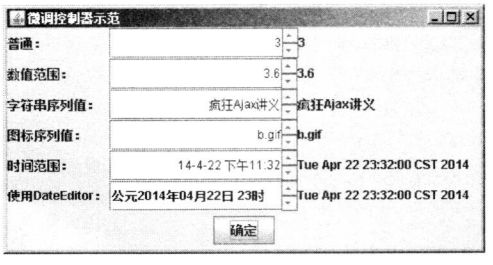


图 12.33　JSpinner 组件的用法示范

运行上面程序，会看到如图 12.33 所示的窗口。

程序中还提供了一个“确定”按钮，当单击该按钮时，

系统会把每个微调控制器的值通过对应的JLabel标签显示出来，如图12.33所示。

12.9 使用JList、JComboBox创建列表框

无论从哪个角度来看，JList和JComboBox都是极其相似的，它们都有一个列表框，只是JComboBox的列表框需要以下拉方式显示出来；JList和JComboBox都可以通过调用setRenderer()方法来改变列表项的表现形式。甚至维护这两个组件的Model都是相似的，JList使用ListModel，JComboBox使用ComboBoxModel，而ComboBoxModel是ListModel的子类。

12.9.1 简单列表框

如果仅仅希望创建一个简单的列表框（包括JList和JComboBox），则直接使用它们的构造器即可，它们的构造器都可接收一个对象数组或元素类型任意的Vector作为参数，这个对象数组或元素类型任意的Vector里的所有元素将转换为列表框的列表项。

使用JList和JComboBox来创建简单列表框非常简单，只需要按如下步骤进行即可。

① 使用JList或者JComboBox的构造器创建一个列表框对象，创建JList或JComboBox时，应该传入一个Vector对象或者Object[]数组作为构造器参数，其中使用JComboBox创建的列表框必须单击右边的向下箭头才会出现。

② 调用JList或JComboBox的各种方法来设置列表框的外观行为，其中JList可以调用如下几个常用的方法。

- addSelectionInterval(int anchor, int lead)：在已经选中列表项的基础上增加选中从anchor到lead索引范围内的所有列表项。
- setFixedCellHeight、setFixedCellWidth：设置每个列表项具有指定的高度和宽度。
- setLayoutOrientation(int layoutOrientation)：设置列表框的布局方向，该属性可以接收三个值，即JList.HORIZONTAL_WRAP、JList.VERTICAL_WRAP和JList.VERTICAL（默认），用于指定当列表框长度不足以显示所有的列表项时，列表框如何排列所有的列表项。
- setSelectedIndex(int index)：设置默认选择哪一个列表项。
- setSelectedIndices(int[] indices)：设置默认选择哪一批列表项（多个）。
- setSelectedValue(Object anObject, boolean shouldScroll)：设置选中哪个列表项的值，第二个参数决定是否滚动到选中项。
- setSelectionBackground(Color selectionBackground)：设置选中项的背景色。
- setSelectionForeground(Color selectionForeground)：设置选中项的前景色。
- setSelectionInterval(int anchor, int lead)：设置选中从anchor到lead索引范围内的所有列表项。
- setSelectionMode(int selectionMode)：设置选中模式。支持如下3个值。
- ListSelectionModel.SINGLE_SELECTION：每次只能选择一个列表项。在这种模式中，setSelectionInterval和addSelectionInterval是等效的。
- ListSelectionModel.SINGLE_INTERVAL_SELECTION：每次只能选择一个连续区域。在此模式中，如果需要添加的区域没有与已选择区域相邻或重叠，则不能添加该区域。简而言之，在这种模式下每次可以选择多个列表项，但多个列表项必须处于连续状态。
- ListSelectionModel.MULTIPLE_INTERVAL_SELECTION：在此模式中，选择没有任何限制。该模式是默认设置。
- setVisibleRowCount(int visibleRowCount)：设置该列表框的可视高度足以显示多少项。

JComboBox则提供了如下几个常用方法。

- setEditable(boolean aFlag)：设置是否允许直接修改JComboBox文本框的值，默认不允许。
- setMaximumRowCount(int count)：设置下拉列表框的可视高度可以显示多少个列表项。

- setSelectedIndex(int anIndex)：根据索引设置默认选中哪一个列表项。
- setSelectedItem(Object anObject)：根据列表项的值设置默认选中哪一个列表项。

提示：

JComboBox 没有设置选择模式的方法，因为 JComboBox 最多只能选中一项，所以没有必要设置选择模式。

③ 如果需要监听列表框选择项的变化，则可以通过添加对应的监听器来实现。通常 JList 使用 addListSelectionListener()方法添加监听器，而 JComboBox 采用 addItemListener()方法添加监听器。

下面程序示范了 JList 和 JCombox 的用法，并允许用户通过单选按钮来控制 JList 的选项布局、选择模式，在用户选择图书之后，这些图书会在窗口下面的文本域里显示出来。

程序清单：codes\12\12.9\ListTest.java

```
public class ListTest
{
    private JFrame mainWin = new JFrame("测试列表框");
    String[] books = new String[]
    {
        "疯狂 Java 讲义"
        , "轻量级 Java EE 企业应用实战"
        , "疯狂 Android 讲义"
        , "疯狂 Ajax 讲义"
        , "经典 Java EE 企业应用实战"
    };
    // 用一个字符串数组来创建一个 JList 对象
    JList<String> bookList = new JList<>(books);
    JComboBox<String> bookSelector;
    // 定义布局选择按钮所在的面板
    JPanel layoutPanel = new JPanel();
    ButtonGroup layoutGroup = new ButtonGroup();
    // 定义选择模式按钮所在的面板
    JPanel selectModePanel = new JPanel();
    ButtonGroup selectModeGroup = new ButtonGroup();
    JTextArea favoriate = new JTextArea(4 , 40);
    public void init()
    {
        // 设置 JList 的可视高度可同时显示 3 个列表项
        bookList.setVisibleRowCount(3);
        // 默认选中第 3 项到第 5 项（第 1 项的索引是 0）
        bookList.setSelectionInterval(2, 4);
        addLayoutButton("纵向滚动", JList.VERTICAL);
        addLayoutButton("纵向换行", JList.VERTICAL_WRAP);
        addLayoutButton("横向换行", JList.HORIZONTAL_WRAP);
        addSelectModelButton("无限制", ListSelectionModel
            .MULTIPLE_INTERVAL_SELECTION);
        addSelectModelButton("单选", ListSelectionModel
            .SINGLE_SELECTION);
        addSelectModelButton("单范围", ListSelectionModel
            .SINGLE_INTERVAL_SELECTION);
        Box listBox = new Box(BoxLayout.Y_AXIS);
        // 将 JList 组件放在 JScrollPane 中，再将该 JScrollPane 添加到 listBox 容器中
        listBox.add(new JScrollPane(bookList));
        // 添加布局选择按钮面板、选择模式按钮面板
        listBox.add(layoutPanel);
        listBox.add(selectModePanel);
        // 为 JList 添加事件监听器
        bookList.addListSelectionListener(e -> {  // ①
            // 获取用户所选择的所有图书
            List<String> books = bookList.getSelectedValuesList();
            favoriate.setText("");
            for (String book : books )
            {
                favoriate.append(book + "\n");
            }
```

```
        });
        Vector<String> bookCollection = new Vector<>();
        bookCollection.add("疯狂 Java 讲义");
        bookCollection.add("轻量级 Java EE 企业应用实战");
        bookCollection.add("疯狂 Android 讲义");
        bookCollection.add("疯狂 Ajax 讲义");
        bookCollection.add("经典 Java EE 企业应用实战");
        // 用一个 Vector 对象来创建一个 JComboBox 对象
        bookSelector = new JComboBox<>(bookCollection);
        // 为 JComboBox 添加事件监听器
        bookSelector.addItemListener(e -> {  // ②
            // 获取 JComboBox 所选中的项
            Object book = bookSelector.getSelectedItem();
            favoriate.setText(book.toString());
        });
        // 设置可以直接编辑
        bookSelector.setEditable(true);
        // 设置下拉列表框的可视高度可同时显示 4 个列表项
        bookSelector.setMaximumRowCount(4);
        JPanel p = new JPanel();
        p.add(bookSelector);
        Box box = new Box(BoxLayout.X_AXIS);
        box.add(listBox);
        box.add(p);
        mainWin.add(box);
        JPanel favoriatePanel = new JPanel();
        favoriatePanel.setLayout(new BorderLayout());
        favoriatePanel.add(new JScrollPane(favoriate));
        favoriatePanel.add(new JLabel("您喜欢的图书：")
            , BorderLayout.NORTH);
        mainWin.add(favoriatePanel , BorderLayout.SOUTH);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    private void addLayoutButton(String label, final int orientation)
    {
        layoutPanel.setBorder(new TitledBorder(new EtchedBorder()
            , "确定选项布局"));
        JRadioButton button = new JRadioButton(label);
        // 把该单选按钮添加到 layoutPanel 面板中
        layoutPanel.add(button);
        // 默认选中第一个按钮
        if(layoutGroup.getButtonCount() == 0)
            button.setSelected(true);
        layoutGroup.add(button);
        button.addActionListener(event ->
            // 改变列表框里列表项的布局方向
            bookList.setLayoutOrientation(orientation));
    }
    private void addSelectModelButton(String label, final int selectModel)
    {
        selectModePanel.setBorder(new TitledBorder(new EtchedBorder()
            , "确定选择模式"));
        JRadioButton button = new JRadioButton(label);
        // 把该单选按钮添加到 selectModePanel 面板中
        selectModePanel.add(button);
        // 默认选中第一个按钮
        if (selectModeGroup.getButtonCount() == 0)
        button.setSelected(true);
        selectModeGroup.add(button);
        button.addActionListener(event ->
            // 改变列表框里的选择模式
            bookList.setSelectionMode(selectModel));
    }
    public static void main(String[] args)
    {
        new ListTest().init();
    }
}
```

图 12.34　JList 和 JComboBox 的用法示范

上面程序中的粗体字代码实现了使用字符串数组创建一个 JList 对象，并通过调用一些方法来改变该 JList 的表现外观；使用 Vector 创建一个 JComboBox 对象，并通过调用一些方法来改变该 JComboBox 的表现外观。

程序中①②号粗体字代码为 JList 对象和 JComboBox 对象添加事件监听器，当用户改变两个列表框里的选择时，程序会把用户选择的图书显示在下面的文本域内。运行上面程序，会看到如图 12.34 所示的效果。

从图 12.34 中可以看出，因为 JComboBox 设置了 setEditable(true)，所以可以直接在该组件中输入用户自己喜欢的图书，当输入结束后，输入的图书名会直接显示在窗口下面的文本域内。

注意：

JList 默认没有滚动条，必须将其放在 JScrollPane 中才有滚动条，通常总是将 JList 放在 JScrollPane 中使用，所以程序中先将 JList 放到 JScrollPane 容器中，再将该 JScrollPane 添加到窗口中。要在 JList 中选中多个选项，可以使用 Ctrl 或 Shift 辅助键，按住 Ctrl 键才可以在原来选中的列表项基础上添加选中新的列表项；按 Shift 键可以选中连续区域的所有列表项。

12.9.2　不强制存储列表项的 ListModel 和 ComboBoxModel

正如前面提到的，Swing 的绝大部分组件都采用了 MVC 的设计模式，其中 JList 和 JComboBox 都只负责组件的外观显示，而组件底层的状态数据维护则由对应的 Model 负责。JList 对应的 Model 是 ListModel 接口，JComboBox 对应的 Model 是 ComboBoxModel 接口，这两个接口负责维护 JList 和 JComboBox 组件里的列表项。其中 ListModel 接口的代码如下：

```
public interface ListModel<E>
{
    // 返回列表项的数量
    int getSize();
    // 返回指定索引处的列表项
    E getElementAt(int index);
    // 为列表项添加一个监听器，当列表项发生变化时将触发该监听器
    void addListDataListener(ListDataListener l);
    // 删除列表项上的指定监听器
    void removeListDataListener(ListDataListener l);
}
```

从上面接口来看，这个 ListModel 不管 JList 里的所有列表项的存储形式，它甚至不强制存储所有的列表项，只要 ListModel 的实现类提供了 getSize()和 getElementAt()两个方法，JList 就可以根据该 ListModel 对象来生成列表框。

ComboBoxModel 继承了 ListModel，它添加了“选择项”的概念，选择项代表 JComboBox 显示区域内可见的列表项。ComboBoxModel 为“选择项”提供了两个方法，下面是 ComboBoxModel 接口的代码。

```
public interface ComboBoxModel<E> extends ListModel<E>
{
    // 设置选中“选择项”
    void setSelectedItem(Object anItem);
    // 获取“选择项”的值
    Object getSelectedItem();
}
```

因为 ListModel 不强制保存所有的列表项，因此可以为它创建一个实现类：NumberListModel，这个实现类只需要传入数字上限、数字下限和步长，程序就可以自动为之实现上面的 getSize()方法和

getElementAt()方法，从而允许直接使用一个数字范围来创建 JList 对象。

实现 getSize()方法的代码如下：

```
public int getSize()
{
    return (int)Math.floor(end.subtract(start)
        .divide(step).doubleValue()) + 1;
}
```

用“(上限–下限) ÷步长+1”即得到该 ListModel 中包含的列表项的个数。

注意：

程序使用 BigDecimal 变量来保存上限、下限和步长，而不是直接使用 double 变量来保存这三个属性，主要是为了实现对数值的精确计算，所以上面程序中的 end、start 和 step 都是 BigDecimal 类型的变量。

实现 getElementAt()方法也很简单，“下限+步长×索引”就是指定索引处的元素，该方法的具体实现请参考 ListModelTest.java。

下面程序为 ListModel 提供了 NumberListModel 实现类，并为 ComboBoxModel 提供了 NumberComboBoxModel 实现类，这两个实现类允许程序使用数值范围来创建 JList 和 JComboBox 对象。

程序清单：codes\12\12.9\ListModelTest.java

```
public class ListModelTest
{
    private JFrame mainWin = new JFrame("测试 ListModel");
    // 根据 NumberListModel 对象来创建一个 JList 对象
    private JList<BigDecimal> numScopeList = new JList<>(
        new NumberListModel(1 , 21 , 2));
    // 根据 NumberComboBoxModel 对象来创建 JComboBox 对象
    private JComboBox<BigDecimal> numScopeSelector = new JComboBox<>(
        new NumberComboBoxModel(0.1 , 1.2 , 0.1));
    private JTextField showVal = new JTextField(10);
    public void init()
    {
        // JList 的可视高度可同时显示 4 个列表项
        numScopeList.setVisibleRowCount(4);
        // 默认选中第 3 项到第 5 项（第 1 项的索引是 0）
        numScopeList.setSelectionInterval(2, 4);
        // 设置每个列表项具有指定的高度和宽度
        numScopeList.setFixedCellHeight(30);
        numScopeList.setFixedCellWidth(90);
        // 为 numScopeList 添加监听器
        numScopeList.addListSelectionListener(e -> {
            // 获取用户所选中的所有数字
            List<BigDecimal> nums = numScopeList.getSelectedValuesList();
            showVal.setText("");
            // 把用户选中的数字添加到单行文本框中
            for (BigDecimal num : nums )
            {
                showVal.setText(showVal.getText()
                    + num.toString() + ", ");
            }
        });
        // 设置列表项的可视高度可显示 5 个列表项
        numScopeSelector.setMaximumRowCount(5);
        Box box = new Box(BoxLayout.X_AXIS);
        box.add(new JScrollPane(numScopeList));
        JPanel p = new JPanel();
        p.add(numScopeSelector);
        box.add(p);
        // 为 numScopeSelector 添加监听器
        numScopeSelector.addItemListener(e -> {
            // 获取 JComboBox 中选中的数字
```

```
            Object num = numScopeSelector.getSelectedItem();
            showVal.setText(num.toString());
        });
        JPanel bottom = new JPanel();
        bottom.add(new JLabel("您选择的值是："));
        bottom.add(showVal);
        mainWin.add(box);
        mainWin.add(bottom , BorderLayout.SOUTH);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ListModelTest().init();
    }
}
class NumberListModel extends AbstractListModel<BigDecimal>
{
    protected BigDecimal start;
    protected BigDecimal end;
    protected BigDecimal step;
    public NumberListModel(double start
        , double end , double step)
    {
        this.start = BigDecimal.valueOf(start);
        this.end = BigDecimal.valueOf(end);
        this.step = BigDecimal.valueOf(step);
    }
    // 返回列表项的个数
    public int getSize()
    {
        return (int)Math.floor(end.subtract(start)
            .divide(step).doubleValue()) + 1;
    }
    // 返回指定索引处的列表项
    public BigDecimal getElementAt(int index)
    {
        return BigDecimal.valueOf(index)
            .multiply(step).add(start);
    }
}
class NumberComboBoxModel extends NumberListModel
    implements ComboBoxModel<BigDecimal>
{
    // 用于保存用户选中项的索引
    private int selectId = 0;
    public NumberComboBoxModel(double start
        , double end , double step)
    {
        super(start , end , step);
    }
    // 设置选中“选择项”
    public void setSelectedItem(Object anItem)
    {
        if (anItem instanceof BigDecimal)
        {
            BigDecimal target = (BigDecimal)anItem;
            // 根据选中的值来修改选中项的索引
            selectId = target.subtract(super.start)
                .divide(step).intValue();
        }
    }
    // 获取“选择项”的值
    public BigDecimal getSelectedItem()
    {
        // 根据选中项的索引来取得选中项
        return BigDecimal.valueOf(selectId)
```

```
                .multiply(step).add(start);
        }
    }
```

上面程序中的粗体字代码分别使用 NumberListModel 和 NumberComboBoxModel 创建了一个 JList 和 JComboBox 对象，创建这两个列表框时无须指定每个列表项，只需给出数值的上限、下限和步长即可。运行上面程序，会看到如图 12.35 所示的窗口。

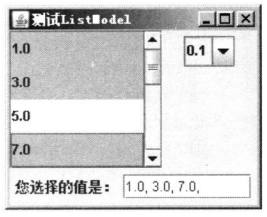


图 12.35 根据数值范围创建的 JList 和 JComboBox

12.9.3 强制存储列表项的 DefaultListModel 和 DefaultComboBoxModel

前面只是介绍了如何创建 JList、JComboBox 对象，当调用 JList 和 JComboBox 构造器时传入数组或 Vector 作为参数，这些数组元素或集合元素将会作为列表项。当使用 JList 或 JComboBox 时常常还需要动态地增加、删除列表项。

对于 JComboBox 类，它提供了如下几个方法来增加、插入和删除列表项。

- addItem(E anObject)：向 JComboBox 中的添加一个列表项。
- insertItemAt(E anObject, int index)：向 JComboBox 的指定索引处插入一个列表项。
- removeAllItems()：删除 JComboBox 中的所有列表项。
- removeItem(E anObject)：删除 JComboBox 中的指定列表项。
- removeItemAt(int anIndex)：删除 JComboBox 指定索引处的列表项。

提示：

上面这些方法的参数类型是 E，这是由于 Java 7 为 JComboBox、JList、ListModel 都增加了泛型支持，这些接口都有形如 JComboBox<E>、JList<E>、ListModel<E>的泛型声明，因此它们里面的方法可使用 E 作为参数或返回值的类型。

通过这些方法就可以增加、插入和删除 JComboBox 中的列表项，但 JList 并没有提供这些类似的方法。实际上，对于直接通过数组或 Vector 创建的 JList 对象，则很难向该 JList 中添加或删除列表项。如果需要创建一个可以增加、删除列表项的 JList 对象，则应该在创建 JList 时显式使用 DefaultListModel 作为构造参数。因为 DefaultListModel 作为 JList 的 Model，它负责维护 JList 组件的所有列表数据，所以可以通过向 DefaultListModel 中添加、删除元素来实现向 JList 对象中增加、删除列表项。DefaultListModel 提供了如下几个方法来添加、删除元素。

- add(int index, E element)：在该 ListModel 的指定位置处插入指定元素。
- addElement(E obj)：将指定元素添加到该 ListModel 的末尾。
- insertElementAt(E obj, int index)：在该 ListModel 的指定位置处插入指定元素。
- Object remove(int index)：删除该 ListModel 中指定位置处的元素。
- removeAllElements()：删除该 ListModel 中的所有元素，并将其的大小设置为零。
- removeElement(E obj)：删除该 ListModel 中第一个与参数匹配的元素。
- removeElementAt(int index)：删除该 ListModel 中指定索引处的元素。
- removeRange(int fromIndex, int toIndex)：删除该 ListModel 中指定范围内的所有元素。
- set(int index, E element)：将该 ListModel 指定索引处的元素替换成指定元素。
- setElementAt(E obj, int index)：将该 ListModel 指定索引处的元素替换成指定元素。

上面这些方法有些功能是重复的，这是由于 Java 的历史原因造成的。如果通过 DefaultListModel 来创建 JList 组件，则就可以通过调用上面的这些方法来添加、删除 DefaultListModel 中的元素，从而实现对 JList 里列表项的增加、删除。下面程序示范了如何向 JList 中添加、删除列表项。

程序清单：codes\12\12.9\DefaultListModelTest.java

```
public class DefaultListModelTest
```

```
{
    private JFrame mainWin = new JFrame("测试 DefaultListModel");
    // 定义一个 JList 对象
    private JList<String> bookList;
    // 定义一个 DefaultListModel 对象
    private DefaultListModel<String> bookModel
        = new DefaultListModel<>();
    private JTextField bookName = new JTextField(20);
    private JButton removeBn = new JButton("删除选中图书") ;
    private JButton addBn = new JButton("添加指定图书");
    public void init()
    {
        // 向 bookModel 中添加元素
        bookModel.addElement("疯狂 Java 讲义");
        bookModel.addElement("轻量级 Java EE 企业应用实战");
        bookModel.addElement("疯狂 Android 讲义");
        bookModel.addElement("疯狂 Ajax 讲义");
        bookModel.addElement("经典 Java EE 企业应用实战");
        // 根据 DefaultListModel 对象创建一个 JList 对象
        bookList = new JList<>(bookModel);
        // 设置最大可视高度
        bookList.setVisibleRowCount(4);
        // 只能单选
        bookList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        // 为添加按钮添加事件监听器
        addBn.addActionListener(evt -> {
            // 当 bookName 文本框的内容不为空时
            if (!bookName.getText().trim().equals(""))
            {
                // 向 bookModel 中添加一个元素
                // 系统会自动向 JList 中添加对应的列表项
                bookModel.addElement(bookName.getText());
            }
        });
        // 为删除按钮添加事件监听器
        removeBn.addActionListener(evt -> {
            // 如果用户已经选中一项
            if (bookList.getSelectedIndex() >= 0)
            {
                // 从 bookModel 中删除指定索引处的元素
                // 系统会自动删除 JList 对应的列表项
                bookModel.removeElementAt(bookList.getSelectedIndex());
            }
        });
        JPanel p = new JPanel();
        p.add(bookName);
        p.add(addBn);
        p.add(removeBn);
        // 添加 bookList 组件
        mainWin.add(new JScrollPane(bookList));
        // 将 p 面板添加到窗口中
        mainWin.add(p , BorderLayout.SOUTH);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new DefaultListModelTest().init();
    }
}
```

上面程序中的粗体字代码通过一个 DefaultListModel 创建了一个 JList 对象，然后在两个按钮的事件监听器中分别向 DefaultListModel 对象中添加、删除元素，从而实现了向 JList 对象中添加、删除列表项。运行上面程序，会看到如图 12.36 所示的窗口。

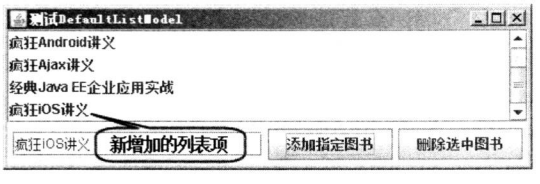
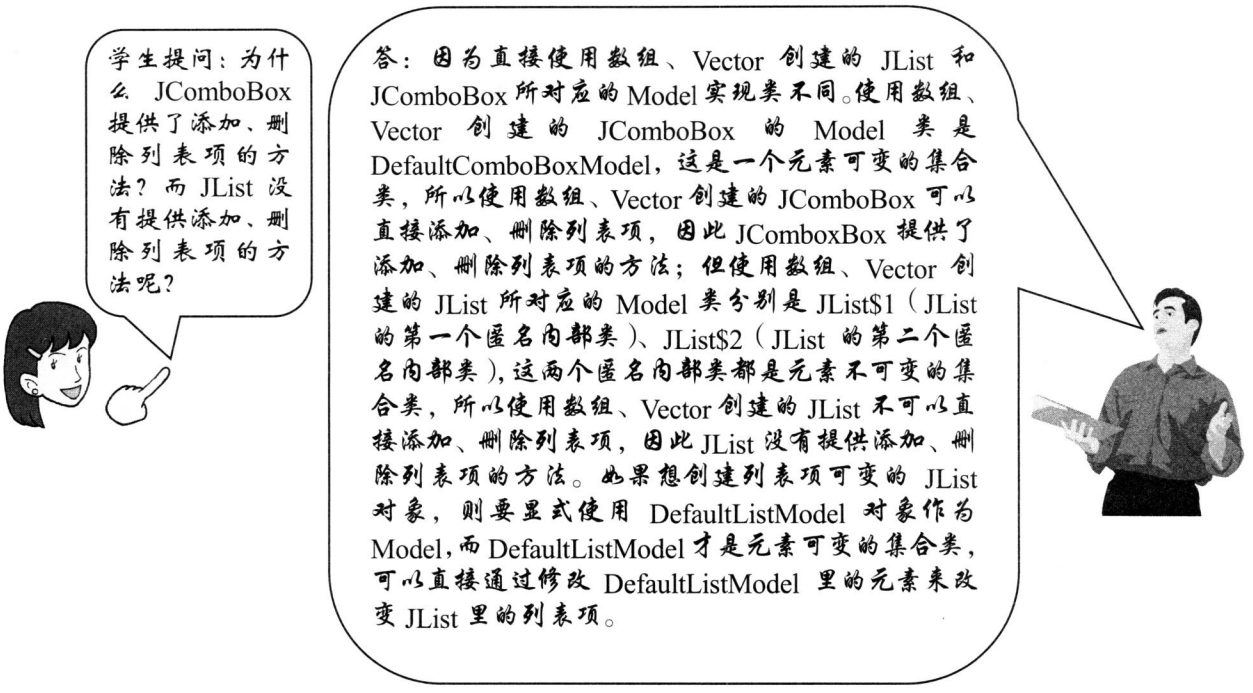


图 12.36　向 JList 中添加、删除列表项



DefaultListModel 和 DefaultComboBoxModel 是两个强制保存所有列表项的 Model 类，它们使用 Vector 来保存所有的列表项。从 DefaultListModelTest 程序中可以看出，DefaultListModel 的用法和 Vector 的用法非常相似。实际上，DefaultListModel 和 DefaultComboBoxModel 从功能上来看，与一个 Vector 并没有太大的区别。如果要创建列表项可变的 JList 组件，使用 DefaultListModel 作为构造参数即可，读者可以把 DefaultListModel 当成一个特殊的 Vector；创建列表项可变的 JComboBox 组件，当然也可以显式使用 DefaultComboBoxModel 作为参数，但这并不是必需的，因为 JComboBox 默认使用 DefaultComboBoxModel 作为对应的 model 对象。

