```
    // 在接口中定义默认方法，需要使用 default 修饰
    default void print(String... msgs)
    {
        for (String msg : msgs)
        {
            System.out.println(msg);
        }
    }
    // 在接口中定义默认方法，需要使用 default 修饰
    default void test()
    {
        System.out.println("默认的 test()方法");
    }
    // 在接口中定义类方法，需要使用 static 修饰
    static String staticTest()
    {
        return "接口里的类方法";
    }
}
```

上面定义了一个 Output 接口，这个接口里包含了一个成员变量：MAX_CACHE_LINE。除此之外，这个接口还定义了两个普通方法：表示取得数据的 getData()方法和表示输出的 out()方法。这就定义了 Output 接口的规范：只要某个类能取得数据，并可以将数据输出，那它就是一个输出设备，至于这个设备的实现细节，这里暂时不关心。

Java 8 允许在接口中定义默认方法，默认方法必须使用 default 修饰，该方法不能使用 static 修饰，无论程序是否指定，默认方法总是使用 public 修饰——如果开发者没有指定 public，系统会自动为默认方法添加 public 修饰符。由于默认方法并不没有 static 修饰，因此不能直接使用接口来调用默认方法，需要使用接口的实现类的实例来调用这些默认方法。

Java 8 允许在接口中定义类方法，类方法必须使用 static 修饰，该方法不能使用 default 修饰，无论程序是否指定，类方法总是使用 public 修饰——如果开发者没有指定 public，系统会自动为类方法添加 public 修饰符。类方法可以直接使用接口来调用。

接口里的成员变量默认是使用 public static final 修饰的，因此即使另一个类处于不同包下，也可以通过接口来访问接口里的成员变量。例如下面程序。

程序清单：codes\06\6.6\OutputFieldTest.java

```
package yeeku;
public class OutputFieldTest
{
    public static void main(String[] args)
    {
        // 访问另一个包中的 Output 接口的 MAX_CACHE_LINE
        System.out.println(lee.Output.MAX_CACHE_LINE);
        // 下面语句将引发"为 final 变量赋值"的编译异常
        // lee.Output.MAX_CACHE_LINE = 20;
        // 使用接口来调用类方法
        System.out.println(lee.Output.staticTest());
    }
}
```

从上面 main()方法中可以看出，OutputFieldTest 与 Output 处于不同包下，但可以访问 Output 的 MAX_CACHE_LINE 常量，这表明该成员变量是 public 访问权限的，而且可通过接口来访问该成员变量，表明这个成员变量是一个类变量；当为这个成员变量赋值时引发"为 final 变量赋值"的编译异常，表明这个成员变量使用了 final 修饰。

注意：

从某个角度来看，接口可被当成一个特殊的类，因此一个 Java 源文件里最多只能有一个 public 接口，如果一个 Java 源文件里定义了一个 public 接口，则该源文件的主文件名必须与该接口名相同。



6.6.3 接口的继承

接口的继承和类继承不一样，接口完全支持多继承，即一个接口可以有多个直接父接口。和类继承相似，子接口扩展某个父接口，将会获得父接口里定义的所有抽象方法、常量。

一个接口继承多个父接口时，多个父接口排在 extends 关键字之后，多个父接口之间以英文逗号（,）隔开。下面程序定义了三个接口，第三个接口继承了前面两个接口。

程序清单：codes\06\6.6\InterfaceExtendsTest.java

```
interface interfaceA
{
    int PROP_A = 5;
    void testA();
}
interface interfaceB
{
    int PROP_B = 6;
    void testB();
}
interface interfaceC extends interfaceA, interfaceB
{
    int PROP_C = 7;
    void testC();
}
public class InterfaceExtendsTest
{
    public static void main(String[] args)
    {
        System.out.println(interfaceC.PROP_A);
        System.out.println(interfaceC.PROP_B);
        System.out.println(interfaceC.PROP_C);
    }
}
```

上面程序中的 interfaceC 接口继承了 interfaceA 和 interfaceB，所以 interfaceC 中获得了它们的常量，因此在 main()方法中看到通过 interfaceC 来访问 PROP_A、PROP_B 和 PROP_C 常量。

6.6.4 使用接口

接口不能用于创建实例，但接口可以用于声明引用类型变量。当使用接口来声明引用类型变量时，这个引用类型变量必须引用到其实现类的对象。除此之外，接口的主要用途就是被实现类实现。归纳起来，接口主要有如下用途。

- 定义变量，也可用于进行强制类型转换。
- 调用接口中定义的常量。
- 被其他类实现。

一个类可以实现一个或多个接口，继承使用 extends 关键字，实现则使用 implements 关键字。因为一个类可以实现多个接口，这也是 Java 为单继承灵活性不足所做的补充。类实现接口的语法格式如下：

```
[修饰符] class 类名 extends 父类 implements 接口1,接口2...
{
    类体部分
}
```

实现接口与继承父类相似，一样可以获得所实现接口里定义的常量（成员变量）、方法（包括抽象方法和默认方法）。

让类实现接口需要类定义后增加 implements 部分，当需要实现多个接口时，多个接口之间以英文逗号（,）隔开。一个类可以继承一个父类，并同时实现多个接口，implements 部分必须放在 extends 部分之后。

一个类实现了一个或多个接口之后，这个类必须完全实现这些接口里所定义的全部抽象方法（也就是重写这些抽象方法）；否则，该类将保留从父接口那里继承到的抽象方法，该类也必须定义成抽象类。

一个类实现某个接口时，该类将会获得接口中定义的常量（成员变量）、方法等，因此可以把实现

接口理解为一种特殊的继承，相当于实现类继承了一个彻底抽象的类（相当于除了默认方法外，所有方法都是抽象方法的类）。

下面看一个实现接口的类。

程序清单：codes\06\6.6\Printer.java

```
// 定义一个 Product 接口
interface Product
{
    int getProduceTime();
}
// 让 Printer 类实现 Output 和 Product 接口
public class Printer implements Output , Product
{
    private String[] printData
        = new String[MAX_CACHE_LINE];
    // 用以记录当前需打印的作业数
    private int dataNum = 0;
    public void out()
    {
        // 只要还有作业，就继续打印
        while(dataNum > 0)
        {
            System.out.println("打印机打印：" + printData[0]);
            // 把作业队列整体前移一位，并将剩下的作业数减 1
            System.arraycopy(printData , 1
                , printData, 0, --dataNum);
        }
    }
    public void getData(String msg)
    {
        if (dataNum >= MAX_CACHE_LINE)
        {
            System.out.println("输出队列已满，添加失败");
        }
        else
        {
            // 把打印数据添加到队列里，已保存数据的数量加 1
            printData[dataNum++] = msg;
        }
    }
    public int getProduceTime()
    {
        return 45;
    }
    public static void main(String[] args)
    {
        // 创建一个 Printer 对象，当成 Output 使用
        Output o = new Printer();
        o.getData("轻量级 Java EE 企业应用实战");
        o.getData("疯狂 Java 讲义");
        o.out();
        o.getData("疯狂 Android 讲义");
        o.getData("疯狂 Ajax 讲义");
        o.out();
        // 调用 Output 接口中定义的默认方法
        o.print("孙悟空" , "猪八戒" , "白骨精");
        o.test();
        // 创建一个 Printer 对象，当成 Product 使用
        Product p = new Printer();
        System.out.println(p.getProduceTime());
        // 所有接口类型的引用变量都可直接赋给 Object 类型的变量
        Object obj = p;
    }
}
```

从上面程序中可以看出，Printer 类实现了 Output 接口和 Product 接口，因此 Printer 对象既可直接

赋给 Output 变量，也可直接赋给 Product 变量。仿佛 Printer 类既是 Output 类的子类，也是 Product 类的子类，这就是 Java 提供的模拟多继承。

上面程序中 Printer 实现了 Output 接口，即可获取 Output 接口中定义的 print()和 test()两个默认方法，因此 Printer 实例可以直接调用这两个默认方法。

注意：

实现接口方法时，必须使用 public 访问控制修饰符，因为接口里的方法都是 public 的，而子类（相当于实现类）重写父类方法时访问权限只能更大或者相等，所以实现类实现接口里的方法时只能使用 public 访问权限。



接口不能显式继承任何类，但所有接口类型的引用变量都可以直接赋给 Object 类型的引用变量。所以在上面程序中可以把 Product 类型的变量直接赋给 Object 类型变量，这是利用向上转型来实现的，因为编译器知道任何 Java 对象都必须是 Object 或其子类的实例，Product 类型的对象也不例外（它必须是 Product 接口实现类的对象，该实现类肯定是 Object 的显式或隐式子类）。

6.6.5 接口和抽象类

接口和抽象类很像，它们都具有如下特征。

- 接口和抽象类都不能被实例化，它们都位于继承树的顶端，用于被其他类实现和继承。
- 接口和抽象类都可以包含抽象方法，实现接口或继承抽象类的普通子类都必须实现这些抽象方法。

但接口和抽象类之间的差别非常大，这种差别主要体现在二者设计目的上。下面具体分析二者的差别。

接口作为系统与外界交互的窗口，接口体现的是一种规范。对于接口的实现者而言，接口规定了实现者必须向外提供哪些服务（以方法的形式来提供）；对于接口的调用者而言，接口规定了调用者可以调用哪些服务，以及如何调用这些服务（就是如何来调用方法）。当在一个程序中使用接口时，接口是多个模块间的耦合标准；当在多个应用程序之间使用接口时，接口是多个程序之间的通信标准。

从某种程度上来看，接口类似于整个系统的“总纲”，它制定了系统各模块应该遵循的标准，因此一个系统中的接口不应该经常改变。一旦接口被改变，对整个系统甚至其他系统的影响将是辐射式的，导致系统中大部分类都需要改写。

抽象类则不一样，抽象类作为系统中多个子类的共同父类，它所体现的是一种模板式设计。抽象类作为多个子类的抽象父类，可以被当成系统实现过程中的中间产品，这个中间产品已经实现了系统的部分功能（那些已经提供实现的方法），但这个产品依然不能当成最终产品，必须有更进一步的完善，这种完善可能有几种不同方式。

除此之外，接口和抽象类在用法上也存在如下差别。

- 接口里只能包含抽象方法和默认方法，不能为普通方法提供方法实现；抽象类则完全可以包含普通方法。
- 接口里不能定义静态方法；抽象类里可以定义静态方法。
- 接口里只能定义静态常量，不能定义普通成员变量；抽象类里则既可以定义普通成员变量，也可以定义静态常量。
- 接口里不包含构造器；抽象类里可以包含构造器，抽象类里的构造器并不是用于创建对象，而是让其子类调用这些构造器来完成属于抽象类的初始化操作。
- 接口里不能包含初始化块；但抽象类则完全可以包含初始化块。
- 一个类最多只能有一个直接父类，包括抽象类；但一个类可以直接实现多个接口，通过实现多个接口可以弥补 Java 单继承的不足。

6.6.6 面向接口编程

前面已经提到，接口体现的是一种规范和实现分离的设计哲学，充分利用接口可以极好地降低程序各模块之间的耦合，从而提高系统的可扩展性和可维护性。

基于这种原则，很多软件架构设计理论都倡导“面向接口”编程，而不是面向实现类编程，希望通过面向接口编程来降低程序的耦合。下面介绍两种常用场景来示范面向接口编程的优势。

1. 简单工厂模式

有一个场景：假设程序中有个 Computer 类需要组合一个输出设备，现在有两个选择：直接让 Computer 类组合一个 Printer，或者让 Computer 类组合一个 Output，那么到底采用哪种方式更好呢？

假设让 Computer 类组合一个 Printer 对象，如果有一天系统需要重构，需要使用 BetterPrinter 来代替 Printer，这就需要打开 Computer 类源代码进行修改。如果系统中只有一个 Computer 类组合了 Printer 还好，但如果系统中有 100 个类组合了 Printer，甚至 1000 个、10000 个……将意味着需要打开 100 个、1000 个、10000 个类进行修改，这是多么大的工作量啊！

为了避免这个问题，工厂模式建议让 Computer 类组合一个 Output 类型的对象，将 Computer 类与 Printer 类完全分离。Computer 对象实际组合的是 Printer 对象还是 BetterPrinter 对象，对 Computer 而言完全透明。当 Printer 对象切换到 BetterPrinter 对象时，系统完全不受影响。下面是这个 Computer 类的定义代码。

程序清单：codes\06\6.6\Computer.java

```
public class Computer
{
    private Output out;
    public Computer(Output out)
    {
        this.out = out;
    }
    // 定义一个模拟获取字符串输入的方法
    public void keyIn(String msg)
    {
        out.getData(msg);
    }
    // 定义一个模拟打印的方法
    public void print()
    {
        out.out();
    }
}
```

上面的 Computer 类已经完全与 Printer 类分离，只是与 Output 接口耦合。Computer 不再负责创建 Output 对象，系统提供一个 Output 工厂来负责生成 Output 对象。这个 OutputFactory 工厂类代码如下。

程序清单：codes\06\6.6\OutputFactory.java

```
public class OutputFactory
{
    public Output getOutput()
    {
        return new Printer();
    }
    public static void main(String[] args)
    {
        OutputFactory of = new OutputFactory();
        Computer c = new Computer(of.getOutput());
        c.keyIn("轻量级 Java EE 企业应用实战");
        c.keyIn("疯狂 Java 讲义");
        c.print();
    }
}
```

在该 OutputFactory 类中包含了一个 getOutput()方法，该方法返回一个 Output 实现类的实例，该方法负责创建 Output 实例，具体创建哪一个实现类的对象由该方法决定（具体由该方法中的粗体部分控制，当然也可以增加更复杂的控制逻辑）。如果系统需要将 Printer 改为 BetterPrinter 实现类，只需让 BetterPrinter 实现 Output 接口，并改变 OutputFactory 类中的 getOutput()方法即可。

下面是 BetterPrinter 实现类的代码，BetterPrinter 只是对原有的 Printer 进行简单修改，以模拟系统

重构后的改进。

程序清单：codes\06\6.6\BetterPrinter.java

```
public class BetterPrinter implements Output
{
    private String[] printData
        = new String[MAX_CACHE_LINE * 2];
    // 用以记录当前需打印的作业数
    private int dataNum = 0;
    public void out()
    {
        // 只要还有作业，就继续打印
        while(dataNum > 0)
        {
            System.out.println("高速打印机正在打印：" + printData[0]);
            // 把作业队列整体前移一位，并将剩下的作业数减 1
            System.arraycopy(printData , 1, printData, 0, --dataNum);
        }
    }
    public void getData(String msg)
    {
        if (dataNum >= MAX_CACHE_LINE * 2)
        {
            System.out.println("输出队列已满，添加失败");
        }
        else
        {
            // 把打印数据添加到队列里，已保存数据的数量加 1
            printData[dataNum++] = msg;
        }
    }
}
```

上面的 BetterPrinter 类也实现了 Output 接口，因此也可当成 Output 对象使用，于是只要把 OutputFactory 工厂类的 getOutput()方法中粗体部分改为如下代码：

```
return new BetterPrinter();
```

再次运行前面的 OutputFactory.java 程序，发现系统运行时已经改为 BetterPrinter 对象，而不再是原来的 Printer 对象。

通过这种方式，即可把所有生成 Output 对象的逻辑集中在 OutputFactory 工厂类中管理，而所有需要使用 Output 对象的类只需与 Output 接口耦合，而不是与具体的实现类耦合。即使系统中有很多类使用了 Printer 对象，只要 OutputFactory 类的 getOutput()方法生成的 Output 对象是 BetterPrinter 对象，则它们全部都会改为使用 BetterPrinter 对象，而所有程序无须修改，只需要修改 OutputFactory 工厂类的 getOutput()方法实现即可。



提示：

上面介绍的就是一种被称为“简单工厂”的设计模式。所谓设计模式，就是对经常出现的软件设计问题的成熟解决方案。很多人把设计模式想象成非常高深的概念，实际上设计模式仅仅是对特定问题的一种惯性思维。有些学员喜欢抱着一本设计模式的书研究，以期成为一个“高手”（估计他肯定是武侠小说看多了），实际上设计模式的理解必须以足够的代码积累量作为基础。最好是经历过某种苦痛，或者正在经历一种苦痛，就会对设计模式有较深的感受。

2. 命令模式

考虑这样一种场景：某个方法需要完成某一个行为，但这个行为的具体实现无法确定，必须等到执行该方法时才可以确定。具体一点：假设有个方法需要遍历某个数组的数组元素，但无法确定在遍历数组元素时如何处理这些元素，需要在调用该方法时指定具体的处理行为。

这个要求看起来有点奇怪：这个方法不仅需要普通数据可以变化，甚至还有方法执行体也需要变化，难道需要把“处理行为”作为一个参数传入该方法？



提示：

在某些编程语言（如Ruby等）里，确实允许传入一个代码块作为参数。现在Java 8已经增加了Lambda表达式，通过Lambda表达式可以传入代码块作为参数。

对于这样一个需求，必须把“处理行为”作为参数传入该方法，这个“处理行为”用编程来实现就是一段代码。那如何把这段代码传入该方法呢？

可以考虑使用一个 Command 接口来定义一个方法，用这个方法来封装“处理行为”。下面是该 Command 接口的代码。

程序清单：codes\06\6.6\Command.java

```
public interface Command
{
    // 接口里定义的 process 方法用于封装“处理行为”
    void process(int[] target);
}
```

上面的 Command 接口里定义了一个 process()方法，这个方法用于封装“处理行为”，但这个方法没有方法体——因为现在还无法确定这个处理行为。

下面是需要处理数组的处理类，在这个处理类中包含一个 process()方法，这个方法无法确定处理数组的处理行为，所以定义该方法时使用了一个 Command 参数，这个 Command 参数负责对数组的处理行为。该类的程序代码如下。

程序清单：codes\06\6.6\ProcessArray.java

```
public class ProcessArray
{
    public void process(int[] target , Command cmd)
    {
        cmd.process(target);
    }
}
```

通过一个 Command 接口，就实现了让 ProcessArray 类和具体“处理行为”的分离，程序使用 Command 接口代表了对数组的处理行为。Command 接口也没有提供真正的处理，只有等到需要调用 ProcessArray 对象的 process()方法时，才真正传入一个 Command 对象，才确定对数组的处理行为。

下面程序示范了对数组的两种处理方式。

程序清单：codes\06\6.6\CommandTest.java

```
public class CommandTest
{
    public static void main(String[] args)
    {
        ProcessArray pa = new ProcessArray();
        int[] target = {3, -4, 6, 4};
        //第一次处理数组，具体处理行为取决于 PrintCommand
        pa.process(target , new PrintCommand());
        System.out.println("------------------");
        //第二次处理数组，具体处理行为取决于 AddCommand
        pa.process(target , new AddCommand());
    }
}
```

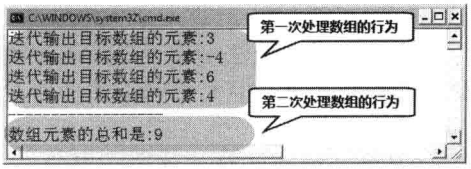


图 6.5　两次处理数组的结果

运行上面程序，看到如图 6.5 所示的界面。

图 6.5 显示了两次不同处理行为的效果，也就实现了 process()方法和“处理行为”的分离，两次不同的处理行为是通过 PrintCommand 类和 AddCommand 类提供的。下面分别是 PrintCommand 类和 AddCommand 类的代码。

程序清单：codes\06\6.6\PrintCommand.java

```
public class PrintCommand implements Command
{
    public void process(int[] target)
    {
        for (int tmp : target )
        {
            System.out.println("迭代输出目标数组的元素:" + tmp);
        }
    }
}
```

程序清单：codes\06\6.6\AddCommand.java

```
public class AddCommand implements Command
{
    public void process(int[] target)
    {
        int sum = 0;
        for (int tmp : target )
        {
            sum += tmp;
        }
        System.out.println("数组元素的总和是:" + sum);
    }
}
```

对于 PrintCommand 和 AddCommand 两个实现类而言，实际有意义的部分就是 process(int[] target)方法，该方法的方法体就是传入 ProcessArray 类里的 process()方法的“处理行为”，通过这种方式就可实现 process()方法和“处理行为”的分离。

6.7 内部类

大部分时候，类被定义成一个独立的程序单元。在某些情况下，也会把一个类放在另一个类的内部定义，这个定义在其他类内部的类就被称为内部类（有的地方也叫嵌套类），包含内部类的类也被称为外部类（有的地方也叫宿主类）。Java 从 JDK 1.1 开始引入内部类，内部类主要有如下作用。

- 内部类提供了更好的封装，可以把内部类隐藏在外部类之内，不允许同一个包中的其他类访问该类。假设需要创建 Cow 类，Cow 类需要组合一个 CowLeg 对象，CowLeg 类只有在 Cow 类里才有效，离开了 Cow 类之后没有任何意义。在这种情况下，就可把 CowLeg 定义成 Cow 的内部类，不允许其他类访问 CowLeg。
- 内部类成员可以直接访问外部类的私有数据，因为内部类被当成其外部类成员，同一个类的成员之间可以互相访问。但外部类不能访问内部类的实现细节，例如内部类的成员变量。
- 匿名内部类适合用于创建那些仅需要一次使用的类。对于前面介绍的命令模式，当需要传入一个 Command 对象时，重新专门定义 PrintCommand 和 AddCommand 两个实现类可能没有太大的意义，因为这两个实现类可能仅需要使用一次。在这种情况下，使用匿名内部类将更方便。

从语法角度来看，定义内部类与定义外部类的语法大致相同，内部类除了需要定义在其他类里面之外，还存在如下两点区别。

- 内部类比外部类可以多使用三个修饰符：private、protected、static——外部类不可以使用这三个修饰符。
- 非静态内部类不能拥有静态成员。

6.7.1 非静态内部类

定义内部类非常简单，只要把一个类放在另一个类内部定义即可。此处的“类内部”包括类中的任何位置，甚至在方法中也可以定义内部类（方法里定义的内部类被称为局部内部类）。内部类定义语法格式如下：

```
public class OuterClass
```

```
{
    // 此处可以定义内部类
}
```

大部分时候，内部类都被作为成员内部类定义，而不是作为局部内部类。成员内部类是一种与成员变量、方法、构造器和初始化块相似的类成员；局部内部类和匿名内部类则不是类成员。

成员内部类分为两种：静态内部类和非静态内部类，使用 static 修饰的成员内部类是静态内部类，没有使用 static 修饰的成员内部类是非静态内部类。

前面经常看到同一个 Java 源文件里定义了多个类，那种情况不是内部类，它们依然是两个互相独立的类。例如下面程序：

```
// 下面A、B两个空类互相独立，没有谁是谁的内部类
class A{}
public class B{}
```

上面两个类定义虽然写在同一个源文件中，但它们互相独立，没有谁是谁的内部类这种关系。内部类一定是放在另一个类的类体部分（也就是类名后的花括号部分）定义。

因为内部类作为其外部类的成员，所以可以使用任意访问控制符如 private、protected 和 public 等修饰。

注意：

外部类的上一级程序单元是包，所以它只有 2 个作用域：同一个包内和任何位置。因此只需 2 种访问权限：包访问权限和公开访问权限，正好对应省略访问控制符和 public 访问控制符。省略访问控制符是包访问权限，即同一包中的其他类可以访问省略访问控制符的成员。因此，如果一个外部类不使用任何访问控制符修饰，则只能被同一个包中其他类访问。而内部类的上一级程序单元是外部类，它就具有 4 个作用域：同一个类、同一个包、父子类和任何位置，因此可以使用 4 种访问控制权限。



下面程序在 Cow 类里定义了一个 CowLeg 非静态内部类，并在 CowLeg 类的实例方法中直接访问 Cow 的 private 访问权限的实例变量。

程序清单：codes\06\6.7\Cow.java

```
public class Cow
{
    private double weight;
    // 外部类的两个重载的构造器
    public Cow(){}
    public Cow(double weight)
    {
        this.weight = weight;
    }
    // 定义一个非静态内部类
    private class CowLeg
    {
        // 非静态内部类的两个实例变量
        private double length;
        private String color;
        // 非静态内部类的两个重载的构造器
        public CowLeg(){}
        public CowLeg(double length , String color)
        {
            this.length = length;
            this.color = color;
        }
        // 下面省略 length、color 的 setter 和 getter 方法
        ...
        // 非静态内部类的实例方法
        public void info()
        {
            System.out.println("当前牛腿颜色是："
                + color + "，高：" + length);
            // 直接访问外部类的 private 修饰的成员变量
```

```
            System.out.println("本牛腿所在奶牛重：" + weight);   //①
        }
    }
    public void test()
    {
        CowLeg cl = new CowLeg(1.12 , "黑白相间");
        cl.info();
    }
    public static void main(String[] args)
    {
        Cow cow = new Cow(378.9);
        cow.test();
    }
}
```

上面程序中粗体字部分是一个普通的类定义，但因为把这个类定义放在了另一个类的内部，所以它就成了一个内部类，可以使用 private 修饰符来修饰这个类。

外部类 Cow 里包含了一个 test()方法，该方法里创建了一个 CowLeg 对象，并调用该对象的 info()方法。读者不难发现，在外部类里使用非静态内部类时，与平时使用普通类并没有太大的区别。

编译上面程序，看到在文件所在路径生成了两个 class 文件，一个是 Cow.class，另一个是 Cow$CowLeg.class，前者是外部类 Cow 的 class 文件，后者是内部类 CowLeg 的 class 文件，即成员内部类（包括静态内部类、非静态内部类）的 class 文件总是这种形式：OuterClass$InnerClass.class。

前面提到过，在非静态内部类里可以直接访问外部类的 private 成员，上面程序中①号粗体代码行，就是在 CowLeg 类的方法内直接访问其外部类的 private 实例变量。这是因为在非静态内部类对象里，保存了一个它所寄生的外部类对象的引用（当调用非静态内部类的实例方法时，必须有一个非静态内部类实例，非静态内部类实例必须寄生在外部类实例里）。图 6.6 显示了上面程序运行时的内存示意图。

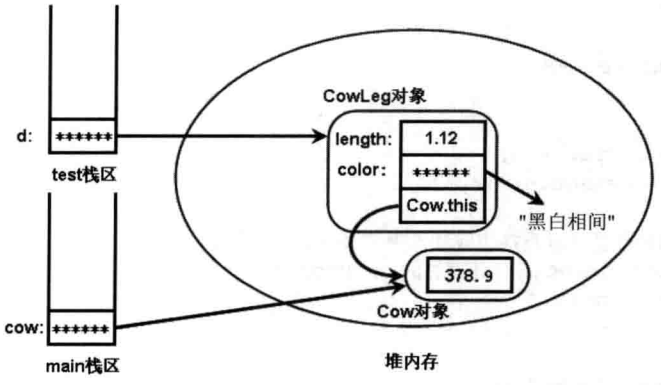


图 6.6 非静态内部类对象中保留外部类对象的引用内存示意图

当在非静态内部类的方法内访问某个变量时，系统优先在该方法内查找是否存在该名字的局部变量，如果存在就使用该变量；如果不存在，则到该方法所在的内部类中查找是否存在该名字的成员变量，如果存在则使用该成员变量；如果不存在，则到该内部类所在的外部类中查找是否存在该名字的成员变量，如果存在则使用该成员变量；如果依然不存在，系统将出现编译错误：提示找不到该变量。

因此，如果外部类成员变量、内部类成员变量与内部类里方法的局部变量同名，则可通过使用 this、外部类类名.this 作为限定来区分。如下程序所示。

程序清单：codes\06\6.7\DiscernVariable.java

```
public class DiscernVariable
{
    private String prop = "外部类的实例变量";
    private class InClass
    {
        private String prop = "内部类的实例变量";
        public void info()
        {
            String prop = "局部变量";
            // 通过外部类类名.this.varName 访问外部类实例变量
```

```
            System.out.println("外部类的实例变量值："
                + DiscernVariable.this.prop);
            // 通过 this.varName 访问内部类实例的变量
            System.out.println("内部类的实例变量值：" + this.prop);
            // 直接访问局部变量
            System.out.println("局部变量的值：" + prop);
        }
    }
    public void test()
    {
        InClass in = new InClass();
        in.info();
    }
    public static void main(String[] args)
    {
        new DiscernVariable().test();
    }
}
```

上面程序中粗体字代码行分别访问外部类的实例变量、非静态内部类的实例变量。通过OutterClass.this.propName的形式访问外部类的实例变量，通过this.propName的形式访问非静态内部类的实例变量。

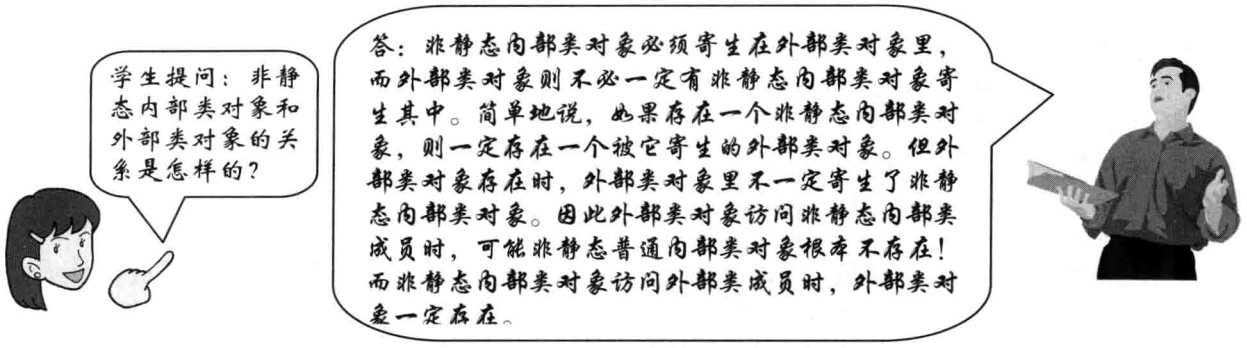
非静态内部类的成员可以访问外部类的private成员，但反过来就不成立了。非静态内部类的成员只在非静态内部类范围内是可知的，并不能被外部类直接使用。如果外部类需要访问非静态内部类的成员，则必须显式创建非静态内部类对象来调用访问其实例成员。下面程序示范了这个规则。

程序清单：codes\06\6.7\Outer.java

```
public class Outer
{
    private int outProp = 9;
    class Inner
    {
        private int inProp = 5;
        public void acessOuterProp()
        {
            // 非静态内部类可以直接访问外部类的 private 成员变量
            System.out.println("外部类的 outProp 值:"
                + outProp);
        }
    }
    public void accessInnerProp()
    {
        // 外部类不能直接访问非静态内部类的实例变量
        // 下面代码出现编译错误
        // System.out.println("内部类的 inProp 值:" + inProp);
        // 如需访问内部类的实例变量，必须显式创建内部类对象
        System.out.println("内部类的 inProp 值:"
            + new Inner().inProp);
    }
    public static void main(String[] args)
    {
        // 执行下面代码，只创建了外部类对象，还未创建内部类对象
        Outer out = new Outer();      //①
        out.accessInnerProp();
    }
}
```

程序中粗体字行试图在外部类方法里访问非静态内部类的实例变量，这将引起编译错误。

外部类不允许访问非静态内部类的实例成员还有一个原因，上面程序中main()方法的①号粗体字代码创建了一个外部类对象，并调用外部类对象的accessInnerProp()方法。此时非静态内部类对象根本不存在，如果允许accessInnerProp()方法访问非静态内部类对象，将肯定引起错误。



根据静态成员不能访问非静态成员的规则，外部类的静态方法、静态代码块不能访问非静态内部类，包括不能使用非静态内部类定义变量、创建实例等。总之，不允许在外部类的静态成员中直接使用非静态内部类。如下程序所示。

程序清单：codes\06\6.7\StaticTest.java

```
public class StaticTest
{
    // 定义一个非静态的内部类，是一个空类
    private class In{}
    // 外部类的静态方法
    public static void main(String[] args)
    {
        // 下面代码引发编译异常，因为静态成员（main()方法）
        // 无法访问非静态成员（In类）
        new In();
    }
}
```

Java 不允许在非静态内部类里定义静态成员。下面程序示范了非静态内部类里包含静态成员将引发编译错误。

程序清单：codes\06\6.7\InnerNoStatic.java

```
public class InnerNoStatic
{
    private class InnerClass
    {
        /*
        下面三个静态声明都将引发如下编译错误：
        非静态内部类不能有静态声明
        */
        static
        {
            System.out.println("==========");
        }
        private static int inProp;
        private static void test(){}
    }
}
```

非静态内部类里不能有静态方法、静态成员变量、静态初始化块，所以上面三个静态声明都会引发错误。

注意：

非静态内部类里不可以有静态初始化块，但可以包含普通初始化块。非静态内部类普通初始化块的作用与外部类初始化块的作用完全相同。



6.7.2 静态内部类

如果使用 static 来修饰一个内部类，则这个内部类就属于外部类本身，而不属于外部类的某个对象。因此使用 static 修饰的内部类被称为类内部类，有的地方也称为静态内部类。

注意：

static 关键字的作用是把类的成员变成类相关，而不是实例相关，即 static 修饰的成员属于整个类，而不属于单个对象。外部类的上一级程序单元是包，所以不可使用 static 修饰；而内部类的上一级程序单元是外部类，使用 static 修饰可以将内部类变成外部类相关，而不是外部类实例相关。因此 static 关键字不可修饰外部类，但可修饰内部类。

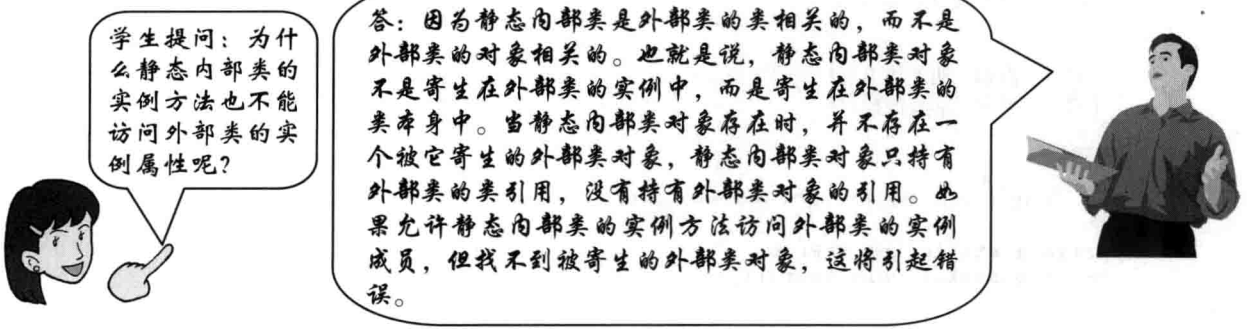


静态内部类可以包含静态成员，也可以包含非静态成员。根据静态成员不能访问非静态成员的规则，静态内部类不能访问外部类的实例成员，只能访问外部类的类成员。即使是静态内部类的实例方法也不能访问外部类的实例成员，只能访问外部类的静态成员。下面程序就演示了这条规则。

程序清单：codes\06\6.7\StaticInnerClassTest.java

```
public class StaticInnerClassTest
{
    private int prop1 = 5;
    private static int prop2 = 9;
    static class StaticInnerClass
    {
        // 静态内部类里可以包含静态成员
        private static int age;
        public void accessOuterProp()
        {
            // 下面代码出现错误
            // 静态内部类无法访问外部类的实例变量
            System.out.println(prop1);
            // 下面代码正常
            System.out.println(prop2);
        }
    }
}
```

上面程序中粗体字代码行定义了一个静态成员变量，因为这个静态成员变量处于静态内部类中，所以完全没有问题。StaticInnerClass 类里定义了一个 accessOuterProp()方法，这是一个实例方法，但依然不能访问外部类的 prop1 成员变量，因为这是实例变量；但可以访问 prop2，因为它是静态成员变量。



静态内部类是外部类的一个静态成员，因此外部类的所有方法、所有初始化块中可以使用静态内部类来定义变量、创建对象等。

外部类依然不能直接访问静态内部类的成员，但可以使用静态内部类的类名作为调用者来访问静态内部类的类成员，也可以使用静态内部类对象作为调用者来访问静态内部类的实例成员。下面程序示范了这条规则。

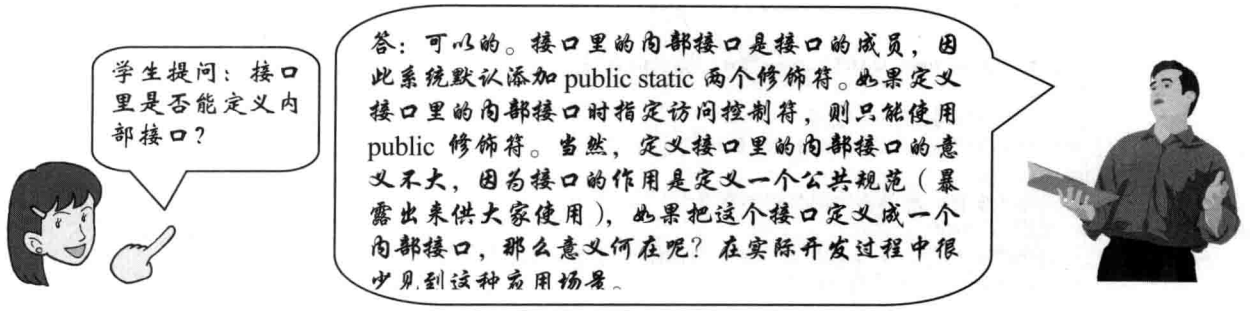
程序清单：codes\06\6.7\AccessStaticInnerClass.java

```
public class AccessStaticInnerClass
{
    static class StaticInnerClass
    {
        private static int prop1 = 5;
```

```
        private int prop2 = 9;
    }
    public void accessInnerProp()
    {
        // System.out.println(prop1);
        // 上面代码出现错误，应改为如下形式
        // 通过类名访问静态内部类的类成员
        System.out.println(StaticInnerClass.prop1);
        // System.out.println(prop2);
        // 上面代码出现错误，应改为如下形式
        // 通过实例访问静态内部类的实例成员
        System.out.println(new StaticInnerClass().prop2);
    }
}
```

除此之外，Java 还允许在接口里定义内部类，接口里定义的内部类默认使用 public static 修饰，也就是说，接口内部类只能是静态内部类。

如果为接口内部类指定访问控制符，则只能指定 public 访问控制符；如果定义接口内部类时省略访问控制符，则该内部类默认是 public 访问控制权限。



6.7.3 使用内部类

定义类的主要作用就是定义变量、创建实例和作为父类被继承。定义内部类的主要作用也如此，但使用内部类定义变量和创建实例则与外部类存在一些小小的差异。下面分三种情况讨论内部类的用法。

1. 在外部类内部使用内部类

从前面程序中可以看出，在外部类内部使用内部类时，与平常使用普通类没有太大的区别。一样可以直接通过内部类类名来定义变量，通过 new 调用内部类构造器来创建实例。

唯一存在的一个区别是：不要在外部类的静态成员（包括静态方法和静态初始化块）中使用非静态内部类，因为静态成员不能访问非静态成员。

在外部类内部定义内部类的子类与平常定义子类也没有太大的区别。

2. 在外部类以外使用非静态内部类

如果希望在外部类以外的地方访问内部类（包括静态和非静态两种），则内部类不能使用 private 访问控制权限，private 修饰的内部类只能在外部类内部使用。对于使用其他访问控制符修饰的内部类，则能在访问控制符对应的访问权限内使用。

- 省略访问控制符的内部类，只能被与外部类处于同一个包中的其他类所访问。
- 使用 protected 修饰的内部类，可被与外部类处于同一个包中的其他类和外部类的子类所访问。
- 使用 public 修饰的内部类，可以在任何地方被访问。

在外部类以外的地方定义内部类（包括静态和非静态两种）变量的语法格式如下：

```
OuterClass.InnerClass varName
```

从上面语法格式可以看出，在外部类以外的地方使用内部类时，内部类完整的类名应该是 OuterClass.InnerClass。如果外部类有包名，则还应该增加包名前缀。

由于非静态内部类的对象必须寄生在外部类的对象里，因此创建非静态内部类对象之前，必须先创建其外部类对象。在外部类以外的地方创建非静态内部类实例的语法如下：

```
OuterInstance.new InnerConstructor()
```

从上面语法格式可以看出，在外部类以外的地方创建非静态内部类实例必须使用外部类实例和 new 来调用非静态内部类的构造器。下面程序示范了如何在外部类以外的地方创建非静态内部类的对象，并把它赋给非静态内部类类型的变量。

程序清单：codes\06\6.7\CreateInnerInstance.java

```
class Out
{
    // 定义一个内部类，不使用访问控制符
    // 即只有同一个包中的其他类可访问该内部类
    class In
    {
        public In(String msg)
        {
            System.out.println(msg);
        }
    }
}
public class CreateInnerInstance
{
    public static void main(String[] args)
    {
        Out.In in = new Out().new In("测试信息");
        /*
        上面代码可改为如下三行代码
        使用 OutterClass.InnerClass 的形式定义内部类变量
        Out.In in;
        创建外部类实例，非静态内部类实例将寄生在该实例中
        Out out = new Out();
        通过外部类实例和 new 来调用内部类构造器创建非静态内部类实例
        in = out.new In("测试信息");
        */
    }
}
```

上面程序中粗体代码行创建了一个非静态内部类的对象。从上面代码可以看出，非静态内部类的构造器必须使用外部类对象来调用。

如果需要在外部类以外的地方创建非静态内部类的子类，则尤其要注意上面的规则：非静态内部类的构造器必须通过其外部类对象来调用。

当创建一个子类时，子类构造器总会调用父类的构造器，因此在创建非静态内部类的子类时，必须保证让子类构造器可以调用非静态内部类的构造器，调用非静态内部类的构造器时，必须存在一个外部类对象。下面程序定义了一个子类继承了 Out 类的非静态内部类 In 类。

程序清单：codes\06\6.7\SubClass.java

```
public class SubClass extends Out.In
{
    // 显示定义 SubClass 的构造器
    public SubClass(Out out)
    {
        // 通过传入的 Out 对象显式调用 In 的构造器
        out.super("hello");
    }
}
```

上面代码中粗体代码行看起来有点奇怪，其实很正常：非静态内部类 In 类的构造器必须使用外部类对象来调用，代码中 super 代表调用 In 类的构造器，而 out 则代表外部类对象（上面的 Out、In 两个类直接来自于前一个 CreateInnerInstance.java）。

从上面代码中可以看出，如果需要创建 SubClass 对象时，必须先创建一个 Out 对象。这是合理的，因为 SubClass 是非静态内部类 In 类的子类，非静态内部类 In 对象里必须有一个对 Out 对象的引用，其子类 SubClass 对象里也应该持有对 Out 对象的引用。当创建 SubClass 对象时传给该构造器的 Out 对象，就是 SubClass 对象里 Out 对象引用所指向的对象。

非静态内部类 In 对象和 SubClass 对象都必须持有指向 Outer 对象的引用，区别是创建两种对象时

传入 Out 对象的方式不同：当创建非静态内部类 In 类的对象时，必须通过 Outer 对象来调用 new 关键字；当创建 SubClass 类的对象时，必须使用 Outer 对象作为调用者来调用 In 类的构造器。

注意：

非静态内部类的子类不一定是内部类，它可以是一个外部类。但非静态内部类的子类实例一样需要保留一个引用，该引用指向其父类所在外部类的对象。也就是说，如果有一个内部类子类的对象存在，则一定存在与之对应的外部类对象。



3. 在外部类以外使用静态内部类

因为静态内部类是外部类类相关的，因此创建静态内部类对象时无须创建外部类对象。在外部类以外的地方创建静态内部类实例的语法如下：

```
new OuterClass.InnerConstructor()
```

下面程序示范了如何在外部类以外的地方创建静态内部类的实例。

程序清单：codes\06\6.7\CreateStaticInnerInstance.java

```
class StaticOut
{
    // 定义一个静态内部类，不使用访问控制符
    // 即同一个包中的其他类可访问该内部类
    static class StaticIn
    {
        public StaticIn()
        {
            System.out.println("静态内部类的构造器");
        }
    }
}
public class CreateStaticInnerInstance
{
    public static void main(String[] args)
    {
        StaticOut.StaticIn in = new StaticOut.StaticIn();
        /*
        上面代码可改为如下两行代码
        使用 OuterClass.InnerClass 的形式定义内部类变量
        StaticOut.StaticIn in;
        通过 new 来调用内部类构造器创建静态内部类实例
        in = new StaticOut.StaticIn();
        */
    }
}
```

从上面代码中可以看出，不管是静态内部类还是非静态内部类，它们声明变量的语法完全一样。区别只是在创建内部类对象时，静态内部类只需使用外部类即可调用构造器，而非静态内部类必须使用外部类对象来调用构造器。

因为调用静态内部类的构造器时无须使用外部类对象，所以创建静态内部类的子类也比较简单，下面代码就为静态内部类 StaticIn 类定义了一个空的子类。

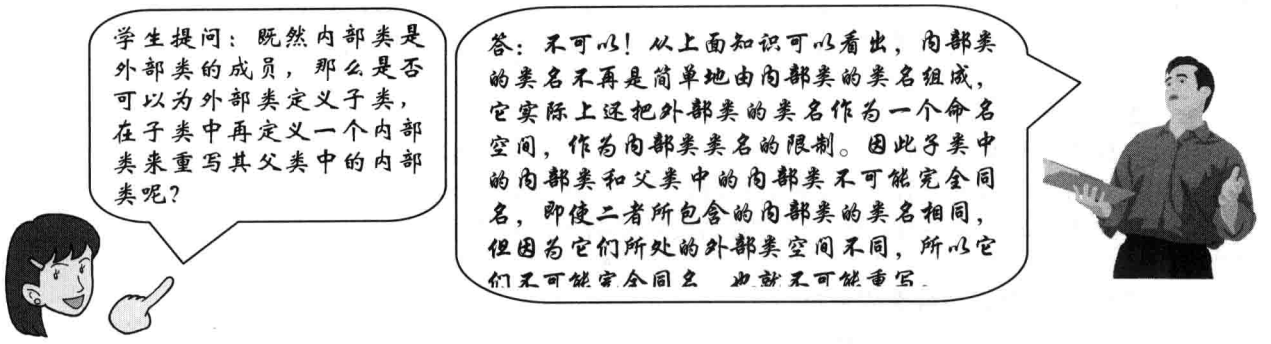
```
public class StaticSubClass extends StaticOut.StaticIn {}
```

从上面代码中可以看出，当定义一个静态内部类时，其外部类非常像一个包空间。

注意：

相比之下，使用静态内部类比使用非静态内部类要简单很多，只要把外部类当成静态内部类的包空间即可。因此当程序需要使用内部类时，应该优先考虑使用静态内部类。





6.7.4　局部内部类

如果把一个内部类放在方法里定义，则这个内部类就是一个局部内部类，局部内部类仅在该方法里有效。由于局部内部类不能在外部类的方法以外的地方使用，因此局部内部类也不能使用访问控制符和 static 修饰符修饰。

注意：

对于局部成员而言，不管是局部变量还是局部内部类，它们的上一级程序单元都是方法，而不是类，使用 static 修饰它们没有任何意义。因此，所有的局部成员都不能使用 static 修饰。不仅如此，因为局部成员的作用域是所在方法，其他程序单元永远也不可能访问另一个方法中的局部成员，所以所有的局部成员都不能使用访问控制符修饰。



如果需要用局部内部类定义变量、创建实例或派生子类，那么都只能在局部内部类所在的方法内进行。

程序清单：codes\06\6.7\LocalInnerClass.java

```
public class LocalInnerClass
{
    public static void main(String[] args)
    {
        // 定义局部内部类
        class InnerBase
        {
            int a;
        }
        // 定义局部内部类的子类
        class InnerSub extends InnerBase
        {
            int b;
        }
        // 创建局部内部类的对象
        InnerSub is = new InnerSub();
        is.a = 5;
        is.b = 8;
        System.out.println("InnerSub 对象的 a 和 b 实例变量是："
            + is.a + "," + is.b);
    }
}
```

编译上面程序，看到生成了三个 class 文件：LocalInnerClass.class、LocalInnerClass$1InnerBase.class 和 LocalInnerClass$1InnerSub.class，这表明局部内部类的 class 文件总是遵循如下命名格式：OuterClass$NInnerClass.class。注意到局部内部类的 class 文件的文件名比成员内部类的 class 文件的文件名多了一个数字，这是因为同一个类里不可能有两个同名的成员内部类，而同一个类里则可能有两个以上同名的局部内部类（处于不同方法中），所以 Java 为局部内部类的 class 文件名中增加了一个数字，用于区分。