12.9.4 使用 ListCellRenderer 改变列表项外观

前面程序中的 JList 和 JComboBox 采用的都是简单的字符串列表项，实际上，JList 和 JComboBox 还可以支持图标列表项，如果在创建 JList 或 JComboBox 时传入图标数组，则创建的 JList 和 JComboBox 的列表项就是图标。

如果希望列表项是更复杂的组件，例如，希望像 QQ 程序那样每个列表项既有图标，也有字符串，那么可以通过调用 JList 或 JComboBox 的 setCellRenderer(ListCellRenderer cr)方法来实现，该方法需要接收一个 ListCellRenderer 对象，该对象代表一个列表项绘制器。

ListCellRenderer 是一个接口，该接口里包含一个方法：

```
public Component getListCellRendererComponent(JList list, Object value
    , int index, bolean isSelected, boolean cellHasFocus)
```

上面的 getListCellRendererComponent()方法返回一个 Component 组件，该组件就代表了 JList 或 JComboBox 的每个列表项。

注意：

自定义绘制 JList 和 JComboBox 的列表项所用的方法相同，所用的列表项绘制器也相同，故本节以 JList 为例。

ListCellRenderer 只是一个接口，它并未强制指定列表项绘制器属于哪种组件，因此可扩展任何组件来实现 ListCellRenderer 接口。通常采用扩展其他容器（如 JPanel）的方式来实现列表项绘制器，实现列表项绘制器时可通过重写 paintComponent()的方法来改变单元格的外观行为。例如下面程序，重写

paintComponent()方法时先绘制好友图像，再绘制好友名字。

程序清单：codes\12\12.9\ListRenderingTest.java

```
public class ListRenderingTest
{
    private JFrame mainWin = new JFrame("好友列表");
    private String[] friends = new String[]
    {
        "李清照",
        "苏格拉底",
        "李白",
        "弄玉",
        "虎头"
    };
    // 定义一个 JList 对象
    private JList friendsList = new JList(friends);
    public void init()
    {
        // 设置该 JList 使用 ImageCellRenderer 作为列表项绘制器
        friendsList.setCellRenderer(new ImageCellRenderer());
        mainWin.add(new JScrollPane(friendsList));
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ListRenderingTest().init();
    }
}
class ImageCellRenderer extends JPanel
    implements ListCellRenderer
{
    private ImageIcon icon;
    private String name;
    // 定义绘制单元格时的背景色
    private Color background;
    // 定义绘制单元格时的前景色
    private Color foreground;
    public Component getListCellRendererComponent(JList list
        , Object value , int index
        , boolean isSelected , boolean cellHasFocus)
    {
        icon = new ImageIcon("ico/" + value + ".gif");
        name = value.toString();
        background = isSelected ? list.getSelectionBackground()
            : list.getBackground();
        foreground = isSelected ? list.getSelectionForeground()
            : list.getForeground();
        // 返回该 JPanel 对象作为列表项绘制器
        return this;
    }
    // 重写 paintComponent()方法，改变 JPanel 的外观
    public void paintComponent(Graphics g)
    {
        int imageWidth = icon.getImage().getWidth(null);
        int imageHeight = icon.getImage().getHeight(null);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
        // 绘制好友图标
        g.drawImage(icon.getImage() , getWidth() / 2
            - imageWidth / 2 , 10 , null);
        g.setFont(new Font("SansSerif" , Font.BOLD , 18));
        // 绘制好友用户名
        g.drawString(name, getWidth() / 2
            - name.length() * 10 , imageHeight + 30 );
    }
```

```
    // 通过该方法来设置该 ImageCellRenderer 的最佳大小
    public Dimension getPreferredSize()
    {
        return new Dimension(60, 80);
    }
}
```

上面程序中的粗体字代码显式指定了该 JList 对象使用 ImageCellRenderer 作为列表项绘制器，ImageCellRenderer 重写了 paintComponent()方法来绘制单元格内容。除此之外，ImageCellRenderer 还重写了 getPreferredSize()方法，该方法返回一个 Dimension 对象，用于描述该列表项绘制器的最佳大小。运行上面程序，会看到如图 12.37 所示的窗口。

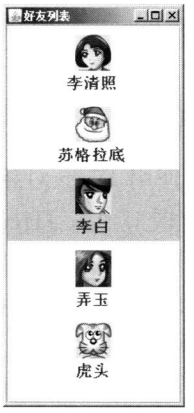


图 12.37 使用 ListCellRenderer 绘制列表项

通过使用自定义的列表项绘制器，可以让 JList 和 JComboBox 的列表项是任意组件，并且可以在该组件上任意添加内容。

12.10 使用 JTree 和 TreeModel 创建树

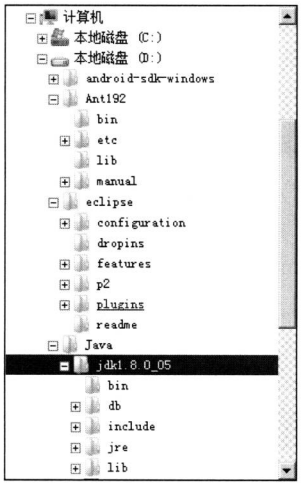


图 12.38 Windows 资源管理器目录树

树也是图形用户界面中使用非常广泛的 GUI 组件，例如使用 Windows 资源管理器时，将看到如图 12.38 所示的目录树。

如图 12.38 所示的树，代表计算机世界里的树，它从自然界实际的树抽象而来。计算机世界里的树是由一系列具有严格父子关系的节点组成的，每个节点既可以是其上一级节点的子节点，也可以是其下一级节点的父节点，因此同一个节点既可以是父节点，也可以是子节点（类似于一个人，他既是他儿子的父亲，又是他父亲的儿子）。

如果按节点是否包含子节点来分，节点分为如下两种。

- 普通节点：包含子节点的节点。
- 叶子节点：没有子节点的节点，因此叶子节点不可作为父节点。

如果按节点是否具有唯一的父节点来分，节点又可分为如下两种。

- 根节点：没有父节点的节点，根节点不可作为子节点。
- 普通节点：具有唯一父节点的节点。

一棵树只能有一个根节点，如果一棵树有了多个根节点，那它就不是一棵树了，而是多棵树的集合，有时也被称为森林。图 12.39 显示了计算机世界里树的一些专业术语。

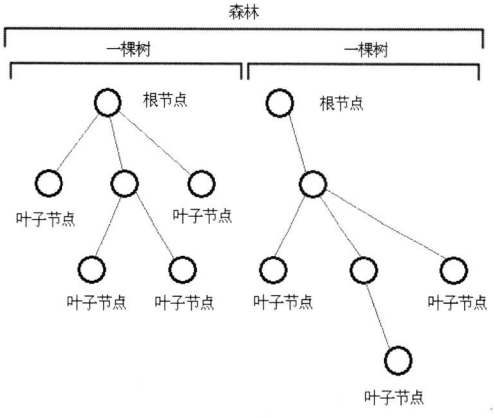


图 12.39 计算机世界里树的示意图

使用 Swing 里的 Jtree、TreeModel 及其相关的辅助类可以很轻松地开发出计算机世界里的树，如图 12.39 所示。

12.10.1 创建树

Swing 使用 JTree 对象来代表一棵树（实际上，JTree 可以代表森林，因为在使用 JTree 创建树时可以传入多个根节点），JTree 树中节点可以使用 TreePath 来标识，该对象封装了当前节点及其所有的父节点。必须指出，节点及其所有的父节点才能唯一地标识一个节点；也可以使用行数来标识，如图 12.39 所示，显示区域的每一行都标识一个节点。

当一个节点具有子节点时，该节点有两种状态。

- 展开状态：当父节点处于展开状态时，其子节点是可见的。
- 折叠状态：当父节点处于折叠状态时，其子节点都是不可见的。

如果某个节点是可见的，则该节点的父节点（包括直接的、间接的父节点）都必须处于展开状态，只要有任意一个父节点处于折叠状态，该节点就是不可见的。

如果希望创建一棵树，则直接使用 JTree 的构造器创建 JTree 对象即可。JTree 提供了如下几个常用构造器。

- JTree(TreeModel newModel)：使用指定的数据模型创建 JTree 对象，它默认显示根节点。
- JTree(TreeNode root)：使用 root 作为根节点创建 JTree 对象，它默认显示根节点。
- JTree(TreeNode root, boolean asksAllowsChildren)：使用 root 作为根节点创建 JTree 对象，它默认显示根节点。asksAllowsChildren 参数控制怎样的节点才算叶子节点，如果该参数为 true，则只有当程序使用 setAllowsChildren(false)显式设置某个节点不允许添加子节点时（以后也不会拥有子节点），该节点才会被 JTree 当成叶子节点；如果该参数为 false，则只要某个节点当时没有子节点（不管以后是否拥有子节点），该节点都会被 JTree 当成叶子节点。

上面的第一个构造器需要显式传入一个 TreeModel 对象，Swing 为 TreeModel 提供了一个 DefaultTreeModel 实现类，通常可先创建 DefaultTreeModel 对象，然后利用 DefaultTreeModel 来创建 JTree，但通过 DefaultTreeModel 的 API 文档会发现，创建 DefaultTreeModel 对象依然需要传入根节点，所以直接通过根节点创建 JTree 更加简洁。

为了利用根节点来创建 JTree，程序需要创建一个 TreeNode 对象。TreeNode 是一个接口，该接口有一个 MutableTreeNode 子接口，Swing 为该接口提供了默认的实现类：DefaultMutableTreeNode，程序可以通过 DefaultMutableTreeNode 来为树创建节点，并通过 DefaultMutableTreeNode 提供的 add()方法建立各节点之间的父子关系，然后调用 JTree 的 JTree(TreeNode root)构造器来创建一棵树。

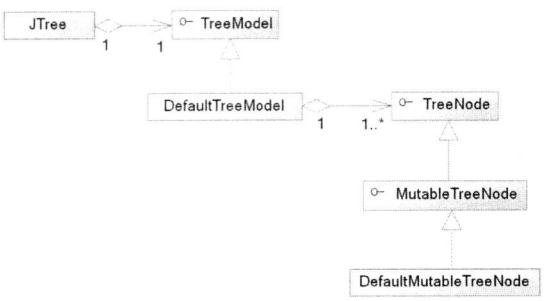


图 12.40　JTree 相关类的关系

图 12.40 显示了 JTree 相关类的关系，从该图可以看出 DefaultTreeModel 是 TreeModel 的默认实现类，当程序通过 TreeNode 类创建 JTree 时，其状态数据实际上由 DefaultTreeModel 对象维护，因为创建 JTree 时传入的 TreeNode 对象，实际上传给了 DefaultTreeModel 对象。

提示：DefaultTreeModel 也提供了 DefaultTreeModel(TreeNode root)构造器，用于接收一个 TreeNode 根节点来创建一个默认的 TreeModel 对象；当程序中通过传入一个根节点来创建 JTree 对象时，实际上是将该节点传入对应的 DefaultTreeModel 对象，并使用该 DefaultTreeModel 对象来创建 JTree 对象。

下面程序创建了一棵最简单的 Swing 树。

程序清单：codes\12\12.10\SimpleJTree.java

```
public class SimpleJTree
{
    JFrame jf = new JFrame("简单树");
    JTree tree;
    DefaultMutableTreeNode root;
    DefaultMutableTreeNode guangdong;
    DefaultMutableTreeNode guangxi;
    DefaultMutableTreeNode foshan;
    DefaultMutableTreeNode shantou;
    DefaultMutableTreeNode guilin;
    DefaultMutableTreeNode nanning;
    public void init()
    {
        // 依次创建树中的所有节点
        root = new DefaultMutableTreeNode("中国");
        guangdong = new DefaultMutableTreeNode("广东");
        guangxi = new DefaultMutableTreeNode("广西");
        foshan = new DefaultMutableTreeNode("佛山");
        shantou = new DefaultMutableTreeNode("汕头");
        guilin = new DefaultMutableTreeNode("桂林");
        nanning = new DefaultMutableTreeNode("南宁");
        // 通过 add()方法建立树节点之间的父子关系
        guangdong.add(foshan);
        guangdong.add(shantou);
        guangxi.add(guilin);
        guangxi.add(nanning);
        root.add(guangdong);
        root.add(guangxi);
        // 以根节点创建树
        tree = new JTree(root);   //①
        jf.add(new JScrollPane(tree));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SimpleJTree().init();
    }
}
```

上面程序中的粗体字代码创建了一系列的 DefaultMutableTreeNode 对象，并通过 add()方法为这些节点建立了相应的父子关系。程序中①号粗体字代码则以一个根节点创建了一个 JTree 对象。当程序把 JTree 对象添加到其他容器中后，JTree 就会在该容器中绘制出一棵 Swing 树。运行上面程序，会看到如图 12.41 所示的窗口。

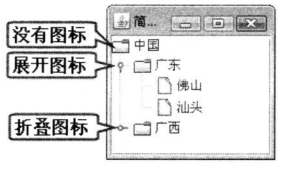


图 12.41 Swing 树的效果

从图 12.41 中可以看出，Swing 树的默认风格是使用一个特殊图标来表示节点的展开、折叠，而不是使用我们熟悉的“+”、“-”图标来表示节点的展开、折叠。如果希望使用“+”、“-”图标来表示节点的展开、折叠，则可以考虑使用 Windows 风格。

从图 12.41 中可以看出，Swing 树默认使用连接线来连接所有节点，程序可以使用如下代码来强制 JTree 不显示节点之间的连接线。

```
// 没有连接线
tree.putClientProperty("JTree.lineStyle" , "None");
```

或者使用如下代码来强制节点之间只有水平分隔线。

```
// 水平分隔线
tree.putClientProperty("JTree.lineStyle" , "Horizontal");
```

图 12.41 中显示的根节点前没有绘制表示节点展开、折叠的特殊图标，如果希望根节点也绘制表示节点展开、折叠的特殊图标，则使用如下代码。

```
// 设置是否显示根节点的“展开、折叠”图标，默认是 false
tree.setShowsRootHandles(true);
```

JTree 甚至允许把整个根节点都隐藏起来，可以通过如下代码来隐藏根节点。

```
// 设置根节点是否可见，默认是 true
tree.setRootVisible(false);
```

DefaultMutableTreeNode 是 JTree 默认的树节点，该类提供了大量的方法来访问树中的节点，包括遍历该节点的所有子节点的两个方法。DefaultMutableTreeNode 提供了深度优先遍历、广度优先遍历两种方法。

- Enumeration breadthFirstEnumeration()/preorderEnumeration()：按广度优先的顺序遍历以此节点为根的子树，并返回所有节点组成的枚举对象。
- Enumeration depthFirstEnumeration()/postorderEnumeration()：按深度优先的顺序遍历以此节点为根的子树，并返回所有节点组成的枚举对象。

提示：

关于树的深度优先和广度优先遍历算法已经不属于本书的介绍范围，读者可以参考《疯狂 Java 程序员的基本修养》学习有关树的更详细内容。

除此之外，DefaultMutableTreeNode 也提供了大量的方法来获取指定节点的兄弟节点、父节点、子节点等，常用的有如下几个方法。

- DefaultMutableTreeNode getNextSibling()：返回此节点的下一个兄弟节点。
- TreeNode getParent()：返回此节点的父节点。如果此节点没有父节点，则返回 null。
- TreeNode[] getPath()：返回从根节点到达此节点的所有节点组成的数组。
- DefaultMutableTreeNode getPreviousSibling()：返回此节点的上一个兄弟节点。
- TreeNode getRoot()：返回包含此节点的树的根节点。
- TreeNode getSharedAncestor(DefaultMutableTreeNode aNode)：返回此节点和 aNode 最近的共同祖先。
- int getSiblingCount()：返回此节点的兄弟节点数。
- boolean isLeaf()：返回该节点是否是叶子节点。
- boolean isNodeAncestor(TreeNode anotherNode)：判断 anotherNode 是否是当前节点的祖先节点（包括父节点）。
- boolean isNodeChild(TreeNode aNode)：如果 aNode 是此节点的子节点，则返回 true。
- boolean isNodeDescendant(DefaultMutableTreeNode anotherNode)：如果 anotherNode 是此节点的后代，包括是此节点本身、此节点的子节点或此节点的子节点的后代，都将返回 true。
- boolean isNodeRelated(DefaultMutableTreeNode aNode)：当 aNode 和当前节点位于同一棵树中时返回 true。
- boolean isNodeSibling(TreeNode anotherNode)：返回 anotherNode 是否是当前节点的兄弟节点。
- boolean isRoot()：返回当前节点是否是根节点。
- Enumeration pathFromAncestorEnumeration(TreeNode ancestor)：返回从指定祖先节点到当前节点的所有节点组成的枚举对象。

12.10.2 拖动、编辑树节点

JTree 生成的树默认是不可编辑的，不可以添加、删除节点，也不可以改变节点数据；如果想让某个 JTree 对象变成可编辑状态，则可以调用 JTree 的 setEditable(boolean b)方法，传入 true 即可把这棵树变成可编辑的树（可以添加、删除节点，也可以改变节点数据）。

一旦将 JTree 对象设置成可编辑状态后，程序就可以为指定节点添加子节点、兄弟节点，也可以修改、删除指定节点。

前面简单提到过，JTree 处理节点有两种方式：一种是根据 TreePath；另一种是根据节点的行号，所有 JTree 显示的节点都有一个唯一的行号（从 0 开始）。只有那些被显示出来的节点才有行号，这就带来一个潜在的问题——如果该节点之前的节点被展开、折叠或增加、删除后，那么该节点的行号就会发生变化，因此通过行号来识别节点可能有一些不确定的地方；相反，使用 TreePath 来识别节点则会更加稳定。

可以使用文件系统来类比 JTree，从图 12.38 中可以看出，实际上所有的文件系统都采用树状结构，其中 Windows 的文件系统是森林，因为 Windows 包含 C、D 等多个根路径，而 UNIX、Linux 的文件系统是一棵树，只有一个根路径。如果直接给出 abc 文件夹（类似于 JTree 中的节点），系统不能准确地定位该路径；如果给出 D:\xyz\abc，系统就可以准确地定位到该路径，这个 D:\xyz\abc 实际上由三个文件夹组成：D:、xyz、abc，其中 D:是该路径的根路径。类似地，TreePath 也采用这种方式来唯一地标识节点。

TreePath 保持着从根节点到指定节点的所有节点，TreePath 由一系列节点组成，而不是单独的一个节点。JTree 的很多方法都用于返回一个 TreePath 对象，当程序得到一个 TreePath 后，可能只需要获取最后一个节点，则可以调用 TreePath 的 getLastPathComponent()方法。例如需要获得 JTree 中被选定的节点，则可以通过如下两行代码来实现。

```
// 获取选中节点所在的 TreePath
TreePath path = tree.getSelectionPath();
// 获取指定 TreePath 的最后一个节点
TreeNode target = (TreeNode)path.getLastPathComponent();
```

又因为 JTree 经常需要查询被选中的节点，所以 JTree 提供了一个 getLastSelectedPathComponent()方法来获取选中的节点。比如采用下面代码也可以获取选中的节点。

```
// 获取选中的节点
TreeNode target = (TreeNode) tree.getLastSelectedPathComponent();
```

可能有读者对上面这行代码感到奇怪，getLastSelectedPathComponent()方法返回的不是 TreeNode 吗？getLastSelectedPathComponent()方法返回的不一定是 TreeNode，该方法的返回值是 Object。因为 Swing 把 JTree 设计得非常复杂，JTree 把所有的状态数据都交给 TreeModel 管理，而 JTree 本身并没有与 TreeNode 发生关联（从图 12.40 可以看出这一点），只是因为 DefaultTreeModel 需要 TreeNode 而已，如果开发者自己提供一个 TreeModel 实现类，这个 TreeModel 实现类完全可以与 TreeNode 没有任何关系。当然，对于大部分 Swing 开发者而言，无须理会 JTree 的这些过于复杂的设计。

如果已经有了从根节点到当前节点的一系列节点所组成的节点数组，也可以通过 TreePath 提供的构造器将这些节点转换成 TreePath 对象，如下代码所示。

```
// 将一个节点数组转换成 TreePath 对象
TreePath tp = new TreePath(nodes);
```

获取了选中的节点之后，即可通过 DefaultTreeModel（它是 Swing 为 TreeModel 提供的唯一一个实现类）提供的一系列方法来插入、删除节点。DefaultTreeModel 类有一个非常优秀的设计，当使用 DefaultTreeModel 插入、删除节点后，该 DefaultTreeModel 会自动通知对应的 JTree 重绘所有节点，用户可以立即看到程序所做的修改。

也可以直接通过 TreeNode 提供的方法来添加、删除和修改节点，但通过 TreeNode 改变节点时，程序必须显式调用 JTree 的 updateUI()通知 JTree 重绘所有节点，让用户看到程序所做的修改。

下面程序实现了增加、修改和删除节点的功能，并允许用户通过拖动将一个节点变成另一个节点的子节点。

程序清单：codes\12\12.10\EditJTree.java

```
public class EditJTree
{
    JFrame jf;
    JTree tree;
    // 上面 JTree 对象对应的 model
```

```
DefaultTreeModel model;
// 定义几个初始节点
DefaultMutableTreeNode root = new DefaultMutableTreeNode("中国");
DefaultMutableTreeNode guangdong = new DefaultMutableTreeNode("广东");
DefaultMutableTreeNode guangxi = new DefaultMutableTreeNode("广西");
DefaultMutableTreeNode foshan = new DefaultMutableTreeNode("佛山");
DefaultMutableTreeNode shantou = new DefaultMutableTreeNode("汕头");
DefaultMutableTreeNode guilin = new DefaultMutableTreeNode("桂林");
DefaultMutableTreeNode nanning = new DefaultMutableTreeNode("南宁");
// 定义需要被拖动的 TreePath
TreePath movePath;
JButton addSiblingButton = new JButton("添加兄弟节点");
JButton addChildButton = new JButton("添加子节点");
JButton deleteButton = new JButton("删除节点");
JButton editButton = new JButton("编辑当前节点");
public void init()
{
    guangdong.add(foshan);
    guangdong.add(shantou);
    guangxi.add(guilin);
    guangxi.add(nanning);
    root.add(guangdong);
    root.add(guangxi);
    jf = new JFrame("可编辑节点的树");
    tree = new JTree(root);
    // 获取 JTree 对应的 TreeModel 对象
    model = (DefaultTreeModel)tree.getModel();
    // 设置 JTree 可编辑
    tree.setEditable(true);
    MouseListener ml = new MouseAdapter()
    {
        // 按下鼠标时获得被拖动的节点
        public void mousePressed(MouseEvent e)
        {
            // 如果需要唯一确定某个节点，则必须通过 TreePath 来获取
            TreePath tp = tree.getPathForLocation(
                e.getX() , e.getY());
            if (tp != null)
            {
                movePath = tp;
            }
        }
        // 松开鼠标时获得需要拖到哪个父节点
        public void mouseReleased(MouseEvent e)
        {
            // 根据松开鼠标时的 TreePath 来获取 TreePath
            TreePath tp = tree.getPathForLocation(
                e.getX(), e.getY());
            if (tp != null && movePath != null)
            {
                // 阻止向子节点拖动
                if (movePath.isDescendant(tp) && movePath != tp)
                {
                    JOptionPane.showMessageDialog(jf,
                        "目标节点是被移动节点的子节点，无法移动！",
                        "非法操作", JOptionPane.ERROR_MESSAGE );
                    return;
                }
                // 不是向子节点移动，鼠标按下、松开的也不是同一个节点
                else if (movePath != tp)
                {
                    // add 方法先将该节点从原父节点下删除，再添加到新父节点下
                    ((DefaultMutableTreeNode)tp.getLastPathComponent())
                        .add((DefaultMutableTreeNode)movePath
                        .getLastPathComponent());
                    movePath = null;
                    tree.updateUI();
                }
            }
        }
```

```
};
// 为 JTree 添加鼠标监听器
tree.addMouseListener(ml);
JPanel panel = new JPanel();
// 实现添加兄弟节点的监听器
addSiblingButton.addActionListener(event -> {
    // 获取选中的节点
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
        tree.getLastSelectedPathComponent();
    // 如果节点为空，则直接返回
    if (selectedNode == null) return;
    // 获取该选中节点的父节点
    DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
        selectedNode.getParent();
    // 如果父节点为空，则直接返回
    if (parent == null) return;
    // 创建一个新节点
    DefaultMutableTreeNode newNode = new
        DefaultMutableTreeNode("新节点");
    // 获取选中节点的选中索引
    int selectedIndex = parent.getIndex(selectedNode);
    // 在选中位置插入新节点
    model.insertNodeInto(newNode, parent, selectedIndex + 1);
    // --------下面代码实现显示新节点（自动展开父节点）-------
    // 获取从根节点到新节点的所有节点
    TreeNode[] nodes = model.getPathToRoot(newNode);
    // 使用指定的节点数组来创建 TreePath
    TreePath path = new TreePath(nodes);
    // 显示指定的 TreePath
    tree.scrollPathToVisible(path);
});
panel.add(addSiblingButton);
// 实现添加子节点的监听器
addChildButton.addActionListener(event -> {
    // 获取选中的节点
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
        tree.getLastSelectedPathComponent();
    // 如果节点为空，则直接返回
    if (selectedNode == null) return;
    // 创建一个新节点
    DefaultMutableTreeNode newNode = new
        DefaultMutableTreeNode("新节点");
    // 通过 model 来添加新节点，则无须调用 JTree 的 updateUI 方法
    // model.insertNodeInto(newNode, selectedNode
    //     , selectedNode.getChildCount());
    // 通过节点添加新节点，则需要调用 tree 的 updateUI 方法
    selectedNode.add(newNode);
    // --------下面代码实现显示新节点（自动展开父节点）-------
    TreeNode[] nodes = model.getPathToRoot(newNode);
    TreePath path = new TreePath(nodes);
    tree.scrollPathToVisible(path);
    tree.updateUI();
});
panel.add(addChildButton);
// 实现删除节点的监听器
deleteButton.addActionListener(event -> {
    DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
        tree.getLastSelectedPathComponent();
    if (selectedNode != null && selectedNode.getParent() != null)
    {
        // 删除指定节点
        model.removeNodeFromParent(selectedNode);
    }
});
panel.add(deleteButton);
// 实现编辑节点的监听器
editButton.addActionListener(event -> {
    TreePath selectedPath = tree.getSelectionPath();
    if (selectedPath != null)
    {
        // 编辑选中的节点
```

```
                tree.startEditingAtPath(selectedPath);
            }
        });
        panel.add(editButton);
        jf.add(new JScrollPane(tree));
        jf.add(panel , BorderLayout.SOUTH);
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new EditJTree().init();
    }
}
```

上面程序中实现拖动节点也比较容易——当用户按下鼠标时获取鼠标事件发生位置的树节点，并把该节点赋给 movePath 变量；当用户松开鼠标时获取鼠标事件发生位置的树节点，作为目标节点需要拖到的父节点，把 movePath 从原来的节点中删除，添加到新的父节点中即可（TreeNode 的 add()方法可以同时完成这两个操作）。程序中的粗体字代码是实现整个程序的关键代码，读者可以结合程序运行效果来研究该代码。运行上面程序，会看到如图 12.42 所示的效果。