局部内部类是一个非常“鸡肋”的语法，在实际开发中很少定义局部内部类，这是因为局部内部类的作用域太小了：只能在当前方法中使用。大部分时候，定义一个类之后，当然希望多次复用这个类，但局部内部类无法离开它所在的方法，因此在实际开发中很少使用局部内部类。



6.7.5 Java 8 改进的匿名内部类

匿名内部类适合创建那种只需要一次使用的类，例如前面介绍命令模式时所需要的Command对象。匿名内部类的语法有点奇怪，创建匿名内部类时会立即创建一个该类的实例，这个类定义立即消失，匿名内部类不能重复使用。

定义匿名内部类的格式如下：

```
new 实现接口() | 父类构造器(实参列表)
{
    //匿名内部类的类体部分
}
```

从上面定义可以看出，匿名内部类必须继承一个父类，或实现一个接口，但最多只能继承一个父类，或实现一个接口。

关于匿名内部类还有如下两条规则。

- 匿名内部类不能是抽象类，因为系统在创建匿名内部类时，会立即创建匿名内部类的对象。因此不允许将匿名内部类定义成抽象类。
- 匿名内部类不能定义构造器。由于匿名内部类没有类名，所以无法定义构造器，但匿名内部类可以定义初始化块，可以通过实例初始化块来完成构造器需要完成的事情。

最常用的创建匿名内部类的方式是需要创建某个接口类型的对象，如下程序所示。

程序清单：codes\06\6.7\AnonymousTest.java

```
interface Product
{
    public double getPrice();
    public String getName();
}
public class AnonymousTest
{
    public void test(Product p)
    {
        System.out.println("购买了一个" + p.getName()
            + ", 花掉了" + p.getPrice());
    }
    public static void main(String[] args)
    {
        AnonymousTest ta = new AnonymousTest();
        // 调用 test()方法时，需要传入一个 Product 参数
        // 此处传入其匿名实现类的实例
        ta.test(new Product()
        {
            public double getPrice()
            {
                return 567.8;
            }
            public String getName()
            {
                return "AGP 显卡";
            }
        });
    }
}
```

上面程序中的AnonymousTest类定义了一个test()方法，该方法需要一个Product对象作为参数，但

Product只是一个接口，无法直接创建对象，因此此处考虑创建一个Product接口实现类的对象传入该方法——如果这个Product接口实现类需要重复使用，则应该将该实现类定义成一个独立类；如果这个Product接口实现类只需一次使用，则可采用上面程序中的方式，定义一个匿名内部类。

正如上面程序中看到的，定义匿名内部类无须class关键字，而是在定义匿名内部类时直接生成该匿名内部类的对象。上面粗体字代码部分就是匿名内部类的类体部分。

由于匿名内部类不能是抽象类，所以匿名内部类必须实现它的抽象父类或者接口里包含的所有抽象方法。

对于上面创建Product实现类对象的代码，可以拆分成如下代码。

```
class AnonymousProduct implements Product
{
    public double getPrice()
    {
        return 567.8;
    }
    public String getName()
    {
        return "AGP显卡";
    }
}
ta.test(new AnonymousProduct());
```

对比两段代码的粗体字代码部分，它们完全一样，但显然采用匿名内部类的写法更加简洁。

当通过实现接口来创建匿名内部类时，匿名内部类也不能显式创建构造器，因此匿名内部类只有一个隐式的无参数构造器，故new接口名后的括号里不能传入参数值。

但如果通过继承父类来创建匿名内部类时，匿名内部类将拥有和父类相似的构造器，此处的相似指的是拥有相同的形参列表。

程序清单：codes\06\6.7\AnonymousInner.java

```
abstract class Device
{
    private String name;
    public abstract double getPrice();
    public Device(){}
    public Device(String name)
    {
        this.name = name;
    }
    // 此处省略了name的setter和getter方法
    ...
}
public class AnonymousInner
{
    public void test(Device d)
    {
        System.out.println("购买了一个" + d.getName()
            + ", 花掉了" + d.getPrice());
    }
    public static void main(String[] args)
    {
        AnonymousInner ai = new AnonymousInner();
        // 调用有参数的构造器创建Device匿名实现类的对象
        ai.test(new Device("电子示波器")
        {
            public double getPrice()
            {
                return 67.8;
            }
        });
        // 调用无参数的构造器创建Device匿名实现类的对象
        Device d = new Device()
        {
            // 初始化块
            {
```

```
            System.out.println("匿名内部类的初始化块...");
        }
        // 实现抽象方法
        public double getPrice()
        {
            return 56.2;
        }
        // 重写父类的实例方法
        public String getName()
        {
            return "键盘";
        }
    };
    ai.test(d);
  }
}
```

上面程序创建了一个抽象父类 Device 类，这个抽象父类里包含两个构造器：一个无参数的，一个有参数的。当创建以 Device 为父类的匿名内部类时，既可以传入参数（如上面程序中第一段粗体字部分），代表调用父类带参数的构造器；也可以不传入参数（如上面程序中第二段粗体字部分），代表调用父类无参数的构造器。

当创建匿名内部类时，必须实现接口或抽象父类里的所有抽象方法。如果有需要，也可以重写父类中的普通方法，如上面程序的第二段粗体字代码部分，匿名内部类重写了抽象父类 Device 类的 getName() 方法，其中 getName()方法并不是抽象方法。

在 Java 8 之前，Java 要求被局部内部类、匿名内部类访问的局部变量必须使用 final 修饰，从 Java 8 开始这个限制被取消了，Java 8 更加智能：如果局部变量被匿名内部类访问，那么该局部变量相当于自动使用了 final 修饰。例如如下程序。

程序清单：codes\06\6.7\ATest.java

```
interface A
{
    void test();
}
public class ATest
{
    public static void main(String[] args)
    {
        int age = 8;     // ①
        A a = new A()
        {
            public void test()
            {
                // 在 Java 8 以前下面语句将提示错误：age 必须使用 final 修饰
                // 从 Java 8 开始，匿名内部类、局部内部类允许访问非 final 的局部变量
                System.out.println(age);
            }
        };
        a.test();
    }
}
```

如果使用 Java 8 的 JDK 来编译、运行上面程序，程序完全正常。但如果使用 Java 8 以前版本的 JDK 编译上面程序，粗体字代码将会引起编译错误，编译器提示用户必须用 final 修饰 age 局部变量。

如果在①号代码后增加如下代码：

```
// 下面代码将会导致编译错误
// 由于 age 局部变量被匿名内部类访问了，因此 age 相当于被 final 修饰了
age = 2;
```

由于程序中①号代码定义 age 局部变量时指定了初始值，而上面代码再次对 age 变量赋值，这会导致 Java 8 无法自动使用 final 修饰 age 局部变量，因此编译器将会报错：被匿名内部类访问的局部变量必须使用 final 修饰。



提示：

Java 8 将这个功能称为“effectively final”，它的意思是对于被匿名内部类访问的局部变量，可以用 final 修饰，也可以不用 final 修饰，但必须按照有 final 修饰的方式来用——也就是一次赋值后，以后不能重新赋值。

6.8 Java 8 新增的 Lambda 表达式

Lambda 表达式是 Java 8 的重要更新，也是一个被广大开发者期待已久的新特性。Lambda 表达式支持将代码块作为方法参数，Lambda 表达式允许使用更简洁的代码来创建只有一个抽象方法的接口（这种接口被称为函数式接口）的实例。

6.8.1 Lambda 表达式入门

下面先使用匿名内部类来改写前面介绍的 command 表达式的例子，改写后的程序如下。

程序清单：codes\06\6.8\CommandTest.java

```
public class CommandTest
{
    public static void main(String[] args)
    {
        ProcessArray pa = new ProcessArray();
        int[] target = {3, -4, 6, 4};
        // 处理数组，具体处理行为取决于匿名内部类
        pa.process(target , new Command()
            {
                public void process(int[] target)
                {
                    int sum = 0;
                    for (int tmp : target )
                    {
                        sum += tmp;
                    }
                    System.out.println("数组元素的总和是:" + sum);
                }
            });
    }
}
```

前面已经提到，ProcessArray 类的 process()方法处理数组时，希望可以动态传入一段代码作为具体的处理行为，因此程序创建了一个匿名内部类实例来封装处理行为。从上面代码可以看出，用于封装处理行为的关键就是实现程序中的粗体字方法。但为了向 process()方法传入这段粗体字代码，程序不得不使用匿名内部类的语法来创建对象。

Lambda 表达式完全可用于简化创建匿名内部类对象，因此可将上面代码改为如下形式。

程序清单：codes\06\6.8\CommandTest2.java

```
public class CommandTest2
{
    public static void main(String[] args)
    {
        ProcessArray pa = new ProcessArray();
        int[] array = {3, -4, 6, 4};
        // 处理数组，具体处理行为取决于匿名内部类
        pa.process(array , (int[] target)->{
                int sum = 0;
                for (int tmp : target )
                {
                    sum += tmp;
                }
                System.out.println("数组元素的总和是:" + sum);
            });
    }
}
```

从上面程序中的粗体字代码可以看出，这段粗体字代码与创建匿名内部类时需要实现的 process(int[] target)方法完全相同，只是不需要 new Xxx(){}这种烦琐的代码，不需要指出重写的方法名字，也不需要给出重写的方法的返回值类型——只要给出重写的方法括号以及括号里的形参列表即可。

从上面介绍可以看出，当使用 Lambda 表达式代替匿名内部类创建对象时，Lambda 表达式的代码块将会代替实现抽象方法的方法体，Lambda 表达式就相当一个匿名方法。

从上面语法格式可以看出，Lambda 表达式的主要作用就是代替匿名内部类的烦琐语法。它由三部分组成。

- 形参列表。形参列表允许省略形参类型。如果形参列表中只有一个参数，甚至连形参列表的圆括号也可以省略。
- 箭头（->）。必须通过英文中画线号和大于符号组成。
- 代码块。如果代码块只包含一条语句，Lambda 表达式允许省略代码块的花括号，那么这条语句就不要用花括号表示语句结束。Lambda 代码块只有一条 return 语句，甚至可以省略 return 关键字。Lambda 表达式需要返回值，而它的代码块中仅有一条省略了 return 的语句，Lambda 表达式会自动返回这条语句的值。

下面程序示范了 Lambda 表达式的几种简化写法。

程序清单：codes\06\6.8\LambdaQs.java

```
interface Eatable
{
    void taste();
}
interface Flyable
{
    void fly(String weather);
}
interface Addable
{
    int add(int a , int b);
}
public class LambdaQs
{
    // 调用该方法需要 Eatable 对象
    public void eat(Eatable e)
    {
        System.out.println(e);
        e.taste();
    }
    // 调用该方法需要 Flyable 对象
    public void drive(Flyable f)
    {
        System.out.println("我正在驾驶：" + f);
        f.fly("【碧空如洗的晴日】");
    }
    // 调用该方法需要 Addable 对象
    public void test(Addable add)
    {
        System.out.println("5 与 3 的和为：" + add.add(5, 3));
    }
    public static void main(String[] args)
    {
        LambdaQs lq = new LambdaQs();
        // Lambda 表达式的代码块只有一条语句，可以省略花括号
        lq.eat(()-> System.out.println("苹果的味道不错！"));
        // Lambda 表达式的形参列表只有一个形参，可以省略圆括号
        lq.drive(weather ->
        {
            System.out.println("今天天气是：" + weather);
            System.out.println("直升机飞行平稳");
        });
        // Lambda 表达式的代码块只有一条语句，可以省略花括号
        // 代码块中只有一条语句，即使该表达式需要返回值，也可以省略 return 关键字
```

```
        lq.test((a , b)->a + b);
    }
}
```

上面程序中的第一段粗体字代码使用 Lambda 表达式相当于不带形参的匿名方法，由于该 Lambda 表达式的代码块只有一行代码，因此可以省略代码块的花括号；第二段粗体字代码使用 Lambda 表达式相当于只带一个形参的匿名方法，由于该 Lambda 表达式的形参列表只有一个形参，因此省略了形参列表的圆括号；第三段粗体字代码的 Lambda 表达式的代码块中只有一行语句，这行语句的返回值将作为该代码块的返回值。

上面程序中的第一处粗体字代码调用 eat()方法，调用该方法需要一个 Eatable 类型的参数，但实际传入的是 Lambda 表达式；第二处粗体字代码调用 drive()方法，调用该方法需要一个 Flyable 类型的参数，但实际传入的是 Lambda 表达式；第三处粗体字代码调用 test()方法，调用该方法需要一个 Addable 类型的参数，但实际传入的是 Lambda 表达式。但上面程序可以正常编译、运行，这说明 Lambda 表达式实际上将会被当成一个“任意类型”的对象，到底需要当成何种类型的对象，这取决于运行环境的需要。下面将详细介绍 Lambda 表达式被当成何种对象。

6.8.2 Lambda 表达式与函数式接口

Lambda 表达式的类型，也被称为“目标类型（target type）”，Lambda 表达式的目标类型必须是“函数式接口（functional interface）”。函数式接口代表只包含一个抽象方法的接口。函数式接口可以包含多个默认方法、类方法，但只能声明一个抽象方法。

如果采用匿名内部类语法来创建函数式接口的实例，则只需要实现一个抽象方法，在这种情况下即可采用 Lambda 表达式来创建对象，该表达式创建出来的对象的目标类型就是这个函数式接口。查询 Java 8 的 API 文档，可以发现大量的函数式接口，例如：Runnable、ActionListener 等接口都是函数式接口。



提示：

Java 8 专门为函数式接口提供了@FunctionalInterface 注解，该注解通常放在接口定义前面，该注解对程序功能没有任何作用，它用于告诉编译器执行更严格检查——检查该接口必须是函数式接口，否则编译器就会报错。

由于 Lambda 表达式的结果就是被当成对象，因此程序中完全可以使用 Lambda 表达式进行赋值，例如如下代码。

程序清单：\codes\06\6.8\LambdaTest.java

```
// Runnable 接口中只包含一个无参数的方法
// Lambda 表达式代表的匿名方法实现了 Runnable 接口中唯一的、无参数的方法
// 因此下面的 Lambda 表达式创建了一个 Runnable 对象
Runnable r = () -> {
    for(int i = 0 ; i < 100 ; i ++)
    {
        System.out.println();
    }
};
```



提示：

Runnable 是 Java 本身提供的一个函数式接口。

从上面粗体字代码可以看出，Lambda 表达式实现的是匿名方法——因此它只能实现特定函数式接口中的唯一方法。这意味着 Lambda 表达式有如下两个限制。

- Lambda 表达式的目标类型必须是明确的函数式接口。
- Lambda 表达式只能为函数式接口创建对象。Lambda 表达式只能实现一个方法，因此它只能为只有一个抽象方法的接口（函数式接口）创建对象。

关于上面第一点限制，看下面代码是否正确（程序清单同上）。

```
Object obj = () -> {
    for(int i = 0 ; i < 100 ; i ++)
```

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

上面代码与前一段代码几乎完全相同，只是此时程序将 Lambda 表达式不再赋值给 Runnable 变量，而是直接赋值给 Object 变量。编译上面代码，会报如下错误：

```
不兼容的类型：Object 不是函数接口
```

从该错误信息可以看出，Lambda 表达式的目标类型必须是明确的函数式接口。上面代码将 Lambda 表达式赋值给 Object 变量，编译器只能确定该 Lambda 表达式的类型为 Object，而 Object 并不是函数式接口，因此上面代码报错。

为了保证 Lambda 表达式的目标类型是一个明确的函数式接口，可以有如下三种常见方式。

- 将 Lambda 表达式赋值给函数式接口类型的变量。
- 将 Lambda 表达式作为函数式接口类型的参数传给某个方法。
- 使用函数式接口对 Lambda 表达式进行强制类型转换。

因此，只要将上面代码改为如下形式即可（程序清单同上）。

```
Object obj1 = (Runnable)() -> {
    for(int i = 0 ; i < 100 ; i ++)
    {
        System.out.println();
    }
};
```

上面代码中的粗体字代码对 Lambda 表达式执行了强制类型转换，这样就可以确定该表达式的目标类型为 Runnable 函数式接口。

需要说明的是，同样的 Lambda 表达式的目标类型完全可能是变化的——唯一的要求是，Lambda 表达式实现的匿名方法与目标类型（函数式接口）中唯一的抽象方法有相同的形参列表。

例如定义了如下接口（程序清单同上）：

```
@FunctionalInterface
interface FkTest
{
    void run();
}
```

上面的函数式接口中仅定义了一个不带参数的方法，因此前面强制转型为 Runnable 的 Lambda 表达式也可强转为 FkTest 类型——因为 FkTest 接口中的唯一的抽象方法是不带参数的，而该 Lambda 表达式也是不带参数的。因此，下面代码是正确的（程序清单同上）。

```
// 同样的 Lambda 表达式可以被当成不同的目标类型，唯一的要求是
// Lambda 表达式的形参列表与函数式接口中唯一的抽象方法的形参列表相同
Object obj2 = (FkTest)() -> {
    for(int i = 0 ; i < 100 ; i ++)
    {
        System.out.println();
    }
};
```

Java 8 在 java.util.function 包下预定义了大量函数式接口，典型地包含如下 4 类接口。

- XxxFunction：这类接口中通常包含一个 apply()抽象方法，该方法对参数进行处理、转换（apply()方法的处理逻辑由 Lambda 表达式来实现），然后返回一个新的值。该函数式接口通常用于对指定数据进行转换处理。
- XxxConsumer：这类接口中通常包含一个 accept()抽象方法，该方法与 XxxFunction 接口中的 apply()方法基本相似，也负责对参数进行处理，只是该方法不会返回处理结果。
- XxxxPredicate：这类接口中通常包含一个 test()抽象方法，该方法通常用来对参数进行某种判断（test()方法的判断逻辑由 Lambda 表达式来实现），然后返回一个 boolean 值。该接口通常用于判断参数是否满足特定条件，经常用于进行筛滤数据。
- XxxSupplier：这类接口中通常包含一个 getAsXxx()抽象方法，该方法不需要输入参数，该方法

会按某种逻辑算法（getAsXxx ()方法的逻辑算法由 Lambda 表达式来实现）返回一个数据。

综上所述，不难发现 Lambda 表达式的本质很简单，就是使用简洁的语法来创建函数式接口的实例——这种语法避免了匿名内部类的烦琐。

6.8.3 方法引用与构造器引用

前面已经介绍过，如果 Lambda 表达式的代码块只有一条代码，程序就可以省略 Lambda 表达式中代码块的花括号。不仅如此，如果 Lambda 表达式的代码块只有一条代码，还可以在代码块中使用方法引用和构造器引用。

方法引用和构造器引用可以让 Lambda 表达式的代码块更加简洁。方法引用和构造器引用都需要使用两个英文冒号。Lambda 表达式支持如表 6.2 所示的几种引用方式。

表 6.2 Lambda 表达式支持的方法引用和构造器引用

种 类	示 例	说 明	对应的 Lambda 表达式
引用类方法	类名::类方法	函数式接口中被实现方法的全部参数传给该类方法作为参数	(a,b,...) -> 类名.类方法(a,b, ...)
引用特定对象的实例方法	特定对象::实例方法	函数式接口中被实现方法的全部参数传给该方法作为参数	(a,b, ...) -> 特定对象.实例方法(a,b, ...)
引用某类对象的实例方法	类名::实例方法	函数式接口中被实现方法的第一个参数作为调用者，后面的参数全部传给该方法作为参数	(a,b, ...) ->a.实例方法(b, ...)
引用构造器	类名::new	函数式接口中被实现方法的全部参数传给该构造器作为参数	(a,b, ...) ->new 类名(a,b, ...)

1. 引用类方法

先看第一种方法引用：引用类方法。例如，定义了如下函数式接口。

程序清单：codes\06\6.8\MethodRefer.java

```
@FunctionalInterface
interface Converter{
    Integer convert(String from);
}
```

该函数式接口中包含一个 convert()抽象方法，该方法负责将 String 参数转换为 Integer。下面代码使用 Lambda 表达式来创建一个 Converter 对象（程序清单同上）。

```
// 下面代码使用 Lambda 表达式创建 Converter 对象
Converter converter1 = from -> Integer.valueOf(from);
```

上面 Lambda 表达式的代码块只有一条语句，因此程序省略了该代码块的花括号；而且由于表达式所实现的 convert()方法需要返回值，因此 Lambda 表达式将会把这条代码的值作为返回值。

接下来程序就可以调用 converter1 对象的 convert()方法将字符串转换为整数了，例如如下代码（程序清单同上）：

```
Integer val = converter1.convert("99");
System.out.println(val); // 输出整数 99
```

上面代码调用 converter1 对象的 conver()方法时——由于 converter1 对象是 Lambda 表达式创建的，convert()方法执行体就是 Lambda 表达式的代码块部分，因此上面程序输出 99。

上面 Lambda 表达式的代码块只有一行调用类方法的代码，因此可以使用如下方法引用进行替换（程序清单同上）。

```
// 方法引用代替 Lambda 表达式：引用类方法
// 函数式接口中被实现方法的全部参数传给该类方法作为参数
Converter converter1 = Integer::valueOf;
```

对于上面的类方法引用，也就是调用 Integer 类的 valueOf()类方法来实现 Converter 函数式接口中唯一的抽象方法，当调用 Converter 接口中的唯一的抽象方法时，调用参数将会传给 Integer 类的 valueOf()类方法。

2. 引用特定对象的实例方法

下面看第二种方法引用：引用特定对象的实例方法。先使用 Lambda 表达式来创建一个 Converter 对象（程序清单同上）。

```
// 下面代码使用 Lambda 表达式创建 Converter 对象
Converter converter2 = from -> "fkit.org".indexOf(from);
```

上面 Lambda 表达式的代码块只有一条语句，因此程序省略了该代码块的花括号；而且由于表达式所实现的 convert()方法需要返回值，因此 Lambda 表达式将会把这条代码的值作为返回值。

接下来程序就可以调用 converter1 对象的 convert()方法将字符串转换为整数了，例如如下代码（程序清单同上）：

```
Integer value = converter2.convert("it");
System.out.println(value); // 输出 2
```

上面代码调用 converter1 对象的 convert()方法时——由于 converter1 对象是 Lambda 表达式创建的，convert()方法执行体就是 Lambda 表达式的代码块部分，因此上面程序输出 2。

上面 Lambda 表达式的代码块只有一行调用"fkit.org"的 indexOf()实例方法的代码，因此可以使用如下方法引用进行替换（程序清单同上）。

```
// 方法引用代替 Lambda 表达式：引用特定对象的实例方法
// 函数式接口中被实现方法的全部参数传给该方法作为参数
Converter converter2 = "fkit.org"::indexOf;
```

对于上面的实例方法引用，也就是调用"fkit.org"对象的 indexOf()实例方法来实现 Converter 函数式接口中唯一的抽象方法，当调用 Converter 接口中的唯一的抽象方法时，调用参数将会传给"fkit.org"对象的 indexOf()实例方法。

3. 引用某类对象的实例方法

下面看第三种方法引用：引用某类对象的实例方法。例如，定义了如下函数式接口（程序清单同上）。

```
@FunctionalInterface
interface MyTest
{
    String test(String a , int b , int c);
}
```

该函数式接口中包含一个 test()抽象方法，该方法负责根据 String、int、int 三个参数生成一个 String 返回值。下面代码使用 Lambda 表达式来创建一个 MyTest 对象（程序清单同上）。

```
// 下面代码使用 Lambda 表达式创建 MyTest 对象
MyTest mt = (a , b , c) -> a.substring(b , c);
```

上面 Lambda 表达式的代码块只有一条语句，因此程序省略了该代码块的花括号；而且由于表达式所实现的 test()方法需要返回值，因此 Lambda 表达式将会把这条代码的值作为返回值。

接下来程序就可以调用 mt 对象的 test()方法了，例如如下代码（程序清单同上）：

```
String str = mt.test("Java I Love you" , 2 , 9);
System.out.println(str); // 输出:va I Lo
```

上面代码调用 mt 对象的 test()方法时——由于 mt 对象是 Lambda 表达式创建的，test()方法执行体就是 Lambda 表达式的代码块部分，因此上面程序输出 va I Lo。