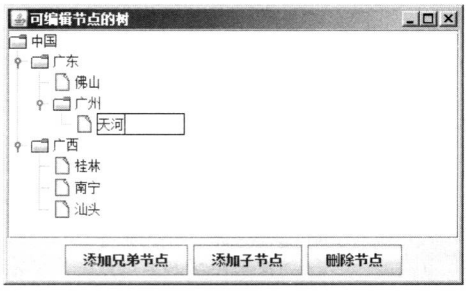


图 12.42　可以拖动、添加、删除节点的 Swing 树

选中图 12.42 中的某个节点并双击，或者单击“编辑当前节点”按钮，就可以进入该节点的编辑状态，系统启动默认的单元格编辑器来编辑该节点，JTree 的单元格编辑器与 JTable 的单元格编辑器都实现了相同的 CellEditor 接口。本书将在下一节与 JTable 一起介绍如何定制节点编辑器。

12.10.3　监听节点事件

JTree 专门提供了一个 TreeSelectionModel 对象来保存该 JTree 选中状态的信息。也就是说，JTree 组件背后隐藏了两个 model 对象，其中 TreeModel 用于保存该 JTree 的所有节点数据，而 TreeSelectionModel 用于保存该 JTree 的所有选中状态的信息。

提示：

对于大部分开发者而言，无须关心 TreeSelectionModel 的存在，程序可以通过 JTree 提供的 getSelectionPath()方法和 getSelectionPaths()方法来获取该 JTree 被选中的 TreePath，但实际上这两个方法底层实现依然依赖于 TreeSelectionModel，只是普通开发者一般无须关心这些底层细节而已。

程序可以改变 JTree 的选择模式，但必须先获取该 JTree 对应的 TreeSelectionModel 对象，再调用该对象的 setSelectionMode()方法来设置该 JTree 的选择模式。该方法支持如下三个参数。

- TreeSelectionModel.CONTINUOUS_TREE_SELECTION：可以连续选中多个 TreePath。
- TreeSelectionModel.DISCONTINUOUS_TREE_SELECTION：该选项对于选择没有任何限制。
- TreeSelectionModel.SINGLE_TREE_SELECTION：每次只能选择一个 TreePath。

与 JList 操作类似，按下 Ctrl 辅助键，用于添加选中多个 JTree 节点；按下 Shift 辅助键，用于选择连续区域里的所有 JTree 节点。

JTree 提供了如下两个常用的添加监听器的方法。

- addTreeExpansionListener(TreeExpansionListener tel)：添加树节点展开/折叠事件的监听器。
- addTreeSelectionListener(TreeSelectionListener tsl)：添加树节点选择事件的监听器。

下面程序设置 JTree 只能选择单个 TreePath，并为节点选择事件添加事件监听器。

程序清单：codes\12\12.10\SelectJTree.java

```
public class SelectJTree
{
    JFrame jf = new JFrame("监听树的选择事件");
    JTree tree;
    // 定义几个初始节点
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("中国");
    DefaultMutableTreeNode guangdong = new DefaultMutableTreeNode("广东");
    DefaultMutableTreeNode guangxi = new DefaultMutableTreeNode("广西");
    DefaultMutableTreeNode foshan = new DefaultMutableTreeNode("佛山");
    DefaultMutableTreeNode shantou = new DefaultMutableTreeNode("汕头");
    DefaultMutableTreeNode guilin = new DefaultMutableTreeNode("桂林");
    DefaultMutableTreeNode nanning = new DefaultMutableTreeNode("南宁");
    JTextArea eventTxt = new JTextArea(5 , 20);
    public void init()
    {
        // 通过 add()方法建立树节点之间的父子关系
        guangdong.add(foshan);
        guangdong.add(shantou);
        guangxi.add(guilin);
        guangxi.add(nanning);
        root.add(guangdong);
        root.add(guangxi);
        // 以根节点创建树
        tree = new JTree(root);
        // 设置只能选择一个 TreePath
        tree.getSelectionModel().setSelectionMode(
            TreeSelectionModel.SINGLE_TREE_SELECTION);
        // 添加监听树节点选择事件的监听器
        // 当 JTree 中被选择节点发生改变时，将触发该方法
        tree.addTreeSelectionListener(e -> {
            if (e.getOldLeadSelectionPath() != null)
                eventTxt.append("原选中的节点路径："
                + e.getOldLeadSelectionPath().toString() + "\n");
            eventTxt.append("新选中的节点路径："
                + e.getNewLeadSelectionPath().toString() + "\n");
        });        // 设置是否显示根节点的展开/折叠图标，默认是 false
        tree.setShowsRootHandles(true);
        // 设置根节点是否可见，默认是 true
        tree.setRootVisible(true);
        Box box = new Box(BoxLayout.X_AXIS);
        box.add(new JScrollPane(tree));
        box.add(new JScrollPane(eventTxt));
        jf.add(box);
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SelectJTree().init();
    }
}
```

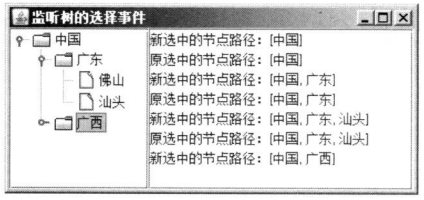


图 12.43 监听树的选择事件

上面程序中的第一行粗体字代码设置了该 JTree 对象采用 SINGLE_TREE_SELECTION 选择模式，即每次只能选中该 JTree 的一个 TreePath。第二段粗体字代码为该 JTree 添加了一个节点选择事件的监听器，当该 JTree 中被选择节点发生改变时，该监听器就会被触发。运行上面程序，会看到如图 12.43 所示的效果。

注意：

不要通过监听鼠标事件来监听所选节点的变化，因为 JTree 中节点的选择完全可以通过键盘来操作，不通过鼠标单击亦可。

12.10.4 使用 DefaultTreeCellRenderer 改变节点外观

对比图 12.38 和图 12.41 所示的两棵树，不难发现图 12.38 所示的树更美观，因为图 12.38 所示的树节点的图标非常丰富，而图 12.41 所示的树节点的图标太过于单一。

实际上，JTree 也可以改变树节点的外观，包括改变节点的图标、字体等，甚至可以自由绘制节点外观。为了改变树节点的外观，可以通过为树指定自己的 CellRenderer 来实现，JTree 默认使用 DefaultTreeCellRenderer 来绘制每个节点。通过查看 API 文档可以发现：DefaultTreeCellRenderer 是 JLabel 的子类，该 JLabel 包含了该节点的图标和文本。

改变树节点的外观样式，可以有如下三种方式。

- 使用 DefaultTreeCellRenderer 直接改变节点的外观，这种方式可以改变整棵树所有节点的字体、颜色和图标。
- 为 JTree 指定 DefaultTreeCellRenderer 的扩展类对象作为 JTree 的节点绘制器，该绘制器负责为不同节点使用不同的字体、颜色和图标。通常使用这种方式来改变节点的外观。
- 为 JTree 指定一个实现 TreeCellRenderer 接口的节点绘制器，该绘制器可以为不同的节点自由绘制任意内容，这是最复杂但最灵活的节点绘制器。

第一种方式最简单，但灵活性最差，因为它会改变整棵树所有节点的外观。在这种情况下，Jtree 的所有节点依然使用相同的图标，相当于整体替换了 Jtree 中节点的所有默认图标。用户指定的节点图标未必就比 JTree 默认的图标美观。

DefaultTreeCellRenderer 提供了如下几个方法来修改节点的外观。

- setBackgroundNonSelectionColor(Color newColor)：设置用于非选定节点的背景颜色。
- setBackgroundSelectionColor(Color newColor)：设置节点在选中状态下的背景颜色。
- setBorderSelectionColor(Color newColor)：设置选中状态下节点的边框颜色。
- setClosedIcon(Icon newIcon)：设置处于折叠状态下非叶子节点的图标。
- setFont(Font font)：设置节点文本的字体。
- setLeafIcon(Icon newIcon)：设置叶子节点的图标。
- setOpenIcon(Icon newIcon)：设置处于展开状态下非叶子节点的图标。
- setTextNonSelectionColor(Color newColor)：设置绘制非选中状态下节点文本的颜色。
- setTextSelectionColor(Color newColor)：设置绘制选中状态下节点文本的颜色。

下面程序直接使用 DefaultTreeCellRenderer 来改变树节点的外观。

程序清单：codes\12\12.10\ChangeAllCellRender.java

```
public class ChangeAllCellRender
{
    JFrame jf = new JFrame("改变所有节点的外观");
    JTree tree;
    // 定义几个初始节点
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("中国");
    DefaultMutableTreeNode guangdong = new DefaultMutableTreeNode("广东");
    DefaultMutableTreeNode guangxi = new DefaultMutableTreeNode("广西");
    DefaultMutableTreeNode foshan = new DefaultMutableTreeNode("佛山");
    DefaultMutableTreeNode shantou = new DefaultMutableTreeNode("汕头");
    DefaultMutableTreeNode guilin = new DefaultMutableTreeNode("桂林");
    DefaultMutableTreeNode nanning = new DefaultMutableTreeNode("南宁");
    public void init()
    {
        // 通过 add()方法建立树节点之间的父子关系
        guangdong.add(foshan);
        guangdong.add(shantou);
        guangxi.add(guilin);
        guangxi.add(nanning);
        root.add(guangdong);
        root.add(guangxi);
        // 以根节点创建树
```

```
        tree = new JTree(root);
        // 创建一个 DefaultTreeCellRenderer 对象
        DefaultTreeCellRenderer cellRender = new DefaultTreeCellRenderer();
        // 设置非选定节点的背景颜色
        cellRender.setBackgroundNonSelectionColor(new
            Color(220 , 220 , 220));
        // 设置节点在选中状态下的背景颜色
        cellRender.setBackgroundSelectionColor(new Color(140 , 140, 140));
        // 设置选中状态下节点的边框颜色
        cellRender.setBorderSelectionColor(Color.BLACK);
        // 设置处于折叠状态下非叶子节点的图标
        cellRender.setClosedIcon(new ImageIcon("icon/close.gif"));
        // 设置节点文本的字体
        cellRender.setFont(new Font("SansSerif" , Font.BOLD , 16));
        // 设置叶子节点的图标
        cellRender.setLeafIcon(new ImageIcon("icon/leaf.png"));
        // 设置处于展开状态下非叶子节点的图标
        cellRender.setOpenIcon(new ImageIcon("icon/open.gif"));
        // 设置绘制非选中状态下节点文本的颜色
        cellRender.setTextNonSelectionColor(new Color(255 , 0 , 0));
        // 设置绘制选中状态下节点文本的颜色
        cellRender.setTextSelectionColor(new Color(0 , 0 , 255));
        tree.setCellRenderer(cellRender);
        // 设置是否显示根节点的展开/折叠图标，默认是 false
        tree.setShowsRootHandles(true);
        // 设置节点是否可见，默认是 true
        tree.setRootVisible(true);
        jf.add(new JScrollPane(tree));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ChangeAllCellRender().init();
    }
}
```

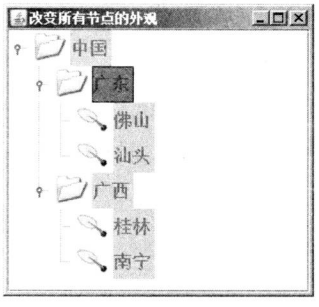


图 12.44 直接使用 DefaultTreeCellRenderer 改变所有节点的外观效果

上面程序中的粗体字代码创建了一个 DefaultTreeCellRenderer 对象，并通过该对象改变了 Jtree 中所有节点的字体、颜色和图标。运行上面程序，会看到如图 12.44 所示的效果。

从图 12.44 中可以看出，Jtree 中的所有节点全部被改变了，相当于完全替代了 Jtree 中所有节点的默认图标、字体和颜色。但所有的叶子节点依然保持相同的外观，所有的非叶子节点也保持相同的外观。这种改变依然不能满足更复杂的需求，例如，如果需要不同类型的节点呈现出不同的外观，则不能直接使用 DefaultTreeCellRenderer 来改变节点的外观，可以采用扩展 DefaultTreeCellRenderer 的方式来实现该需求。

提示：

不要试图通过 TreeCellRenderer 来改变表示节点展开/折叠的图标，因为该图标是由 Metal 风格决定的。如果需要改变该图标，则可以考虑改变该 JTree 的外观风格。

12.10.5 扩展 DefaultTreeCellRenderer 改变节点外观

DefaultTreeCellRenderer 实现类实现了 TreeCellRenderer 接口，该接口里只有一个用于绘制节点内容的方法：getTreeCellRendererComponent()，该方法负责绘制 JTree 节点。如果读者还记得前面介绍的绘制 JList 的列表项外观的内容，应该对该方法非常熟悉——与 ListCellRenderer 接口类似的是，getTreeCellRendererComponent()方法返回一个 Component 对象，该对象就是 JTree 的节点组件。

DefaultTreeCellRenderer 类继承了 JLabel，实现 getTreeCellRendererComponent()方法时返回 this，即返回一个特殊的 JLabel 对象。如果需要根据节点内容来改变节点的外观，则可以再次扩展 DefaultTreeCellRenderer 类，并再次重写它提供的 getTreeCellRendererComponent()方法。

下面程序模拟了一个数据库对象导航树，程序可以根据节点的类型来绘制节点的图标。在本程序中为了给每个节点指定节点类型，程序不再使用 String 作为节点数据，而是使用 NodeData 来封装节点数据，并重写了 NodeData 的 toString()方法。

注意：

使用 Object 类型的对象来创建 TreeNode 对象时，DefaultTreeCellRenderer 默认使用该对象的 toString()方法返回的字符串作为该节点的标签。

程序清单：codes\12\12.10\ExtendsDefaultTreeCellRenderer.java

```
public class ExtendsDefaultTreeCellRenderer
{
    JFrame jf = new JFrame("根据节点类型定义图标");
    JTree tree;
    // 定义几个初始节点
    DefaultMutableTreeNode root = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.ROOT , "数据库导航"));
    DefaultMutableTreeNode salaryDb = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.DATABASE , "公司工资数据库"));
    DefaultMutableTreeNode customerDb = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.DATABASE , "公司客户数据库"));
    // 定义 salaryDb 的两个子节点
    DefaultMutableTreeNode employee = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.TABLE , "员工表"));
    DefaultMutableTreeNode attend = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.TABLE , "考勤表"));
    // 定义 customerDb 的一个子节点
    DefaultMutableTreeNode contact = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.TABLE , "联系方式表"));
    // 定义 employee 的三个子节点
    DefaultMutableTreeNode id = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.INDEX , "员工 ID"));
    DefaultMutableTreeNode name = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.COLUMN , "姓名"));
    DefaultMutableTreeNode gender = new DefaultMutableTreeNode(
        new NodeData(DBObjectType.COLUMN , "性别"));
    public void init()
    {
        // 通过 add()方法建立树节点之间的父子关系
        root.add(salaryDb);
        root.add(customerDb);
        salaryDb.add(employee);
        salaryDb.add(attend);
        customerDb.add(contact);
        employee.add(id);
        employee.add(name);
        employee.add(gender);
        // 以根节点创建树
        tree = new JTree(root);
        // 设置该 JTree 使用自定义的节点绘制器
        tree.setCellRenderer(new MyRenderer());
        // 设置是否显示根节点的展开/折叠图标，默认是 false
        tree.setShowsRootHandles(true);
        // 设置节点是否可见，默认是 true
        tree.setRootVisible(true);
        try
        {
            // 设置使用 Windows 风格外观
            UIManager.setLookAndFeel("com.sun.java.swing.plaf."
                + "windows.WindowsLookAndFeel");
        }
```

```
        catch (Exception ex){}
        // 更新 JTree 的 UI 外观
        SwingUtilities.updateComponentTreeUI(tree);
        jf.add(new JScrollPane(tree));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new ExtendsDefaultTreeCellRenderer().init();
    }
}
// 定义一个 NodeData 类，用于封装节点数据
class NodeData
{
    public int nodeType;
    public String nodeData;
    public NodeData(int nodeType , String nodeData)
    {
        this.nodeType = nodeType;
        this.nodeData = nodeData;
    }
    public String toString()
    {
        return nodeData;
    }
}
// 定义一个接口，该接口里包含数据库对象类型的常量
interface DBObjectType
{
    int ROOT = 0;
    int DATABASE = 1;
    int TABLE = 2;
    int COLUMN = 3;
    int INDEX = 4;
}
class MyRenderer extends DefaultTreeCellRenderer
{
    // 初始化 5 个图标
    ImageIcon rootIcon = new ImageIcon("icon/root.gif");
    ImageIcon databaseIcon = new ImageIcon("icon/database.gif");
    ImageIcon tableIcon = new ImageIcon("icon/table.gif");
    ImageIcon columnIcon = new ImageIcon("icon/column.gif");
    ImageIcon indexIcon = new ImageIcon("icon/index.gif");
    public Component getTreeCellRendererComponent(JTree tree
        , Object value , boolean sel , boolean expanded
        , boolean leaf , int row , boolean hasFocus)
    {
        // 执行父类默认的节点绘制操作
        super.getTreeCellRendererComponent(tree , value
            , sel, expanded , leaf , row , hasFocus);
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
        NodeData data = (NodeData)node.getUserObject();
        // 根据数据节点里的 nodeType 数据决定节点图标
        ImageIcon icon = null;
        switch(data.nodeType)
        {
            case DBObjectType.ROOT:
                icon = rootIcon;
                break;
            case DBObjectType.DATABASE:
                icon = databaseIcon;
                break;
            case DBObjectType.TABLE:
                icon = tableIcon;
                break;
            case DBObjectType.COLUMN:
                icon = columnIcon;
                break;
```

```
            case DBObjectType.INDEX:
                icon = indexIcon;
                break;
        }
        // 改变图标
        this.setIcon(icon);
        return this;
    }
}
```

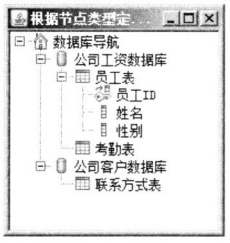


图 12.45　根据节点类型绘制节点图标

程序中的粗体字代码强制 JTree 使用自定义的节点绘制器：MyRenderer，该节点绘制器继承了 DefaultTreeCellRenderer 类，并重写了 getTreeCellRendererComponent()方法。该节点绘制器重写该节点时根据节点的 nodeType 属性改变其图标。运行上面程序，会看到如图 12.45 所示的效果。

从图 12.45 中可以看出，JTree 中表示节点展开、折叠的图标已经改为了“+”和“-”，这是因为本程序强制 JTree 使用了 Windows 风格。

12.10.6　实现 TreeCellRenderer 改变节点外观

这种方式是最灵活的方式，程序实现 TreeCellRenderer 接口时同样需要实现 getTreeCellRendererComponent()方法，该方法可以返回任意类型的组件，该组件将作为 JTree 的节点。通过这种方式可以最大程度地改变 JTree 的节点外观。

与前面实现 ListCellRenderer 接口类似的是，本实例程序同样通过扩展 JPanel 来实现 TreeCellRenderer，实现 TreeCellRenderer 的方式与前面实现 ListCellRenderer 的方式基本相似，所以读者将会看到一个完全不同的 JTree。

程序清单：codes\12\12.10\CustomTreeNode.java

```
public class CustomTreeNode
{
    JFrame jf = new JFrame("定制树的节点");
    JTree tree;
    // 定义几个初始节点
    DefaultMutableTreeNode friends = new DefaultMutableTreeNode("我的好友");
    DefaultMutableTreeNode qingzhao = new DefaultMutableTreeNode("李清照");
    DefaultMutableTreeNode suge = new DefaultMutableTreeNode("苏格拉底");
    DefaultMutableTreeNode libai = new DefaultMutableTreeNode("李白");
    DefaultMutableTreeNode nongyu = new DefaultMutableTreeNode("弄玉");
    DefaultMutableTreeNode hutou = new DefaultMutableTreeNode("虎头");
    public void init()
    {
        // 通过 add()方法建立树节点之间的父子关系
        friends.add(qingzhao);
        friends.add(suge);
        friends.add(libai);
        friends.add(nongyu);
        friends.add(hutou);
        // 以根节点创建树
        tree = new JTree(friends);
        // 设置是否显示根节点的展开/折叠图标，默认是 false
        tree.setShowsRootHandles(true);
        // 设置节点是否可见，默认是 true
        tree.setRootVisible(true);
        // 设置使用定制的节点绘制器
        tree.setCellRenderer(new ImageCellRenderer());
        jf.add(new JScrollPane(tree));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
```

```
    public static void main(String[] args)
    {
        new CustomTreeNode().init();
    }
}
// 实现自己的节点绘制器
class ImageCellRenderer extends JPanel implements TreeCellRenderer
{
    private ImageIcon icon;
    private String name;
    // 定义绘制单元格时的背景色
    private Color background;
    // 定义绘制单元格时的前景色
    private Color foreground;
    public Component getTreeCellRendererComponent(JTree tree
        , Object value , boolean sel , boolean expanded
        , boolean leaf , int row , boolean hasFocus)
    {
        icon = new ImageIcon("icon/" + value + ".gif");
        name = value.toString();
        background = hasFocus ? new Color(140 , 200 ,235)
            : new Color(255 , 255 , 255);
        foreground = hasFocus ? new Color(255 , 255 ,3)
            : new Color(0 , 0 , 0);
        // 返回该 JPanel 对象作为单元格绘制器
        return this;
    }
    // 重写 paintComponent 方法，改变 JPanel 的外观
    public void paintComponent(Graphics g)
    {
        int imageWidth = icon.getImage().getWidth(null);
        int imageHeight = icon.getImage().getHeight(null);
        g.setColor(background);
        g.fillRect(0 , 0 , getWidth() , getHeight());
        g.setColor(foreground);
        // 绘制好友图标
        g.drawImage(icon.getImage() , getWidth() / 2
            - imageWidth / 2 , 10 , null);
        g.setFont(new Font("SansSerif" , Font.BOLD , 18));
        // 绘制好友用户名
        g.drawString(name, getWidth() / 2
            - name.length() * 10 , imageHeight + 30 );
    }
    // 通过该方法来设置该 ImageCellRenderer 的最佳大小
    public Dimension getPreferredSize()
    {
        return new Dimension(80, 80);
    }
}
```

上面程序中的粗体字代码设置 JTree 对象使用定制的节点绘制器：ImageCellRenderer，该节点绘制器实现了 TreeCellRenderer 接口的 getTreeCellRendererComponent()方法，该方法返回 this，也就是一个特殊的 JPanel 对象，这个特殊的 JPanel 重写了 paintComponent()方法，重新绘制了 JPanel 的外观——根据节点数据来绘制图标和文本。运行上面程序，会看到如图 12.46 所示的树。

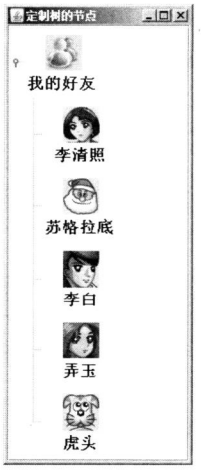


图 12.46 自行定制树节点的外观

这看上去似乎不太像一棵树，但可从每个节点前的连接线、表示节点的展开/折叠的图标中看出这依然是一棵树。

12.11 使用 JTable 和 TableModel 创建表格

表格也是 GUI 程序中常用的组件，表格是一个由多行、多列组成的二维显示区。Swing 的 JTable 以及相关类提供了这种表格支持，通

过使用 JTable 以及相关类，程序既可以使用简单的代码创建出表格来显示二维数据，也可以开发出功能丰富的表格，还可以为表格定制各种显示外观、编辑特性。

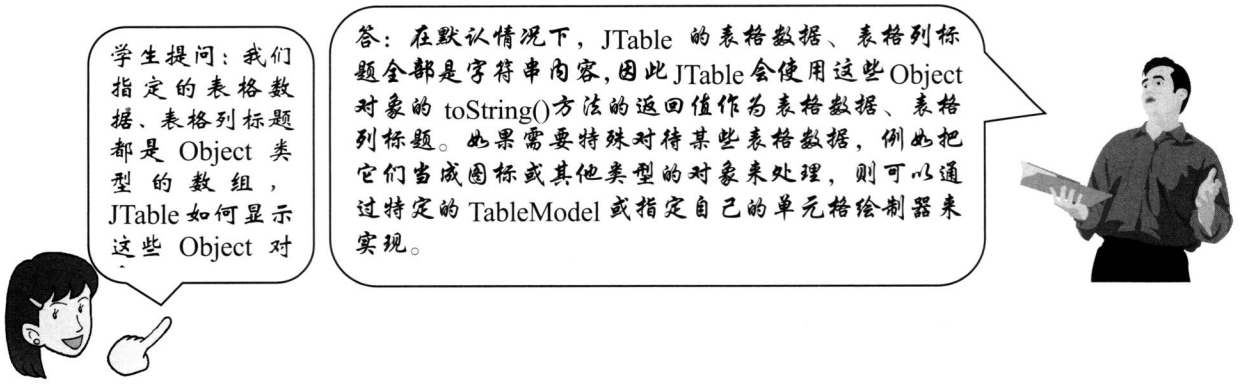
12.11.1 创建表格

使用 JTable 来创建表格是非常容易的事情，JTable 可以把一个二维数据包装成一个表格，这个二维数据既可以是一个二维数组，也可以是集合元素为 Vector 的 Vector 对象（Vector 里包含 Vector 形成二维数据）。除此之外，为了给该表格的每一列指定列标题，还需要传入一个一维数据作为列标题，这个一维数据既可以是一维数组，也可以是 Vector 对象。下面程序使用二维数组和一维数组来创建一个简单表格。

程序清单：codes\12\12.11\SimpleTable.java

```
public class SimpleTable
{
    JFrame jf = new JFrame("简单表格");
    JTable table;
    // 定义二维数组作为表格数据
    Object[][] tableData =
    {
        new Object[]{"李清照" , 29 , "女"},
        new Object[]{"苏格拉底", 56 , "男"},
        new Object[]{"李白", 35 , "男"},
        new Object[]{"弄玉", 18 , "女"},
        new Object[]{"虎头" , 2 , "男"}
    };
    // 定义一维数据作为列标题
    Object[] columnTitle = {"姓名" , "年龄" , "性别"};
    public void init()
    {
        // 以二维数组和一维数组来创建一个 JTable 对象
        table = new JTable(tableData , columnTitle);
        // 将 JTable 对象放在 JScrollPane 中
        // 并将该 JScrollPane 放在窗口中显示出来
        jf.add(new JScrollPane(table));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SimpleTable().init();
    }
}
```

上面程序中的粗体字代码创建了两个 Object 数组，第一个二维数组作为 JTable 的数据，第二个一维数组作为 JTable 的列标题。创建二维数组时利用了 JDK 1.5 提供的自动装箱功能——虽然直接指定的数组元素是 int 类型的整数，但系统会将它包装成 Integer 对象。



在默认情况下，JTable 的所有单元格、列标题显示的全部是字符串内容。除此之外，通常应该将 JTable

对象放在 JScrollPane 容器中，由 JScrollPane 为 JTable 提供 ViewPort。

注意：

通常总是会把 JTable 对象放在 JScrollPane 中显示，使用 JScrollPane 来包装 JTable 不仅可以为 JTable 增加滚动条，而且可以让 JTable 的列标题显示出来；如果不把 JTable 放在 JScrollPane 中显示，JTable 默认不会显示列标题。