上面 Lambda 表达式的代码块只有一行 a.substring(b , c);，因此可以使用如下方法引用进行替换（程序清单同上）。

```
// 方法引用代替 Lambda 表达式：引用某类对象的实例方法
// 函数式接口中被实现方法的第一个参数作为调用者
// 后面的参数全部传给该方法作为参数
MyTest mt = String::substring;
```

对于上面的实例方法引用，也就是调用某个 String 对象的 substring()实例方法来实现 MyTest 函数式接口中唯一的抽象方法，当调用 MyTest 接口中的唯一的抽象方法时，第一个调用参数将作为 substring()

方法的调用者，剩下的调用参数会作为 substring()实例方法的调用参数。

4. 引用构造器

下面看构造器引用。例如，定义了如下函数式接口（程序清单同上）。

```
@FunctionalInterface
interface YourTest
{
    JFrame win(String title);
}
```

该函数式接口中包含一个 win()抽象方法，该方法负责根据 String 参数生成一个 JFrame 返回值。下面代码使用 Lambda 表达式来创建一个 YourTest 对象（程序清单同上）。

```
// 下面代码使用 Lambda 表达式创建 YourTest 对象
YourTest yt = (String a) -> new JFrame(a);
```

上面 Lambda 表达式的代码块只有一条语句，因此程序省略了该代码块的花括号；而且由于表达式所实现的 win()方法需要返回值，因此 Lambda 表达式将会把这条代码的值作为返回值。

接下来程序就可以调用 yt 对象的 win()方法了，例如如下代码（程序清单同上）：

```
JFrame jf = yt.win("我的窗口");
System.out.println(jf);
```

上面代码调用 yt 对象的 win()方法时——由于 yt 对象是 Lambda 表达式创建的，因此 win()方法执行体就是 Lambda 表达式的代码块部分，即执行体就是执行 new JFrame(a);语句，并将这条语句的值作为方法的返回值。

上面 Lambda 表达式的代码块只有一行 new JFrame(a);，因此可以使用如下构造器引用进行替换（程序清单同上）。

```
// 构造器引用代替 Lambda 表达式
// 函数式接口中被实现方法的全部参数传给该构造器作为参数
YourTest yt = JFrame::new;
```

对于上面的构造器引用，也就是调用某个 JFrame 类的构造器来实现 YourTest 函数式接口中唯一的抽象方法，当调用 YourTest 接口中的唯一的抽象方法时，调用参数将会传给 JFrame 构造器。从上面程序中可以看出，调用 YourTest 对象的 win()抽象方法时，实际只传入了一个 String 类型的参数，这个 String 类型的参数会被传给 JFrame 构造器——这就确定了是调用 JFrame 类的、带一个 String 参数的构造器。

6.8.4 Lambda 表达式与匿名内部类的联系和区别

从前面介绍可以看出，Lambda 表达式是匿名内部类的一种简化，因此它可以部分取代匿名内部类的作用，Lambda 表达式与匿名内部类存在如下相同点。

- Lambda 表达式与匿名内部类一样，都可以直接访问“effectively final”的局部变量，以及外部类的成员变量（包括实例变量和类变量）。
- Lambda 表达式创建的对象与匿名内部类生成的对象一样，都可以直接调用从接口中继承的默认方法。

下面程序示范了 Lambda 表达式与匿名内部类的相似之处。

程序清单：codes\06\6.8\LambdaAndInner.java

```
@FunctionalInterface
interface Displayable
{
    // 定义一个抽象方法和默认方法
    void display();
    default int add(int a , int b)
    {
        return a + b;
    }
}
public class LambdaAndInner
```

```
{
    private int age = 12;
    private static String name = "疯狂软件教育中心";
    public void test()
    {
        String book = "疯狂 Java 讲义";
        Displayable dis = ()->{
            // 访问“effectively final”的局部变量
            System.out.println("book 局部变量为：" + book);
            // 访问外部类的实例变量和类变量
            System.out.println("外部类的 age 实例变量为：" + age);
            System.out.println("外部类的 name 类变量为：" + name);
        };
        dis.display();
        // 调用 dis 对象从接口中继承的 add()方法
        System.out.println(dis.add(3 , 5));      // ①
    }
    public static void main(String[] args)
    {
        LambdaAndInner lambda = new LambdaAndInner();
        lambda.test();
    }
}
```

上面程序使用 Lambda 表达式创建了一个 Displayable 的对象，Lambda 表达式的代码块中的三行粗体字代码分别示范了访问“effectively final”的局部变量、外部类的实例变量和类变量。从这点来看，Lambda 表达式的代码块与匿名内部类的方法体是相同的。

与匿名内部类相似的是，由于 Lambda 表达式访问了 book 局部变量，因此该局部变量相当于有一个隐式的 final 修饰，因此同样不允许对 book 局部变量重新赋值。

当程序使用 Lambda 表达式创建了 Displayable 的对象之后，该对象不仅可调用接口中唯一的抽象方法，也可调用接口中的默认方法，如上面程序中①号粗体字代码所示。

Lambda 表达式与匿名内部类主要存在如下区别。

- 匿名内部类可以为任意接口创建实例——不管接口包含多少个抽象方法，只要匿名内部类实现所有的抽象方法即可；但 Lambda 表达式只能为函数式接口创建实例。
- 匿名内部类可以为抽象类甚至普通类创建实例；但 Lambda 表达式只能为函数式接口创建实例。
- 匿名内部类实现的抽象方法的方法体允许调用接口中定义的默认方法；但 Lambda 表达式的代码块不允许调用接口中定义的默认方法。

对于 Lambda 表达式的代码块不允许调用接口中定义的默认方法的限制，可以尝试对上面的 LambdaAndInner.java 程序稍做修改，在 Lambda 表达式的代码块中增加如下一行：

```
// 尝试调用接口中的默认方法，编译器会报错
System.out.println(add(3 , 5));
```

虽然 Lambda 表达式的目标类型：Displayable 中包含了 add()方法，但 Lambda 表达式的代码块不允许调用这个方法；如果将上面的 Lambda 表达式改为匿名内部类的写法，当匿名内部类实现 display()抽象方法时，则完全可以调用这个 add()方法。

6.8.5 使用 Lambda 表达式调用 Arrays 的类方法

前面介绍 Array 类的功能时已经提到，Arrays 类的有些方法需要 Comparator、XxxOperator、XxxFunction 等接口的实例，这些接口都是函数式接口，因此可以使用 Lambda 表达式来调用 Arrays 的方法。例如如下程序。

程序清单：codes\06\6.8\LambdaArrays.java

```
public class LambdaArrays
{
    public static void main(String[] args)
    {
        String[] arr1 = new String[]{"java" , "fkava" , "fkit", "ios" , "android"};
        Arrays.parallelSort(arr1, (o1, o2) -> o1.length() - o2.length());
```

```
        System.out.println(Arrays.toString(arr1));
        int[] arr2 = new int[]{3, -4 , 25, 16, 30, 18};
        // left 代表数组中前一个索引处的元素，计算第一个元素时，left 为 1
        // right 代表数组中当前索引处的元素
        Arrays.parallelPrefix(arr2, (left, right)-> left * right);
        System.out.println(Arrays.toString(arr2));
        long[] arr3 = new long[5];
        // operand 代表正在计算的元素索引
        Arrays.parallelSetAll(arr3 , operand -> operand * 5);
        System.out.println(Arrays.toString(arr3));
    }
}
```

上面程序中的粗体字代码就是 Lambda 表达式，第一段粗体字代码的 Lambda 表达式的目标类型是 Comparator，该 Comparator 指定了判断字符串大小的标准：字符串越长，即可认为该字符串越大；第二段粗体字代码的 Lambda 表达式的目标类型是 IntBinaryOperator，该对象将会根据前后两个元素来计算当前元素的值；第三段粗体字代码的 Lambda 表达式的目标类型是 IntToLongFunction，该对象将会根据元素的索引来计算当前元素的值。编译、运行该程序，即可看到如下输出：

```
[ios, java, fkit, fkava, android]
[3, -12, -300, -4800, -144000, -2592000]
[0, 5, 10, 15, 20]
```

通过该程序不难看出：Lambda 表达式可以让程序更加简洁。

6.9 枚举类

在某些情况下，一个类的对象是有限而且固定的，比如季节类，它只有 4 个对象；再比如行星类，目前只有 8 个对象。这种实例有限而且固定的类，在 Java 里被称为枚举类。

6.9.1 手动实现枚举类

在早期代码中，可能会直接使用简单的静态常量来表示枚举，例如如下代码：

```
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL = 3;
public static final int SEASON_WINTER = 4;
```

这种定义方法简单明了，但存在如下几个问题。

- 类型不安全：因为上面的每个季节实际上是一个 int 整数，因此完全可以把一个季节当成一个 int 整数使用，例如进行加法运算 SEASON_SPRING + SEASON_SUMMER，这样的代码完全正常。
- 没有命名空间：当需要使用季节时，必须在 SPRING 前使用 SEASON_前缀，否则程序可能与其他类中的静态常量混淆。
- 打印输出的意义不明确：当输出某个季节时，例如输出 SEASON_SPRING，实际上输出的是 1，这个 1 很难猜测它代表了春天。

但枚举又确实有存在的意义，因此早期也可采用通过定义类的方式来实现，可以采用如下设计方式。

- 通过 private 将构造器隐藏起来。
- 把这个类的所有可能实例都使用 public static final 修饰的类变量来保存。
- 如果有必要，可以提供一些静态方法，允许其他程序根据特定参数来获取与之匹配的实例。
- 使用枚举类可以使程序更加健壮，避免创建对象的随意性。

但通过定义类来实现枚举的代码量比较大，实现起来也比较麻烦，Java 从 JDK 1.5 后就增加了对枚举类的支持。



提示：

如果读者确实需要了解通过定义类的方法来实现枚举，可参考本书的第 2 版或第 1 版，也可参考本书光盘 codes\06\6.9 目录下的 Season.java 文件。

6.9.2 枚举类入门

Java 5 新增了一个 enum 关键字（它与 class、interface 关键字的地位相同），用以定义枚举类。正如前面看到的，枚举类是一种特殊的类，它一样可以有自己的成员变量、方法，可以实现一个或者多个接口，也可以定义自己的构造器。一个 Java 源文件中最多只能定义一个 public 访问权限的枚举类，且该 Java 源文件也必须和该枚举类的类名相同。

但枚举类终究不是普通类，它与普通类有如下简单区别。

- 枚举类可以实现一个或多个接口，使用 enum 定义的枚举类默认继承了 java.lang.Enum 类，而不是默认继承 Object 类，因此枚举类不能显式继承其他父类。其中 java.lang.Enum 类实现了 java.lang.Serializable 和 java.lang. Comparable 两个接口。
- 使用 enum 定义、非抽象的枚举类默认会使用 final 修饰，因此枚举类不能派生子类。
- 枚举类的构造器只能使用 private 访问控制符，如果省略了构造器的访问控制符，则默认使用 private 修饰；如果强制指定访问控制符，则只能指定 private 修饰符。
- 枚举类的所有实例必须在枚举类的第一行显式列出，否则这个枚举类永远都不能产生实例。列出这些实例时，系统会自动添加 public static final 修饰，无须程序员显式添加。

枚举类默认提供了一个 values()方法，该方法可以很方便地遍历所有的枚举值。

下面程序定义了一个 SeasonEnum 枚举类。

程序清单：codes\06\6.9\SeasonEnum.java

```
public enum SeasonEnum
{
    // 在第一行列出 4 个枚举实例
    SPRING,SUMMER,FALL,WINTER;
}
```

编译上面 Java 程序，将生成一个 SeasonEnum.class 文件，这表明枚举类是一个特殊的 Java 类。由此可见，enum 关键字和 class、interface 关键字的作用大致相似。

定义枚举类时，需要显式列出所有的枚举值，如上面的 SPRING,SUMMER,FALL,WINTER;所示，所有的枚举值之间以英文逗号（,）隔开，枚举值列举结束后以英文分号作为结束。这些枚举值代表了该枚举类的所有可能的实例。

如果需要使用该枚举类的某个实例，则可使用 EnumClass.variable 的形式，如 SeasonEnum.SPRING。

程序清单：codes\06\6.9\EnumTest.java

```
public class EnumTest
{
    public void judge(SeasonEnum s)
    {
        // switch 语句里的表达式可以是枚举值
        switch (s)
        {
            case SPRING:
                System.out.println("春暖花开，正好踏青");
                break;
            case SUMMER:
                System.out.println("夏日炎炎，适合游泳");
                break;
            case FALL:
                System.out.println("秋高气爽，进补及时");
                break;
            case WINTER:
                System.out.println("冬日雪飘，围炉赏雪");
                break;
        }
    }
    public static void main(String[] args)
    {
        // 枚举类默认有一个 values()方法，返回该枚举类的所有实例
        for (SeasonEnum s : SeasonEnum.values())
        {
```

```
            System.out.println(s);
        }
        // 使用枚举实例时，可通过 EnumClass.variable 形式来访问
        new EnumTest().judge(SeasonEnum.SPRING);
    }
}
```

上面程序测试了 SeasonEnum 枚举类的用法，该类通过 values()方法返回了 SeasonEnum 枚举类的所有实例，并通过循环迭代输出了 SeasonEnum 枚举类的所有实例。

不仅如此，上面程序的 switich 表达式中还使用了 SeasonEnum 对象作为表达式，这是 JDK 1.5 增加枚举后对 switch 的扩展：switch 的控制表达式可以是任何枚举类型。不仅如此，当 switch 控制表达式使用枚举类型时，后面 case 表达式中的值直接使用枚举值的名字，无须添加枚举类作为限定。

前面已经介绍过，所有的枚举类都继承了 java.lang.Enum 类，所以枚举类可以直接使用 java.lang.Enum 类中所包含的方法。java.lang.Enum 类中提供了如下几个方法。

- int compareTo(E o)：该方法用于与指定枚举对象比较顺序，同一个枚举实例只能与相同类型的枚举实例进行比较。如果该枚举对象位于指定枚举对象之后，则返回正整数；如果该枚举对象位于指定枚举对象之前，则返回负整数，否则返回零。
- String name()：返回此枚举实例的名称，这个名称就是定义枚举类时列出的所有枚举值之一。与此方法相比，大多数程序员应该优先考虑使用 toString()方法，因为 toString()方法返回更加用户友好的名称。
- int ordinal()：返回枚举值在枚举类中的索引值（就是枚举值在枚举声明中的位置，第一个枚举值的索引值为零）。
- String toString()：返回枚举常量的名称，与 name 方法相似，但 toString()方法更常用。
- public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)：这是一个静态方法，用于返回指定枚举类中指定名称的枚举值。名称必须与在该枚举类中声明枚举值时所用的标识符完全匹配，不允许使用额外的空白字符。

正如前面看到的，当程序使用 System.out.println(s)语句来打印枚举值时，实际上输出的是该枚举值的 toString()方法，也就是输出该枚举值的名字。

6.9.3 枚举类的成员变量、方法和构造器

枚举类也是一种类，只是它是一种比较特殊的类，因此它一样可以定义成员变量、方法和构造器。下面程序将定义一个 Gender 枚举类，该枚举类里包含了一个 name 实例变量。

程序清单：codes\06\6.9\Gender.java

```
public enum Gender
{
    MALE,FEMALE;
    // 定义一个 public 修饰的实例变量
    public String name;
}
```

上面的 Gender 枚举类里定义了一个名为 name 的实例变量，并且将它定义成一个 public 访问权限的。下面通过如下程序来使用该枚举类。

程序清单：codes\06\6.9\GenderTest.java

```
public class GenderTest
{
    public static void main(String[] args)
    {
        // 通过 Enum 的 valueOf()方法来获取指定枚举类的枚举值
        Gender g = Enum.valueOf(Gender.class , "FEMALE");
        // 直接为枚举值的 name 实例变量赋值
        g.name = "女";
        // 直接访问枚举值的 name 实例变量
        System.out.println(g + "代表:" + g.name);
    }
}
```

上面程序使用 Gender 枚举类时与使用一个普通类没有太大的差别，差别只是产生 Gender 对象的方式不同，枚举类的实例只能是枚举值，而不是随意地通过 new 来创建枚举类对象。

正如前面提到的，Java 应该把所有类设计成良好封装的类，所以不应该允许直接访问 Gender 类的 name 成员变量，而是应该通过方法来控制对 name 的访问。否则可能出现很混乱的情形，例如上面程序恰好设置了 g.name = "女"，要是采用 g.name = "男"，那程序就会非常混乱了，可能出现 FEMALE 代表男的局面。可以按如下代码来改进 Gender 类的设计。

程序清单：codes\06\6.9\better\Gender.java

```
public enum Gender
{
    MALE,FEMALE;
    private String name;
    public void setName(String name)
    {
        switch (this)
        {
            case MALE:
                if (name.equals("男"))
                {
                    this.name = name;
                }
                else
                {
                    System.out.println("参数错误");
                    return;
                }
                break;
            case FEMALE:
                if (name.equals("女"))
                {
                    this.name = name;
                }
                else
                {
                    System.out.println("参数错误");
                    return;
                }
                break;
        }
    }
    public String getName()
    {
        return this.name;
    }
}
```

上面程序把 name 设置成 private，从而避免其他程序直接访问该 name 成员变量，必须通过 setName() 方法来修改 Gender 实例的 name 变量，而 setName()方法就可以保证不会产生混乱。上面程序中粗体字部分保证 FEMALE 枚举值的 name 变量只能设置为"女"，而 MALE 枚举值的 name 变量则只能设置为"男"。看如下程序。

程序清单：codes\06\6.9\better\GenderTest.java

```
public class GenderTest
{
    public static void main(String[] args)
    {
        Gender g = Gender.valueOf("FEMALE");
        g.setName("女");
        System.out.println(g + "代表:" + g.getName());
        // 此时设置 name 值时将会提示参数错误
        g.setName("男");
        System.out.println(g + "代表:" + g.getName());
    }
}
```

上面代码中粗体字部分试图将一个 FEMALE 枚举值的 name 变量设置为"男"，系统将会提示参数错误。

实际上这种做法依然不够好，枚举类通常应该设计成不可变类，也就是说，它的成员变量值不应该允许改变，这样会更安全，而且代码更加简洁。因此建议将枚举类的成员变量都使用 private final 修饰。

如果将所有的成员变量都使用了 final 修饰符来修饰，所以必须在构造器里为这些成员变量指定初始值（或者在定义成员变量时指定默认值，或者在初始化块中指定初始值，但这两种情况并不常见），因此应该为枚举类显式定义带参数的构造器。

一旦为枚举类显式定义了带参数的构造器，列出枚举值时就必须对应地传入参数。

程序清单：codes\06\6.9\best\Gender.java

```
public enum Gender
{
    // 此处的枚举值必须调用对应的构造器来创建
    MALE("男"),FEMALE("女");
    private final String name;
    // 枚举类的构造器只能使用 private 修饰
    private Gender(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
}
```

从上面程序中可以看出，当为 Gender 枚举类创建了一个 Gender(String name)构造器之后，列出枚举值就应该采用粗体字代码来完成。也就是说，在枚举类中列出枚举值时，实际上就是调用构造器创建枚举类对象，只是这里无须使用 new 关键字，也无须显式调用构造器。前面列出枚举值时无须传入参数，甚至无须使用括号，仅仅是因为前面的枚举类包含无参数的构造器。

不难看出，上面程序中粗体字代码实际上等同于如下两行代码：

```
public static final Gender MALE = new Gender("男");
public static final Gender FEMALE = new Gender("女");
```

6.9.4 实现接口的枚举类

枚举类也可以实现一个或多个接口。与普通类实现一个或多个接口完全一样，枚举类实现一个或多个接口时，也需要实现该接口所包含的方法。下面程序定义了一个 GenderDesc 接口。

程序清单：codes\06\6.9\interface\GenderDesc.java

```
public interface GenderDesc
{
    void info();
}
```

在上面 GenderDesc 接口中定义了一个 info()方法，下面的 Gender 枚举类实现了该接口，并实现了该接口里包含的 info()方法。下面是 Gender 枚举类的代码。

程序清单：codes\06\6.9\interface\Gender.java

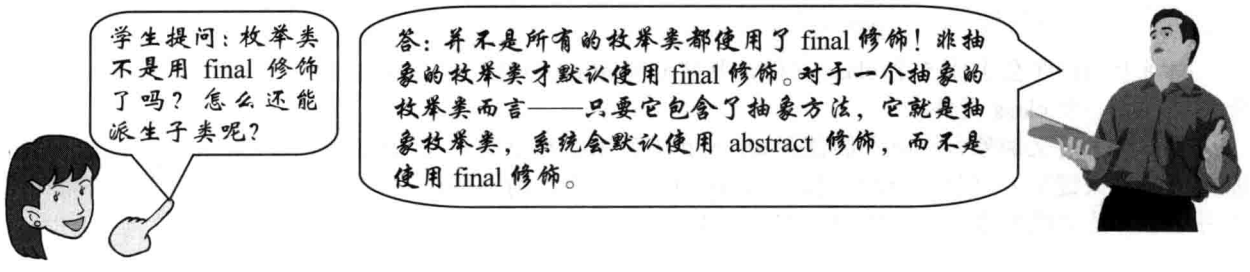
```
public enum Gender implements GenderDesc
{
    // 其他部分与 codes\06\6.9\best\Gender.java 中的 Gender 类完全相同
    ...
    // 增加下面的 info()方法，实现 GenderDesc 接口必须实现的方法
    public void info()
    {
        System.out.println(
            "这是一个用于定义性别的枚举类");
    }
}
```

读者可能会发现，枚举类实现接口不过如此，与普通类实现接口完全一样：使用 implements 实现接口，并实现接口里包含的抽象方法。

如果由枚举类来实现接口里的方法，则每个枚举值在调用该方法时都有相同的行为方式（因为方法体完全一样）。如果需要每个枚举值在调用该方法时呈现出不同的行为方式，则可以让每个枚举值分别来实现该方法，每个枚举值提供不同的实现方式，从而让不同的枚举值调用该方法时具有不同的行为方式。在下面的 Gender 枚举类中，不同的枚举值对 info()方法的实现各不相同（程序清单同上）。

```
public enum Gender implements GenderDesc
{
    // 此处的枚举值必须调用对应的构造器来创建
    MALE("男")
    // 花括号部分实际上是一个类体部分
    {
        public void info()
        {
            System.out.println("这个枚举值代表男性");
        }
    },
    FEMALE("女")
    {
        public void info()
        {
            System.out.println("这个枚举值代表女性");
        }
    };
    //枚举类的其他部分与 codes\06\6.9\best\Gender.java 中的 Gender 类完全相同
    ...
}
```

上面代码的粗体字部分看起来有些奇怪：当创建 MALE 和 FEMALE 两个枚举值时，后面又紧跟了一对花括号，这对花括号里包含了一个 info()方法定义。如果读者还记得匿名内部类语法的话，则可能对这样的语法有点印象了，花括号部分实际上就是一个类体部分，在这种情况下，当创建 MALE、FEMALE 枚举值时，并不是直接创建 Gender 枚举类的实例，而是相当于创建 Gender 的匿名子类的实例。因为粗体字括号部分实际上是匿名内部类的类体部分，所以这个部分的代码语法与前面介绍的匿名内部类语法大致相似，只是它依然是枚举类的匿名内部子类。



编译上面的程序，可以看到生成了 Gender.class、Gender$1.class 和 Gender$2.class 三个文件，这样的三个 class 文件正好证明了上面的结论：MALE 和 FEMALE 实际上是 Gender 匿名子类的实例，而不是 Gender 类的实例。当调用 MALE 和 FEMALE 两个枚举值的方法时，就会看到两个枚举值的方法表现不同的行为方式。

6.9.5 包含抽象方法的枚举类

假设有一个 Operation 枚举类，它的 4 个枚举值 PLUS, MINUS, TIMES, DIVIDE 分别代表加、减、乘、除 4 种运算，该枚举类需要定义一个 eval()方法来完成计算。

从上面描述可以看出，Operation 需要让 PLUS、MINUS、TIMES、DIVIDE 四个值对 eval()方法各有不同的实现。此时可考虑为 Operation 枚举类定义一个 eval()抽象方法，然后让 4 个枚举值分别为 eval()提供不同的实现。例如如下代码。