运行上面程序，会看到如图 12.47 所示的简单表格。

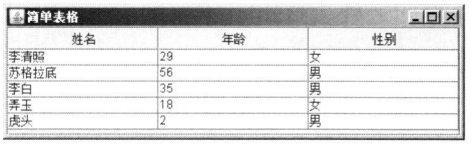


图 12.47 简单表格

虽然生成如图 12.47 所示表格的代码非常简单，但这个表格已经表现出丰富的功能。该表格具有如下几个功能。

- 当表格高度不足以显示所有的数据行时，该表格会自动显示滚动条。
- 当把鼠标移动到两列之间的分界符时，鼠标形状会变成可调整大小的形状，表明用户可以自由调整表格列的大小。
- 当在表格列上按下鼠标并拖动时，可以将表格的整列拖动到其他位置。
- 当单击某一个单元格时，系统会自动选中该单元格所在的行。
- 当双击某一个单元格时，系统会自动进入该单元格的修改状态。

运行 SimpleTable.java 程序，当拖动两列分界线来调整某列的列宽时，将看到该列后面的所有列的列宽都会发生相应的改变，但该列前面的所有列的列宽都不会发生改变，整个表格的宽度不会发生改变。

JTable 提供了一个 setAutoResizeMode()方法来控制这种调整方式，该方法可以接收如下几个值。

- JTable.AUTO_RESIZE_OFF：关闭 JTable 的自动调整功能，当调整某一列的宽度时，其他列的宽度不会发生改变，只有表格的宽度会随之改变。
- JTable.AUTO_RESIZE_NEXT_COLUMN：只调整下一列的宽度，其他列及表格的宽度不会发生改变。
- JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS：平均调整当前列后面所有列的宽度，当前列的前面所有列及表格的宽度都不会发生变化，这是默认的调整方式。
- JTable.AUTO_RESIZE_LAST_COLUMN：只调整最后一列的宽度，其他列及表格的宽度不会发生改变。
- JTable.AUTO_RESIZE_ALL_COLUMNS：平均调整表格中所有列的宽度，表格的宽度不会发生改变。

JTable 默认采用平均调整当前列后面所有列的宽度的方式，这种方式允许用户从左到右依次调整每一列的宽度，以达到最好的显示效果。

注意：

尽量避免使用平均调整表格中所有列的宽度的方式，这种方式将会导致用户调整某一列时，其余所有列都随之发生改变，从而使得用户很难把每一列的宽度都调整到具有最好的显示效果。

如果需要精确控制每一列的宽度，则可通过 TableColumn 对象来实现。JTable 使用 TableColumn 来表示表格中的每一列，JTable 中表格列的所有属性，如最佳宽度、是否可调整宽度、最小和最大宽度等都保存在该 TableColumn 中。此外，TableColumn 还允许为该列指定特定的单元格绘制器和单元格编辑器（这些内容将在后面讲解）。TableColumn 具有如下方法。

- setMaxWidth(int maxWidth)：设置该列的最大宽度。如果指定的 maxWidth 小于该列的最小宽度，则 maxWidth 被设置成最小宽度。

- setMinWidth(int minWidth)：设置该列的最小宽度。
- setPreferredWidth(int preferredWidth)：设置该列的最佳宽度。
- setResizable(boolean isResizable)：设置是否可以调整该列的宽度。
- sizeWidthToFit()：调整该列的宽度，以适合其标题单元格的宽度。

在默认情况下，当用户单击 JTable 的任意一个单元格时，系统默认会选中该单元格所在行的整行，也就是说，JTable 表格默认的选择单元是行。当然也可通过 JTable 提供的 setRowSelectionAllowed()方法来改变这种设置，如果为该方法传入 false 参数，则可以关闭这种每次选择一行的方式。

除此之外，JTable 还提供了一个 setColumnSelectionAllowed()方法，该方法用于控制选择单元是否是列，如果为该方法传入 true 参数，则当用户单击某个单元格时，系统会选中该单元格所在的列。

如果同时调用 setColumnSelectionAllowed(true)和 setRowSelectionAllowed(true)方法，则该表格的选择单元是单元格。实际上，同时调用这两个方法相当于调用 setCellSelectionEnabled(true)方法。与此相反，如果调用 setCellSelectionEnabled(false)方法，则相当于同时调用 setColumnSelectionAllowed(false)和 setRowSelectionAllowed (false)方法，即用户无法选中该表格的任何地方。

与 JList、JTree 类似的是，JTable 使用了一个 ListSelectionModel 表示该表格的选择状态，程序可以通过 ListSelectionModel 来控制 JTable 的选择模式。JTable 的选择模式有如下三种。

- ListSelectionModel.MULTIPLE_INTERVAL_SELECTION：没有任何限制，可以选择表格中任何表格单元，这是默认的选择模式。通过 Shift 和 Ctrl 辅助键的帮助可以选择多个表格单元。
- ListSelectionModel.SINGLE_INTERVAL_SELECTION：选择单个连续区域，该选项可以选择多个表格单元，但多个表格单元之间必须是连续的。通过 Shift 辅助键的帮助来选择连续区域。
- ListSelectionModel.SINGLE_SELECTION：只能选择单个表格单元。

程序通常通过如下代码来改变 JTable 的选择模式。

```
// 设置该表格只能选中单个表格单元
table.getSelectionModel().setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

注意：

保存 JTable 选择状态的 model 类就是 ListSelectionModel，这并不是笔误。

下面程序示范了如何控制每列的宽度、控制表格的宽度调整模式、改变表格的选择单元和表格的选择模式。

程序清单：codes\12\12.11\AdjustingWidth.java

```
public class AdjustingWidth
{
    JFrame jf = new JFrame("调整表格列宽");
    JMenuBar menuBar = new JMenuBar();
    JMenu adjustModeMenu = new JMenu("调整方式");
    JMenu selectUnitMenu = new JMenu("选择单元");
    JMenu selectModeMenu = new JMenu("选择方式");
    // 定义5个单选框按钮，用以控制表格的宽度调整方式
    JRadioButtonMenuItem[] adjustModesItem = new JRadioButtonMenuItem[5];
    // 定义3个单选框按钮，用以控制表格的选择方式
    JRadioButtonMenuItem[] selectModesItem = new JRadioButtonMenuItem[3];
    JCheckBoxMenuItem rowsItem = new JCheckBoxMenuItem("选择行");
    JCheckBoxMenuItem columnsItem = new JCheckBoxMenuItem("选择列");
    JCheckBoxMenuItem cellsItem = new JCheckBoxMenuItem("选择单元格");
    ButtonGroup adjustBg = new ButtonGroup();
    ButtonGroup selectBg = new ButtonGroup();
    // 定义一个int类型的数组，用于保存表格所有的宽度调整方式
    int[] adjustModes = new int[]{
        JTable.AUTO_RESIZE_OFF
        , JTable.AUTO_RESIZE_NEXT_COLUMN
        , JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS
        , JTable.AUTO_RESIZE_LAST_COLUMN
```

```
    , JTable.AUTO_RESIZE_ALL_COLUMNS
};
int[] selectModes = new int[]{
    ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
    , ListSelectionModel.SINGLE_INTERVAL_SELECTION
    , ListSelectionModel.SINGLE_SELECTION
};
JTable table;
// 定义二维数组作为表格数据
Object[][] tableData =
{
    new Object[]{"李清照" , 29 , "女"},
    new Object[]{"苏格拉底", 56 , "男"},
    new Object[]{"李白", 35 , "男"},
    new Object[]{"弄玉", 18 , "女"},
    new Object[]{"虎头" , 2 , "男"}
};
// 定义一维数据作为列标题
Object[] columnTitle = {"姓名" , "年龄" , "性别"};
public void init()
{
    // 以二维数组和一维数组来创建一个 JTable 对象
    table = new JTable(tableData , columnTitle);
    // -----------为窗口安装设置表格调整方式的菜单-----------
    adjustModesItem[0] = new JRadioButtonMenuItem("只调整表格");
    adjustModesItem[1] = new JRadioButtonMenuItem("只调整下一列");
    adjustModesItem[2] = new JRadioButtonMenuItem("平均调整余下列");
    adjustModesItem[3] = new JRadioButtonMenuItem("只调整最后一列");
    adjustModesItem[4] = new JRadioButtonMenuItem("平均调整所有列");
    menuBar.add(adjustModeMenu);
    for (int i = 0; i < adjustModesItem.length ; i++)
    {
        // 默认选中第三个菜单项，即对应表格默认的宽度调整方式
        if (i == 2)
        {
            adjustModesItem[i].setSelected(true);
        }
        adjustBg.add(adjustModesItem[i]);
        adjustModeMenu.add(adjustModesItem[i]);
        final int index = i;
        // 为设置调整方式的菜单项添加监听器
        adjustModesItem[i].addActionListener(evt -> {
            // 如果当前菜单项处于选中状态，表格使用对应的调整方式
            if (adjustModesItem[index].isSelected())
            {
                table.setAutoResizeMode(adjustModes[index]); // ①
            }
        });
    }
    // -----------为窗口安装设置表格选择方式的菜单-----------
    selectModesItem[0] = new JRadioButtonMenuItem("无限制");
    selectModesItem[1] = new JRadioButtonMenuItem("单独的连续区");
    selectModesItem[2] = new JRadioButtonMenuItem("单选");
    menuBar.add(selectModeMenu);
    for (int i = 0; i < selectModesItem.length ; i++)
    {
        // 默认选中第一个菜单项，即对应表格默认的选择方式
        if (i == 0)
        {
        selectModesItem[i].setSelected(true);
        }
        selectBg.add(selectModesItem[i]);
        selectModeMenu.add(selectModesItem[i]);
        final int index = i;
        // 为设置选择方式的菜单项添加监听器
        selectModesItem[i].addActionListener(evt -> {
            // 如果当前菜单项处于选中状态，表格使用对应的选择方式
            if (selectModesItem[index].isSelected())
            {
                table.getSelectionModel().setSelectionMode
```

```
                    (selectModes[index]);     // ②
                }
            });
        }
        menuBar.add(selectUnitMenu);
        // -----为窗口安装设置表格选择单元的菜单-----
        rowsItem.setSelected(table.getRowSelectionAllowed());
        columnsItem.setSelected(table.getColumnSelectionAllowed());
        cellsItem.setSelected(table.getCellSelectionEnabled());
        rowsItem.addActionListener(event -> {
            table.clearSelection();
            // 如果该菜单项处于选中状态，设置表格的选择单元是行
            table.setRowSelectionAllowed(rowsItem.isSelected());
            // 如果选择行、选择列同时被选中，其实质是选择单元格
            cellsItem.setSelected(table.getCellSelectionEnabled());
        });
        selectUnitMenu.add(rowsItem);
        columnsItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                table.clearSelection();
                // 如果该菜单项处于选中状态，设置表格的选择单元是列
                table.setColumnSelectionAllowed(columnsItem.isSelected());
                // 如果选择行、选择列同时被选中，其实质是选择单元格
                cellsItem.setSelected(table.getCellSelectionEnabled());
            }
        });
        selectUnitMenu.add(columnsItem);
        cellsItem.addActionListener(event -> {
            table.clearSelection();
            // 如果该菜单项处于选中状态，设置表格的选择单元是单元格
            table.setCellSelectionEnabled(cellsItem.isSelected());
            // 该选项的改变会同时影响选择行、选择列两个菜单
            rowsItem.setSelected(table.getRowSelectionAllowed());
            columnsItem.setSelected(table.getColumnSelectionAllowed());
        });
        selectUnitMenu.add(cellsItem);
        jf.setJMenuBar(menuBar);
        // 分别获取表格的三个表格列，并设置三列的最小宽、最佳宽度和最大宽度
        TableColumn nameColumn = table.getColumn(columnTitle[0]);
        nameColumn.setMinWidth(40);
        TableColumn ageColumn = table.getColumn(columnTitle[1]);
        ageColumn.setPreferredWidth(50);
        TableColumn genderColumn = table.getColumn(columnTitle[2]);
        genderColumn.setMaxWidth(50);
        // 将 JTable 对象放在 JScrollPane 中，并将该 JScrollPane 放在窗口中显示出来
        jf.add(new JScrollPane(table));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new AdjustingWidth().init();
    }
}
```

上面程序中的①号粗体字代码根据单选钮菜单来设置表格的宽度调整方式，②号粗体字代码根据单选钮菜单来设置表格的选择模式，最后一段粗体字代码通过 JTable 的 getColumn()方法获取指定列，并分别设置三列的最佳、最大、最小宽度。如果选中“只调整表格”菜单项，并把第一列宽度拖大，将看到如图 12.48 所示的界面。

上面程序中还有三段粗体字代码，分别用于为三个复选框菜单添加监听器，根据复选框菜单的选中状态来决定表格的选择单元。如果程序采用 JTable 默认的选择模式（无限制的选择模式），并设置表格的选择单元是单元格，则可看到如图 12.49 所示的界面。

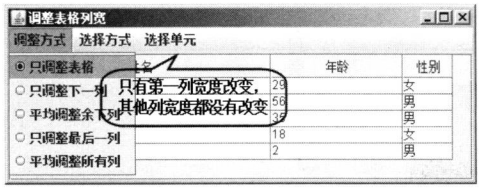


图 12.48　采用只调整表格宽度的方式

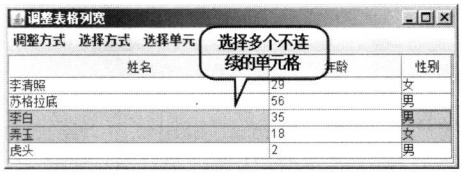


图 12.49　选择多个不连续的单元格

12.11.2　TableModel 和监听器

与 JList、JTree 类似的是，JTable 采用了 TableModel 来保存表格中的所有状态数据；与 ListModel 类似的是，TableModel 也不强制保存该表格显示的数据。虽然在前面程序中看到的是直接利用一个二维数组来创建 JTable 对象，但也可以通过 TableModel 对象来创建表格。如果需要利用 TableModel 来创建表格对象，则可以利用 Swing 提供的 AbstractTableModel 抽象类，该抽象类已经实现了 TableModel 接口里的大部分方法，程序只需要为该抽象类实现如下三个抽象方法即可。

- getColumnCount()：返回该 TableModel 对象的列数量。
- getRowCount()：返回该 TableModel 对象的行数量。
- getValueAt()：返回指定行、指定列的单元格值。

重写这三个方法后只是告诉 JTable 生成该表格所需的基本信息，如果想指定 JTable 生成表格的列名，还需要重写 getColumnName(int c)方法，该方法返回一个字符串，该字符串将作为第 c+1 列的列名。

在默认情况下，AbstractTableModel 的 boolean isCellEditable(int rowIndex, int columnIndex)方法返回 false，表明该表格的单元格处于不可编辑状态，如果想让用户直接修改单元格的内容，则需要重写该方法，并让该方法返回 true。重写该方法后，只实现了界面上单元格的可编辑，如果需要控制实际的编辑操作，还需要重写该类的 setValueAt(Object aValue, int rowIndex, int columnIndex)方法。

关于 TableModel 的典型应用就是用于封装 JDBC 编程里的 ResultSet，程序可以利用 TableModel 来封装数据库查询得到的结果集，然后使用 JTable 把该结果集显示出来。还可以允许用户直接编辑表格的单元格，当用户编辑完成后，程序将用户所做的修改写入数据库。下面程序简单实现了这种功能——当用户选择了指定的数据表后，程序将显示该数据表中的全部数据，用户可以直接在该表格内修改数据表的记录。

程序清单：codes\12\12.11\TableModelTest.java

```
public class TableModelTest
{
    JFrame jf = new JFrame("数据表管理工具");
    private JScrollPane scrollPane;
    private ResultSetTableModel model;
    // 用于装载数据表的 JComboBox
    private JComboBox<String> tableNames = new JComboBox<>();
    private JTextArea changeMsg = new JTextArea(4, 80);
    private ResultSet rs;
    private Connection conn;
    private Statement stmt;
    public void init()
    {
        // 为 JComboBox 添加事件监听器，当用户选择某个数据表时，触发该方法
        tableNames.addActionListener(event -> {
            try
            {
                // 如果装载 JTable 的 JScrollPane 不为空
                if (scrollPane != null)
                {
                    // 从主窗口中删除表格
                    jf.remove(scrollPane);
                }
                // 从 JComboBox 中取出用户试图管理的数据表的表名
                String tableName = (String) tableNames.getSelectedItem();
```

```
            // 如果结果集不为空，则关闭结果集
            if (rs != null)
            {
                rs.close();
            }
            String query = "select * from " + tableName;
            // 查询用户选择的数据表
            rs = stmt.executeQuery(query);
            // 使用查询到的 ResultSet 创建 TableModel 对象
            model = new ResultSetTableModel(rs);
            // 为 TableModel 添加监听器，监听用户的修改
            model.addTableModelListener(evt -> {
                int row = evt.getFirstRow();
                int column = evt.getColumn();
                changeMsg.append("修改的列:" + column
                    + ",修改的行:" + row + "修改后的值:"
                    + model.getValueAt(row , column));
            });
            // 使用 TableModel 创建 JTable，并将对应表格添加到窗口中
            JTable table = new JTable(model);
            scrollPane = new JScrollPane(table);
            jf.add(scrollPane, BorderLayout.CENTER);
            jf.validate();
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    });
    JPanel p = new JPanel();
    p.add(tableNames);
    jf.add(p, BorderLayout.NORTH);
    jf.add(new JScrollPane(changeMsg), BorderLayout.SOUTH);
    try
    {
        // 获取数据库连接
        conn = getConnection();
        // 获取数据库的 MetaData 对象
        DatabaseMetaData meta = conn.getMetaData();
        // 创建 Statement
        stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE
            , ResultSet.CONCUR_UPDATABLE);
        // 查询当前数据库的全部数据表
        ResultSet tables = meta.getTables(null, null, null
            , new String[] { "TABLE" });
        // 将全部数据表添加到 JComboBox 中
        while (tables.next())
        {
            tableNames.addItem(tables.getString(3));
        }
        tables.close();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    jf.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent event)
        {
            try
            {
                if (conn != null) conn.close();
            }
            catch (SQLException e)
            {
```

```
                    e.printStackTrace();
                }
            }
        });
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    private static Connection getConnection()
        throws SQLException, IOException , ClassNotFoundException
    {
        // 通过加载 conn.ini 文件来获取数据库连接的详细信息
        Properties props = new Properties();
        FileInputStream in = new FileInputStream("conn.ini");
        props.load(in);
        in.close();
        String drivers = props.getProperty("jdbc.drivers");
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.username");
        String password = props.getProperty("jdbc.password");
        // 加载数据库驱动
        Class.forName(drivers);
        // 取得数据库连接
        return DriverManager.getConnection(url, username, password);
    }
    public static void main(String[] args)
    {
        new TableModelTest().init();
    }
}
// 扩展 AbstractTableModel，用于将一个 ResultSet 包装成 TableModel
class ResultSetTableModel extends AbstractTableModel   // ①
{
    private ResultSet rs;
    private ResultSetMetaData rsmd;
    // 构造器，初始化 rs 和 rsmd 两个属性
    public ResultSetTableModel(ResultSet aResultSet)
    {
        rs = aResultSet;
        try
        {
            rsmd = rs.getMetaData();
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
    // 重写 getColumnName 方法，用于为该 TableModel 设置列名
    public String getColumnName(int c)
    {
        try
        {
            return rsmd.getColumnName(c + 1);
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            return "";
        }
    }
    // 重写 getColumnCount 方法，用于设置该 TableModel 的列数
    public int getColumnCount()
    {
        try
        {
            return rsmd.getColumnCount();
        }
        catch (SQLException e)
        {
```

```
            e.printStackTrace();
            return 0;
        }
    }
    // 重写 getValueAt 方法，用于设置该 TableModel 指定单元格的值
    public Object getValueAt(int r, int c)
    {
        try
        {
            rs.absolute(r + 1);
            return rs.getObject(c + 1);
        }
        catch(SQLException e)
        {
            e.printStackTrace();
            return null;
        }
    }
    // 重写 getRowCount 方法，用于设置该 TableModel 的行数
    public int getRowCount()
    {
        try
        {
            rs.last();
            return rs.getRow();
        }
        catch(SQLException e)
        {
            e.printStackTrace();
            return 0;
        }
    }
    // 重写 isCellEditable 返回 true，让每个单元格可编辑
    public boolean isCellEditable(int rowIndex, int columnIndex)
    {
        return true;
    }
    // 重写 setValueAt()方法，当用户编辑单元格时，将会触发该方法
    public void setValueAt(Object aValue , int row,int column)
    {
        try
        {
            // 结果集定位到对应的行数
            rs.absolute(row + 1);
            // 修改单元格对应的值
            rs.updateObject(column + 1 , aValue);
            // 提交修改
            rs.updateRow();
            // 触发单元格的修改事件
            fireTableCellUpdated(row, column);
        }
        catch (SQLException evt)
        {
            evt.printStackTrace();
        }
    }
}
```

上面程序的关键在于①号粗体字代码所扩展的 ResultSetTableModel 类，该类继承了 AbstractTableModel 父类，根据其 ResultSet 来重写 getColumnCount()、getRowCount()和 getValueAt()三个方法，从而允许该表格可以将该 ResultSet 里的所有记录显示出来。除此之外，该扩展类还重写了 isCellEditable()和 setValueAt()两个方法——重写前一个方法实现允许用户编辑单元格的功能，重写后一个方法实现当用户编辑单元格时将所做的修改同步到数据库的功能。

程序中的粗体字代码使用 ResultSet 创建了一个 TableModel 对象，并为该 TableModel 添加事件监听器，然后把该 TableModel 使用 JTable 显示出来。当用户修改该 JTable 对应表格里单元格的内容时，该监听器会检测到这种修改，并将这种修改信息通过下面的文本域显示出来。

提示：

上面程序大量使用了 JDBC 编程中的 JDBC 连接数据库、获取可更新的结果集、ResultSetMetaData、DatabaseMetaData 等知识，读者可能一时难以读懂，可以参考本书第 13 章的内容来阅读本程序。该程序的运行需要底层数据库的支持，所以读者应按第 13 章的内容正常安装 MySQL 数据库，并将 codes\12\12.11\路径下的 mysql.sql 脚本导入数据库，修改 conn.ini 文件中的数据库连接信息才可运行该程序。使用 JDBC 连接数据库还需要加载 JDBC 驱动，所以本章为运行该程序提供了一个 run.cmd 批处理文件，读者可以通过该文件来运行该程序。不要直接运行该程序，否则可能出现 java.lang.ClassNotFoundException: com.mysql.jdbc.Driver 异常。

运行上面程序，会看到如图 12.50 所示的界面。

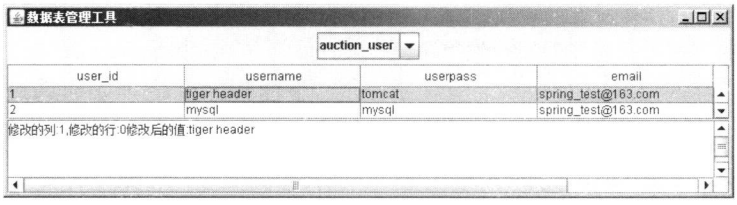


图 12.50 使用 JTable 管理数据表记录

从图 12.50 中可以看出，当修改指定单元格的记录时，添加在 TableModel 上的监听器就会被触发。当修改 JTable 单元格里的内容时，底层数据表里的记录也会做出相应的改变。

不仅用户可以扩展 AbstractTableModel 抽象类，Swing 本身也为 AbstractTableModel 提供了一个 DefaultTableModel 实现类，程序可以通过使用 DefaultTableModel 实现类来创建 JTable 对象。通过 DefaultTableModel 对象创建 JTable 对象后，就可以调用它提供的方法来添加数据行、插入数据行、删除数据行和移动数据行。DefaultTableModel 提供了如下几个方法来控制数据行操作。

- addColumn()：该方法用于为 TableModel 增加一列，该方法有三个重载的版本，实际上该方法只是将原来隐藏的数据列显示出来。
- addRow()：该方法用于为 TableModel 增加一行，该方法有两个重载的版本。
- insertRow()：该方法用于在 TableModel 的指定位置插入一行，该方法有两个重载的版本。
- removeRow(int row)：该方法用于删除 TableModel 中的指定行。
- moveRow(int start, int end, int to)：该方法用于移动 TableModel 中指定范围的数据行。

通过 DefaultTableModel 提供的这样几个方法，程序就可以动态地改变表格里的数据行。

提示：

Swing 为 TableModel 提供了两个实现类，其中一个是 DefaultTableModel，另一个是 JTable 的匿名内部类。如果直接使用二维数组来创建 JTable 对象，维护该 JTable 状态信息的 model 对象就是 JTable 匿名内部类的实例；当使用 Vector 来创建 JTable 对象时，维护该 JTable 状态信息的 model 对象就是 DefaultTableModel 实例。

12.11.3 TableColumnModel 和监听器

JTable 使用 TableColumnModel 来保存该表格所有数据列的状态数据，如果程序需要访问 JTable 的所有列状态信息，则可以通过获取该 JTable 的 TableColumnModel 来实现。TableColumnModel 提供了如下几个方法来增加、删除和移动数据列。

- addColumn(TableColumn aColumn)：该方法用于为 TableModel 添加一列。该方法主要用于将原来隐藏的数据列显示出来。
- moveColumn(int columnIndex, int newIndex)：该方法用于将指定列移动到其他位置。

- removeColumn(TableColumn column)：该方法用于从 TableModel 中删除指定列。实际上，该方法并未真正删除指定列，只是将该列在 TableColumnModel 中隐藏起来，使之不可见。

注意：

当调用 removeColumn()删除指定列之后，调用 TableColumnModel 的 getColumnCount()方法也会看到返回的列数减少了，看起来很像真正删除了该列。但使用 setValueAt()方法为该列设置值时，依然可以设置成功，这表明这些列依然是存在的。

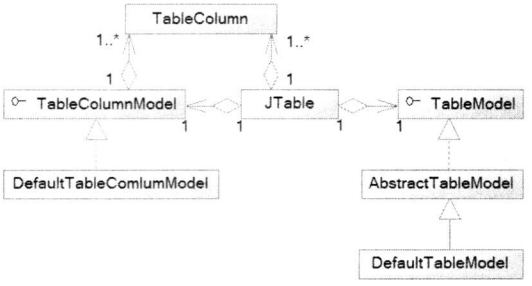


图 12.51　JTable 及其主要辅助类之间的关系

实际上，JTable 也提供了对应的方法来增加、删除和移动数据列，不过 JTable 的这些方法实际上还是需要委托给它所对应的 TableColumnModel 来完成。图 12.51 显示了 JTable 及其主要辅助类之间的关系。

下面程序示范了如何通过 DefaultTableModel 和 TableColumnModel 动态地改变表格的行、列。

程序清单：codes\12\12.11\DefaultTableModelTest.java

```
public class DefaultTableModelTest
{
    JFrame mainWin = new JFrame("管理数据行、数据列");
    final int COLUMN_COUNT = 5;
    DefaultTableModel model;
    JTable table;
    // 用于保存被隐藏列的 List 集合
    ArrayList<TableColumn> hiddenColumns = new ArrayList<>();
    public void init()
    {
        model = new DefaultTableModel(COLUMN_COUNT ,COLUMN_COUNT);
        for (int i = 0; i < COLUMN_COUNT ; i++ )
        {
            for (int j = 0; j < COLUMN_COUNT ; j++ )
            {
                model.setValueAt("老单元格值 " + i + " " + j , i , j);
            }
        }
        table = new JTable(model);
        mainWin.add(new JScrollPane(table), BorderLayout.CENTER);
        // 为窗口安装菜单
        JMenuBar menuBar = new JMenuBar();
        mainWin.setJMenuBar(menuBar);
        JMenu tableMenu = new JMenu("管理");
        menuBar.add(tableMenu);
        JMenuItem hideColumnsItem = new JMenuItem("隐藏选中列");
        hideColumnsItem.addActionListener(event -> {
            // 获取所有选中列的索引
            int[] selected = table.getSelectedColumns();
            TableColumnModel columnModel = table.getColumnModel();
            // 依次把每一个选中的列隐藏起来，并使用 List 保存这些列
            for (int i = selected.length - 1; i >= 0; i--)
            {
                TableColumn column = columnModel.getColumn(selected[i]);
                // 隐藏指定列
                table.removeColumn(column);
                // 把隐藏的列保存起来，确保以后可以显示出来
                hiddenColumns.add(column);
            }
        });
        tableMenu.add(hideColumnsItem);
        JMenuItem showColumnsItem = new JMenuItem("显示隐藏列");
        showColumnsItem.addActionListener(event -> {
            // 把所有隐藏的列依次显示出来
```

```
            for (TableColumn tc : hiddenColumns)
            {
                // 依次把所有隐藏的列显示出来
                table.addColumn(tc);
            }
            // 清空保存隐藏列的 List 集合
            hiddenColumns.clear();
        });
        tableMenu.add(showColumnsItem);
        JMenuItem addColumnItem = new JMenuItem("插入选中列");
        addColumnItem.addActionListener(event -> {
            // 获取所有选中列的索引
            int[] selected = table.getSelectedColumns();
            TableColumnModel columnModel = table.getColumnModel();
            // 依次把选中的列添加到 JTable 之后
            for (int i = selected.length - 1; i >= 0; i--)
            {
                TableColumn column = columnModel
                    .getColumn(selected[i]);
                table.addColumn(column);
            }
        });
        tableMenu.add(addColumnItem);
        JMenuItem addRowItem = new JMenuItem("增加行");
        addRowItem.addActionListener(event -> {
            // 创建一个 String 数组作为新增行的内容
            String[] newCells = new String[COLUMN_COUNT];
            for (int i = 0; i < newCells.length; i++)
            {
                newCells[i] = "新单元格值 " + model.getRowCount()
                    + " " + i;
            }
            // 向 TableModel 中新增一行
            model.addRow(newCells);
        });
        tableMenu.add(addRowItem);
        JMenuItem removeRowsItem = new  JMenuItem("删除选中行");
        removeRowsItem.addActionListener(event -> {
            // 获取所有选中行
            int[] selected = table.getSelectedRows();
            // 依次删除所有选中行
            for (int i = selected.length - 1; i >= 0; i--)
            {
                model.removeRow(selected[i]);
            }
        });
        tableMenu.add(removeRowsItem);
        mainWin.pack();
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new DefaultTableModelTest().init();
    }
}
```

上面程序中的粗体字代码部分就是程序控制隐藏列、显示隐藏列、增加数据行和删除数据行的代码。除此之外，程序还实现了一个功能：当用户选中某个数据列之后，还可以将该数据列添加到该表格的后面——但不要忘记了 add()方法的功能，它只是将已有的数据列显示出来，并不是真正添加数据列。运行上面程序，会看到如图 12.52 所示的界面。

从图 12.52 中可以看出，虽然程序新增了一列，但新增列的列名依然是 B，如果修改新增列内的单元格的值时，看到原来的 B 列的值也随之改变，由此可见，addColumn()方法只是将原有的列显示出来而已。程序还允许新增数据行，当执行 addRows()方法时需要传入数组或 Vector 参数，该参数里包含的多个数值将作为新增行的数据。

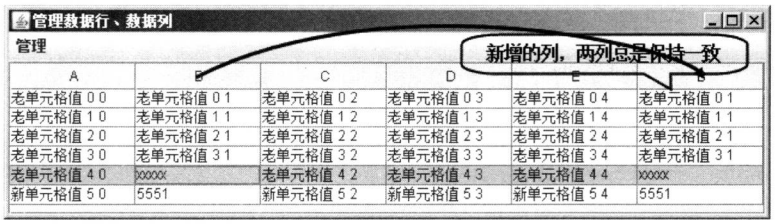


图 12.52　新增数据行、数据列的效果

如果程序需要监听 JTable 里列状态的改变，例如监听列的增加、删除、移动等改变，则必须使用该 JTable 所对应的 TableColumnModel 对象，该对象提供了一个 addColumnModelListener()方法来添加监听器，该监听器接口里包含如下几个方法。

- columnAdded(TableColumnModelEvent e)：当向 TableColumnModel 里添加数据列时将会触发该方法。
- columnMarginChanged(ChangeEvent e)：当由于页面距（Margin）的改变引起列状态改变时将会触发该方法。
- columnMoved(TableColumnModelEvent e)：当移动 TableColumnModel 里的数据列时将会触发该方法。
- columnRemoved(TableColumnModelEvent e)：当删除 TableColumnModel 里的数据列时将会触发该方法。
- columnSelectionChanged(ListSelectionEvent e)：当改变表格的选择模式时将会触发该方法。

但表格的数据列通常需要程序来控制增加、删除，用户操作通常无法直接为表格增加、删除数据列，所以使用监听器来监听 TableColumnModel 改变的情况比较少见。

12.11.4　实现排序

使用 JTable 实现的表格并没有实现根据指定列排序的功能，但开发者可以利用 AbstractTableModel 类来实现该功能。由于 TableModel 不强制要求保存表格里的数据，只要 TableModel 实现了 getValueAt()、getColumnCount()和 getRowCount()三个方法，JTable 就可以根据该 TableModel 生成表格。因此可以创建一个 SortableTableModel 实现类，它可以将原 TableModel 包装起来，并实现根据指定列排序的功能。

程序创建的 SortableTableModel 实现类会对原 TableModel 进行包装，但它实际上并不保存任何数据，它会把所有的方法实现委托给原 TableModel 完成。SortableTableModel 仅保存原 TableModel 里每行的行索引，当程序对 SortableTableModel 的指定列排序时，实际上仅仅对 SortableTableModel 里的行索引进行排序——这样造成的结果是：SortableTableModel 里的数据行的行索引与原 TableModel 里数据行的行索引不一致，所以对于 TableModel 的那些涉及行索引的方法都需要进行相应的转换。下面程序实现了 SortableTableModel 类，并使用该类来实现对表格根据指定列排序的功能。

程序清单：codes\12\12.11\SortTable.java

```
public class SortTable
{
    JFrame jf = new JFrame("可按列排序的表格");
    // 定义二维数组作为表格数据
    Object[][] tableData =
    {
        new Object[]{"李清照" , 29 , "女"},
        new Object[]{"苏格拉底", 56 , "男"},
        new Object[]{"李白", 35 , "男"},
        new Object[]{"弄玉", 18 , "女"},
        new Object[]{"虎头" , 2 , "男"}
    };
    // 定义一维数据作为列标题
    Object[] columnTitle = {"姓名" , "年龄" , "性别"};
    // 以二维数组和一维数组来创建一个 JTable 对象
    JTable table = new JTable(tableData , columnTitle);
```

```
        // 将原表格里的model包装成新的SortTableModel对象
        SortableTableModel sorterModel = new SortableTableModel(
            table.getModel());
        public void init()
        {
            // 使用包装后的SortableTableModel对象作为JTable的model对象
            table.setModel(sorterModel);
            // 为每列的列头增加鼠标监听器
            table.getTableHeader().addMouseListener(new MouseAdapter()
            {
                public void mouseClicked(MouseEvent event)    // ①
                {
                    // 如果单击次数小于2，即不是双击，直接返回
                    if (event.getClickCount() < 2)
                    {
                        return;
                    }
                    // 找出鼠标双击事件所在的列索引
                    int tableColumn = table.columnAtPoint(event.getPoint());
                    // 将JTable中的列索引转换成对应TableModel中的列索引
                    int modelColumn = table.convertColumnIndexToModel(tableColumn);
                    // 根据指定列进行排序
                    sorterModel.sort(modelColumn);
                }
            });
            // 将JTable对象放在JScrollPane中，并将该JScrollPane显示出来
            jf.add(new JScrollPane(table));
            jf.pack();
            jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            jf.setVisible(true);
        }
        public static void main(String[] args)
        {
            new SortTable().init();
        }
    }
    class SortableTableModel extends AbstractTableModel
    {
        private TableModel model;
        private int sortColumn;
        private Row[] rows;
        // 将一个已经存在的TableModel对象包装成SortableTableModel对象
        public SortableTableModel(TableModel m)
        {
            // 将被封装的TableModel传入
            model = m;
            rows = new Row[model.getRowCount()];
            // 将原TableModel中每行记录的索引使用Row数组保存起来
            for (int i = 0; i < rows.length; i++)
            {
                rows[i] = new Row(i);
            }
        }
        // 实现根据指定列进行排序
        public void sort(int c)
        {
            sortColumn = c;
            java.util.Arrays.sort(rows);
            fireTableDataChanged();
        }
        // 下面三个方法需要访问model中的数据，所以涉及本model中数据
        // 和被包装model数据中的索引转换，程序使用rows数组完成这种转换
        public Object getValueAt(int r, int c)
        {
            return model.getValueAt(rows[r].index, c);
        }
        public boolean isCellEditable(int r, int c)
        {
            return model.isCellEditable(rows[r].index, c);
        }
```

```
    public void setValueAt(Object aValue, int r, int c)
    {
        model.setValueAt(aValue, rows[r].index, c);
    }
    // 下面方法的实现把该 model 的方法委托给原封装的 model 来实现
    public int getRowCount()
    {
        return model.getRowCount();
    }
    public int getColumnCount()
    {
        return model.getColumnCount();
    }
    public String getColumnName(int c)
    {
        return model.getColumnName(c);
    }
    public Class getColumnClass(int c)
    {
        return model.getColumnClass(c);
    }
    // 定义一个 Row 类，该类用于封装 JTable 中的一行
    // 实际上它并不封装行数据，它只封装行索引
    private class Row implements Comparable<Row>
    {
        // 该 index 保存着被封装 Model 里每行记录的行索引
        public int index;
        public Row(int index)
        {
            this.index = index;
        }
        // 实现两行之间的大小比较
        public int compareTo(Row other)
        {
            Object a = model.getValueAt(index, sortColumn);
            Object b = model.getValueAt(other.index, sortColumn);
            if (a instanceof Comparable)
            {
                return ((Comparable)a).compareTo(b);
            }
            else
            {
                return a.toString().compareTo(b.toString());
            }
        }
    }
}
```

上面程序是在 SimpleTable 程序的基础上改变而来的，改变的部分就是增加了两行粗体字代码和①号粗体字代码块。其中粗体字代码负责把原 JTable 的 model 对象包装成 SortableTableModel 实例，并设置原 JTable 使用 SortableTableModel 实例作为对应的 model 对象；而①号粗体字代码部分则用于为该表格的列头增加鼠标监听器：当用鼠标双击指定列时，SortableTableModel 对象根据指定列进行排序。

注意：

程序中还使用了 convertColumnIndexToModel()方法把 JTable 中的列索引转换成 TableModel 中的列索引。这是因为 JTable 中的列允许用户随意拖动，因此可能造成 JTable 中的列索引与 TableModel 中的列索引不一致。

运行上面程序，并双击“年龄”列头，将看到如图 12.53 所示的排序效果。

实际上，上面程序的关键在于 SortableTableModel 类，该类使用 rows[]数组来保存原 TableModel 里的行索引。

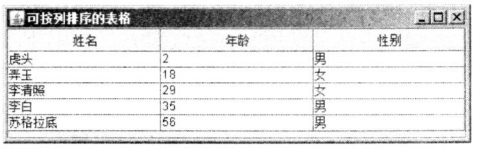


图 12.53 根据“年龄”列排序的效果

为了让程序可以对 rows[]数组元素根据指定列排序，程序使用了 Row 类来封装行索引，并实现了 compareTo()方法，该方法实现了根据指定列来比较两行大小的功能，从而允许程序根据指定列对 rows[]数组元素进行排序。

12.11.5 绘制单元格内容

前面看到的所有表格的单元格内容都是字符串，实际上表格的单元格内容也可以是更复杂的内容。JTable 使用 TableCellRenderer 绘制单元格，Swing 为该接口提供了一个实现类：DefaultTableCellRenderer，该单元格绘制器可以绘制如下三种类型的单元格值（根据其 TableModel 的 getColumnClass()方法来决定该单元格值的类型）。

- Icon：默认的单元格绘制器会把该类型的单元格值绘制成该 Icon 对象所代表的图标。
- Boolean：默认的单元格绘制器会把该类型的单元格值绘制成复选按钮。
- Object：默认的单元格绘制器在单元格内绘制出该对象的 toString()方法返回的字符串。

在默认情况下，如果程序直接使用二维数组或 Vector 来创建 JTable，程序将会使用 JTable 的匿名内部类或 DefaultTableModel 充当该表格的 model 对象，这两个 TableModel 的 getColumnClass()方法的返回值都是 Object。这意味着，即使该二维数组里值的类型是 Icon，但由于两个默认的 TableModel 实现类的 getColumnClass()方法总是返回 Object，这将导致默认的单元格绘制器把 Icon 值当成 Object 值处理——只是绘制出其 toString()方法返回的字符串。

为了让默认的单元格绘制器可以将 Icon 类型的值绘制成图标，把 Boolean 类型的值绘制成复选框，创建 JTable 时所使用的 TableModel 绝不能采用默认的 TableModel，必须采用扩展后的 TableModel 类，如下所示。

```
// 定义一个 DefaultTableModel 类的子类
class ExtendedTableModel extends DefaultTableModel
{
    ...
    // 重写 getColumnClass 方法，根据每列的第一个值来返回每列真实的数据类型
    public Class getColumnClass(int c)
    {
        return getValueAt(0 , c).getClass();
    }
}
```

提供了上面的 ExtendedTableModel 类之后，程序应该先创建 ExtendedTableModel 对象，再利用该对象来创建 JTable，这样就可以保证 JTable 的 model 对象的 getColumnClass()方法会返回每列真实的数据类型，默认的单元格绘制器就会将 Icon 类型的单元格值绘制成图标，将 Boolean 类型的单元格值绘制成复选框。

如果希望程序采用自己定制的单元格绘制器，则必须实现自己的单元格绘制器，单元格绘制器必须实现 TableCellRenderer 接口。与前面的 TreeCellRenderer 接口完全相似，该接口里也只包含一个 getTableCellRendererComponent()方法，该方法返回的 Component 将会作为指定单元格绘制的组件。

提示：

Swing 提供了一致的编程模型，不管是 JList、JTree 还是 JTable，它们所使用的单元格绘制器都有一致的编程模型，分别需要扩展 ListCellRenderer、TreeCellRenderer 或 TableCellRenderer，扩展这三个基类时都需要重写 getXxxCellRendererComponent()方法，该方法的返回值将作为被绘制的组件。

一旦实现了自己的单元格绘制器之后，还必须将该单元格绘制器安装到指定的 JTable 对象上，为指定的 JTable 对象安装单元格绘制器有如下两种方式。

- 局部方式（列级）：调用 TableColumn 的 setCellRenderer()方法为指定列安装指定的单元格绘制器。
- 全局方式（表级）：调用 JTable 的 setDefaultRenderer()方法为指定的 JTable 对象安装单元格绘制器。setDefaultRenderer()方法需要传入两个参数，即列类型和单元格绘制器，表明指定类型的数

据列才会使用该单元格绘制器。

注意：

当某一列既符合全局绘制器的规则，又符合局部绘制器的规则时，局部绘制器将会负责绘制该单元格，全局绘制器不会产生任何作用。除此之外，TableColumn 还包含了一个 setHeaderRenderer()方法，该方法可以为指定列的列头安装单元格绘制器。

下面程序提供了一个 ExtendedTableModel 类，该类扩展了 DefaultTableModel，重写了父类的 getColumnClass()方法，该方法根据每列的第一个值来决定该列的数据类型；下面程序还提供了一个定制的单元格绘制器，它使用图标来形象地表明每个好友的性别。

程序清单：codes\12\12.11\TableCellRendererTest.java

```
public class TableCellRendererTest
{
    JFrame jf = new JFrame("使用单元格绘制器");
    JTable table;
    // 定义二维数组作为表格数据
    Object[][] tableData =
    {
        new Object[]{"李清照" , 29 , "女"
            , new ImageIcon("icon/3.gif") , true},
        new Object[]{"苏格拉底", 56 , "男"
            , new ImageIcon("icon/1.gif") , false},
        new Object[]{"李白", 35 , "男"
            , new ImageIcon("icon/4.gif") , true},
        new Object[]{"弄玉", 18 , "女"
            , new ImageIcon("icon/2.gif") , true},
        new Object[]{"虎头" , 2 , "男"
            , new ImageIcon("icon/5.gif") , false}
    };
    // 定义一维数据作为列标题
    String[] columnTitle = {"姓名" , "年龄" , "性别"
        , "主头像" , "是否中国人"};
    public void init()
    {
        // 以二维数组和一维数组来创建一个 ExtendedTableModel 对象
        ExtendedTableModel model = new ExtendedTableModel(columnTitle
            , tableData);
        // 以 ExtendedTableModel 来创建 JTable
        table = new JTable( model);
        table.setRowSelectionAllowed(false);
        table.setRowHeight(40);
        // 获取第三列
        TableColumn lastColumn = table.getColumnModel().getColumn(2);
        // 对第三列采用自定义的单元格绘制器
        lastColumn.setCellRenderer(new GenderTableCellRenderer());
        // 将 JTable 对象放在 JScrollPane 中，并将该 JScrollPane 显示出来
        jf.add(new JScrollPane(table));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TableCellRendererTest().init();
    }
}
class ExtendedTableModel extends DefaultTableModel
{
    // 重新提供一个构造器，该构造器的实现委托给 DefaultTableModel 父类
    public ExtendedTableModel(String[] columnNames , Object[][] cells)
    {
        super(cells , columnNames);
    }
```

```
    // 重写 getColumnClass 方法，根据每列的第一个值来返回其真实的数据类型
    public Class getColumnClass(int c)
    {
        return getValueAt(0 , c).getClass();
    }
}
// 定义自定义的单元格绘制器
class GenderTableCellRenderer extends JPanel
    implements TableCellRenderer
{
    private String cellValue;
    // 定义图标的宽度和高度
    final int ICON_WIDTH = 23;
    final int ICON_HEIGHT = 21;
    public Component getTableCellRendererComponent(JTable table
        , Object value , boolean isSelected , boolean hasFocus
        , int row , int column)
    {
        cellValue = (String)value;
        // 设置选中状态下绘制边框
        if (hasFocus)
        {
            setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
        }
        else
        {
            setBorder(null);
        }
        return this;
    }
    // 重写 paint()方法，负责绘制该单元格内容
    public void paint(Graphics g)
    {
        // 如果表格值为"男"或"male"，则绘制一个男性图标
        if (cellValue.equalsIgnoreCase("男")
            || cellValue.equalsIgnoreCase("male"))
        {
            drawImage(g , new ImageIcon("icon/male.gif").getImage());
        }
        // 如果表格值为"女"或"female"，则绘制一个女性图标
        if (cellValue.equalsIgnoreCase("女")
            || cellValue.equalsIgnoreCase("female"))
        {
            drawImage(g , new ImageIcon("icon/female.gif").getImage());
        }
    }
    // 绘制图标的方法
    private void drawImage(Graphics g , Image image)
    {
        g.drawImage(image, (getWidth() - ICON_WIDTH ) / 2
            , (getHeight() - ICON_HEIGHT) / 2 , null);
    }
}
```

上面程序中没有直接使用二维数组和一维数组来创建 JTable 对象，而是采用 ExtendedTableModel 对象来创建 JTable 对象（如第一段粗体字代码所示）。ExtendedTableModel 类重写了父类的 getColumnClass()方法，该方法将会根据每列实际的值来返回该列的类型（如第二段粗体字代码所示）。

程序提供了一个 GenderTableCellRenderer 类，该类实现了 TableCellRenderer 接口，可以作为单元格绘制器使用。该类继承了 JPanel 容器，重写 getTableCellRendererComponent()方法时返回 this，这表明它会使用 JPanel 对象作为单元格绘制器。

提示：

读者可以将 ExtendedTableModel 补充得更加完整——主要是将 DefaultTableModel 中的几个构造器重新暴露出来，以后程序中可以使用 ExtendedTableModel 类作为 JTable 的 model 类，这样创建的 JTable 就可以将 Icon 列、Boolean 列绘制成图标和复选框。

运行上面程序，会看到如图 12.54 所示的效果。

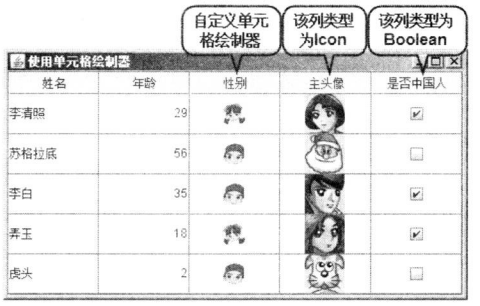


图 12.54　重写 getColumnClass()方法和定制单元格绘制器

12.11.6　编辑单元格内容

如果用户双击 JTable 表格的指定单元格，系统将会开始编辑该单元格的内容。在默认情况下，系统会使用文本框来编辑该单元格的内容，包括如图 12.54 所示表格的图标单元格。与此类似的是，如果用户双击 JTree 的节点，默认也会采用文本框来编辑节点的内容。

但如果单元格内容不是文字内容，而是如图 12.54 所示的图形类型时，用户当然不希望使用文本编辑器来编辑该单元格的内容，因为这种编辑方式非常不直观，用户体验相当差。为了避免这种情况，可以实现自己的单元格编辑器，从而可以给用户提供更好的操作界面。

实现 JTable 的单元格编辑器应该实现 TableCellEditor 接口，实现 JTree 的节点编辑器需要实现 TreeCellEditor 接口，这两个接口有非常紧密的联系。它们有一个共同的父接口：CellEditor；而且它们有一个共同的实现类：DefaultCellEditor。关于 TableCellEditor 和 TreeCellEditor 两个接口及其实现类之间的关系如图 12.55 所示。

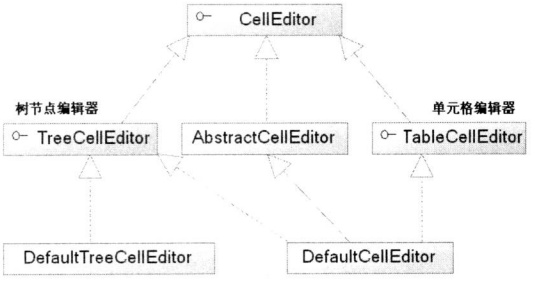


图 12.55　TableCellEditor 和 TreeCellEditor 的关系

从图12.55中可以看出，Swing为TableCellEditor提供了 DefaultCellEditor 实现类（也可作为TreeCellEditor的实现类），DefaultCellEditor类有三个构造器，它们分别使用文本框、复选框和JComboBox作为单元格编辑器，其中使用文本框编辑器是最常见的情形，如果单元格的值是 Boolean 类型，则系统默认使用复选框编辑器（如图 12.54 中最右边一列所示），这两种情形都是前面见过的情形。如果想指定某列使用 JComboBox 作为单元格编辑器，则需要显式创建 JComboBox 实例，然后以此实例来创建 DefaultCellEditor 编辑器。

实现 TableCellEditor 接口可以开发自己的单元格编辑器，但这种做法比较烦琐；通常会使用扩展 DefaultCellEditor 类的方式，这种方式比较简单。TableCellEditor 接口定义了一个 getTableCellEditorComponent()方法，该方法返回一个 Component 对象，该对象就是该单元格的编辑器。

一旦实现了自己的单元格编辑器，就可以为 JTable 对象安装该单元格编辑器，与安装单元格绘制器类似，安装单元格编辑器也有两种方式。

- 局部方式（列级）：为特定列指定单元格编辑器，通过调用 TableColumn 的 setCellEditor()方法为该列安装单元格编辑器。
- 全局方式（表级）：调用 JTable 的 setDefaultEditor()方法为该表格安装默认的单元格编辑器。该方法需要两个参数，即列类型和单元格编辑器，这两个参数表明对于指定类型的数据列使用该单元格编辑器。

与单元格绘制器相似的是，如果有一列同时满足列级单元格编辑器和表级单元格编辑器的要求，系统将采用列级单元格编辑器。

下面程序实现了一个 ImageCellEditor 编辑器，该编辑器由一个不可直接编辑的文本框和一个按钮组成，当用户单击该按钮时，该编辑器弹出一个文件选择器，方便用户选择图标文件。除此之外，下面程序还创建了一个基于 JComboBox 的 DefaultCellEditor 类，该编辑器允许用户通过下拉列表来选择图标。

程序清单：codes\12\12.11\TableCellEditorTest.java

```
public class TableCellEditorTest
{
    JFrame jf = new JFrame("使用单元格编辑器");
```

```
    JTable table;
    // 定义二维数组作为表格数据
    Object[][] tableData =
    {
        new Object[]{"李清照" , 29 , "女" , new ImageIcon("icon/3.gif")
            , new ImageIcon("icon/3.gif") , true},
        new Object[]{"苏格拉底", 56 , "男" , new ImageIcon("icon/1.gif")
            , new ImageIcon("icon/1.gif") , false},
        new Object[]{"李白", 35 , "男" , new ImageIcon("icon/4.gif")
            , new ImageIcon("icon/4.gif") , true},
        new Object[]{"弄玉", 18 , "女" , new ImageIcon("icon/2.gif")
            , new ImageIcon("icon/2.gif") , true},
        new Object[]{"虎头" , 2 , "男" , new ImageIcon("icon/5.gif")
            , new ImageIcon("icon/5.gif") , false}
    };
    // 定义一维数据作为列标题
    String[] columnTitle = {"姓名" , "年龄" , "性别" , "主头像"
        , "次头像" , "是否中国人"};
    public void init()
    {
        // 以二维数组和一维数组来创建一个 ExtendedTableModel 对象
        ExtendedTableModel model = new ExtendedTableModel(
            columnTitle , tableData);
        // 以 ExtendedTableModel 来创建 JTable
        table = new JTable(model);
        table.setRowSelectionAllowed(false);
        table.setRowHeight(40);
        // 为该表格指定默认的编辑器
        table.setDefaultEditor(ImageIcon.class, new ImageCellEditor());
        // 获取第 5 列
        TableColumn lastColumn = table.getColumnModel().getColumn(4);
        // 创建 JComboBox 对象，并添加多个图标列表项
        JComboBox<ImageIcon> editCombo = new JComboBox<>();
        for (int i = 1; i <= 10; i++)
        {
            editCombo.addItem(new ImageIcon("icon/" + i + ".gif"));
        }
        // 设置第 5 列使用基于 JComboBox 的 DefaultCellEditor
        lastColumn.setCellEditor(new DefaultCellEditor(editCombo));
        // 将 JTable 对象放在 JScrollPane 中，并将该 JScrollPane 放在窗口中显示出来
        jf.add(new JScrollPane(table));
        jf.pack();
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TableCellEditorTest().init();
    }
}
class ExtendedTableModel extends DefaultTableModel
{
    // 重新提供一个构造器，该构造器的实现委托给 DefaultTableModel 父类
    public ExtendedTableModel(String[] columnNames , Object[][] cells)
    {
        super(cells , columnNames);
    }
    // 重写 getColumnClass 方法，根据每列的第一个值返回该列真实的数据类型
    public Class getColumnClass(int c)
    {
        return getValueAt(0 , c).getClass();
    }
}
// 扩展 DefaultCellEditor 来实现 TableCellEditor 类
class ImageCellEditor extends DefaultCellEditor
{
    // 定义文件选择器
    private JFileChooser fDialog = new JFileChooser(); ;
    private JTextField field = new JTextField(15);
    private JButton button = new JButton("...");
```