程序清单：codes\06\6.9\abstract\Operation.java

```
public enum Operation
{
    PLUS
    {
        public double eval(double x , double y)
        {
            return x + y;
        }
    },
    MINUS
    {
        public double eval(double x , double y)
        {
            return x - y;
        }
    },
    TIMES
    {
        public double eval(double x , double y)
        {
            return x * y;
        }
    },
    DIVIDE
    {
        public double eval(double x , double y)
        {
            return x / y;
        }
    };
    // 为枚举类定义一个抽象方法
    // 这个抽象方法由不同的枚举值提供不同的实现
    public abstract double eval(double x, double y);
    public static void main(String[] args)
    {
        System.out.println(Operation.PLUS.eval(3, 4));
        System.out.println(Operation.MINUS.eval(5, 4));
        System.out.println(Operation.TIMES.eval(5, 4));
        System.out.println(Operation.DIVIDE.eval(5, 4));
    }
}
```

编译上面程序会生成5个class文件，其实Operation对应一个class文件，它的4个匿名内部子类分别各对应一个class文件。

枚举类里定义抽象方法时不能使用abstract关键字将枚举类定义成抽象类（因为系统自动会为它添加abstract关键字），但因为枚举类需要显式创建枚举值，而不是作为父类，所以定义每个枚举值时必须为抽象方法提供实现，否则将出现编译错误。

6.10 对象与垃圾回收

第1章已经介绍过，Java的垃圾回收是Java语言的重要功能之一。当程序创建对象、数组等引用类型实体时，系统都会在堆内存中为之分配一块内存区，对象就保存在这块内存区中，当这块内存不再被任何引用变量引用时，这块内存就变成垃圾，等待垃圾回收机制进行回收。垃圾回收机制具有如下特征。

- 垃圾回收机制只负责回收堆内存中的对象，不会回收任何物理资源（例如数据库连接、网络IO等资源）。
- 程序无法精确控制垃圾回收的运行，垃圾回收会在合适的时候进行。当对象永久性地失去引用后，系统就会在合适的时候回收它所占的内存。
- 在垃圾回收机制回收任何对象之前，总会先调用它的finalize()方法，该方法可能使该对象重新复活（让一个引用变量重新引用该对象），从而导致垃圾回收机制取消回收。

6.10.1 对象在内存中的状态

当一个对象在堆内存中运行时，根据它被引用变量所引用的状态，可以把它所处的状态分成如下三种。

- 可达状态：当一个对象被创建后，若有一个以上的引用变量引用它，则这个对象在程序中处于可达状态，程序可通过引用变量来调用该对象的实例变量和方法。
- 可恢复状态：如果程序中某个对象不再有任何引用变量引用它，它就进入了可恢复状态。在这种状态下，系统的垃圾回收机制准备回收该对象所占用的内存，在回收该对象之前，系统会调用所有可恢复状态对象的 finalize()方法进行资源清理。如果系统在调用 finalize()方法时重新让一个引用变量引用该对象，则这个对象会再次变为可达状态；否则该对象将进入不可达状态。
- 不可达状态：当对象与所有引用变量的关联都被切断，且系统已经调用所有对象的 finalize()方法后依然没有使该对象变成可达状态，那么这个对象将永久性地失去引用，最后变成不可达状态。只有当一个对象处于不可达状态时，系统才会真正回收该对象所占有的资源。

图 6.7 显示了对象的三种状态的转换示意图。

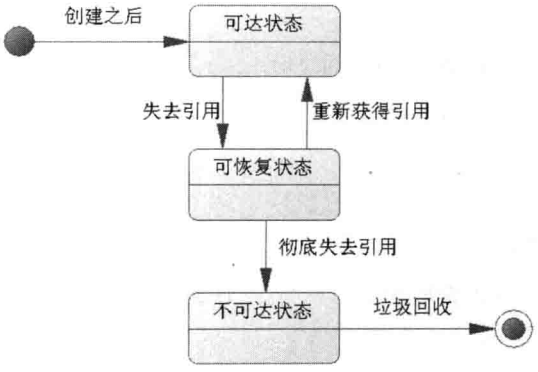


图 6.7 对象的状态转换示意图

例如，下面程序简单地创建了两个字符串对象，并创建了一个引用变量依次指向两个对象。

程序清单：codes\06\6.10\StatusTranfer.java

```
public class StatusTranfer
{
    public static void test()
    {
        String a = new String("轻量级 Java EE 企业应用实战"); // ①
        a = new String("疯狂 Java 讲义");   // ②
    }
    public static void main(String[] args)
    {
        test();     // ③
    }
}
```

当程序执行 test 方法的①代码时，代码定义了一个 a 变量，并让该变量指向"轻量级 Java EE 企业应用实战"字符串，该代码执行结束后，"轻量级 Java EE 企业应用实战"字符串对象处于可达状态。

当程序执行了 test 方法的②代码后，代码再次创建了"疯狂 Java 讲义"字符串对象，并让 a 变量指向该对象。此时，"轻量级 Java EE 企业应用实战"字符串对象处于可恢复状态，而"疯狂 Java 讲义"字符串对象处于可达状态。

一个对象可以被一个方法的局部变量引用，也可以被其他类的类变量引用，或被其他对象的实例变量引用。当某个对象被其他类的类变量引用时，只有该类被销毁后，该对象才会进入可恢复状态；当某个对象被其他对象的实例变量引用时，只有当该对象被销毁后，该对象才会进入可恢复状态。

6.10.2 强制垃圾回收

当一个对象失去引用后，系统何时调用它的 finalize()方法对它进行资源清理，何时它会变成不可达

状态，系统何时回收它所占有的内存，对于程序完全透明。程序只能控制一个对象何时不再被任何引用变量引用，绝不能控制它何时被回收。

程序无法精确控制 Java 垃圾回收的时机，但依然可以强制系统进行垃圾回收——这种强制只是通知系统进行垃圾回收，但系统是否进行垃圾回收依然不确定。大部分时候，程序强制系统垃圾回收后总会有一些效果。强制系统垃圾回收有如下两种方式。

- 调用 System 类的 gc()静态方法：System.gc()。
- 调用 Runtime 对象的 gc()实例方法：Runtime.getRuntime().gc()。



提示：

关于 System 和 Runtime 请参考本书第 7 章的内容。

下面程序创建了 4 个匿名对象，每个对象创建之后立即进入可恢复状态，等待系统回收，但直到程序退出，系统依然不会回收该资源。

程序清单：codes\06\6.10\GcTest.java

```
public class GcTest
{
    public static void main(String[] args)
    {
        for (int i = 0 ; i < 4; i++)
        {
            new GcTest();
        }
    }
    public void finalize()
    {
        System.out.println("系统正在清理 GcTest 对象的资源...");
    }
}
```

编译、运行上面程序，看不到任何输出，可见直到系统退出，系统都不曾调用 GcTest 对象的 finalize()方法。但如果将程序修改成如下形式（程序清单同上）：

```
public class GcTest
{
    public static void main(String[] args)
    {
        for (int i = 0 ; i < 4; i++)
        {
            new GcTest();
            // 下面两行代码的作用完全相同，强制系统进行垃圾回收
            // System.gc();
            Runtime.getRuntime().gc();
        }
    }
    public void finalize()
    {
        System.out.println("系统正在清理 GcTest 对象的资源...");
    }
}
```

上面程序与前一个程序相比，只是增加了粗体字代码行，此代码行强制系统进行垃圾回收。编译上面程序，使用如下命令来运行此程序：

```
java -verbose:gc GcTest
```

运行 java 命令时指定-verbose:gc 选项，可以看到每次垃圾回收后的提示信息，如图 6.8 所示。



图 6.8　垃圾回收的运行提示信息

从图 6.8 中可以看出，每次调用了 Runtime.getRuntime().gc()代码后，系统垃圾回收机制还是“有所动作”的，可以看出垃圾回收之前、回收之后的内

存占用对比。

虽然图 6.8 显示了程序强制垃圾回收的效果，但这种强制只是建议系统立即进行垃圾回收，系统完全有可能并不立即进行垃圾回收，垃圾回收机制也不会对程序的建议完全置之不理：垃圾回收机制会在收到通知后，尽快进行垃圾回收。

6.10.3 finalize 方法

在垃圾回收机制回收某个对象所占用的内存之前，通常要求程序调用适当的方法来清理资源，在没有明确指定清理资源的情况下，Java 提供了默认机制来清理该对象的资源，这个机制就是 finalize()方法。该方法是定义在 Object 类里的实例方法，方法原型为：

```
protected void finalize() throws Throwable
```

当 finalize()方法返回后，对象消失，垃圾回收机制开始执行。方法原型中的 throws Throwable 表示它可以抛出任何类型的异常。

任何 Java 类都可以重写 Object 类的 finalize()方法，在该方法中清理该对象占用的资源。如果程序终止之前始终没有进行垃圾回收，则不会调用失去引用对象的 finalize()方法来清理资源。垃圾回收机制何时调用对象的 finalize()方法是完全透明的，只有当程序认为需要更多的额外内存时，垃圾回收机制才会进行垃圾回收。因此，完全有可能出现这样一种情形：某个失去引用的对象只占用了少量内存，而且系统没有产生严重的内存需求，因此垃圾回收机制并没有试图回收该对象所占用的资源，所以该对象的 finalize()方法也不会得到调用。

finalize()方法具有如下 4 个特点。

- 永远不要主动调用某个对象的 finalize()方法，该方法应交给垃圾回收机制调用。
- finalize()方法何时被调用，是否被调用具有不确定性，不要把 finalize()方法当成一定会被执行的方法。
- 当 JVM 执行可恢复对象的 finalize()方法时，可能使该对象或系统中其他对象重新变成可达状态。
- 当 JVM 执行 finalize()方法时出现异常时，垃圾回收机制不会报告异常，程序继续执行。

注意：

由于 finalize()方法并不一定会被执行，因此如果想清理某个类里打开的资源，则不要放在 finalize()方法中进行清理，后面会介绍专门用于清理资源的方法。



下面程序演示了如何在 finalize()方法里复活自身，并可通过该程序看出垃圾回收的不确定性。

程序清单：codes\06\6.10\FinalizeTest.java

```
public class FinalizeTest
{
    private static FinalizeTest ft = null;
    public void info()
    {
        System.out.println("测试资源清理的 finalize 方法");
    }
    public static void main(String[] args) throws Exception
    {
        // 创建 FinalizeTest 对象立即进入可恢复状态
        new FinalizeTest();
        // 通知系统进行资源回收
        System.gc();    // ①
        // 强制垃圾回收机制调用可恢复对象的 finalize()方法
//        Runtime.getRuntime().runFinalization();  // ②
        System.runFinalization();   // ③
        ft.info();
    }
    public void finalize()
    {
        // 让 tf 引用到试图回收的可恢复对象，即可恢复对象重新变成可达
```

```
            ft = this;
        }
    }
```

上面程序中定义了一个 FinalizeTest 类，重写了该类的 finalize()方法，在该方法中把需要清理的可恢复对象重新赋给 ft 引用变量，从而让该可恢复对象重新变成可达状态。

上面程序中的 main()方法创建了一个 FinalizeTest 类的匿名对象，因为创建后没有把这个对象赋给任何引用变量，所以该对象立即进入可恢复状态。进入可恢复状态后，系统调用①号粗体字代码通知系统进行垃圾回收，②号粗体字代码强制系统立即调用可恢复对象的 finalize()方法，再次调用 ft 对象的 info()方法。编译、运行上面程序，看到 ft 的 info()方法被正常执行。

如果删除①行代码，取消强制垃圾回收。再次编译、运行上面程序，将会看到如图 6.9 所示的结果。

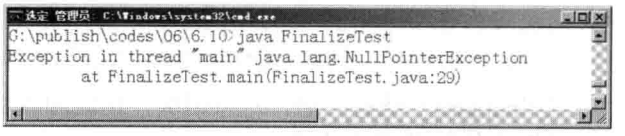


图 6.9　调用 ft.info()方法时引发空指针异常

从图 6.9 所示的运行结果可以看出，如果取消①号粗体字代码，程序并没有通知系统开始执行垃圾回收（而且程序内存也没有紧张），因此系统通常不会立即进行垃圾回收，也就不会调用 FinalizeTest 对象的 finalize()方法，这样 FinalizeTest 的 ft 类变量将依然保持为 null，这样就导致了空指针异常。

上面程序中②号代码和③号代码都用于强制垃圾回收机制调用可恢复对象的 finalize()方法，如果程序仅执行 System.gc();代码，而不执行②号或③号粗体字代码——由于 JVM 垃圾回收机制的不确定性，JVM 往往并不立即调用可恢复对象的 finalize()方法，这样 FinalizeTest 的 ft 类变量可能依然为 null，可能依然会导致空指针异常

➤➤ 6.10.4　对象的软、弱和虚引用

对大部分对象而言，程序里会有一个引用变量引用该对象，这是最常见的引用方式。除此之外，java.lang.ref 包下提供了 3 个类：SoftReference、PhantomReference 和 WeakReference，它们分别代表了系统对对象的 3 种引用方式：软引用、虚引用和弱引用。因此，Java 语言对对象的引用有如下 4 种方式。

1．强引用（StrongReference）

这是 Java 程序中最常见的引用方式。程序创建一个对象，并把这个对象赋给一个引用变量，程序通过该引用变量来操作实际的对象，前面介绍的对象和数组都采用了这种强引用的方式。当一个对象被一个或一个以上的引用变量所引用时，它处于可达状态，不可能被系统垃圾回收机制回收。

2．软引用（SoftReference）

软引用需要通过 SoftReference 类来实现，当一个对象只有软引用时，它有可能被垃圾回收机制回收。对于只有软引用的对象而言，当系统内存空间足够时，它不会被系统回收，程序也可使用该对象；当系统内存空间不足时，系统可能会回收它。软引用通常用于对内存敏感的程序中。

3．弱引用（WeakReference）

弱引用通过 WeakReference 类实现，弱引用和软引用很像，但弱引用的引用级别更低。对于只有弱引用的对象而言，当系统垃圾回收机制运行时，不管系统内存是否足够，总会回收该对象所占用的内存。当然，并不是说当一个对象只有弱引用时，它就会立即被回收——正如那些失去引用的对象一样，必须等到系统垃圾回收机制运行时才会被回收。

4．虚引用（PhantomReference）

虚引用通过 PhantomReference 类实现，虚引用完全类似于没有引用。虚引用对对象本身没有太大影响，对象甚至感觉不到虚引用的存在。如果一个对象只有一个虚引用时，那么它和没有引用的效果大致相同。虚引用主要用于跟踪对象被垃圾回收的状态，虚引用不能单独使用，虚引用必须和引用队列（ReferenceQueue）联合使用。

上面三个引用类都包含了一个 get()方法，用于获取被它们所引用的对象。



提示： 如果需要掌握 JDK 系统类的详细用法，例如，包含哪些可用的成员变量和方法（protected 和 public 权限的成员变量和方法），以及包含哪些构造器都应该查阅 Java 提供的 API 文档。

引用队列由 java.lang.ref.ReferenceQueue 类表示，它用于保存被回收后对象的引用。当联合使用软引用、弱引用和引用队列时，系统在回收被引用的对象之后，将把被回收对象对应的引用添加到关联的引用队列中。与软引用和弱引用不同的是，虚引用在对象被释放之前，将把它对应的虚引用添加到它关联的引用队列中，这使得可以在对象被回收之前采取行动。

软引用和弱引用可以单独使用，但虚引用不能单独使用，单独使用虚引用没有太大的意义。虚引用的主要作用就是跟踪对象被垃圾回收的状态，程序可以通过检查与虚引用关联的引用队列中是否已经包含了该虚引用，从而了解虚引用所引用的对象是否即将被回收。

下面程序示范了弱引用所引用的对象被系统垃圾回收过程。

程序清单：codes\06\6.10\ReferenceTest.java

```
public class ReferenceTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个字符串对象
        String str = new String("疯狂 Java 讲义");
        // 创建一个弱引用，让此弱引用引用到"疯狂 Java 讲义"字符串
        WeakReference wr = new WeakReference(str); // ①
        // 切断 str 引用和"疯狂 Java 讲义"字符串之间的引用
        str = null;   // ②
        // 取出弱引用所引用的对象
        System.out.println(wr.get()); // ③
        // 强制垃圾回收
        System.gc();
        System.runFinalization();
        // 再次取出弱引用所引用的对象
        System.out.println(wr.get()); // ④
    }
}
```

上面程序先创建了一个"疯狂 Java 讲义"字符串对象，并让 str 引用变量引用它，执行①行粗体字代码时，系统创建了一个弱引用对象，并让该对象和 str 引用同一个对象。当程序执行到②行代码时，程序切断了 str 和"疯狂 Java 讲义"字符串对象之间的引用关系。此时系统内存如图 6.10 所示。

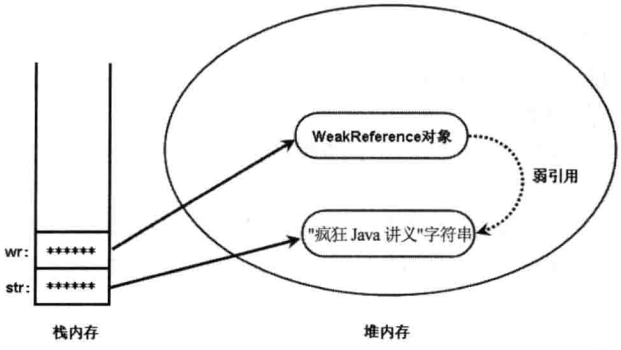


图 6.10 仅被弱引用引用的字符串对象



提示： 编译上面程序时会出现一个警告提示，这个警告提示是一个泛型提示。此处先不要理它。不仅如此，上面程序创建"疯狂 Java 讲义"字符串对象时，不要使用 String str = "疯狂 Java 讲义";，否则将看不到运行效果。因为采用 String str = "疯狂 Java 讲义";代码定义字

> 符串时，系统会使用常量池来管理这个字符串直接量（会使用强引用来引用它），系统不会回收这个字符串直接量。

当程序执行到③号粗体字代码时，由于本程序不会导致内存紧张，此时程序通常还不会回收弱引用wr所引用的对象，因此在③号代码处可以看到输出"疯狂Java讲义"字符串。

执行到③号粗体字代码之后，程序调用了System.gc();和System.runFinalization();通知系统进行垃圾回收，如果系统立即进行垃圾回收，那么就会将弱引用wr所引用的对象回收。接下来在④号粗体字代码处将看到输出null。

下面程序与上面程序基本相似，只是使用了虚引用来引用字符串对象，虚引用无法获取它引用的对象。下面程序还将虚引用和引用队列结合使用，可以看到被虚引用所引用的对象被垃圾回收后，虚引用将被添加到引用队列中。

程序清单：codes\06\6.10\PhantomReferenceTest.java

```
public class PhantomReferenceTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个字符串对象
        String str = new String("疯狂 Java 讲义");
        // 创建一个引用队列
        ReferenceQueue rq = new ReferenceQueue();
        // 创建一个虚引用，让此虚引用引用到"疯狂 Java 讲义"字符串
        PhantomReference pr = new PhantomReference (str , rq);
        // 切断 str 引用和"疯狂 Java 讲义"字符串之间的引用
        str = null;
        // 取出虚引用所引用的对象，并不能通过虚引用获取被引用的对象，所以此处输出 null
        System.out.println(pr.get()); // ①
        // 强制垃圾回收
        System.gc();
        System.runFinalization();
        // 垃圾回收之后，虚引用将被放入引用队列中
        // 取出引用队列中最先进入队列的引用与 pr 进行比较
        System.out.println(rq.poll() == pr);  // ②
    }
}
```

因为系统无法通过虚引用来获得被引用的对象，所以执行①处的输出语句时，程序将输出null（即使此时并未强制进行垃圾回收）。当程序强制垃圾回收后，只有虚引用引用的字符串对象将会被垃圾回收，当被引用的对象被回收后，对应的虚引用将被添加到关联的引用队列中，因而将在②代码处看到输出true。

使用这些引用类可以避免在程序执行期间将对象留在内存中。如果以软引用、弱引用或虚引用的方式引用对象，垃圾回收器就能够随意地释放对象。如果希望尽可能减小程序在其生命周期中所占用的内存大小时，这些引用类就很有用处。

必须指出：要使用这些特殊的引用类，就不能保留对对象的强引用；如果保留了对对象的强引用，就会浪费这些引用类所提供的任何好处。

由于垃圾回收的不确定性，当程序希望从软、弱引用中取出被引用对象时，可能这个被引用对象已经被释放了。如果程序需要使用那个被引用的对象，则必须重新创建该对象。这个过程可以采用两种方式完成，下面代码显示了其中一种方式。

```
// 取出弱引用所引用的对象
obj = wr.get();
// 如果取出的对象为 null
if (obj == null)
{
    // 重新创建一个新的对象，再次让弱引用去引用该对象
    wr = new WeakReference(recreateIt());     // ①
    // 取出弱引用所引用的对象，将其赋给 obj 变量
```

```
    obj = wr.get();    // ②
}
...// 操作 obj 对象
// 再次切断 obj 和对象之间的关联
obj = null;
```

下面代码显示了另一种取出被引用对象的方式。

```
// 取出弱引用所引用的对象
obj = wr.get();
// 如果取出的对象为 null
if (obj == null)
{
    // 重新创建一个新的对象，并使用强引用来引用它
    obj = recreateIt();
    // 取出弱引用所引用的对象，将其赋给 obj 变量
    wr = new WeakReference(obj);
}
...// 操作 obj 对象
// 再次切断 obj 和对象之间的关联
obj = null;
```

上面两段代码采用的都是伪码，其中 recreateIt()方法用于生成一个 obj 对象。这两段代码都是先判断 obj 对象是否已经被回收，如果已经被回收，则重新创建该对象。如果弱引用引用的对象已经被垃圾回收释放了，则重新创建该对象。但第一段代码存在一定的问题：当 if 块执行完成后，obj 还是有可能为 null。因为垃圾回收的不确定性，假设系统在①和②行代码之间进行垃圾回收，则系统会再次将 wr 所引用的对象回收，从而导致 obj 依然为 null。第二段代码则不会存在这个问题，当 if 块执行结束后，obj 一定不为 null。

6.11 修饰符的适用范围

到目前为止，已经学习了 Java 中的大部分修饰符，如访问控制符、static 和 final 等。还有其他的一些修饰符将会在后面的章节里继续介绍，此处给出 Java 修饰符适用范围总表（见表 6.3）。

表 6.3 Java 修饰符适用范围总表

	外部类/接口	成员属性	方法	构造器	初始化块	成员内部类	局部成员
public	√	√	√	√		√	
protected		√	√	√		√	
包访问控制符	√	√	√	√	○	√	○
private		√	√	√		√	
abstract	√		√			√	
final	√	√	√			√	√
static		√	√		√	√	
strictfp	√		√			√	
synchronized			√				
native			√				
transient		√					
volatile		√					
default			√				

在表 6.3 中，包访问控制符是一个特殊的修饰符，不用任何访问控制符的就是包访问控制。对于初始化块和局部成员而言，它们不能使用任何访问控制符，所以看起来像使用了包访问控制符。

strictfp 关键字的含义是 FP-strict，也就是精确浮点的意思。在 Java 虚拟机进行浮点运算时，如果没有指定 strictfp 关键字，Java 的编译器和运行时环境在浮点运算上不一定令人满意。一旦使用了 strictfp 来修饰类、接口或者方法时，那么在所修饰的范围内 Java 的编译器和运行时环境会完全依照浮点规范

IEEE-754 来执行。因此，如果想让浮点运算更加精确，就可以使用 strictfp 关键字来修饰类、接口和方法。

native 关键字主要用于修饰一个方法，使用 native 修饰的方法类似于一个抽象方法。与抽象方法不同的是，native 方法通常采用 C 语言来实现。如果某个方法需要利用平台相关特性，或者访问系统硬件等，则可以使用 native 修饰该方法，再把该方法交给 C 去实现。一旦 Java 程序中包含了 native 方法，这个程序将失去跨平台的功能。

其他修饰符如 synchronized、transient 将在后面章节中有更详细的介绍，此处不再赘述。

在表 6.3 列出的所有修饰符中，4 个访问控制符是互斥的，最多只能出现其中之一。不仅如此，还有 abstract 和 final 永远不能同时使用；abstract 和 static 不能同时修饰方法，可以同时修饰内部类；abstract 和 private 不能同时修饰方法，可以同时修饰内部类。private 和 final 同时修饰方法虽然语法是合法的，但没有太大的意义——由于 private 修饰的方法不可能被子类重写，因此使用 final 修饰没什么意义。