```
public ImageCellEditor()
{
    // 因为 DefaultCellEditor 没有无参数的构造器
    // 所以这里显式调用父类有参数的构造器
    super(new JTextField());
    initEditor();
}
private void initEditor()
{
    field.setEditable(false);
    // 为按钮添加监听器，当用户单击该按钮时
    // 系统将出现一个文件选择器让用户选择图标文件
    button.addActionListener(e -> browse());
    // 为文件选择器安装文件过滤器
    fDialog.addChoosableFileFilter(new FileFilter()
    {
        public boolean accept(File f)
        {
            if (f.isDirectory())
            {
                return true;
            }
            String extension = Utils.getExtension(f);
            if (extension != null)
            {
                if (extension.equals(Utils.tiff)
                    || extension.equals(Utils.tif)
                    || extension.equals(Utils.gif)
                    || extension.equals(Utils.jpeg)
                    || extension.equals(Utils.jpg)
                    || extension.equals(Utils.png))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
            return false;
        }
        public String getDescription()
        {
            return "有效的图片文件";
        }
    });
    fDialog.setAcceptAllFileFilterUsed(false);
}
// 重写 TableCellEditor 接口的 getTableCellEditorComponent 方法
// 该方法返回单元格编辑器，该编辑器是一个 JPanel
// 该容器包含一个文本框和一个按钮
public Component getTableCellEditorComponent(JTable table
    , Object value , boolean isSelected , int row , int column)  // ①
{
    this.button.setPreferredSize(new Dimension(20, 20));
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    field.setText(value.toString());
    panel.add(this.field, BorderLayout.CENTER);
    panel.add(this.button, BorderLayout.EAST);
    return panel;
}
public Object getCellEditorValue()
{
    return new ImageIcon(field.getText());
}
private void browse()
{
    // 设置、打开文件选择器
    fDialog.setCurrentDirectory(new File("icon"));
```

```
            int result = fDialog.showOpenDialog(null);
            // 如果单击了文件选择器的“取消”按钮
            if (result == JFileChooser.CANCEL_OPTION)
            {
                // 取消编辑
                super.cancelCellEditing();
                return;
            }
            // 如果单击了文件选择器的“确定”按钮
            else
            {
                // 设置 field 的内容
                field.setText("icon/" + fDialog.getSelectedFile().getName());
            }
        }
    }
    class Utils
    {
        public final static String jpeg = "jpeg";
        public final static String jpg = "jpg";
        public final static String gif = "gif";
        public final static String tiff = "tiff";
        public final static String tif = "tif";
        public final static String png = "png";
        // 获取文件扩展名的方法
        public static String getExtension(File f)
        {
            String ext = null;
            String s = f.getName();
            int i = s.lastIndexOf('.');
            if (i > 0 &&  i < s.length() - 1)
            {
                ext = s.substring(i + 1).toLowerCase();
            }
            return ext;
        }
    }
```

上面程序中实现了一个 ImageCellEditor 编辑器，程序中的粗体字代码将该单元格编辑器注册成 ImageIcon 类型的单元格编辑器，如果某一列的数据类型是 ImageIcon，则默认使用该单元格编辑器。ImageCellEditor 扩展了 DefaultCellEditor 基类，重写 getTableCellEditorComponent()方法返回一个 JPanel，该 JPanel 里包含一个文本框和一个按钮。

除此之外，程序中的粗体字代码还为最后一列安装了一个基于 JComboBox 的 DefaultCellEditor。

运行上面程序，双击倒数第 3 列的任意单元格，开始编辑该单元格，将看到如图 12.56 所示的窗口。

双击第 5 列的任意单元格，开始编辑该单元格，将看到如图 12.57 所示的窗口。

图 12.56 自定义单元格编辑器

图 12.57 基于 JComboBox 的 DefaultCellEditor

通过图 12.56 和图 12.57 可以看出，如果单元格的值需要从多个枚举值之中选择，则使用 DefaultCellEditor 即可。使用自定义的单元格编辑器则非常灵活，可以取得单元格编辑器的全部控制权。

12.12 使用 JFormattedTextField 和 JTextPane 创建格式文本

Swing 使用 JTextComponent 作为所有文本输入组件的父类，从图 12.1 中可以看出，Swing 为该类提供了三个子类：JTextArea、JTextField 和 JEditorPane，并为 JEditorPane 提供了一个 JTextPane 子类，

JEditorPane 和 JTextPane 是两个典型的格式文本编辑器，也是本节介绍的重点。JTextArea 和 JTextField 是两个常见的文本组件，比较简单，本节不会再次介绍它们。

JTextField 派生了两个子类：JPasswordField 和 JFormattedTextField，它们代表密码输入框和格式化文本输入框。

与其他的 Swing 组件类似，所有的文本输入组件也遵循了 MVC 的设计模式，即每个文本输入组件都有对应的 model 来保存其状态数据；与其他的 Swing 组件不同的是，文本输入组件的 model 接口不是 XxxModel 接口，而是 Document 接口，Document 既包括有格式的文本，也包括无格式的文本。不同的文本输入组件对应的 Document 不同。

12.12.1 监听 Document 的变化

如果希望检测到任何文本输入组件里所输入内容的变化，则可以通过监听该组件对应的 Document 来实现。JTextComponent 类里提供了一个 getDocument()方法，该方法用于获取所有文本输入组件对应的 Document 对象。

Document 提供了一个 addDocumentListener()方法来为 Document 添加监听器，该监听器必须实现 DocumentListener 接口，该接口里提供了如下三个方法。

- changedUpdate(DocumentEvent e)：当 Document 里的属性或属性集发生了变化时触发该方法。
- insertUpdate(DocumentEvent e)：当向 Document 中插入文本时触发该方法。
- removeUpdate(DocumentEvent e)：当从 Document 中删除文本时触发该方法。

对于上面的三个方法而言，如果仅需要检测文本的变化，则无须实现第一个方法。但 Swing 并没有为 DocumentListener 接口提供适配器(难道是 Oracle 的疏忽)，所以程序依然要为第一个方法提供空实现。

除此之外，还可以为文件输入组件添加一个撤销监听器，这样就允许用户撤销以前的修改。添加撤销监听器的方法是 addUndoableEditListener()，该方法需要接收一个 UndoableEditListener 监听器，该监听器里包含了 undoableEditHappened()方法，当文档里发生了可撤销的编辑操作时将会触发该方法。

下面程序示范了如何为一个普通文本域的 Document 添加监听器，当用户在目标文本域里输入、删除文本时，程序会显示出用户所做的修改。该文本域还支持撤销操作，当用户按“Ctrl+Z”键时，该文本域会撤销用户刚刚输入的内容。

程序清单：codes\12\12.12\MonitorText.java

```
public class MonitorText
{
    JFrame mainWin = new JFrame("监听 Document 对象");
    JTextArea target = new JTextArea(4, 35);
    JTextArea msg = new JTextArea(5, 35);
    JLabel label = new JLabel("文本域的修改信息");
    Document doc = target.getDocument();
    // 保存撤销操作的 List 对象
    LinkedList<UndoableEdit> undoList = new LinkedList<>();
    // 最多允许撤销多少次
    final int UNDO_COUNT = 20;
    public void init()
    {
        msg.setEditable(false);
        // 添加 DocumentListener
        doc.addDocumentListener(new DocumentListener()
        {
            // 当 Document 的属性或属性集发生了变化时触发该方法
            public void changedUpdate(DocumentEvent e){}
            // 当向 Document 中插入文本时触发该方法
            public void insertUpdate(DocumentEvent e)
            {
                int offset = e.getOffset();
                int len = e.getLength();
                // 取得插入事件的位置
                msg.append("插入文本的长度: " + len + "\n");
```

```
            msg.append("插入文本的起始位置：" + offset + "\n");
            try
            {
                msg.append("插入文本内容："
                    + doc.getText(offset, len) + "\n");
            }
            catch (BadLocationException evt)
            {
                evt.printStackTrace();
            }
        }
        // 当从 Document 中删除文本时触发该方法
        public void removeUpdate(DocumentEvent e)
        {
            int offset = e.getOffset();
            int len = e.getLength();
            // 取得插入事件的位置
            msg.append("删除文本的长度：" + len + "\n");
            msg.append("删除文本的起始位置：" + offset + "\n");
        }
    });
    // 添加可撤销操作的监听器
    doc.addUndoableEditListener(e -> {
        // 每次发生可撤销操作时都会触发该代码块      // ①
        UndoableEdit edit = e.getEdit();
        if (edit.canUndo() && undoList.size() < UNDO_COUNT)
        {
            // 将撤销操作装入 List 内
            undoList.add(edit);
        }
        // 已经达到了最大撤销次数
        else if (edit.canUndo() && undoList.size() >= UNDO_COUNT)
        {
            // 弹出第一个撤销操作
            undoList.pop();
            // 将撤销操作装入 List 内
            undoList.add(edit);
        }
    });
    // 为“Ctrl+Z”按键添加监听器
    target.addKeyListener(new KeyAdapter()
    {
        public void keyTyped(KeyEvent e)     // ②
        {
            // 如果按键是“Ctrl + Z”
            if (e.getKeyChar() == 26)
            {
                if (undoList.size() > 0)
                {
                    // 移出最后一个可撤销操作，并取消该操作
                    undoList.removeLast().undo();
                }
            }
        }
    });
    Box box = new Box(BoxLayout.Y_AXIS);
    box.add(new JScrollPane(target));
    JPanel panel = new JPanel();
    panel.add(label);
    box.add(panel);
    box.add(new JScrollPane(msg));
    mainWin.add(box);
    mainWin.pack();
    mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainWin.setVisible(true);
}
public static void main(String[] args) throws Exception
{
```

```
            new MonitorText().init();
        }
    }
```

上面程序中的两段粗体字代码实现了 Document 中插入文本、删除文本的事件处理器，当用户向 Document 中插入文本、删除文本时，程序将会把这些修改信息添加到下面的一个文本域里。

程序中①号粗体字代码是可撤销操作的事件处理器，当用户在该文本域内进行可撤销操作时，这段代码将会被触发，这段代码把用户刚刚进行的可撤销操作以 List 保存起来，以便在合适的时候撤销用户所做的修改。

程序中②号粗体字代码主要用于为“Ctrl+Z”按键添加按键监听器，当用户按下“Ctrl+Z”键时，程序从保存可撤销操作的 List 中取出最后一个可撤销操作，并撤销该操作的修改。

运行上面程序，会看到如图 12.58 所示的运行结果。

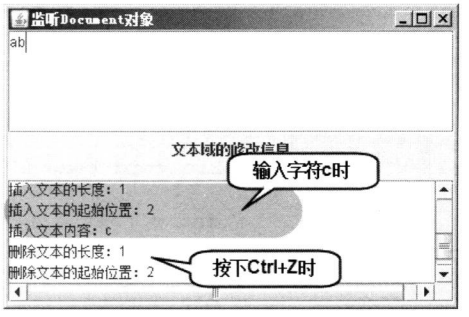


图 12.58　为 Document 添加监听器

12.12.2　使用 JPasswordField

JPasswordField 是 JTextField 的一个子类，它是 Swing 的 MVC 设计的产品——JPasswordField 和 JTextField 的各种特征几乎完全一样，只是当用户向 JPasswordField 输入内容时，JPasswordField 并不会显示出用户输入的内容，而是以 echo 字符（通常是星号和黑点）来代替用户输入的所有字符。

JPasswordField 和 JTextField 的用法几乎完全一样，连构造器的个数和参数都完全一样。但是 JPasswordField 多了一个 setEchoChar(Char ch)方法，该方法用于设置该密码框的 echo 字符——当用户在密码输入框内输入时，每个字符都会使用该 echo 字符代替。

除此之外，JPasswordField 重写了 JTextComponent 的 getText()方法，并且不再推荐使用 getText()方法返回字符串密码框的字符串，因为 getText()方法所返回的字符串会一直停留在虚拟机中，直到垃圾回收，这可能导致存在一些安全隐患，所以 JPasswordField 提供了一个 getPassword()方法，该方法返回一个字符数组，而不是返回字符串，从而提供了更好的安全机制。

注意：

当程序使用完 getPassword()方法返回的字符数组后，应该立即清空该字符数组的内容，以防该数组泄露密码信息。

12.12.3　使用 JFormattedTextField

在有些情况下，程序不希望用户在输入框内随意地输入，例如，程序需要用户输入一个有效的时间，或者需要用户输入一个有效的物品价格，如果用户输入不合理，程序应该阻止用户输入。对于这种需求，通常的做法是为该文本框添加失去焦点的监听器，再添加回车按键的监听器，当该文本框失去焦点时，或者该用户在该文本框内按回车键时，就检测用户输入是否合法。这种做法基本可以解决该问题，但编程比较烦琐！Swing 提供的 JFormattedTextField 可以更优雅地解决该问题。

使用 JFormattedTextField 与使用普通文本行有一个区别——它需要指定一个文本格式，只有当用户的输入满足该格式时，JFormattedTextField 才会接收用户输入。JFormattedTextField 可以使用如下两种类型的格式。

- JFormattedTextField.AbstractFormatter：该内部类有一个子类 DefaultFormatter，而 DefaultFormatter 又有一个非常实用的 MaskFormatter 子类，允许程序以掩码的形式指定文本格式。
- Format：主要由 DateFormat 和 NumberFormat 两个格式器组成，这两个格式器可以指定 JFormattedTextField 所能接收的格式字符串。

创建 JFormattedTextField 对象时可以传入上面任意一个格式器，成功地创建了 JFormattedTextField

对象之后，JFormattedTextField 对象的用法和普通 TextField 的用法基本相似，一样可以调用 setColumns()来设置该文本框的宽度，调用 setFont()来设置该文本框内的字体等。除此之外，JFormattedTextField 还包含如下三个特殊方法。

- Object getValue()：获取该格式化文本框里的值。
- void setValue(Object obj)：设置该格式化文本框的初始值。
- void setFocusLostBehavior(int behavior)：设置该格式化文本框失去焦点时的行为，该方法可以接收如下 4 个值。
  - JFormattedTextField.COMMIT：如果用户输入的内容满足格式器的要求，则该格式化文本框显示的文本变成用户输入的内容，调用 getValue()方法返回的是该文本框内显示的内容；如果用户输入的内容不满足格式器的要求，则该格式化文本框显示的依然是用户输入的内容，但调用 getValue()方法返回的不是该文本框内显示的内容，而是上一个满足要求的值。
  - JFormattedTextField.COMMIT_OR_REVERT：这是默认值。如果用户输入的内容满足格式器的要求，则该格式化文本框显示的文本、getValue()方法返回的都是用户输入的内容；如果用户输入的内容不满足格式器的要求，则该格式化文本框显示的文本、getValue()方法返回的都是上一个满足要求的值。
  - JFormattedTextField.PERSIST：不管用户输入的内容是否满足格式器的要求，该格式化文本框都显示用户输入的内容，getValue()方法返回的都是上一个满足要求的值。
  - JFormattedTextField.REVERT：不管用户输入的内容是否满足格式器的要求，该格式化文本框显示的内容、getValue()方法返回的都是上一个满足要求的值。在这种情况下，不管用户输入什么内容对该文本框都没有任何影响。

上面三个方法中获取格式化文本框内容的方法返回 Object 类型，而不是返回 String 类型；与之对应的是，设置格式化文本框初始值的方法需要传入 Object 类型参数，而不是 String 类型参数，这都是因为格式化文本框会将文本框内容转换成指定格式对应的对象，而不再是普通字符串。

DefaultFormatter 是一个功能非常强大的格式器，它可以格式化任何类的实例，只要该类包含一个带一个字符串参数的构造器，并提供对应的 toString()方法（该方法的返回值就是传入给构造器字符串参数的值）即可。

例如，URL 类包含一个 URL(String spec)构造器，且 URL 对象的 toString()方法恰好返回刚刚传入的 spec 参数，因此可以使用 DefaultFormatter 来格式化 URL 对象。当格式化文本框失去焦点时，该格式器就会调用带一个字符串参数的构造器来创建新的对象，如果构造器抛出了异常，即表明用户输入无效。

注意：

DefaultFormatter 格式器默认采用改写方式来处理用户输入，即当用户在格式化文本框内输入时，每输入一个字符就会替换文本框内原来的一个字符。如果想关闭这种改写方式，采用插入方式，则可通过调用它的 setOverwriteMode(false)方法来实现。

MaskFormatter 格式器的功能有点类似于正则表达式，它要求用户在格式化文本框内输入的内容必须匹配一定的掩码格式。例如，若要匹配广州地区的电话号码，则可采用 020-########的格式，这个掩码字符串和正则表达式有一定的区别，因为该掩码字符串只支持如下通配符。

- #：代表任何有效数字。
- '：转义字符，用于转义具有特殊格式的字符。例如，若想匹配#，则应该写成'#。
- U：任何字符，将所有小写字母映射为大写。
- L：任何字符，将所有大写字母映射为小写。
- A：任何字符或数字。
- ?：任何字符。

- *：可以匹配任何内容。
- H：任何十六进制字符（0~9、a~f或A~F）。

值得指出的是，格式化文本框内的字符串总是和掩码具有相同的格式，连长度也完全相同。如果用户删除了格式化文本框内的字符，这些被删除的字符将由占位符替代。默认使用空格作为占位符，当然也可以调用MaskFormatter的setPlaceholderCharacter()方法来设置该格式器的占位符。例如如下代码：

```
formatter.setPlaceholderCharacter('□');
```

下面程序示范了关于JFormattedTextField的简单用法。

程序清单：codes\12\12.12\JFormattedTextFieldTest.java

```
public class JFormattedTextFieldTest
{
    private JFrame mainWin = new JFrame("测试格式化文本框");
    private JButton okButton = new JButton("确定");
    // 定义用于添加格式化文本框的容器
    private JPanel mainPanel = new JPanel();
    JFormattedTextField[] fields = new JFormattedTextField[6];
    String[] behaviorLabels = new String[]
    {
        "COMMIT",
        "COMMIT_OR_REVERT",
        "PERSIST",
        "REVERT"
    };
    int[] behaviors = new int[]
    {
        JFormattedTextField.COMMIT,
        JFormattedTextField.COMMIT_OR_REVERT,
        JFormattedTextField.PERSIST,
        JFormattedTextField.REVERT
    };
    ButtonGroup bg = new ButtonGroup();
    public void init()
    {
        // 添加按钮
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(okButton);
        mainPanel.setLayout(new GridLayout(0, 3));
        mainWin.add(mainPanel, BorderLayout.CENTER);
        // 使用NumberFormat的integerInstance创建一个JformattedTextField对象
        fields[0] = new JFormattedTextField(NumberFormat
            .getIntegerInstance());
        // 设置初始值
        fields[0].setValue(100);
        addRow("整数格式文本框 :", fields[0]);
        // 使用NumberFormat的currencyInstance创建一个JFormattedTextField对象
        fields[1] = new JFormattedTextField(NumberFormat
            .getCurrencyInstance());
        fields[1].setValue(100.0);
        addRow("货币格式文本框:", fields[1]);
        // 使用默认的日期格式创建一个JFormattedTextField对象
        fields[2] = new JFormattedTextField(DateFormat.getDateInstance());
        fields[2].setValue(new Date());
        addRow("默认的日期格式器:", fields[2]);
        // 使用SHORT类型的日期格式创建一个JFormattedTextField对象
        // 且要求采用严格日期格式
        DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);
        // 要求采用严格的日期格式语法
        format.setLenient(false);
        fields[3] = new JFormattedTextField(format);
        fields[3].setValue(new Date());
        addRow("SHORT类型的日期格式器（语法严格）:", fields[3]);
        try
        {
            // 创建默认的DefaultFormatter对象
            DefaultFormatter formatter = new DefaultFormatter();
```

```
            // 关闭 overwrite 状态
            formatter.setOverwriteMode(false);
            fields[4] = new JFormattedTextField(formatter);
            // 使用 DefaultFormatter 来格式化 URL
            fields[4].setValue(new URL("http://www.crazyit.org"));
            addRow("URL:", fields[4]);
        }
        catch (MalformedURLException e)
        {
            e.printStackTrace();
        }
        try
        {
            MaskFormatter formatter = new MaskFormatter("020-########");
            // 设置占位符
            formatter.setPlaceholderCharacter('□');
            fields[5] = new JFormattedTextField(formatter);
            // 设置初始值
            fields[5].setValue("020-28309378");
            addRow("电话号码: ", fields[5]);
        }
        catch (ParseException ex)
        {
            ex.printStackTrace();
        }

        JPanel focusLostPanel = new JPanel();
        // 采用循环方式加入失去焦点行为的单选按钮
        for (int i = 0; i < behaviorLabels.length ; i++ )
        {
            final int index = i;
            final JRadioButton radio = new JRadioButton(behaviorLabels[i]);
            // 默认选中第二个单选按钮
            if (i == 1)
            {
                radio.setSelected(true);
            }
            focusLostPanel.add(radio);
            bg.add(radio);
            // 为所有的单选按钮添加事件监听器
            radio.addActionListener(e -> {
                // 如果当前该单选按钮处于选中状态
                if (radio.isSelected())
                {
                    // 设置所有的格式化文本框失去焦点的行为
                    for (int j = 0 ; j < fields.length ; j++)
                    {
                      fields[j].setFocusLostBehavior(behaviors[index]);
                    }
                }
            });
        }
        focusLostPanel.setBorder(new TitledBorder(new EtchedBorder(),
            "请选择焦点失去后的行为"));
        JPanel p = new JPanel();
        p.setLayout(new BorderLayout());
        p.add(focusLostPanel , BorderLayout.NORTH);
        p.add(buttonPanel , BorderLayout.SOUTH);

        mainWin.add(p , BorderLayout.SOUTH);
        mainWin.pack();
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.setVisible(true);
    }
    // 定义添加一行格式化文本框的方法
    private void addRow(String labelText, final JFormattedTextField field)
    {
        mainPanel.add(new JLabel(labelText));
        mainPanel.add(field);
        final JLabel valueLabel = new JLabel();
```

```
        mainPanel.add(valueLabel);
        // 为“确定”按钮添加事件监听器
        // 当用户单击“确定”按钮时，文本框后显示文本框的值
        okButton.addActionListener(event -> {
            Object value = field.getValue();
            // 输出格式化文本框的值
            valueLabel.setText(value.toString());
        });
    }
    public static void main(String[] args)
    {
        new JFormattedTextFieldTest().init();
    }
}
```

上面程序添加了 6 个格式化文本框，其中两个是基于 NumberFormat 生成的整数格式器、货币格式器，两个是基于 DateFormat 生成的日期格式器，一个是使用 DefaultFormatter 创建的 URL 格式器，最后一个是使用 MaskFormatter 创建的掩码格式器，程序中的粗体字代码是创建这些格式器的关键代码。

除此之外，程序还添加了 4 个单选按钮，用于控制这些格式化文本框失去焦点后的行为。运行上面程序，并选中“COMMIT”行为，将看到如图 12.59 所示的界面。

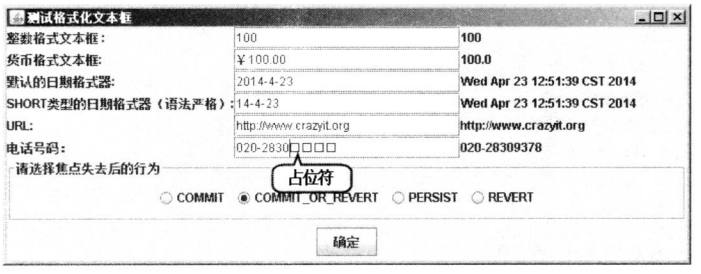


图 12.59　COMMIT 行为下的格式化文本框

从图 12.59 中可以看出，虽然用户向格式化文本框内输入的内容与该文本框所要求的格式不符，但该文本框依然显示了用户输入的内容，只是后面显示该文本框的 getValue()方法返回值时看到的依然是 100，即上一个符合格式的值。

大部分时候，使用基于 Format 的格式器，DefaultFormatter 和 MaskFormatter 已经能满足绝大部分要求；但对于一些特殊的要求，则可以采用扩展 DefaultFormatter 的方式来定义自己的格式器。定义自己的格式器通常需要重写如下两个方法。

- Object stringToValue(String string)：根据格式化文本框内的字符串来创建符合指定格式的对象。
- String valueToString(Object value)：将符合格式的对象转换成文本框中显示的字符串。

例如，若需要创建一个只能接收 IP 地址的格式化文本框，则可以创建一个自定义的格式化文本框，因为 IP 地址是由 4 个 0~255 之间的整数表示的，所以程序采用长度为 4 的 byte[]数组来保存 IP 地址。程序可以采用如下方法将用户输入的字符串转换成 byte[]数组。

```
public Object stringToValue(String text) throws ParseException
{
    // 将格式化文本框内的字符串以点号（.）分成 4 节
    String[] nums = text.split("\\.");
    if (nums.length != 4)
    {
        throw new ParseException("IP 地址必须是 4 个整数", 0);
    }
    byte[] a = new byte[4];
    for (int i = 0; i < 4; i++)
    {
        int b = 0;
        try
        {
            b = Integer.parseInt(nums[i]);
        }
        catch (NumberFormatException e)
```

```
            {
                throw new ParseException("IP地址必须是整数", 0);
            }
            if (b < 0 || b >= 256)
            {
                throw new ParseException("IP地址值只能在0~255之间", 0);
            }
            a[i] = (byte) b;
        }
        return a;
    }
```

除此之外，Swing还提供了如下两种机制来保证用户输入的有效性。

- 输入过滤：输入过滤机制允许程序拦截用户的插入、替换、删除等操作，并改变用户所做的修改。
- 输入校验：输入验证机制允许用户离开输入组件时，验证机制自动触发——如果用户输入不符合要求，校验器强制用户重新输入。

输入过滤器需要继承DocumentFilter类，程序可以重写该类的如下三个方法来拦截用户的插入、删除和替换等操作。

- insertString(DocumentFilter.FilterBypass fb, int offset, String string, AttributeSet attr)：该方法会拦截用户向文档中插入字符串的操作。
- remove(DocumentFilter.FilterBypass fb, int offset, int length)：该方法会拦截用户从文档中删除字符串的操作。
- replace(DocumentFilter.FilterBypass fb, int offset, int length, String text, AttributeSet attrs)：该方法会拦截用户替换文档中字符串的操作。

为了创建自己的输入校验器，可以通过扩展InputVerifier类来实现。实际上，InputVerifier输入校验器可以绑定到任何输入组件，InputVerifier类里包含了一个verify(JComponent component)方法，当用户在该输入组件内输入完成，且该组件失去焦点时，该方法被调用——如果该方法返回false，即表明用户输入无效，该输入组件将自动得到焦点。也就是说，如果某个输入组件绑定了InputVerifier，则用户必须为该组件输入有效内容，否则用户无法离开该组件。

注意：

有一种情况例外，如果输入焦点离开了带InputVerifier输入校验器的组件后，立即单击某个按钮，则该按钮的事件监听器将会在焦点重新回到原组件之前被触发。

下面程序示范了如何为格式化文本框添加输入过滤器、输入校验器，程序还自定义了一个IP地址格式器，该IP地址格式器扩展了DefaultFormatter格式器。

程序清单：codes\12\12.12\JFormattedTextFieldTest2.java

```
public class JFormattedTextFieldTest2
{
    private JFrame mainWin = new JFrame("测试格式化文本框");
    private JButton okButton = new JButton("确定");
    // 定义用于添加格式化文本框的容器
    private JPanel mainPanel = new JPanel();
    public void init()
    {
        // 添加按钮
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(okButton);
        mainPanel.setLayout(new GridLayout(0, 3));
        mainWin.add(mainPanel, BorderLayout.CENTER);
        JFormattedTextField intField0 = new JFormattedTextField(
            new InternationalFormatter(NumberFormat.getIntegerInstance())
            {
                protected DocumentFilter getDocumentFilter()
                {
                    return new NumberFilter();
```

```
            }
        });
        intField0.setValue(100);
        addRow("只接受数字的文本框", intField0);
        JFormattedTextField intField1 = new JFormattedTextField
            (NumberFormat.getIntegerInstance());
        intField1.setValue(100);
        // 添加输入校验器
        intField1.setInputVerifier(new FormattedTextFieldVerifier());
        addRow("带输入校验器的文本框", intField1);
        // 创建自定义格式器对象
        IPAddressFormatter ipFormatter = new IPAddressFormatter();
        ipFormatter.setOverwriteMode(false);
        // 以自定义格式器对象创建格式化文本框
        JFormattedTextField ipField = new JFormattedTextField(ipFormatter);
        ipField.setValue(new byte[]{(byte)192, (byte)168, 4, 1});
        addRow("IP 地址格式", ipField);
        mainWin.add(buttonPanel , BorderLayout.SOUTH);
        mainWin.pack();
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.setVisible(true);
    }
    // 定义添加一行格式化文本框的方法
    private void addRow(String labelText, final JFormattedTextField field)
    {
        mainPanel.add(new JLabel(labelText));
        mainPanel.add(field);
        final JLabel valueLabel = new JLabel();
        mainPanel.add(valueLabel);
        // 为“确定”按钮添加事件监听器
        // 当用户单击“确定”按钮时，文本框后显示文本框内的值
        okButton.addActionListener(event -> {
            Object value = field.getValue();
            // 如果该值是数组，则使用 Arrays 的 toString()方法输出数组
            if (value.getClass().isArray())
            {
                StringBuilder builder = new StringBuilder();
                builder.append('{');
                for (int i = 0; i < Array.getLength(value); i++)
                {
                    if (i > 0)
                        builder.append(',');
                    builder.append(Array.get(value, i).toString());
                }
                builder.append('}');
                valueLabel.setText(builder.toString());
            }
            else
            {
                // 输出格式化文本框的值
                valueLabel.setText(value.toString());
            }
        });
    }
    public static void main(String[] args)
    {
        new JFormattedTextFieldTest2().init();
    }
}
// 输入校验器
class FormattedTextFieldVerifier extends InputVerifier
{
    // 当输入组件失去焦点时，该方法被触发
    public boolean verify(JComponent component)
    {
        JFormattedTextField field = (JFormattedTextField)component;
        // 返回用户输入是否有效
        return field.isEditValid();
    }
}
```

```
// 数字过滤器
class NumberFilter extends DocumentFilter
{
    public void insertString(FilterBypass fb , int offset
        , String string , AttributeSet attr)throws BadLocationException
    {
        StringBuilder builder = new StringBuilder(string);
        // 过滤用户输入的所有字符
        filterInt(builder);
        super.insertString(fb, offset, builder.toString(), attr);
    }
    public void replace(FilterBypass fb , int offset , int length
        , String string , AttributeSet attr)throws BadLocationException
    {
        if (string != null)
        {
            StringBuilder builder = new StringBuilder(string);
            // 过滤用户替换的所有字符
            filterInt(builder);
            string = builder.toString();
        }
        super.replace(fb, offset, length, string, attr);
    }
    // 过滤整数字符，把所有非0～9的字符全部删除
    private void filterInt(StringBuilder builder)
    {
        for (int i = builder.length() - 1; i >= 0; i--)
        {
            int cp = builder.codePointAt(i);
            if (cp > '9' || cp < '0')
            {
                builder.deleteCharAt(i);
            }
        }
    }
}
class IPAddressFormatter extends DefaultFormatter
{
    public String valueToString(Object value)
        throws ParseException
    {
        if (!(value instanceof byte[]))
        {
            throw new ParseException("该IP地址的值只能是字节数组", 0);
        }
        byte[] a = (byte[])value;
        if (a.length != 4)
        {
            throw new ParseException("IP地址必须是4个整数", 0);
        }
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < 4; i++)
        {
            int b = a[i];
            if (b < 0) b += 256;
                builder.append(String.valueOf(b));
            if (i < 3) builder.append('.');
        }
        return builder.toString();
    }
    public Object stringToValue(String text) throws ParseException
    {
        // 将格式化文本框内的字符串以点号（.）分成4节
        String[] nums = text.split("\\.");
        if (nums.length != 4)
        {
            throw new ParseException("IP地址必须是4个整数", 0);
        }
        byte[] a = new byte[4];
        for (int i = 0; i < 4; i++)
```

```
        {
            int b = 0;
            try
            {
                b = Integer.parseInt(nums[i]);
            }
            catch (NumberFormatException e)
            {
                throw new ParseException("IP地址必须是整数", 0);
            }
            if (b < 0 || b >= 256)
            {
                throw new ParseException("IP地址值只能在0~255之间", 0);
            }
            a[i] = (byte) b;
        }
        return a;
    }
}
```

运行上面程序，会看到窗口中出现三个格式化文本框，其中第一个格式化文本框只能输入数字，其他字符无法输入到该文本框内；第二个格式化文本框有输入校验器，只有当用户输入的内容符合该文本框的要求时，用户才可以离开该文本框；第三个格式化文本框的格式器是自定义的格式器，它要求用户输入的内容是一个合法的 IP 地址。

12.12.4 使用 JEditorPane

Swing 提供了一个 JEditorPane 类，该类可以编辑各种文本内容，包括有格式的文本。在默认情况下，JEditorPane 支持如下三种文本内容。

- text/plain：纯文本，当 JEditorPane 无法识别给定内容的类型时，使用这种文本格式。在这种模式下，文本框的内容是带换行符的无格式文本。
- text/html：HTML 文本格式。该文本组件仅支持 HTML 3.2 格式，因此对互联网上复杂的网页支持非常有限。
- text/rtf：RTF（富文本格式）文本格式。实际上，它对 RTF 的支持非常有限。

通过上面介绍不难看出，其实 JEditorPane 类的用途非常有限，使用 JEditorPane 作为纯文本的编辑器，还不如使用 JTextArea；如果使用 JEditorPane 来支持 RTF 文本格式，但它对这种文本格式的支持又相当有限；JEditorPane 唯一可能的用途就是显示自己的 HTML 文档，前提是这份 HTML 文档比较简单，只包含 HTML 3.2 或更早的元素。

JEditorPane 组件支持三种方法来加载文本内容。

- 使用 setText()方法直接设置 JEditorPane 的文本内容。
- 使用 read()方法从输入流中读取 JEditorPane 的文本内容。
- 使用 setPage()方法来设置 JEditorPane 从哪个 URL 处读取文本内容。在这种情况下，将根据该 URL 来确定内容类型。

在默认状态下，使用 JEditorPane 装载的文本内容是可编辑的，即使装载互联网上的网页也是如此，可以使用 JEditorPane 的 setEditable(false)方法阻止用户编辑该 JEditorPane 里的内容。

当使用 JEditorPane 打开 HTML 页面时，该页面的超链接是活动的，用户可以单击超链接。如果程序想监听用户单击超链接的事件，则必须使用 addHyperlinkListener()方法为 JEditorPane 添加一个 HyperlinkListener 监听器。

从目前的功能来看，JEditorPane 确实没有太大的实用价值，所以本书不打算给出此类的用法示例，有兴趣的读者可以参考光盘 codes\12\12.12\路径下的 JEditorPaneTest.java 来学习该类的用法。相比之下，该类的子类 JTextPane 则功能丰富多了，下面详细介绍 JTextPane 类的用法。

12.12.5 使用 JTextPane

使用 EditPlus、Eclipse 等工具时会发现，当在这些工具中输入代码时，如果输入的单词是程序关键

字、类名等，则这些关键字将会自动变色。使用 JTextPane 组件，就可以开发出这种带有语法高亮的编辑器。

JTextPane 使用 StyledDocument 作为它的 model 对象，而 StyleDocument 允许对文档的不同段落分别设置不同的颜色、字体属性。Document 使用 Element 来表示文档中的组成部分，Element 可以表示章（chapter）、段落（paragraph）等，在普通文档中，Element 也可以表示一行。为了设置 StyledDocument 中文字的字体、颜色，Swing 提供了 AttributeSet 接口来表示文档字体、颜色等属性。

Swing 为 StyledDocument 提供了 DefaultStyledDocument 实现类，该实现类就是 JTextPane 的 model 实现类；为 AttributeSet 接口提供了 MutableAttributeSet 子接口，并为该接口提供了 SimpleAttributeSet 实现类，程序通过这些接口和实现类就可以很好地控制 JTextPane 中文字的字体和颜色。

StyledDocument 提供了如下一个方法来设置文档中局部文字的字体、颜色。

- setParagraphAttributes(int offset, int length, AttributeSet s, boolean replace)：设置文档中从 offset 开始，长度为 length 处的文字使用 s 属性（控制字体、颜色等），最后一个参数控制新属性是替换原有属性，还是将新属性累加到原有属性上。

AttributeSet 的常用实现类是 MutableAttributeSet，为了给 MutableAttributeSet 对象设置字体、颜色等属性，Swing 提供了 StyleConstants 工具类，该工具类里大致包含了如下常用的静态方法来设置 MutableAttributeSet 里的字体、颜色等。

- setAlignment(MutableAttributeSet a, int align)：设置文本对齐方式。
- setBackground(MutableAttributeSet a, Color fg)：设置背景色。
- setBold(MutableAttributeSet a, boolean b)：设置是否使用粗体字。
- setFirstLineIndent(MutableAttributeSet a, float i)：设置首行缩进的大小。
- setFontFamily(MutableAttributeSet a, String fam)：设置字体。
- setFontSize(MutableAttributeSet a, int s)：设置字体大小。
- setForeground(MutableAttributeSet a, Color fg)：设置字体前景色。
- setItalic(MutableAttributeSet a, boolean b)：设置是否采用斜体字。
- setLeftIndent(MutableAttributeSet a, float i)：设置左边缩进大小。
- setLineSpacing(MutableAttributeSet a, float i)：设置行间距。
- setRightIndent(MutableAttributeSet a, float i)：设置右边缩进大小。
- setStrikeThrough(MutableAttributeSet a, boolean b)：设置是否为文字添加删除线。
- setSubscript(MutableAttributeSet a, boolean b)：设置将指定文字设置成下标。
- setSuperscript(MutableAttributeSet a, boolean b)：设置将指定文字设置成上标。
- setUnderline(MutableAttributeSet a, boolean b)：设置是否为文字添加下画线。

提示：

上面这些方法用于控制文档中文字的外观样式，如果读者对这些外观样式不是太熟悉，则可以参考 Word 里设置“字体”属性的设置效果。

图 12.60 显示了 Document 及其相关实现类，以及相关辅助类的类关系图。

下面程序简单地定义了三个 SimpleAttributeSet 对象，并为这三个对象设置了对应的文字、颜色、字体等属性，并使用三个 SimpleAttributeSet 对象设置文档中三段文字的外观。

程序清单：codes\12\12.12\JTextPaneTest.java

```
public class JTextPaneTest
{
    JFrame mainWin = new JFrame("测试 JTextPane");
    JTextPane txt = new JTextPane();
    StyledDocument doc = txt.getStyledDocument();
    // 定义 3SimpleAttributeSet 对象
    SimpleAttributeSet android = new SimpleAttributeSet();
```

```
    SimpleAttributeSet java = new SimpleAttributeSet();
    SimpleAttributeSet javaee = new SimpleAttributeSet();
    public void init()
    {
        // 为 android 属性集设置颜色、字体大小、字体和下画线
        StyleConstants.setForeground(android, Color.RED);
        StyleConstants.setFontSize(android, 24);
        StyleConstants.setFontFamily(android, "Dialog");
        StyleConstants.setUnderline(android, true);
        // 为 java 属性集设置颜色、字体大小、字体和粗体字
        StyleConstants.setForeground(java, Color.BLUE);
        StyleConstants.setFontSize(java, 30);
        StyleConstants.setFontFamily(java, "Arial Black");
        StyleConstants.setBold(java, true);
        // 为 javaee 属性集设置颜色、字体大小、斜体字
        StyleConstants.setForeground(javaee, Color.GREEN);
        StyleConstants.setFontSize(javaee, 32);
        StyleConstants.setItalic(javaee, true);
        // 设置不允许编辑
        txt.setEditable(false);
        txt.setText("疯狂 Android 讲义\n"
            + "疯狂 Java 讲义\n" + "轻量级 Java EE 企业应用实战\n");
        // 分别为文档中 3 段文字设置不同的外观样式
        doc.setCharacterAttributes(0 , 12 , android, true);
        doc.setCharacterAttributes(12 , 12 , java, true);
        doc.setCharacterAttributes(24 , 30 , javaee, true);
        mainWin.add(new JScrollPane(txt), BorderLayout.CENTER);
        // 获取屏幕尺寸
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int inset = 100;
        // 设置主窗口的大小
        mainWin.setBounds(inset, inset, screenSize.width - inset * 2
            , screenSize.height - inset * 2);
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new JTextPaneTest().init();
    }
}
```

上面程序其实很简单，程序中的第一段粗体字代码为三个 SimpleAttributeSet 对象设置了字体、字体大小、颜色等外观样式，第二段粗体字代码使用前面的三个 SimpleAttributeSet 对象来控制文档中三段文字的外观样式。运行上面程序，将看到如图 12.61 所示的界面。

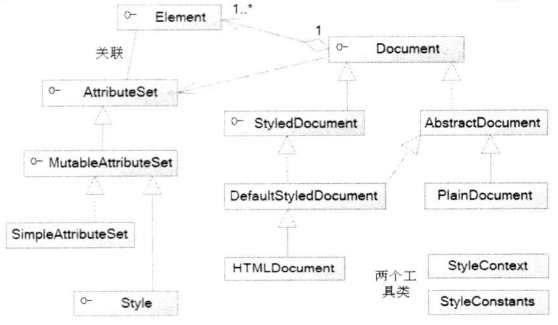


图 12.60　Document 及其相关实现类，以及相关辅助类的类关系图

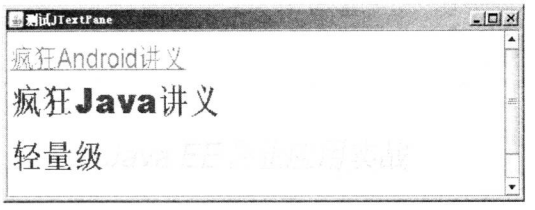


图 12.61　使用 JTextPane 的效果

从图 12.61 中可以看出，窗口中文字具有丰富的外观，而且还可以选中这些文字，表明它们依然是文字，而不是直接绘制上去的图形。

如果希望开发出类似于 EditPlus、Eclipse 等的代码编辑窗口，程序可以扩展 JTextPane 的子类，为该对象添加按键监听器和文档监听器。当文档内容被修改时，或者用户在该文档内进行击键动作时，程序负责分析该文档的内容，对特殊关键字设置字体颜色。

为了保证具有较好的性能，程序并不总是分析文档中的所有内容，而是只分析文档中被改变的部分，这个要求看似简单，只为文档添加文档监听器即可——当文档内容改变时分析被改变部分，并设置其中关键字的颜色。问题是：DocumentListener 监听器里的三个方法不能改变文档本身，所以程序还是必须通过监听按键事件来启动语法分析，DocumentListener 监听器中仅仅记录文档改变部分的位置和长度。

除此之外，程序还提供了一个 SyntaxFormatter 类根据语法文件来设置文档中的文字颜色。

程序清单：codes\12\12.12\MyTextPane.java

```
public class MyTextPane extends JTextPane
{
    protected StyledDocument doc;
    protected SyntaxFormatter formatter = new SyntaxFormatter("my.stx");
    // 定义该文档的普通文本的外观属性
    private SimpleAttributeSet normalAttr =
        formatter.getNormalAttributeSet();
    private SimpleAttributeSet quotAttr = new SimpleAttributeSet();
    // 保存文档改变的开始位置
    private int docChangeStart = 0;
    // 保存文档改变的长度
    private int docChangeLength = 0;
    public MyTextPane()
    {
        StyleConstants.setForeground(quotAttr
            , new Color(255, 0 , 255));
        StyleConstants.setFontSize(quotAttr, 16);
        this.doc = super.getStyledDocument();
        // 设置该文档的页边距
        this.setMargin(new Insets(3, 40, 0, 0));
        // 添加按键监听器，当按键松开时进行语法分析
        this.addKeyListener(new KeyAdapter()
        {
            public void keyReleased(KeyEvent ke)
            {
                syntaxParse();
            }
        });
        // 添加文档监听器
        doc.addDocumentListener(new DocumentListener()
        {
            // 当 Document 的属性或属性集发生了变化时触发该方法
            public void changedUpdate(DocumentEvent e){}
            // 当向 Document 中插入文本时触发该方法
            public void insertUpdate(DocumentEvent e)
            {
                docChangeStart = e.getOffset();
                docChangeLength = e.getLength();
            }
            // 当从 Document 中删除文本时触发该方法
            public void removeUpdate(DocumentEvent e){}
        });
    }
    public void syntaxParse()
    {
        try
        {
            // 获取文档的根元素，即文档内的全部内容
            Element root = doc.getDefaultRootElement();
            // 获取文档中光标插入符的位置
            int cursorPos = this.getCaretPosition();
            int line = root.getElementIndex(cursorPos);
            // 获取光标所在位置的行
            Element para = root.getElement(line);
            // 定义光标所在行的行头在文档中的位置
            int start = para.getStartOffset();
            // 让 start 等于 start 与 docChangeStart 中的较小值
            start = start > docChangeStart ? docChangeStart :start;
```

```
            // 定义被修改部分的长度
            int length = para.getEndOffset() - start;
            length = length < docChangeLength ? docChangeLength + 1
                : length;
            // 取出所有可能被修改的字符串
            String s = doc.getText(start, length);
            // 以空格、点号等作为分隔符
            String[] tokens = s.split("\\s+|\\.|\\(|\\)|\\{|\\}|\\[|\\]");
            // 定义当前分析单词在 s 字符串中的开始位置
            int curStart = 0;
            // 定义单词是否处于引号内
            boolean isQuot = false;
            for (String token : tokens)
            {
                // 找出当前分析单词在 s 字符串中的位置
                int tokenPos = s.indexOf(token , curStart);
                if (isQuot && (token.endsWith("\"") || token.endsWith("\'")))
                {
                    doc.setCharacterAttributes(start + tokenPos
                        , token.length(), quotAttr, false);
                    isQuot = false;
                }
                else if (isQuot && !(token.endsWith("\"")
                    || token.endsWith("\'")))
                {
                    doc.setCharacterAttributes(start + tokenPos
                        , token.length(), quotAttr, false);
                }
                else if ((token.startsWith("\"") || token.startsWith("\'"))
                    && (token.endsWith("\"") || token.endsWith("\'")))
                {
                    doc.setCharacterAttributes(start + tokenPos
                        , token.length(), quotAttr, false);
                }
                else if ((token.startsWith("\"") || token.startsWith("\'"))
                    && !(token.endsWith("\"") || token.endsWith("\'")))
                {
                    doc.setCharacterAttributes(start + tokenPos
                        , token.length(), quotAttr, false);
                    isQuot = true;
                }
                else
                {
                    // 使用格式器对当前单词设置颜色
                    formatter.setHighLight(doc , token , start + tokenPos
                        , token.length());
                }
                // 开始分析下一个单词
                curStart = tokenPos + token.length();
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
    // 重画该组件，设置行号
    public void paint(Graphics g)
    {
        super.paint(g);
        Element root = doc.getDefaultRootElement();
        // 获得行号
        int line = root.getElementIndex(doc.getLength());
        // 设置颜色
        g.setColor(new Color(230, 230, 230));
        // 绘制显示行数的矩形框
        g.fillRect(0 , 0 , this.getMargin().left - 10 , getSize().height);
        // 设置行号的颜色
        g.setColor(new Color(40, 40, 40));
        // 每行绘制一个行号
```

```
            for (int count = 0, j = 1; count <= line; count++, j++)
            {
                g.drawString(String.valueOf(j), 3, (int)((count + 1)
                    * 1.535 * StyleConstants.getFontSize(normalAttr)));
            }
        }
        public static void main(String[] args)
        {
            JFrame frame = new JFrame("文本编辑器");
            // 使用 MyTextPane
            frame.getContentPane().add(new JScrollPane(new MyTextPane()));
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            final int inset = 50;
            Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
            frame.setBounds(inset, inset, screenSize.width - inset*2
                , screenSize.height - inset * 2);
            frame.setVisible(true);
        }
    }
    // 定义语法格式器
    class SyntaxFormatter
    {
        // 以一个 Map 保存关键字和颜色的对应关系
        private Map<SimpleAttributeSet , ArrayList<String>> attMap
            = new HashMap<>();
        // 定义文档的正常文本的外观属性
        SimpleAttributeSet normalAttr = new SimpleAttributeSet();
        public SyntaxFormatter(String syntaxFile)
        {
            // 设置正常文本的颜色、大小
            StyleConstants.setForeground(normalAttr, Color.BLACK);
            StyleConstants.setFontSize(normalAttr, 16);
            // 创建一个 Scanner 对象，负责根据语法文件加载颜色信息
            Scanner scaner = null;
            try
            {
                scaner = new Scanner(new File(syntaxFile));
            }
            catch (FileNotFoundException e)
            {
                throw new RuntimeException("丢失语法文件："
                    + e.getMessage());
            }
            int color = -1;
            ArrayList<String> keywords = new ArrayList<>();
            // 不断读取语法文件的内容行
            while(scaner.hasNextLine())
            {
                String line = scaner.nextLine();
                // 如果当前行以#开头
                if (line.startsWith("#"))
                {
                    if (keywords.size() > 0 && color > -1)
                    {
                        // 取出当前行的颜色值，并封装成 SimpleAttributeSet 对象
                        SimpleAttributeSet att = new SimpleAttributeSet();
                        StyleConstants.setForeground(att, new Color(color));
                        StyleConstants.setFontSize(att, 16);
                        // 将当前颜色和关键字 List 对应起来
                        attMap.put(att , keywords);
                    }
                    // 重新创建新的关键字 List，为下一个语法格式做准备
                    keywords = new ArrayList<>();
                    color = Integer.parseInt(line.substring(1) , 16);
                }
                else
                {
                    // 对于普通行，将每行内容添加到关键字 List 里
```