6.12 使用 JAR 文件

JAR 文件的全称是 Java Archive File，意思就是 Java 档案文件。通常 JAR 文件是一种压缩文件，与常见的 ZIP 压缩文件兼容，通常也被称为 JAR 包。JAR 文件与 ZIP 文件的区别就是在 JAR 文件中默认包含了一个名为 META-INF/MANIFEST.MF 的清单文件，这个清单文件是在生成 JAR 文件时由系统自动创建的。

当开发了一个应用程序后，这个应用程序包含了很多类，如果需要把这个应用程序提供给别人使用，通常会将这些类文件打包成一个 JAR 文件，把这个 JAR 文件提供给别人使用。只要别人在系统的 CLASSPATH 环境变量中添加这个 JAR 文件，则 Java 虚拟机就可以自动在内存中解压这个 JAR 包，把这个 JAR 文件当成一个路径，在这个路径中查找所需要的类或包层次对应的路径结构。

使用 JAR 文件有以下好处。

- 安全。能够对 JAR 文件进行数字签名，只让能够识别数字签名的用户使用里面的东西。
- 加快下载速度。在网上使用 Applet 时，如果存在多个文件而不打包，为了能够把每个文件都下载到客户端，需要为每个文件单独建立一个 HTTP 连接，这是非常耗时的工作。将这些文件压缩成一个 JAR 包，只要建立一次 HTTP 连接就能够一次下载所有的文件。
- 压缩。使文件变小，JAR 的压缩机制和 ZIP 完全相同。
- 包封装。能够让 JAR 包里面的文件依赖于统一版本的类文件。
- 可移植性。JAR 包作为内嵌在 Java 平台内部处理的标准，能够在各种平台上直接使用。

把一个 JAR 文件添加到系统的 CLASSPATH 环境变量中后，Java 将会把这个 JAR 文件当成一个路径来处理。实际上 JAR 文件就是一个路径，JAR 文件通常使用 jar 命令压缩而成，当使用 jar 命令压缩生成 JAR 文件时，可以把一个或多个路径全部压缩成一个 JAR 文件。

例如，test 目录下包含如下目录结构和文件。

```
test
    |—a
        |—Test.class
        |—Test.java
    |—b
        |—Test.class
        |—Test.java
```

如果把上面 test 路径下的所有文件压缩成一个 JAR 文件，则 JAR 文件的内部目录结构为：

```
test.jar
    |—META-INF
        |—MANIFEST.MF
    |—a
        |—Test.class
        |—Test.java
    |—b
        |—Test.class
        |—Test.java
```

6.12.1 jar 命令详解

jar 是随 JDK 自动安装的，在 JDK 安装目录下的 bin 目录中（本书中就是 D:\Java\jdk1.8.0\bin 路径下），Windows 下文件名为 jar.exe，Linux 下文件名为 jar。运行它需要用到 JDK 安装目录下 lib 目录中的 tools.jar 文件，通常系统会自动加载 tools.jar，无须显式设置。

如果在命令行窗口运行不带任何参数的 jar 命令，系统将会提示 jar 命令的用法，提示信息如图 6.11 所示。

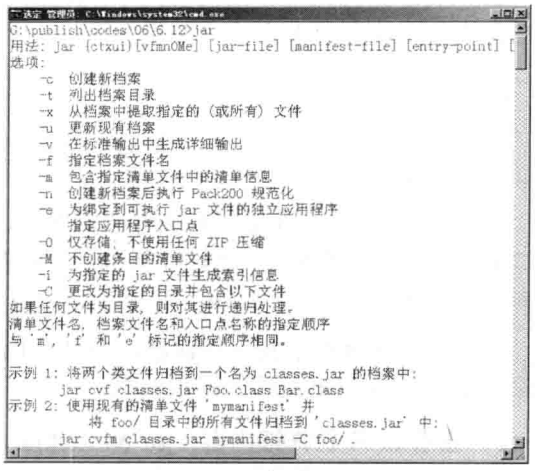


图 6.11 jar 命令用法详细信息

下面通过一些例子来说明 jar 命令的用法。

1. 创建 JAR 文件：jar cf test.jar test

该命令没有显示压缩过程，执行结果是将当前路径下的 test 路径下的全部内容生成一个 test.jar 文件。如果当前目录中已经存在 test.jar 文件，那么该文件将被覆盖。

2. 创建 JAR 文件，并显示压缩过程：jar cvf test.jar test

该命令的结果与第 1 个命令相同，但是由于 v 参数的作用，显示出了打包过程，如下所示：

```
已添加清单
正在添加: test/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: test/Test.class(输入 = 414) (输出 = 289)(压缩了 30%)
正在添加: test/Test.java(输入 = 409) (输出 = 305)(压缩了 25%)
```

3. 不使用清单文件：jar cvfM test.jar test

该命令的结果与第 2 个命令类似，其中 M 选项表明不生成清单文件。因此生成的 test.jar 中没有包含 META-INF/MANIFEST.MF 文件，打包过程的信息也略有差别。

```
正在添加: test/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: test/Test.class(输入 = 414) (输出 = 289)(压缩了 30%)
正在添加: test/Test.java(输入 = 409) (输出 = 305)(压缩了 25%)
```

4. 自定义清单文件内容：jar cvfm test.jar manifest.mf test

运行结果与第 2 个命令相似，显示信息也相同，其中 m 选项指定读取用户清单文件信息。因此在生成的 JAR 包中清单文件 META-INF/MANIFEST.MF 的内容有所不同，它会在原有清单文件基础上增加 MANIFEST.MF 文件的内容。

当开发者向 MANIFEST.MF 清单文件中增加自己的内容时，就需要借助于自己的清单文件了，清单文件只是一个普通的文本文件，使用记事本编辑即可。清单文件的内容由如下格式的多个 key-value 对组成。

```
key:<空格>value
```

清单文件的内容格式要求如下。

- 每行只能定义一个 key-value 对，每行的 key-value 对之前不能有空格，即 key-value 对必须顶格写。
- 每组 key-value 对之间以“:”（英文冒号后紧跟一个英文空格）分隔，少写了冒号或者空格都是错误的。
- 文件开头不能有空行。
- 文件必须以一个空行结束。

可以将上面文件保存在任意位置，以任意文件名存放。例如将上面文件保存在当前路径下，文件名为 a.txt。使用如下命令即可将清单文件中的 key-value 对提取到 META-INF/MANIFEST.MF 文件中。

```
jar cvfm test.jar a.txt test
```

5．查看 JAR 包内容：jar tf test.jar

在 test.jar 文件已经存在的前提下，使用此命令可以查看 test.jar 中的内容。例如，对使用第 2 个命令生成的 test.jar 执行此命令，结果如下：

```
META-INF/
META-INF/MANIFEST.MF
test/
test/Test.class
test/Test.java
```

当 JAR 包中的文件路径和文件非常多时，直接执行该命令将无法看到包的全部内容（因为命令行窗口能显示的行数有限），此时可利用重定向将显示结果保存到文件中。例如，采用如下命令：

```
jar tf test.jar > a.txt
```

执行上面命令看不到任何输出，但命令执行结束后，将在当前路径下生成一个 a.txt 文件，该文件中保存了 test.jar 包里文件的详细信息。

6．查看 JAR 包详细内容：jar tvf test.jar

该命令与第 5 个命令基本相似，但它更详细。所以除了显示第 5 个命令中显示的内容外，还包括包内文件的详细信息。例如：

```
  0 Wed Aug 24 15:29:42 CST 2011 META-INF/
 79 Wed Aug 24 15:29:42 CST 2011 META-INF/MANIFEST.MF
  0 Wed Aug 24 15:26:42 CST 2011 test/
414 Wed Aug 24 15:26:42 CST 2011 test/Test.class
409 Wed Aug 24 15:26:40 CST 2011 test/Test.java
```

7．解压缩：jar xf test.jar

将 test.jar 文件解压缩到当前目录下，不显示任何信息。假设将第 2 个命令生成的 test.jar 解压缩，将看到如下目录结构：

```
|—META-INF
    |—MANIFEST.MF
|—test
    |—Test.java
    |—Test.class
```

8．带提示信息解压缩：jar xvf test.jar

解压缩效果与第 7 个命令相同，但系统会显示解压过程的详细信息。例如：

```
已创建: META-INF/
已解压: META-INF/MANIFEST.MF
已创建: test/
已解压: test/Test.class
已解压: test/Test.java
```

9．更新 JAR 文件：jar uf test.jar Hello.class

更新 test.jar 中的 Hello.class 文件。如果 test.jar 中已有 Hello.class 文件，则使用新的 Hello.class 文件替换原来的 Hello.class 文件；如果 test.jar 中没有 Hello.class 文件，则把新的 Hello.class 文件添加到 test.jar 文件中。

10．更新时显示详细信息：jar uvf test.jar Hello.class

这个命令与第 9 个命令相同，也用于更新 test.jar 文件中的 Hello.class 文件，但它会显示详细的压缩信息。例如：

```
增加: Hello.class(读入= 51) (写出= 28)(压缩了 45%)
```

6.12.2　创建可执行的 JAR 包

当一个应用程序开发成功后，大致有如下三种发布方式。

- 使用平台相关的编译器将整个应用编译成平台相关的可执行性文件。这种方式常常需要第三方编译器的支持，而且编译生成的可执行性文件丧失了跨平台特性，甚至可能有一定的性能下降。
- 为应用编辑一个批处理文件。以 Windows 操作系统为例，批处理文件中只需要定义如下命令：

```
java package.MainClass
```

当用户单击上面的批处理文件时，系统将执行批处理文件的 java 命令，从而运行程序的主类。如果不想保留运行 Java 程序的命令行窗口，也可在批处理文件中定义如下命令：

```
start javaw package.MainClass
```

- 将一个应用程序制作成可执行的 JAR 包，通过 JAR 包来发布应用程序。

把应用程序压缩成 JAR 包来发布是比较典型的做法，如果开发者把整个应用制作成一个可执行的 JAR 包交给用户，那么用户使用起来就方便了。在 Windows 下安装 JRE 时，安装文件会将*.jar 文件映射成由 javaw.exe 打开。对于一个可执行的 JAR 包，用户只需要双击它就可以运行程序了，和阅读*.chm 文档一样方便（*.chm 文档默认是由 hh.exe 打开的）。下面介绍如何制作可执行的 JAR 包。

创建可执行的 JAR 包的关键在于：让 javaw 命令知道 JAR 包中哪个类是主类，javaw 命令可以通过运行该主类来运行程序。

jar 命令有一个-e 选项，该选项指定 JAR 包中作为程序入口的主类的类名。因此，制作一个可执行的 JAR 包只要增加-e 选项即可。例如如下命令：

```
jar cvfe test.jar test.Test test
```

上面命令把 test 目录下的所有文件都压缩到 test.jar 包中，并指定使用 test.Test 类（如果主类带包名，此处必须指定完整类名）作为程序的入口。

运行上面的 JAR 包有两种方式。

- 使用 java 命令，使用 java 运行时的语法是：java -jar test.jar。
- 使用 javaw 命令，使用 javaw 运行时的语法是：javaw test.jar。

当创建 JAR 包时，所有的类都必须放在与包结构对应的目录结构中，就像上面-e 选项指定的 Test 类，表明入口类为 Test。因此，必须在 JAR 包下包含 Test.class 文件。

6.12.3 关于 JAR 包的技巧

介绍 JAR 文件时就已经说过，JAR 文件实际上就是 ZIP 文件，所以可以使用一些常见的解压缩工具来解压缩 JAR 文件，如 Windows 下的 WinRAR、WinZip 等，以及 Linux 下的 unzip 等。使用 WinRAR 和 WinZip 等工具比使用 JAR 命令更加直观、方便；而使用 unzip 则可通过-d 选项来指定目标目录。

解压缩一个 JAR 文件时不能使用 jar 的-C 选项来指定解压的目标目录，因为-C 选项只在创建或者更新包时可用。如果需要将文件解压缩到指定目录下，则需要先将该 JAR 文件拷贝到目标目录下，再进行解压缩。如果使用 unzip，就无须这么麻烦了，只需要指定一个-d 选项即可。例如：

```
unzip test.jar -d dest/
```

使用 WinRAR 则更加方便，它不仅可以解压缩 JAR 文件，而且便于浏览 JAR 文件的任意目录。图 6.12 显示了使用 WinRAR 查看 test.jar 包的界面。

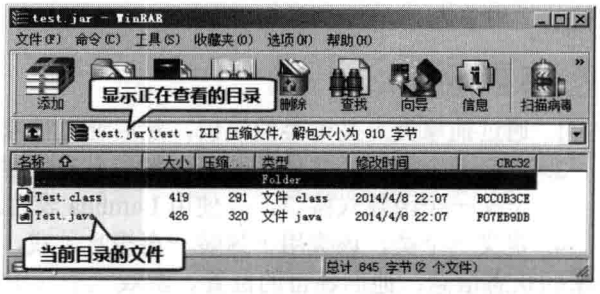


图 6.12 使用 WinRAR 查看 JAR 包

如果不喜欢 jar 命令的字符界面，也可以使用 WinRAR 工具来创建 JAR 包。因为 WinRAR 工具创建压缩文件时不会自动添加清单文件，所以需要手动添加清单文件，即需要手动建立 META-INF 路径，并在该路径下建立一个 MANIFEST.MF 文件，该文件中至少需要如下两行：

```
Manifest-Version: 1.0
Created-By: 1.8.0 (Oracle Corporation)
```

上面的 MANIFEST.MF 文件是一个格式敏感的文件，该文件的格式要求与前面自定义清单的格式要求完全一样。

接下来选中需要被压缩的文件、文件夹和 META-INF 文件夹，单击右键弹出右键菜单，单击“添加到压缩文件(A)...”菜单项，将看到如图 6.13 所示的压缩界面。

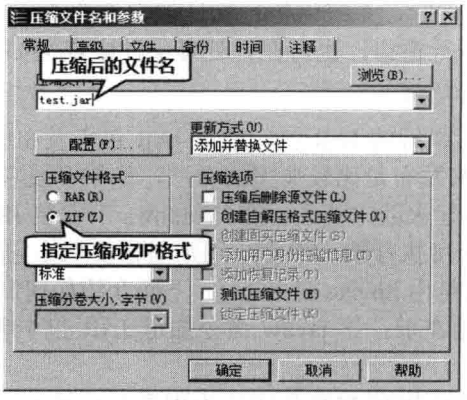


图 6.13　使用 WinRAR 压缩 JAR 包

按图 6.13 选择压缩成 ZIP 格式，并输入压缩后的文件名，然后单击“确定”按钮，即可生成一个 JAR 包，与使用 jar 命令生成的 JAR 包没有区别。

除此之外，Java 还可能生成两种压缩包：WAR 包和 EAR 包。其中 WAR 文件是 Web Archive File，它对应一个 Web 应用文档；而 EAR 文件就是 Enterprise Archive File，它对应于一个企业应用文档（通常由 Web 应用和 EJB 两个部分组成）。实际上，WAR 包和 EAR 包的压缩格式及压缩方式与 JAR 包完全一样，只是改变了文件后缀而已。

6.13　本章小结

本章主要介绍了 Java 面向对象的深入部分，包括 Java 里 8 个基本类型的包装类，以及系统直接输出一个对象时的处理方式，比较了对象相等时所用的==和 equals 方法的区别。本章详细介绍了使用 final 修饰符修饰变量、方法和类的用法，讲解了抽象类和接口的用法，并深入比较了接口和抽象类之间的联系和区别，以便读者能掌握接口和抽象类在用法上的区别。

本章还介绍了内部类的概念和用法，包括静态内部类、非静态内部类、局部内部类和匿名内部类等，并深入讲解了内部类的作用。枚举类是 Java 新提供的一个功能，这也是本章讲解的知识点，本章详细讲解了如何手动定义枚举类，以及通过 enum 来定义枚举类的各种相关知识。本章还重点介绍了 Java 8 新增的 Lambda 表达式，包括 Lambda 表达式的用法和本质，以及如何在 Lambda 表达式中使用方法引用、构造器引用。

本章最后介绍了对象的几种引用方式，以及系统垃圾回收的各种相关知识，还总结了 Java 所有修饰符的适用总表。

➤➤ 本章练习

1. 通过抽象类定义车类的模板，然后通过抽象的车类来派生拖拉机、卡车、小轿车。
2. 定义一个接口，并使用匿名内部类方式创建接口的实例。
3. 定义一个函数式接口，并使用 Lambda 表达式创建函数式接口的实例。
4. 定义一个类，该类用于封装一桌梭哈游戏，这个类应该包含桌上剩下的牌的信息，并包含 5 个玩家的状态信息：他们各自的位置、游戏状态（正在游戏或已放弃）、手上已有的牌等信息。如果有可能，这个类还应该实现发牌方法，这个方法需要控制从谁开始发牌，不要发牌给放弃的人，并修改桌上剩下的牌。

CHAPTER

7

第7章 Java基础类库

本章要点

- Java 程序的参数
- 程序运行过程中接收用户输入
- System 类相关用法
- Runtime 类的相关用法
- Object 与 Objects 类
- 使用 String、StringBuffer、StringBuilder 类
- 使用 Math 类进行数学计算
- 使用 BigDecimal 保存精确浮点数
- 使用 Random 类生成各种伪随机数
- Date、Calendar 的用法及之间的联系
- Java 8 新增的日期、时间 API 的功能和用法
- 创建正则表达式
- 通过 Pattern 和 Matcher 使用正则表达式
- 通过 String 类使用正则表达式
- 程序国际化的思路
- 程序国际化
- 使用 NumberFormat 格式化数字
- 使用 DateTimeFormatter 解析日期、时间字符串
- 使用 DateTimeFormatter 格式化日期、时间
- 使用 DateFormat、SimpleDateFormat 格式化日期

Oracle 为 Java 提供了丰富的基础类库，Java 8 提供了 4000 多个基础类（包括下一章将要介绍的集合框架），通过这些基础类库可以提高开发效率，降低开发难度。对于合格的 Java 程序员而言，至少要熟悉 Java SE 中 70%以上的类（当然本书并不是让读者去背诵 Java API 文档），但在反复查阅 API 文档的过程中，会自动记住大部分类的功能、方法，因此程序员一定要多练，多敲代码。

Java 提供了 String、StringBuffer 和 StringBuilder 来处理字符串，它们之间存在少许差别，本章会详细介绍它们之间的差别，以及如何选择合适的字符串类。Java 还提供了 Date 和 Calendar 来处理日期、时间，其中 Date 是一个已经过时的 API，通常推荐使用 Calendar 来处理日期、时间。

正则表达式是一个强大的文本处理工具，通过正则表达式可以对文本内容进行查找、替换、分割等操作。从 JDK 1.4 以后，Java 也增加了对正则表达式的支持，包括新增的 Pattern 和 Matcher 两个类，并改写了 String 类，让 String 类增加了正则表达式支持，增加了正则表达式功能后的 String 类更加强大。

Java 还提供了非常简单的国际化支持，Java 使用 Locale 对象封装一个国家、语言环境，再使用 ResourceBundle 根据 Locale 加载语言资源包，当 ResourceBundle 加载了指定 Locale 对应的语言资源文件后，ResourceBundle 对象就可调用 getString()方法来取出指定 key 所对应的消息字符串。

7.1 与用户互动

如果一个程序总是按既定的流程运行，无须处理用户动作，这个程序总是比较简单的。实际上，绝大部分程序都需要处理用户动作，包括接收用户的键盘输入、鼠标动作等。因为现在还未涉及图形用户接口（GUI）编程，故本节主要介绍程序如何获得用户的键盘输入。

7.1.1 运行 Java 程序的参数

回忆 Java 程序的入口——main()方法的方法签名：

```
// Java 程序入口：main()方法
public static void main(String[] args){....}
```

下面详细讲解 main()方法为什么采用这个方法签名。

- public 修饰符：Java 类由 JVM 调用，为了让 JVM 可以自由调用这个 main()方法，所以使用 public 修饰符把这个方法暴露出来。
- static 修饰符：JVM 调用这个主方法时，不会先创建该主类的对象，然后通过对象来调用该主方法。JVM 直接通过该类来调用主方法，因此使用 static 修饰该主方法。
- void 返回值：因为主方法被 JVM 调用，该方法的返回值将返回给 JVM，这没有任何意义，因此 main()方法没有返回值。

上面方法中还包括一个字符串数组形参，根据方法调用的规则：谁调用方法，谁负责为形参赋值。也就是说，main()方法由 JVM 调用，即 args 形参应该由 JVM 负责赋值。但 JVM 怎么知道如何为 args 数组赋值呢？先看下面程序。

程序清单：codes\07\7.1\ArgsTest.java

```
public class ArgsTest
{
    public static void main(String[] args)
    {
        // 输出 args 数组的长度
        System.out.println(args.length);
        // 遍历 args 数组的每个元素
        for (String arg : args)
        {
            System.out.println(arg);
        }
    }
}
```

上面程序几乎是最简单的“HelloWorld”程序，只是这个程序增加了输出 args 数组的长度，遍历 args 数组元素的代码。使用 java ArgsTest 命令运行上面程序，看到程序仅仅输出一个 0，这表明 args 数组是

一个长度为 0 的数组——这是合理的。因为计算机是没有思考能力的，它只能忠实地执行用户交给它的任务，既然程序没有给 args 数组设定参数值，那么 JVM 就不知道 args 数组的元素，所以 JVM 将 args 数组设置成一个长度为 0 的数组。

改为如下命令来运行上面程序：

```
java ArgsTest Java Spring
```

将看到如图 7.1 所示的运行结果。

从图 7.1 中可以看出，如果运行 Java 程序时在类名后紧跟一个或多个字符串（多个字符串之间以空格隔开），JVM 就会把这些字符串依次赋给 args 数组元素。运行 Java 程序时的参数与 args 数组之间的对应关系如图 7.2 所示。

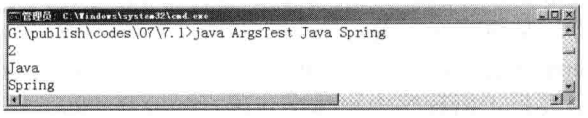


图 7.1　为 main()方法的形参数组赋值

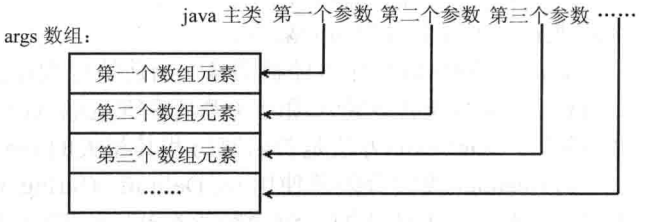


图 7.2　运行 Java 程序时参数与 args 数组的关系

如果某参数本身包含了空格，则应该将该参数用双引号（""）括起来，否则 JVM 会把这个空格当成参数分隔符，而不是当成参数本身。例如，采用如下命令来运行上面程序：

```
java ArgsTest "Java Spring"
```

看到 args 数组的长度是 1，只有一个数组元素，其值是 Java Spring。

7.1.2　使用 Scanner 获取键盘输入

运行 Java 程序时传入参数只能在程序开始运行之前就设定几个固定的参数。对于更复杂的情形，程序需要在运行过程中取得输入，例如，前面介绍的五子棋游戏、梭哈游戏都需要在程序运行过程中获得用户的键盘输入。

使用 Scanner 类可以很方便地获取用户的键盘输入，Scanner 是一个基于正则表达式的文本扫描器，它可以从文件、输入流、字符串中解析出基本类型值和字符串值。Scanner 类提供了多个构造器，不同的构造器可以接收文件、输入流、字符串作为数据源，用于从文件、输入流、字符串中解析数据。

Scanner 主要提供了两个方法来扫描输入。

- hasNextXxx()：是否还有下一个输入项，其中 Xxx 可以是 Int、Long 等代表基本数据类型的字符串。如果只是判断是否包含下一个字符串，则直接使用 hasNext()。
- nextXxx()：获取下一个输入项。Xxx 的含义与前一个方法中的 Xxx 相同。

在默认情况下，Scanner 使用空白（包括空格、Tab 空白、回车）作为多个输入项之间的分隔符。下面程序使用 Scanner 来获得用户的键盘输入。

程序清单：codes\07\7.1\ScannerKeyBoardTest.java

```
public class ScannerKeyBoardTest
{
    public static void main(String[] args)
    {
        // System.in 代表标准输入，就是键盘输入
        Scanner sc = new Scanner(System.in);
        // 增加下面一行将只把回车作为分隔符
        // sc.useDelimiter("\n");
        // 判断是否还有下一个输入项
        while(sc.hasNext())
        {
            // 输出输入项
            System.out.println("键盘输入的内容是："
```

```
                + sc.next());
        }
    }
}
```