```
                if (line.trim().length() > 0)
                {
                    keywords.add(line.trim());
                }
            }
        }
        // 把所有的关键字和颜色对应起来
        if (keywords.size() > 0 && color > -1)
        {
            SimpleAttributeSet att = new SimpleAttributeSet();
            StyleConstants.setForeground(att, new Color(color));
            StyleConstants.setFontSize(att, 16);
            attMap.put(att , keywords);
        }
    }
    // 返回该格式器里正常文本的外观属性
    public SimpleAttributeSet getNormalAttributeSet()
    {
        return normalAttr;
    }
    // 设置语法高亮
    public void setHighLight(StyledDocument doc , String token
        , int start , int length)
    {
        // 保存当前单词对应的外观属性
        SimpleAttributeSet currentAttributeSet = null;
        outer :
        for (SimpleAttributeSet att : attMap.keySet())
        {
            // 取出当前颜色对应的所有关键字
            ArrayList<String> keywords = attMap.get(att);
            // 遍历所有关键字
            for (String keyword : keywords)
            {
                // 如果该关键字与当前单词相同
                if (keyword.equals(token))
                {
                    // 跳出循环，并设置当前单词对应的外观属性
                    currentAttributeSet = att;
                    break outer;
                }
            }
        }
        // 如果当前单词对应的外观属性不为空
        if (currentAttributeSet != null)
        {
            // 设置当前单词的颜色
            doc.setCharacterAttributes(start , length
                , currentAttributeSet , false);
        }
        // 否则使用普通外观来设置该单词
        else
        {
            doc.setCharacterAttributes(start , length , normalAttr , false);
        }
    }
}
```

上面程序中的粗体字代码负责分析当前单词与哪种颜色关键字匹配，并为这段文字设置字体颜色。其实这段程序为文档中的单词设置颜色并不难，难点在于找出每个单词与哪种关键字匹配，并要标识出该单词在文档中的位置，然后才可以为该单词设置颜色。

运行上面程序，会看到如图 12.62 所示的带语法高亮的文本编辑器。

上面程序已经完成了对不同类型的单词进行着色，所以会看到如图 12.62 示的运行界面。如果进行改进，则可以为上面的编辑器增加括号配对、代码折叠等功能，这些都可以通过 JTextPane 组件来完成。对于此文本编辑器，只要传入不同的语法文件，程序就可以为不同的源代码显示语法高亮。

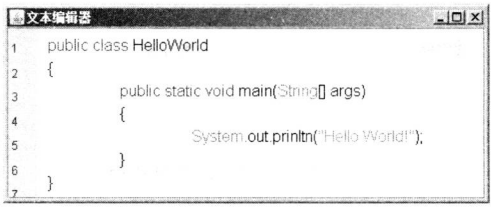


图 12.62 带语法高亮的文本编辑器

12.13 本章小结

本章与前一章内容的结合性非常强，本章主要介绍了以 AWT 为基础的 Swing 编程知识。本章简要介绍了 Swing 基本组件如对话框、按钮的用法，还详细介绍了 Swing 所提供的特殊容器。除此之外，本章重点介绍了 Swing 提供的特殊控件：JList、JComboBox、JSpinner、JSlider、JTable、JTree 等，介绍 JTable、JTree 时深入介绍了 Swing 的 MVC 实现机制，并通过提供自定义的 Render 来改变页面 JTable、JTree 的外观效果。

本章练习

1．设计俄罗斯方块游戏。

2．设计仿 ACDSee 的看图程序。

3．结合 JTree、JList、JSplitPane、JDesktopPane、JInternalFrame、JTextPane 等组件，开发仿 EditPlus 的文字编辑程序界面，可以暂时不提供文字保存、文字打开等功能。

CHAPTER

13

第13章 MySQL数据库与JDBC编程

本章要点

- 关系数据库和 SQL 语句
- DML 语句的语法
- DDL 语句的语法
- 简单查询语句的语法
- 多表连接查询
- 子查询
- JDBC 数据库编程步骤
- 执行 SQL 语句的三种方法
- 使用 PreparedStatement 执行 SQL 语句
- 使用 CallableStatement 调用存储过程
- 使用 ResultSetMetaData 分析结果集元数据
- 理解并掌握 RowSet、RowSetFactory
- 离线 RowSet
- 使用 RowSet 控制分页
- 使用 DatabaseMetaData 分析数据库元数据
- 事务的基础知识
- SQL 语句中的事务控制
- JDBC 编程中的事务控制

通过使用 JDBC，Java 程序可以非常方便地操作各种主流数据库，这是 Java 语言的巨大魅力所在。由于 Java 语言的跨平台特性，所以使用 JDBC 编写的程序不仅可以实现跨数据库，还可以跨平台，具有非常优秀的可移植性。

程序使用 JDBC API 以统一的方式来连接不同的数据库，然后通过 Statement 对象来执行标准的 SQL 语句，并可以获得 SQL 语句访问数据库的结果，因此掌握标准的 SQL 语句是学习 JDBC 编程的基础。本章将会简要介绍关系数据库理论基础，并以 MySQL 数据库为例来讲解标准的 SQL 语句的语法细节，包括基本查询语句、多表连接查询和子查询等。

本章将重点介绍 JDBC 连接数据库的详细步骤，并讲解使用 JDBC 执行 SQL 语句的各种方式，包括使用 CallableStatement 调用存储过程等。本章还会介绍 ResultSetMetaData、DatabaseMetaData 两个接口的用法。事务也是数据库编程中的重要概念，本章不仅会介绍标准 SQL 语句中的事务控制语句，而且会讲解如何利用 JDBC API 进行事务控制。

13.1　JDBC 基础

JDBC 的全称是 Java Database Connectivity，即 Java 数据库连接，它是一种可以执行 SQL 语句的 Java API。程序可通过 JDBC API 连接到关系数据库，并使用结构化查询语言（SQL，数据库标准的查询语言）来完成对数据库的查询、更新。

与其他数据库编程环境相比，JDBC 为数据库开发提供了标准的 API，所以使用 JDBC 开发的数据库应用可以跨平台运行，而且可以跨数据库（如果全部使用标准的 SQL）。也就是说，如果使用 JDBC 开发一个数据库应用，则该应用既可以在 Windows 平台上运行，也可以在 UNIX 等其他平台上运行；既可以使用 MySQL 数据库，也可以使用 Oracle 等数据库，而程序无须进行任何修改。

13.1.1　JDBC 简介

通过使用 JDBC，就可以使用同一种 API 访问不同的数据库系统。换言之，有了 JDBC API，就不必为访问 Oracle 数据库学习一组 API，为访问 DB2 数据库又学习一组 API……开发人员面向 JDBC API 编写应用程序，然后根据不同的数据库，使用不同的数据库驱动程序即可。

提示：

最早的时候，Sun 公司希望自己开发一组 Java API，程序员通过这组 Java API 即可操作所有的数据库系统，但后来 Sun 发现这个目标具有不可实现性——因为数据库系统太多了，而且各数据库系统的内部特性又各不相同。后来 Sun 就制定了一组标准的 API，它们只是接口，没有提供实现类——这些实现类由各数据库厂商提供实现，这些实现类就是驱动程序。而程序员使用 JDBC 时只要面向标准的 JDBC API 编程即可，当需要在数据库之间切换时，只要更换不同的实现类（即更换数据库驱动程序）就行，这是面向接口编程的典型应用。

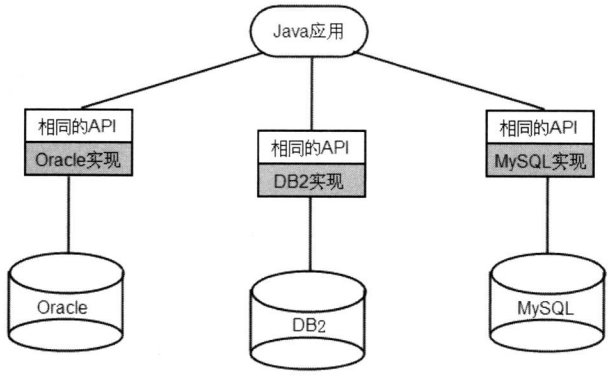


图 13.1　JDBC 驱动示意图

Java 语言的各种跨平台特性，都采用相似的结构，因为它们都需要让相同的程序在不同的平台上运行，所以都需要中间的转换程序（为了实现 Java 程序的跨平台性，Java 为不同的操作系统提供了不同的虚拟机）。同样，为了使 JDBC 程序可以跨平台，则需要不同的数据库厂商提供相应的驱动程序。图 13.1 显示了 JDBC 驱动示意图。

正是通过 JDBC 驱动的转换，才使得使用相同 JDBC API 编写的程序，在不同的数据库系统上运行良好。Sun 提供的 JDBC 可以完成以下三个基本工作。

- 建立与数据库的连接。
- 执行 SQL 语句。
- 获得 SQL 语句的执行结果。

通过 JDBC 的这三个功能，应用程序即可访问、操作数据库系统。

13.1.2 JDBC 驱动程序

数据库驱动程序是 JDBC 程序和数据库之间的转换层，数据库驱动程序负责将 JDBC 调用映射成特定的数据库调用。图 13.2 显示了 JDBC 示意图。

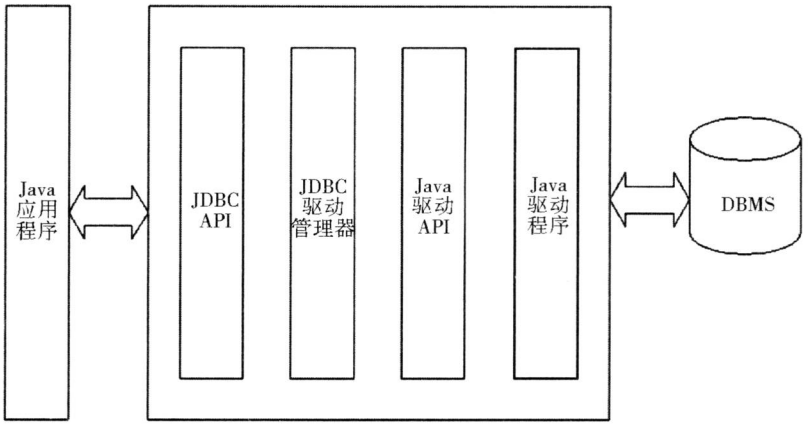


图 13.2　JDBC 访问示意图

大部分数据库系统，例如 Oracle 和 Sybase 等，都有相应的 JDBC 驱动程序，当需要连接某个特定的数据库时，必须有相应的数据库驱动程序。

提示：

还有一种名为 ODBC 的技术，其全称是 Open Database Connectivity，即开放数据库连接。ODBC 和 JDBC 很像，严格来说，应该是 JDBC 模仿了 ODBC 的设计。ODBC 也允许应用程序通过一组通用的 API 访问不同的数据库管理系统，从而使得基于 ODBC 的应用程序可以在不同的数据库之间切换。同样，ODBC 也需要各数据库厂商提供相应的驱动程序，而 ODBC 则负责管理这些驱动程序。

JDBC 驱动通常有如下 4 种类型。

- 第 1 种 JDBC 驱动：称为 JDBC-ODBC 桥，这种驱动是最早实现的 JDBC 驱动程序，主要目的是为了快速推广 JDBC。这种驱动将 JDBC API 映射到 ODBC API。这种方式在 Java 8 中已经被删除了。
- 第 2 种 JDBC 驱动：直接将 JDBC API 映射成数据库特定的客户端 API。这种驱动包含特定数据库的本地代码，用于访问特定数据库的客户端。
- 第 3 种 JDBC 驱动：支持三层结构的 JDBC 访问方式，主要用于 Applet 阶段，通过 Applet 访问数据库。
- 第 4 种 JDBC 驱动：是纯 Java 的，直接与数据库实例交互。这种驱动是智能的，它知道数据库使用的底层协议。这种驱动是目前最流行的 JDBC 驱动。

注意：

早期为了让 Java 程序操作 Access 这种伪数据库，可能需要使用 JDBC-ODBC 桥，但 JDBC-ODBC 桥不适合在并发访问数据库的情况下使用，其固有的性能和扩展能力也非常有限，因此 Java 8 删除了 JDBC-ODBC 桥驱动。基本上 Java 应用也很少使用 Access 这种伪数据库。

通常建议选择第 4 种 JDBC 驱动，这种驱动避开了本地代码，减少了应用开发的复杂性，也减少了

产生冲突和出错的可能。如果对性能有严格的要求，则可以考虑使用第 2 种 JDBC 驱动，但使用这种驱动，则势必增加编码和维护的困难。

相对于 ODBC 而言，JDBC 更加简单。总结起来，JDBC 比 ODBC 多了如下几个优势。

- ODBC 更复杂，ODBC 中有几个命令需要配置很多复杂的选项，而 JDBC 则采用简单、直观的方式来管理数据库连接。
- JDBC 比 ODBC 安全性更高，更易部署。

13.2 SQL 语法

SQL 语句是对所有关系数据库都通用的命令语句，而 JDBC API 只是执行 SQL 语句的工具，JDBC 允许对不同的平台、不同的数据库采用相同的编程接口来执行 SQL 语句。在开始 JDBC 编程之前必须掌握基本的 SQL 知识，本节将以 MySQL 数据库为例详细介绍 SQL 语法知识。

提示：

除标准的 SQL 语句之外，所有的数据库都会在标准 SQL 语句基础上进行扩展，增加一些额外的功能，这些额外的功能属于特定的数据库系统，不能在所有的数据库系统上都通用。因此，如果想让数据库应用程序可以跨数据库运行，则应该尽量少用这些属于特定数据库的扩展。

13.2.1 安装数据库

对于基于 JDBC 的应用程序，如果使用标准的 SQL 语句进行数据库操作，则应用程序可以在所有的数据库之间切换，只要为程序提供不同的数据库驱动程序即可。从这个角度来看，我们可以使用任何一种数据库来学习 JDBC 编程。本章将以 MySQL 为例来介绍 JDBC 编程，因为 MySQL 数据库非常小巧，而且使用相当简单。

提示：

对初学者不推荐使用 Microsoft 的 SQL Server 作为 JDBC 应用的数据库，因为 Microsoft 为 SQL Server 提供的 JDBC 驱动偶尔会出现未知异常，这些异常会影响初学者学习的心情。

安装 MySQL 数据库与安装普通程序并没有太大的区别，关键是配置 MySQL 数据库时需要注意选择支持中文的编码集。下面简要介绍在 Windows 平台上下载和安装 MySQL 数据库系统的步骤。

① 登录 http://dev.mysql.com/downloads/mysql/站点，下载 MySQL 数据库的最新版本。本书成书之时，MySQL 数据库的最新稳定版本是 MySQL 5.7.19，建议下载该版本的 MySQL 安装文件。读者可根据自己所用的 Windows 平台选择下载相应的 MSI Installer 安装文件。

② 下载完成后，得到一个 mysql-installer-community-5.7.19.0.msi 文件，双击该文件，开始安装 MySQL 数据库系统，安装 MySQL 数据库系统与安装普通的 Windows 软件没有太大差别。

③ 开始安装 MySQL 后，在出现的对话框中单击“Install MySQL Products”按钮，然后看到“License Agreement”界面，该界面要求用户必须接受该协议才能安装 MySQL 数据库系统。勾选该界面下方的“I accept the license terms”复选框，然后单击“Next”按钮。

④ 显示如图 13.3 所示的安装选项对话框，勾选“Custom”单选钮，然后单击“Next”按钮。在该界面可以选择安装 MySQL 所需的组件和选择 MySQL 数据库及数据文件的安装路径，本书选择将 MySQL 数据库和数据文件都安装在 D 盘下。单击“Next”按钮，将显示选择安装组件对话框，选择安装 MySQL 服务器和文档，如图 13.4 所示。

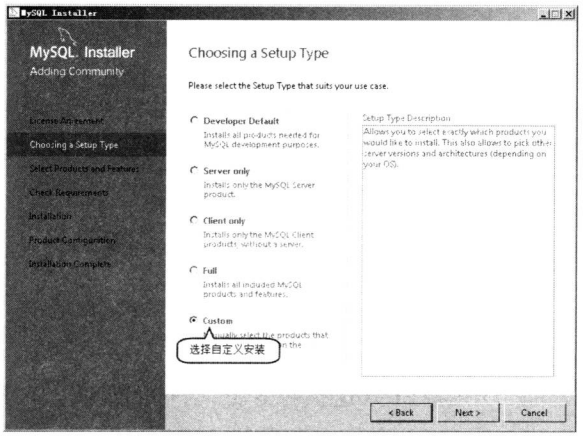


图 13.3　选择自定义安装

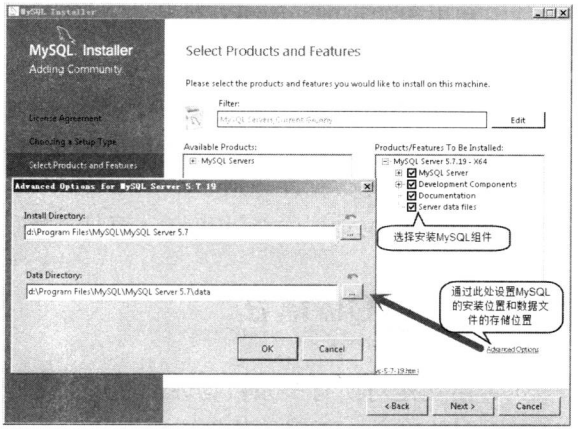


图 13.4　只安装 MySQL 服务器和文档并设置数据库和数据文件的安装位置

⑤ 单击“Next”按钮，MySQL Installer 会检查系统环境是否满足安装 MySQL 的要求。如果满足要求，则可以直接单击“Next”按钮开始安装；如果不符合条件，请根据 MySQL 提示先安装相应的系统组件，然后再重新安装 MySQL。开始安装 MySQL 数据库系统。

⑥ 成功安装 MySQL 数据库系统后，会看到如图 13.5 所示的成功安装对话框。

注意：

MySQL 需要 Visual Studio 2013 Redistributable，而且不管你的操作系统是 32 位还是 64 位的，它始终需要 32 位的 Visual Studio 2013 Redistributable，否则会安装失败。

⑦ MySQL 数据库程序安装成功后，系统还要求配置 MySQL 数据库。单击如图 13.5 所示对话框中下方的“Next”按钮，开始配置 MySQL 数据库。在如图 13.6 所示的对话框中，勾选“Standalone MySQL Server/Classic MySQL Replication”复选框，这样即可进行更详细的配置。

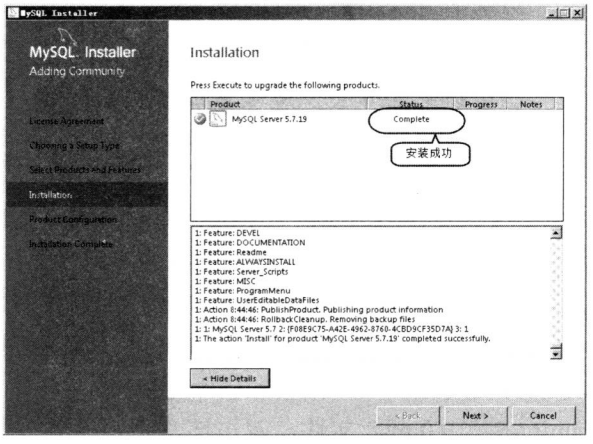


图 13.5　成功安装 MySQL 数据库

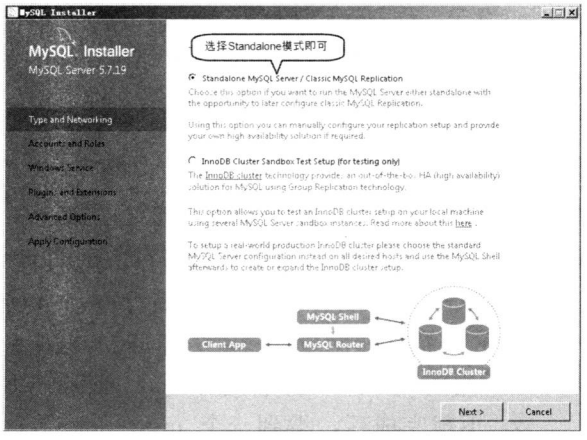


图 13.6　选择进行详细配置

⑧ 单击“Next”按钮，将出现如图 13.7 所示的对话框，允许用户设置 MySQL 的 root 账户密码，也允许添加更多的用户。

⑨ 如果需要为 MySQL 数据库添加更多的用户，则可单击“Add User”按钮进行添加。设置完成后单击“Next”按钮，在配置中将依次出现一系列对话框，但这些对话框对配置影响不大，直接单击“Next”按钮直至 MySQL 配置成功。

MySQL 可通过命令行客户端来管理 MySQL 数据库及数据库里的数据。经过上面 10 个步骤之后，应该在 Windows 的“开始”菜单中看到“MySQL”→“MySQL Server 5.7”→“MySQL 5.7 Command Line

Client - Unicode”菜单项，单击该菜单项将启动 MySQL 的命令行客户端窗口，进入该窗口将会提示输入 root 账户密码。

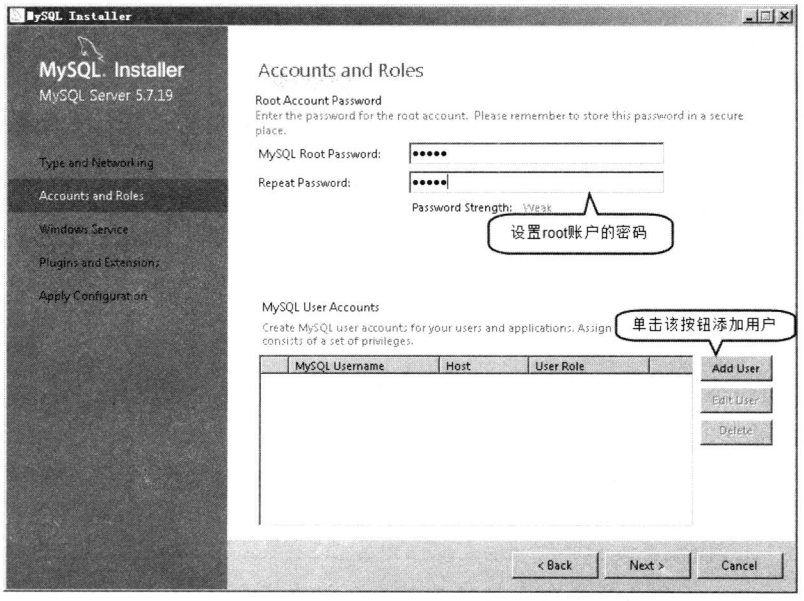


图 13.7　设置 root 账户密码和添加新用户

提示：由于 MySQL 默认使用 UTF-8 字符串，因此应该通过“MySQL 5.7 Command Line Client - Unicode”菜单项启动命令行工具，该工具将会使用 UTF-8 字符集。

提示：市面上有一个名为 SQLyog 的工具程序提供了较好的图形用户界面来管理 MySQL 数据库的数据。除此之外，MySQL 也提供了 MySQLAdministrator 工具来管理 MySQL 数据库。读者可以自行下载这两个工具，并使用这两个工具来管理 MySQL 数据库。但本书依然推荐读者使用命令行窗口，因为这种“恶劣”的工具会强制读者记住 SQL 命令的详细用法。

在命令行客户端工具中输入在如图 13.7 所示对话框中为 root 账户设定的密码，系统进入 MySQL 数据库系统，通过执行 SQL 命令就可以管理 MySQL 数据库系统了。

13.2.2　关系数据库基本概念和 MySQL 基本命令

严格来说，数据库（Database）仅仅是存放用户数据的地方。当用户访问、操作数据库中的数据时，就需要数据库管理系统的帮助。数据库管理系统的全称是 Database Management System，简称 DBMS。习惯上常常把数据库和数据库管理系统笼统地称为数据库，通常所说的数据库既包括存储用户数据的部分，也包括管理数据库的管理系统。

DBMS 是所有数据的知识库，它负责管理数据的存储、安全、一致性、并发、恢复和访问等操作。DBMS 有一个数据字典（有时也被称为系统表），用于存储它拥有的每个事务的相关信息，例如名字、结构、位置和类型，这种关于数据的数据也被称为元数据（metadata）。

在数据库发展历史中，按时间顺序主要出现了如下几种类型的数据库系统。

- 网状型数据库
- 层次型数据库
- 关系数据库
- 面向对象数据库

在上面 4 种数据库系统中，关系数据库是理论最成熟、应用最广泛的数据库。从 20 世纪 70 年代末

开始，关系数据库理论逐渐成熟，随之涌现出大量商用的关系数据库。关系数据库理论经过30多年的发展已经相当完善，在大量数据的查找、排序操作上非常成熟且快速，并对数据库系统的并发、隔离有非常完善的解决方案。

面向对象数据库则是由面向对象编程语言催生的新型数据库，目前有些数据库系统如Oracle 11g等开始增加面向对象特性，但面向对象数据库还没有大规模地商业应用。

对于关系数据库而言，最基本的数据存储单元就是数据表，因此可以简单地把数据库想象成大量数据表的集合（当然，数据库绝不仅由数据表组成）。

数据表是存储数据的逻辑单元，可以把数据表想象成由行和列组成的表格，其中每一行也被称为一条记录，每一列也被称为一个字段。为数据库建表时，通常需要指定该表包含多少列，每列的数据类型信息，无须指定该数据表包含多少行——因为数据库表的行是动态改变的，每行用于保存一条用户数据。除此之外，还应该为每个数据表指定一个特殊列，该特殊列的值可以唯一地标识此行的记录，则该特殊列被称为主键列。

MySQL数据库的一个实例（Server Instance）可以同时包含多个数据库，MySQL使用如下命令来查看当前实例下包含多少个数据库：

```
show databases;
```

注意：

MySQL默认以分号作为每条命令的结束符，所以在每条MySQL命令结束后都应该输一个英文分号（;）。

如果用户需要创建新的数据库，则可以使用如下命令：

```
create database [IF NOT EXISTS] 数据库名;
```

如果用户需要删除指定数据库，则可以使用如下命令：

```
drop database 数据库名;
```

建立了数据库之后，如果想操作该数据库（例如为该数据库建表，在该数据库中执行查询等操作），则需要进入该数据库。进入指定数据库可以使用如下命令：

```
use 数据库名;
```

进入指定数据库后，如果需要查询该数据库下包含多少个数据表，则可以使用如下命令：

```
show tables;
```

如果想查看指定数据表的表结构（查看该表有多少列，每列的数据类型等信息），则可以使用如下命令：

```
desc 表名
```

图13.8显示了使用MySQL命令行客户端执行这些命令的效果。

正如在图13.8中看到的，MySQL的命令行客户端依次执行了show databases;、drop database abc;等命令，如果将多条MySQL命令写在一份SQL脚本文件里，然后将这份SQL脚本的内容一次复制到该窗口里，将可以看到该命令行客户端一次性执行所有SQL命令的效果——这种一次性执行多条SQL命令的方式也被称为导入SQL脚本。

提示：

本章的大量程序需要相应数据库的支持，因为本章的大部分程序都会提供对应的SQL脚本，因此运行这些程序之前，应该先向MySQL数据库中导入这些SQL脚本。

MySQL数据库安装成功后，在其安装目录下有一个bin路径（本书中该路径为D:\Program Files\MySQL\MySQL Server 5.7\bin），该路径下包含一个mysql命令，该命令用于启动MySQL命令行客户端。执行mysql命令的语法如下：

```
mysql -p 密码 -u 用户名 -h 主机名 --default-character-set=utf8
```

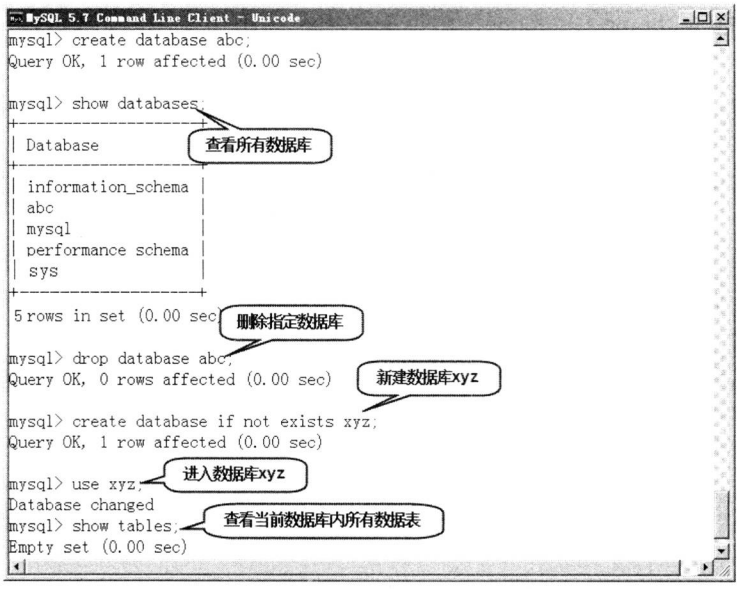


图 13.8　执行 MySQL 常用命令

执行上面命令可以连接远程主机的 MySQL 服务。为了保证有较好的安全性，执行上面命令时可以省略-p 后面的密码，执行该命令后系统会提示输入密码。

提示：

为了更方便地使用该命令，可以将该 MySQL 安装路径下的 bin 目录添加到系统 PATH 环境变量中。实际上，“开始”菜单中的“MySQL 5.7 Command Line Client - Unicode”菜单项就是一条 mysql 命令。

MySQL 数据库通常支持如下两种存储机制。

- MyISAM：这是 MySQL 早期默认的存储机制，对事务支持不够好。
- InnoDB：InnoDB 提供事务安全的存储机制。InnoDB 通过建立行级锁来保证事务完整性，并以 Oracle 风格的共享锁来处理 Select 语句。系统默认启动 InnoDB 存储机制，如果不想使用 InnoDB 表，则可以使用 skip-innodb 选项。

对比两种存储机制，不难发现 InnoDB 比 MyISAM 多了事务支持的功能，而事务支持是 Java EE 最重要的特性，因此通常推荐使用 InnoDB 存储机制。如果使用 5.0 以上版本的 MySQL 数据库系统，通常无须指定数据表的存储机制，因为系统默认使用 InnoDB 存储机制。如果需要在建表时显式指定存储机制，则可在标准建表语法的后面添加下面任意一句。

- ENGINE=MyISAM——强制使用 MyISAM 存储机制。
- ENGINE=InnoDB——强制使用 InnoDB 存储机制。

13.2.3　SQL 语句基础

SQL 的全称是 Structured Query Language，也就是结构化查询语言。SQL 是操作和检索关系数据库的标准语言，标准的 SQL 语句可用于操作任何关系数据库。

使用 SQL 语句，程序员和数据库管理员（DBA）可以完成如下任务。

- 在数据库中检索信息。
- 对数据库的信息进行更新。
- 改变数据库的结构。
- 更改系统的安全设置。
- 增加或回收用户对数据库、表的许可权限。

在上面 5 个任务中，一般程序员可以管理前 3 个任务，后面 2 个任务通常由 DBA 来完成。

标准的 SQL 语句通常可分为如下几种类型。

- 查询语句：主要由 select 关键字完成，查询语句是 SQL 语句中最复杂、功能最丰富的语句。