运行上面程序，程序通过 Scanner 不断从键盘读取键盘输入，每次读到键盘输入后，直接将输入内容打印在控制台。上面程序的运行效果如图 7.3 所示。

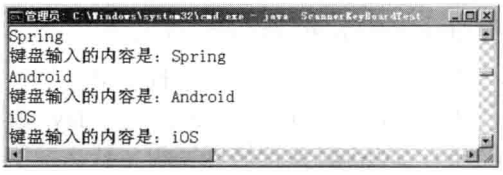


图 7.3　使用 Scanner 获取键盘输入

如果希望改变 Scanner 的分隔符（不使用空白作为分隔符），例如，程序需要每次读取一行，不管这一行中是否包含空格，Scanner 都把它当成一个输入项。在这种需求下，可以把 Scanner 的分隔符设置为回车符，不再使用默认的空白作为分隔符。

Scanner 的读取操作可能被阻塞（当前执行顺序流暂停）来等待信息的输入。如果输入源没有结束，Scanner 又读不到更多输入项时（尤其在键盘输入时比较常见），Scanner 的 hasNext()和 next()方法都有可能阻塞，hasNext()方法是否阻塞与和其相关的 next()方法是否阻塞无关。

为 Scanner 设置分隔符使用 useDelimiter(String pattern)方法即可，该方法的参数应该是一个正则表达式，关于正则表达式的介绍请参考本章后面的内容。只要把上面程序中粗体字代码行的注释去掉，该程序就会把键盘的每行输入当成一个输入项，不会以空格、Tab 空白等作为分隔符。

事实上，Scanner 提供了两个简单的方法来逐行读取。

- boolean hasNextLine()：返回输入源中是否还有下一行。
- String nextLine()：返回输入源中下一行的字符串。

Scanner 不仅可以获取字符串输入项，也可以获取任何基本类型的输入项，如下程序所示。

程序清单：codes\07\7.1\ScannerLongTest.java

```
public class ScannerLongTest
{
    public static void main(String[] args)
    {
        // System.in 代表标准输入，就是键盘输入
        Scanner sc = new Scanner(System.in);
        // 判断是否还有下一个 long 型整数
        while(sc.hasNextLong())
        {
            // 输出输入项
            System.out.println("键盘输入的内容是："
                + sc.nextLong());
        }
    }
}
```

注意上面程序中粗体字代码部分，正如通过 hasNextLong()和 nextLong()两个方法，Scanner 可以直接从输入流中获得 long 型整数输入项。与此类似的是，如果需要获取其他基本类型的输入项，则可以使用相应的方法。

注意：

上面程序不如 ScannerKeyBoardTest 程序适应性强，因为 ScannerLongTest 程序要求键盘输入必须是整数，否则程序就会退出。



Scanner 不仅能读取用户的键盘输入，还可以读取文件输入。只要在创建 Scanner 对象时传入一个 File 对象作为参数，就可以让 Scanner 读取该文件的内容。例如如下程序。

程序清单：codes\07\7.1\ScannerFileTest.java

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

```
        throws Exception
    {
        // 将一个 File 对象作为 Scanner 的构造器参数，Scanner 读取文件内容
        Scanner sc = new Scanner(new File("ScannerFileTest.java"));
        System.out.println("ScannerFileTest.java 文件内容如下：");
        // 判断是否还有下一行
        while(sc.hasNextLine())
        {
            // 输出文件中的下一行
            System.out.println(sc.nextLine());
        }
    }
}
```

上面程序创建 Scanner 对象时传入一个 File 对象作为参数（如粗体字代码所示），这表明该程序将会读取 ScannerFileTest.java 文件中的内容。上面程序使用了 hasNextLine()和 nextLine()两个方法来读取文件内容（如粗体字代码所示），这表明该程序将逐行读取 ScannerFileTest.java 文件的内容。

因为上面程序涉及文件输入，可能引发文件 IO 相关异常，故主程序声明 throws Exception 表明 main 方法不处理任何异常。关于异常处理请参考第 10 章内容。

7.2 系统相关

Java 程序在不同操作系统上运行时，可能需要取得平台相关的属性，或者调用平台命令来完成特定功能。Java 提供了 System 类和 Runtime 类来与程序的运行平台进行交互。

7.2.1 System 类

System 类代表当前 Java 程序的运行平台，程序不能创建 System 类的对象，System 类提供了一些类变量和类方法，允许直接通过 System 类来调用这些类变量和类方法。

System 类提供了代表标准输入、标准输出和错误输出的类变量，并提供了一些静态方法用于访问环境变量、系统属性的方法，还提供了加载文件和动态链接库的方法。下面程序通过 System 类来访问操作的环境变量和系统属性。

注意：

加载文件和动态链接库主要对 native 方法有用，对于一些特殊的功能（如访问操作系统底层硬件设备等）Java 程序无法实现，必须借助 C 语言来完成，此时需要使用 C 语言为 Java 方法提供实现。其实现步骤如下：

① Java 程序中声明 native 修饰的方法，类似于 abstract 方法，只有方法签名，没有实现。编译该 Java 程序，生成一个 class 文件。

② 用 javah 编译第 1 步生成的 class 文件，将产生一个.h 文件。

③ 写一个.cpp 文件实现 native 方法，这一步需要包含第 2 步产生的.h 文件（这个.h 文件中又包含了 JDK 带的 jni.h 文件）。

④ 将第 3 步的.cpp 文件编译成动态链接库文件。

⑤ 在 Java 中用 System 类的 loadLibrary..()方法或 Runtime 类的 loadLibrary()方法加载第 4 步产生的动态链接库文件，Java 程序中就可以调用这个 native 方法了。



程序清单：codes\07\7.2\SystemTest.java

```
public class SystemTest
{
    public static void main(String[] args) throws Exception
    {
        // 获取系统所有的环境变量
        Map<String,String> env = System.getenv();
        for (String name : env.keySet())
        {
```

```
            System.out.println(name + " ---> " + env.get(name));
        }
        // 获取指定环境变量的值
        System.out.println(System.getenv("JAVA_HOME"));
        // 获取所有的系统属性
        Properties props = System.getProperties();
        // 将所有的系统属性保存到 props.txt 文件中
        props.store(new FileOutputStream("props.txt")
            , "System Properties");
        // 输出特定的系统属性
        System.out.println(System.getProperty("os.name"));
    }
}
```

上面程序通过调用 System 类的 getenv()、getProperties()、getProperty()等方法来访问程序所在平台的环境变量和系统属性，程序运行的结果会输出操作系统所有的环境变量值，并输出 JAVA_HOME 环境变量，以及 os.name 系统属性的值，运行结果如图 7.4 所示。

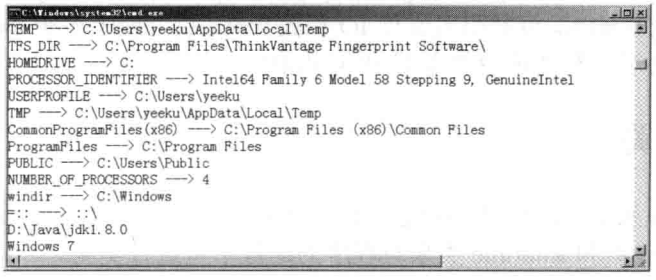


图 7.4　访问环境变量和系统属性的效果

该程序运行结束后还会在当前路径下生成一个 props.txt 文件，该文件中记录了当前平台的所有系统属性。



提示：

System 类提供了通知系统进行垃圾回收的 gc()方法，以及通知系统进行资源清理的 runFinalization()方法。关于这两个方法的用法请参考本书 6.10 节的内容。

System 类还有两个获取系统当前时间的方法：currentTimeMillis()和 nanoTime()，它们都返回一个 long 型整数。实际上它们都返回当前时间与 UTC 1970 年 1 月 1 日午夜的时间差，前者以毫秒作为单位，后者以纳秒作为单位。必须指出的是，这两个方法返回的时间粒度取决于底层操作系统，可能所在的操作系统根本不支持以毫秒、纳秒作为计时单位。例如，许多操作系统以几十毫秒为单位测量时间，currentTimeMillis()方法不可能返回精确的毫秒数；而 nanoTime()方法很少用，因为大部分操作系统都不支持使用纳秒作为计时单位。

除此之外，System 类的 in、out 和 err 分别代表系统的标准输入（通常是键盘）、标准输出（通常是显示器）和错误输出流，并提供了 setIn()、setOut()和 setErr()方法来改变系统的标准输入、标准输出和标准错误输出流。



提示：

关于如何改变系统的标准输入、输出的方法，可以参考本书第 15 章的内容。

System 类还提供了一个 identityHashCode(Object x)方法，该方法返回指定对象的精确 hashCode 值，也就是根据该对象的地址计算得到的 hashCode 值。当某个类的 hashCode()方法被重写后，该类实例的 hashCode()方法就不能唯一地标识该对象；但通过 identityHashCode()方法返回的 hashCode 值，依然是根据该对象的地址计算得到的 hashCode 值。所以，如果两个对象的 identityHashCode 值相同，则两个对象绝对是同一个对象。如下程序所示。

程序清单：codes\07\7.2\IdentityHashCodeTest.java

```
public class IdentityHashCodeTest
{
```

```
    public static void main(String[] args)
    {
        // 下面程序中 s1 和 s2 是两个不同的对象
        String s1 = new String("Hello");
        String s2 = new String("Hello");
        // String 重写了 hashCode()方法——改为根据字符序列计算 hashCode 值
        // 因为 s1 和 s2 的字符序列相同，所以它们的 hashCode()方法返回值相同
        System.out.println(s1.hashCode()
            + "----" + s2.hashCode());
        // s1 和 s2 是不同的字符串对象，所以它们的 identityHashCode 值不同
        System.out.println(System.identityHashCode(s1)
            + "----" + System.identityHashCode(s2));
        String s3 = "Java";
        String s4 = "Java";
        // s3 和 s4 是相同的字符串对象，所以它们的 identityHashCode 值相同
        System.out.println(System.identityHashCode(s3)
            + "----" + System.identityHashCode(s4));
    }
}
```

通过 identityHashCode(Object x)方法可以获得对象的 identityHashCode 值，这个特殊的 identityHashCode 值可以唯一地标识该对象。因为 identityHashCode 值是根据对象的地址计算得到的，所以任何两个对象的 identityHashCode 值总是不相等。

7.2.2 Runtime 类

Runtime 类代表 Java 程序的运行时环境，每个 Java 程序都有一个与之对应的 Runtime 实例，应用程序通过该对象与其运行时环境相连。应用程序不能创建自己的 Runtime 实例，但可以通过 getRuntime()方法获取与之关联的 Runtime 对象。

与 System 类似的是，Runtime 类也提供了 gc()方法和 runFinalization()方法来通知系统进行垃圾回收、清理系统资源，并提供了 load(String filename)和 loadLibrary(String libname)方法来加载文件和动态链接库。

Runtime 类代表 Java 程序的运行时环境，可以访问 JVM 的相关信息，如处理器数量、内存信息等。如下程序所示。

程序清单：codes\07\7.2\RuntimeTest.java

```
public class RuntimeTest
{
    public static void main(String[] args)
    {
        // 获取 Java 程序关联的运行时对象
        Runtime rt = Runtime.getRuntime();
        System.out.println("处理器数量："
            + rt.availableProcessors());
        System.out.println("空闲内存数："
            + rt.freeMemory());
        System.out.println("总内存数："
            + rt.totalMemory());
        System.out.println("可用最大内存数："
            + rt.maxMemory());
    }
}
```

上面程序中粗体字代码就是 Runtime 类提供的访问 JVM 相关信息的方法。除此之外，Runtime 类还有一个功能——它可以直接单独启动一个进程来运行操作系统的命令，如下程序所示。

程序清单：codes\07\7.2\ExecTest.java

```
public class ExecTest
{
    public static void main(String[] args)
        throws Exception
    {
        Runtime rt = Runtime.getRuntime();
        // 运行记事本程序
```

```
            rt.exec("notepad.exe");
        }
    }
```

上面程序中粗体字代码将启动 Windows 系统里的“记事本”程序。Runtime 提供了一系列 exec()方法来运行操作系统命令，关于它们之间的细微差别，请读者自行查阅 API 文档。

7.3 常用类

本节将介绍 Java 提供的一些常用类，如 String、Math、BigDecimal 等的用法。

7.3.1 Object 类

Object 类是所有类、数组、枚举类的父类，也就是说，Java 允许把任何类型的对象赋给 Object 类型的变量。当定义一个类时没有使用 extends 关键字为它显式指定父类，则该类默认继承 Object 父类。

因为所有的 Java 类都是 Object 类的子类，所以任何 Java 对象都可以调用 Object 类的方法。Object 类提供了如下几个常用方法。

- boolean equals(Object obj)：判断指定对象与该对象是否相等。此处相等的标准是，两个对象是同一个对象，因此该 equals()方法通常没有太大的实用价值。
- protected void finalize()：当系统中没有引用变量引用到该对象时，垃圾回收器调用此方法来清理该对象的资源。
- Class<?> getClass()：返回该对象的运行时类，该方法在本书第 18 章还有更详细的介绍。
- int hashCode()：返回该对象的 hashCode 值。在默认情况下，Object 类的 hashCode()方法根据该对象的地址来计算（即与 System.identityHashCode(Object x)方法的计算结果相同）。但很多类都重写了 Object 类的 hashCode()方法，不再根据地址来计算其 hashCode()方法值。
- String toString()：返回该对象的字符串表示，当程序使用 System.out.println()方法输出一个对象，或者把某个对象和字符串进行连接运算时，系统会自动调用该对象的 toString()方法返回该对象的字符串表示。Object 类的 toString()方法返回“运行时类名@十六进制 hashCode 值”格式的字符串，但很多类都重写了 Object 类的 toString()方法，用于返回可以表述该对象信息的字符串。

除此之外，Object 类还提供了 wait()、notify()、notifyAll()几个方法，通过这几个方法可以控制线程的暂停和运行。本书将在第 16 章介绍这几个方法的详细用法。

Java 还提供了一个 protected 修饰的 clone()方法，该方法用于帮助其他对象来实现“自我克隆”，所谓“自我克隆”就是得到一个当前对象的副本，而且二者之间完全隔离。由于 Object 类提供的 clone()方法使用了 protected 修饰，因此该方法只能被子类重写或调用。

自定义类实现“克隆”的步骤如下。

① 自定义类实现 Cloneable 接口。这是一个标记性的接口，实现该接口的对象可以实现“自我克隆”，接口里没有定义任何方法。

② 自定义类实现自己的 clone()方法。

③ 实现 clone()方法时通过 super.clone()；调用 Object 实现的 clone()方法来得到该对象的副本，并返回该副本。如下程序示范了如何实现“自我克隆”。

程序清单：codes\07\7.3\CloneTest.java

```
class Address
{
    String detail;
    public Address(String detail)
    {
        this.detail = detail;
    }
}
// 实现 Cloneable 接口
class User implements Cloneable
{
    int age;
    Address address;
```

```
    public User(int age)
    {
        this.age = age;
        address = new Address("广州天河");
    }
    // 通过调用 super.clone()来实现 clone()方法
    public User clone()
        throws CloneNotSupportedException
    {
        return (User)super.clone();
    }
}
public class CloneTest
{
    public static void main(String[] args)
        throws CloneNotSupportedException
    {
        User u1 = new User(29);
        // clone 得到 u1 对象的副本
        User u2 = u1.clone();
        // 判断 u1、u2 是否相同
        System.out.println(u1 == u2);    // ①
        // 判断 u1、u2 的 address 是否相同
        System.out.println(u1.address == u2.address);    // ②
    }
}
```

上面程序让 User 类实现了 Cloneable 接口，而且实现了 clone()方法，因此 User 对象就可实现“自我克隆”——克隆出来的对象只是原有对象的副本。程序在①号粗体字代码处判断原有的 User 对象与克隆出来的 User 对象是否相同，程序返回 false。

Object类提供的Clone机制只对对象里各实例变量进行“简单复制”，如果实例变量的类型是引用类型，Object 的 Clone 机制也只是简单地复制这个引用变量，这样原有对象的引用类型的实例变量与克隆对象的引用类型的实例变量依然指向内存中的同一个实例，所以上面程序在②号代码处输出 true。上面程序“克隆”出来的 u1、u2 所指向的对象在内存中的存储示意图如图 7.5 所示。

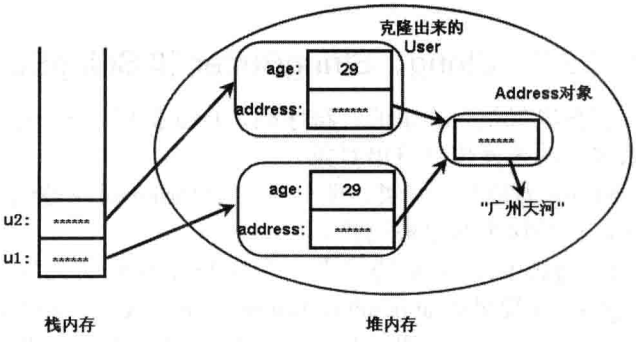


图 7.5 Object 类提供的克隆机制

Object类提供的clone()方法不仅能简单地处理“复制”对象的问题，而且这种“自我克隆”机制十分高效。比如 clone 一个包含 100 个元素的 int[]数组，用系统默认的 clone 方法比静态 copy 方法快近 2 倍。

需要指出的是，Object 类的 clone()方法虽然简单、易用，但它只是一种“浅克隆”——它只克隆该对象的所有成员变量值，不会对引用类型的成员变量值所引用的对象进行克隆。如果开发者需要对对象进行“深克隆”，则需要开发者自己进行“递归”克隆，保证所有引用类型的成员变量值所引用的对象都被复制了。

7.3.2 Java 7 新增的 Objects 类

Java 7 新增了一个 Objects 工具类，它提供了一些工具方法来操作对象，这些工具方法大多是“空指针”安全的。比如你不能确定一个引用变量是否为 null，如果贸然地调用该变量的 toString()方法，则可能引发 NullPointerExcetpion 异常；但如果使用 Objects 类提供的 toString(Object o)方法，就不会引发空指针异常，当 o 为 null 时，程序将返回一个"null"字符串。



提示：

Java 为工具类的命名习惯是添加一个字母 s，比如操作数组的工具类是 Arrays，操作集合的工具类是 Collections。

如下程序示范了 Objects 工具类的用法。

程序清单：codes\07\7.3\ObjectsTest.java

```
public class ObjectsTest
{
    // 定义一个 obj 变量，它的默认值是 null
    static ObjectsTest obj;
    public static void main(String[] args)
    {
        // 输出一个 null 对象的 hashCode 值，输出 0
        System.out.println(Objects.hashCode(obj));
        // 输出一个 null 对象的 toString，输出 null
        System.out.println(Objects.toString(obj));
        // 要求 obj 不能为 null，如果 obj 为 null 则引发异常
        System.out.println(Objects.requireNonNull(obj
            , "obj 参数不能是 null! "));
    }
}
```

上面程序还示范了 Objects 提供的 requireNonNull()方法，当传入的参数不为 null 时，该方法返回参数本身；否则将会引发 NullPointerException 异常。该方法主要用来对方法形参进行输入校验，例如如下代码：

```
public Foo(Bar bar)
{
    // 校验 bar 参数，如果 bar 参数为 null 将引发异常；否则 this.bar 被赋值为 bar 参数
    this.bar = Objects.requireNonNull(bar);
}
```

7.3.3 String、StringBuffer 和 StringBuilder 类

字符串就是一连串的字符序列，Java 提供了 String 和 StringBuffer 两个类来封装字符串，并提供了一系列方法来操作字符串对象。

String 类是不可变类，即一旦一个 String 对象被创建以后，包含在这个对象中的字符序列是不可改变的，直至这个对象被销毁。

StringBuffer 对象则代表一个字符序列可变的字符串，当一个 StringBuffer 被创建以后，通过 StringBuffer 提供的 append()、insert()、reverse()、setCharAt()、setLength()等方法可以改变这个字符串对象的字符序列。一旦通过 StringBuffer 生成了最终想要的字符串，就可以调用它的 toString()方法将其转换为一个 String 对象。

JDK 1.5 又新增了一个 StringBuilder 类，它也代表字符串对象。实际上，StringBuilder 和 StringBuffer 基本相似，两个类的构造器和方法也基本相同。不同的是，StringBuffer 是线程安全的，而 StringBuilder 则没有实现线程安全功能，所以性能略高。因此在通常情况下，如果需要创建一个内容可变的字符串对象，则应该优先考虑使用 StringBuilder 类。



提示：

String、StringBuilder、StringBuffer 都实现了 CharSequence 接口，因此 CharSequence 可认为是一个字符串的协议接口。

String 类提供了大量构造器来创建 String 对象，其中如下几个有特殊用途。

- String()：创建一个包含 0 个字符串序列的 String 对象（并不是返回 null）。
- String(byte[] bytes, Charset charset)：使用指定的字符集将指定的 byte[]数组解码成一个新的 String 对象。
- String(byte[] bytes, int offset, int length)：使用平台的默认字符集将指定的 byte[]数组从 offset 开

始、长度为 length 的子数组解码成一个新的 String 对象。

- String(byte[] bytes, int offset, int length, String charsetName)：使用指定的字符集将指定的 byte[]数组从 offset 开始、长度为 length 的子数组解码成一个新的 String 对象。
- String(byte[] bytes, String charsetName)：使用指定的字符集将指定的 byte[]数组解码成一个新的 String 对象。
- String(char[] value, int offset, int count)：将指定的字符数组从 offset 开始、长度为 count 的字符元素连缀成字符串。
- String(String original)：根据字符串直接量来创建一个 String 对象。也就是说，新创建的 String 对象是该参数字符串的副本。
- String(StringBuffer buffer)：根据 StringBuffer 对象来创建对应的 String 对象。
- String(StringBuilder builder)：根据 StringBuilder 对象来创建对应的 String 对象。

String 类也提供了大量方法来操作字符串对象，下面详细介绍这些常用方法。

- char charAt(int index)：获取字符串中指定位置的字符。其中，参数 index 指的是字符串的序数，字符串的序数从 0 开始到 length()－1。如下代码所示。

```
String s = new String("fkit.org");
System.out.println("s.charAt(5): " + s.charAt(5) );
```

结果为：

```
s.charAt(5): o
```

- int compareTo(String anotherString)：比较两个字符串的大小。如果两个字符串的字符序列相等，则返回 0；不相等时，从两个字符串第 0 个字符开始比较，返回第一个不相等的字符差。另一种情况，较长字符串的前面部分恰巧是较短的字符串，则返回它们的长度差。

```
String s1 = new String("abcdefghijklmn");
String s2 = new String("abcdefghij");
String s3 = new String("abcdefghijalmn");
System.out.println("s1.compareTo(s2): " + s1.compareTo(s2) );// 返回长度差
System.out.println("s1.compareTo(s3): " + s1.compareTo(s3) );// 返回'k'-'a'的差
```

结果为：

```
s1.compareTo(s2): 4
s1.compareTo(s3): 10
```

- String concat(String str)：将该 String 对象与 str 连接在一起。与 Java 提供的字符串连接运算符“+”的功能相同。
- boolean contentEquals(StringBuffer sb)：将该 String 对象与 StringBuffer 对象 sb 进行比较，当它们包含的字符序列相同时返回 true。
- static String copyValueOf(char[] data)：将字符数组连缀成字符串，与 String(char[] content)构造器的功能相同。
- static String copyValueOf(char[] data, int offset, int count)：将 char 数组的子数组中的元素连缀成字符串，与 String(char[] value, int offset, int count)构造器的功能相同。
- boolean endsWith(String suffix)：返回该 String 对象是否以 suffix 结尾。

```
String s1 = "fkit.org"; String s2 = ".org";
System.out.println("s1.endsWith(s2): " + s1.endsWith(s2) );
```

结果为：

```
s1.endsWith(s2): true
```

- boolean equals(Object anObject)：将该字符串与指定对象比较，如果二者包含的字符序列相等，则返回 true；否则返回 false。
- boolean equalsIgnoreCase(String str)：与前一个方法基本相似，只是忽略字符的大小写。
- byte[] getBytes()：将该 String 对象转换成 byte 数组。
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)：该方法将字符串中从 srcBegin 开始，

到 srcEnd 结束的字符复制到 dst 字符数组中，其中 dstBegin 为目标字符数组的起始复制位置。

```
char[] s1 = {'I',' ','l','o','v','e',' ','j','a','v','a'}; // s1=I love java
String s2 = new String("ejb");
s2.getChars(0,3,s1,7);    // s1=I love ejba
System.out.println( s1 );
```

结果为：

```
I love ejba
```

- int indexOf(int ch)：找出 ch 字符在该字符串中第一次出现的位置。
- int indexOf(int ch, int fromIndex)：找出 ch 字符在该字符串中从 fromIndex 开始后第一次出现的位置。
- int indexOf(String str)：找出 str 子字符串在该字符串中第一次出现的位置。
- int indexOf(String str, int fromIndex)：找出 str 子字符串在该字符串中从 fromIndex 开始后第一次出现的位置。

```
String s = "www.fkit.org"; String ss = "it";
System.out.println("s.indexOf('r'): " + s.indexOf('r') );
System.out.println("s.indexOf('r',2): " + s.indexOf('r',2) );
System.out.println("s.indexOf(ss): " + s.indexOf(ss));
```

结果为：

```
s.indexOf('r'): 10
s.indexOf('r',2): 10
s.indexOf(ss): 6
```

- int lastIndexOf(int ch)：找出 ch 字符在该字符串中最后一次出现的位置。
- int lastIndexOf(int ch, int fromIndex)：找出 ch 字符在该字符串中从 fromIndex 开始后最后一次出现的位置。
- int lastIndexOf(String str)：找出 str 子字符串在该字符串中最后一次出现的位置。
- int lastIndexOf(String str, int fromIndex)：找出 str 子字符串在该字符串中从 fromIndex 开始后最后一次出现的位置。
- int length()：返回当前字符串长度。
- String replace(char oldChar, char newChar)：将字符串中的第一个 oldChar 替换成 newChar。
- boolean startsWith(String prefix)：该 String 对象是否以 prefix 开始。
- boolean startsWith(String prefix, int toffset)：该 String 对象从 toffset 位置算起，是否以 prefix 开始。

```
String s = "www.fkit.org"; String ss = "www"; String sss = "fkit";
System.out.println("s.startsWith(ss): " + s.startsWith(ss));
System.out.println("s.startsWith(sss,4): " + s.startsWith(sss,4));
```

结果为：

```
s.startsWith(ss): true
s.startsWith(sss,4): true
```

- String substring(int beginIndex)：获取从 beginIndex 位置开始到结束的子字符串。
- String substring(int beginIndex, int endIndex)：获取从 beginIndex 位置开始到 endIndex 位置的子字符串。
- char[] toCharArray()：将该 String 对象转换成 char 数组。
- String toLowerCase()：将字符串转换成小写。
- String toUpperCase()：将字符串转换成大写。

```
String s = "fkjava.org";
System.out.println("s.toUpperCase(): " + s.toUpperCase());
System.out.println("s.toLowerCase(): " + s.toLowerCase());
```

结果为：

```
s.toUpperCase(): FKJAVA.ORG
s.toLowerCase(): fkjava.org
```

- static String valueOf(X x)：一系列用于将基本类型值转换为 String 对象的方法。

本书详细列出String类的各种方法时，有读者可能会觉得烦琐，因为这些方法都可以从API文档中找到，所以后面介绍各常用类时不会再列出每个类里所有方法的详细用法了，读者应该自行查阅API文档来掌握各方法的用法。

String类是不可变的，String的实例一旦生成就不会再改变了，例如如下代码。

```
String str1 = "java";
str1 = str1 + "struts";
str1 = str1 + "spring"
```

上面程序除了使用了3个字符串直接量之外，还会额外生成2个字符串直接量——"java"和"struts"连接生成的"javastruts"，接着"javastruts"与"spring"连接生成的"javastrutsspring"，程序中的str1依次指向3个不同的字符串对象。

因为String是不可变的，所以会额外产生很多临时变量，使用StringBuffer或StringBuilder就可以避免这个问题。

StringBuilder提供了一系列插入、追加、改变该字符串里包含的字符序列的方法。而StringBuffer与其用法完全相同，只是StringBuffer是线程安全的。

StringBuilder、StringBuffer有两个属性：length和capacity，其中length属性表示其包含的字符序列的长度。与String对象的length不同的是，StringBuilder、StringBuffer的length是可以改变的，可以通过length()、setLength(int len)方法来访问和修改其字符序列的长度。capacity属性表示StringBuilder的容量，capacity通常比length大，程序通常无须关心capacity属性。如下程序示范了StringBuilder类的用法。

程序清单：codes\07\7.3\StringBuilderTest.java

```
public class StringBuilderTest
{
    public static void main(String[] args)
    {
        StringBuilder sb = new StringBuilder();
        // 追加字符串
        sb.append("java");// sb = "java"
        // 插入
        sb.insert(0 , "hello "); // sb="hello java"
        // 替换
        sb.replace(5, 6, ","); // sb="hello, java"
        // 删除
        sb.delete(5, 6); // sb="hellojava"
        System.out.println(sb);
        // 反转
        sb.reverse(); // sb="avajolleh"
        System.out.println(sb);
        System.out.println(sb.length()); // 输出 9
        System.out.println(sb.capacity()); // 输出 16
        // 改变 StringBuilder 的长度，将只保留前面部分
        sb.setLength(5); // sb="avajo"
        System.out.println(sb);
    }
}
```

上面程序中粗体字部分示范了StringBuilder类的追加、插入、替换、删除等操作，这些操作改变了StringBuilder里的字符序列，这就是StringBuilder与String之间最大的区别：StringBuilder的字符序列是可变的。从程序看到StringBuilder的length()方法返回其字符序列的长度，而capacity()返回值则比length()返回值大。

7.3.4 Math类

Java提供了基本的+、－、*、/、%等基本算术运算的运算符，但对于更复杂的数学运算，例如，三角函数、对数运算、指数运算等则无能为力。Java提供了Math工具类来完成这些复杂的运算，Math类是一个工具类，它的构造器被定义成private的，因此无法创建Math类的对象；Math类中的所有方法都是类方法，可以直接通过类名来调用它们。Math类除了提供了大量静态方法之外，还提供了两个

类变量：PI 和 E，正如它们名字所暗示的，它们的值分别等于π和 e。

Math 类的所有方法名都明确标识了该方法的作用，读者可自行查阅 API 来了解 Math 类各方法的说明。下面程序示范了 Math 类的用法。

程序清单：codes\07\7.3\MathTest.java

```
public class MathTest
{
    public static void main(String[] args)
    {
        /*---------下面是三角运算---------*/
        // 将弧度转换成角度
        System.out.println("Math.toDegrees(1.57): "
            + Math.toDegrees(1.57));
        // 将角度转换为弧度
        System.out.println("Math.toRadians(90): "
            + Math.toRadians(90));
        // 计算反余弦，返回的角度范围在 0.0 到 pi 之间
        System.out.println("Math.acos(1.2): " + Math.acos(1.2));
        // 计算反正弦，返回的角度范围在 -pi/2 到 pi/2 之间
        System.out.println("Math.asin(0.8): " + Math.asin(0.8));
        // 计算反正切，返回的角度范围在 -pi/2 到 pi/2 之间
        System.out.println("Math.atan(2.3): " + Math.atan(2.3));
        // 计算三角余弦
        System.out.println("Math.cos(1.57): " + Math.cos(1.57));
        // 计算双曲余弦
        System.out.println("Math.cosh(1.2 ): " + Math.cosh(1.2 ));
        // 计算正弦
        System.out.println("Math.sin(1.57 ): " + Math.sin(1.57 ));
        // 计算双曲正弦
        System.out.println("Math.sinh(1.2 ): " + Math.sinh(1.2 ));
        // 计算三角正切
        System.out.println("Math.tan(0.8 ): " + Math.tan(0.8 ));
        // 计算双曲正切
        System.out.println("Math.tanh(2.1 ): " + Math.tanh(2.1 ));
        // 将矩形坐标 (x, y) 转换成极坐标 (r, thet))
        System.out.println("Math.atan2(0.1, 0.2): " + Math.atan2(0.1, 0.2));
        /*---------下面是取整运算---------*/
        // 取整，返回小于目标数的最大整数
        System.out.println("Math.floor(-1.2 ): " + Math.floor(-1.2 ));
        // 取整，返回大于目标数的最小整数
        System.out.println("Math.ceil(1.2): " + Math.ceil(1.2));
        // 四舍五入取整
        System.out.println("Math.round(2.3 ): " + Math.round(2.3 ));
        /*---------下面是乘方、开方、指数运算---------*/
        // 计算平方根
        System.out.println("Math.sqrt(2.3 ): " + Math.sqrt(2.3 ));
        // 计算立方根
        System.out.println("Math.cbrt(9): " + Math.cbrt(9));
        // 返回欧拉数 e 的 n 次幂
        System.out.println("Math.exp(2): " + Math.exp(2));
        // 返回 sqrt(x2 +y2)，没有中间溢出或下溢
        System.out.println("Math.hypot(4 , 4): " + Math.hypot(4 , 4));
        // 按照 IEEE 754 标准的规定，对两个参数进行余数运算
        System.out.println("Math.IEEEremainder(5 , 2): "
            + Math.IEEEremainder(5 , 2));
        // 计算乘方
        System.out.println("Math.pow(3, 2): " + Math.pow(3, 2));
        // 计算自然对数
        System.out.println("Math.log(12): " + Math.log(12));
        // 计算底数为 10 的对数
        System.out.println("Math.log10(9): " + Math.log10(9));
        // 返回参数与 1 之和的自然对数
```

```
        System.out.println("Math.log1p(9): " + Math.log1p(9));
        /*---------下面是符号相关的运算---------*/
        // 计算绝对值
        System.out.println("Math.abs(-4.5): " + Math.abs(-4.5));
        // 符号赋值，返回带有第二个浮点数符号的第一个浮点参数
        System.out.println("Math.copySign(1.2, -1.0): "
            + Math.copySign(1.2, -1.0));
        // 符号函数，如果参数为 0，则返回 0；如果参数大于 0
        // 则返回 1.0；如果参数小于 0，则返回 -1.0
        System.out.println("Math.signum(2.3): " + Math.signum(2.3));
        /*---------下面是大小相关的运算---------*/
        // 找出最大值
        System.out.println("Math.max(2.3 , 4.5): " + Math.max(2.3 , 4.5));
        // 计算最小值
        System.out.println("Math.min(1.2 , 3.4): " + Math.min(1.2 , 3.4));
        // 返回第一个参数和第二个参数之间与第一个参数相邻的浮点数
        System.out.println("Math.nextAfter(1.2, 1.0): "
            + Math.nextAfter(1.2, 1.0));
        // 返回比目标数略大的浮点数
        System.out.println("Math.nextUp(1.2 ): " + Math.nextUp(1.2 ));
        // 返回一个伪随机数，该值大于等于 0.0 且小于 1.0
        System.out.println("Math.random(): " + Math.random());
    }
}
```

上面程序中关于 Math 类的用法几乎覆盖了 Math 类的所有数学计算功能，读者可参考上面程序来学习 Math 类的用法。

7.3.5 Java 7 的 ThreadLocalRandom 与 Random

Random 类专门用于生成一个伪随机数，它有两个构造器：一个构造器使用默认的种子（以当前时间作为种子），另一个构造器需要程序员显式传入一个 long 型整数的种子。

ThreadLocalRandom 类是 Java 7 新增的一个类，它是 Random 的增强版。在并发访问的环境下，使用 ThreadLocalRandom 来代替 Random 可以减少多线程资源竞争，最终保证系统具有更好的线程安全性。



提示：

关于多线程编程的知识，请参考本书第 16 章的内容。

ThreadLocalRandom 类的用法与 Random 类的用法基本相似，它提供了一个静态的 current()方法来获取 ThreadLocalRandom 对象，获取该对象之后即可调用各种 nextXxx()方法来获取伪随机数了。

ThreadLocalRandom 与 Random 都比 Math 的 random()方法提供了更多的方式来生成各种伪随机数，可以生成浮点类型的伪随机数，也可以生成整数类型的伪随机数，还可以指定生成随机数的范围。关于 Random 类的用法如下程序所示。

程序清单：codes\07\7.3\RandomTest.java

```
public class RandomTest
{
    public static void main(String[] args)
    {
        Random rand = new Random();
        System.out.println("rand.nextBoolean(): "
            + rand.nextBoolean());
        byte[] buffer = new byte[16];
        rand.nextBytes(buffer);
        System.out.println(Arrays.toString(buffer));
        // 生成 0.0~1.0 之间的伪随机 double 数
        System.out.println("rand.nextDouble(): "
            + rand.nextDouble());
        // 生成 0.0~1.0 之间的伪随机 float 数
        System.out.println("rand.nextFloat(): "
            + rand.nextFloat());
```

```
        // 生成平均值是 0.0，标准差是 1.0 的伪高斯数
        System.out.println("rand.nextGaussian(): "
            + rand.nextGaussian());
        // 生成一个处于 int 整数取值范围的伪随机整数
        System.out.println("rand.nextInt(): " + rand.nextInt());
        // 生成 0~26 之间的伪随机整数
        System.out.println("rand.nextInt(26): " + rand.nextInt(26));
        // 生成一个处于 long 整数取值范围的伪随机整数
        System.out.println("rand.nextLong(): " +  rand.nextLong());
    }
}
```

从上面程序中可以看出，Random 可以提供很多选项来生成伪随机数。

Random 使用一个 48 位的种子，如果这个类的两个实例是用同一个种子创建的，对它们以同样的顺序调用方法，则它们会产生相同的数字序列。

下面就对上面的介绍做一个实验，可以看到当两个 Random 对象种子相同时，它们会产生相同的数字序列。值得指出的，当使用默认的种子构造 Random 对象时，它们属于同一个种子。

程序清单：codes\07\7.3\SeedTest.java

```
public class SeedTest
{
    public static void main(String[] args)
    {
        Random r1 = new Random(50);
        System.out.println("第一个种子为 50 的 Random 对象");
        System.out.println("r1.nextBoolean():\t" + r1.nextBoolean());
        System.out.println("r1.nextInt():\t\t" + r1.nextInt());
        System.out.println("r1.nextDouble():\t" + r1.nextDouble());
        System.out.println("r1.nextGaussian():\t" + r1.nextGaussian());
        System.out.println("----------------------------");
        Random r2 = new Random(50);
        System.out.println("第二个种子为 50 的 Random 对象");
        System.out.println("r2.nextBoolean():\t" + r2.nextBoolean());
        System.out.println("r2.nextInt():\t\t" + r2.nextInt());
        System.out.println("r2.nextDouble():\t" + r2.nextDouble());
        System.out.println("r2.nextGaussian():\t" + r2.nextGaussian());
        System.out.println("----------------------------");
        Random r3 = new Random(100);
        System.out.println("种子为 100 的 Random 对象");
        System.out.println("r3.nextBoolean():\t" + r3.nextBoolean());
        System.out.println("r3.nextInt():\t\t" + r3.nextInt());
        System.out.println("r3.nextDouble():\t" + r3.nextDouble());
        System.out.println("r3.nextGaussian():\t" + r3.nextGaussian());
    }
}
```

运行上面程序，看到如下结果：

```
第一个种子为 50 的 Random 对象
r1.nextBoolean():       true
r1.nextInt():           -1727040520
r1.nextDouble():        0.6141579720626675
r1.nextGaussian():      2.377650302287946
----------------------------
第二个种子为 50 的 Random 对象
r2.nextBoolean():       true
r2.nextInt():           -1727040520
r2.nextDouble():        0.6141579720626675
r2.nextGaussian():      2.377650302287946
----------------------------
种子为 100 的 Random 对象
r3.nextBoolean():       true
r3.nextInt():           -1139614796
r3.nextDouble():        0.19497605734770518
r3.nextGaussian():      0.6762208162903859
```

从上面运行结果来看，只要两个 Random 对象的种子相同，而且方法的调用顺序也相同，它们就会

产生相同的数字序列。也就是说，Random产生的数字并不是真正随机的，而是一种伪随机。

为了避免两个Random对象产生相同的数字序列，通常推荐使用当前时间作为Random对象的种子，如下代码所示。

```
Random rand = new Random(System.currentTimeMillis());
```

在多线程环境下使用ThreadLocalRandom的方式与使用Random基本类似，如下程序片段示范了ThreadLocalRandom的用法。

```
ThreadLocalRandom rand = ThreadLocalRandom.current();
// 生成一个4~20之间的伪随机整数
int val1 = rand.nextInt(4 , 20);
// 生成一个2.0~10.0之间的伪随机浮点数
int val2 = rand.nextDouble(2.0, 10.0);
```

7.3.6 BigDecimal类

前面在介绍float、double两种基本浮点类型时已经指出，这两个基本类型的浮点数容易引起精度丢失。先看如下程序。

程序清单：codes\07\7.3\DoubleTest.java

```
public class DoubleTest
{
    public static void main(String args[])
    {
        System.out.println("0.05 + 0.01 = " + (0.05 + 0.01));
        System.out.println("1.0 - 0.42 = " + (1.0 - 0.42));
        System.out.println("4.015 * 100 = " + (4.015 * 100));
        System.out.println("123.3 / 100 = " + (123.3 / 100));
    }
}
```

程序输出结果是：

```
0.05 + 0.01 = 0.060000000000000005
1.0 - 0.42 = 0.5800000000000001
4.015 * 100 = 401.49999999999994
123.3 / 100 = 1.2329999999999999
```

上面程序运行结果表明，Java的double类型会发生精度丢失，尤其在进行算术运算时更容易发生这种情况。不仅是Java，很多编程语言也存在这样的问题。

为了能精确表示、计算浮点数，Java提供了BigDecimal类，该类提供了大量的构造器用于创建BigDecimal对象，包括把所有的基本数值型变量转换成一个BigDecimal对象，也包括利用数字字符串、数字字符数组来创建BigDecimal对象。

查看BigDecimal类的BigDecimal(double val)构造器的详细说明时，可以看到不推荐使用该构造器的说明，主要是因为使用该构造器时有一定的不可预知性。当程序使用new BigDecimal(0.1)来创建一个BigDecimal对象时，它的值并不是0.1，它实际上等于一个近似0.1的数。这是因为0.1无法准确地表示为double浮点数，所以传入BigDecimal构造器的值不会正好等于0.1（虽然表面上等于该值）。

如果使用BigDecimal(String val)构造器的结果是可预知的——写入new BigDecimal("0.1")将创建一个BigDecimal，它正好等于预期的0.1。因此通常建议优先使用基于String的构造器。

如果必须使用double浮点数作为BigDecimal构造器的参数时，不要直接将该double浮点数作为构造器参数创建BigDecimal对象，而是应该通过BigDecimal.valueOf(double value)静态方法来创建BigDecimal对象。

BigDecimal类提供了add()、subtract()、multiply()、divide()、pow()等方法对精确浮点数进行常规算术运算。下面程序示范了BigDecimal的基本运算。

程序清单：codes\07\7.3\BigDecimalTest.java

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

```
        BigDecimal f1 = new BigDecimal("0.05");
        BigDecimal f2 = BigDecimal.valueOf(0.01);
        BigDecimal f3 = new BigDecimal(0.05);
        System.out.println("使用 String 作为 BigDecimal 构造器参数：");
        System.out.println("0.05 + 0.01 = " + f1.add(f2));
        System.out.println("0.05 - 0.01 = " + f1.subtract(f2));
        System.out.println("0.05 * 0.01 = " + f1.multiply(f2));
        System.out.println("0.05 / 0.01 = " + f1.divide(f2));
        System.out.println("使用 double 作为 BigDecimal 构造器参数：");
        System.out.println("0.05 + 0.01 = " + f3.add(f2));
        System.out.println("0.05 - 0.01 = " + f3.subtract(f2));
        System.out.println("0.05 * 0.01 = " + f3.multiply(f2));
        System.out.println("0.05 / 0.01 = " + f3.divide(f2));
    }
}
```

上面程序中 f1 和 f3 都是基于 0.05 创建的 BigDecimal 对象，其中 f1 是基于"0.05"字符串，但 f3 是基于 0.05 的 double 浮点数。运行上面程序，看到如下运行结果：

```
使用 String 作为 BigDecimal 构造器参数：
0.05 + 0.01 = 0.06
0.05 - 0.01 = 0.04
0.05 * 0.01 = 0.0005
0.05 / 0.01 = 5
使用 double 作为 BigDecimal 构造器参数：
0.05 + 0.01 = 0.060000000000000002775557561562891351059079170227050781 25
0.05 - 0.01 = 0.040000000000000002775557561562891351059079170227050781 25
0.05 * 0.01 = 0.000500000000000000027755575615628913510590791702270507812 5
0.05 / 0.01 = 5.000000000000000027755575615628913510590791702270507812 5
```

从上面运行结果可以看出 BigDecimal 进行算术运算的效果，而且可以看出创建 BigDecimal 对象时，一定要使用 String 对象作为构造器参数，而不是直接使用 double 数字。

注意：

创建 BigDecimal 对象时，不要直接使用 double 浮点数作为构造器参数来调用 BigDecimal 构造器，否则同样会发生精度丢失的问题。



如果程序中要求对 double 浮点数进行加、减、乘、除基本运算，则需要先将 double 类型数值包装成 BigDecimal 对象，调用 BigDecimal 对象的方法执行运算后再将结果转换成 double 型变量。这是比较烦琐的过程，可以考虑以 BigDecimal 为基础定义一个 Arith 工具类，该工具类代码如下。

程序清单：codes\07\7.3\Arith.java

```
public class Arith
{
    // 默认除法运算精度
    private static final int DEF_DIV_SCALE = 10;
    // 构造器私有，让这个类不能实例化
    private Arith()    {}
    // 提供精确的加法运算
    public static double add(double v1,double v2)
    {
        BigDecimal b1 = BigDecimal.valueOf(v1);
        BigDecimal b2 = BigDecimal.valueOf(v2);
        return b1.add(b2).doubleValue();
    }
    // 提供精确的减法运算
    public static double sub(double v1,double v2)
    {
        BigDecimal b1 = BigDecimal.valueOf(v1);
        BigDecimal b2 = BigDecimal.valueOf(v2);
        return b1.subtract(b2).doubleValue();
    }
    // 提供精确的乘法运算
    public static double mul(double v1,double v2)
    {
```

```
        BigDecimal b1 = BigDecimal.valueOf(v1);
        BigDecimal b2 = BigDecimal.valueOf(v2);
        return b1.multiply(b2).doubleValue();
    }
    // 提供（相对）精确的除法运算，当发生除不尽的情况时
    // 精确到小数点以后10位的数字四舍五入
    public static double div(double v1,double v2)
    {
        BigDecimal b1 = BigDecimal.valueOf(v1);
        BigDecimal b2 = BigDecimal.valueOf(v2);
        return b1.divide(b2 , DEF_DIV_SCALE
            , BigDecimal.ROUND_HALF_UP).doubleValue();
    }
    public static void main(String[] args)
    {
        System.out.println("0.05 + 0.01 = "
            + Arith.add(0.05 , 0.01));
        System.out.println("1.0 - 0.42 = "
            + Arith.sub(1.0 , 0.42));
        System.out.println("4.015 * 100 = "
            + Arith.mul(4.015 , 100));
        System.out.println("123.3 / 100 = "
            + Arith.div(123.3 , 100));
    }
}
```

Arith 工具类还提供了 main 方法用于测试加、减、乘、除等运算。运行上面程序将看到如下运行结果：

```
0.05 + 0.01 = 0.06
1.0 - 0.42 = 0.58
4.015 * 100 = 401.5
123.3 / 100 = 1.233
```

上面的运行结果才是期望的结果，这也正是使用 BigDecimal 类的作用。

7.4 Java 8 的日期、时间类

Java 原本提供了 Date 和 Calendar 用于处理日期、时间的类，包括创建日期、时间对象，获取系统当前日期、时间等操作。但 Date 不仅无法实现国际化，而且它对不同属性也使用了前后矛盾的偏移量，比如月份与小时都是从 0 开始的，月份中的天数则是从 1 开始的，年又是从 1900 开始的，而 java.util.Calendar 则显得过于复杂，从下面介绍中会看到传统 Java 对日期、时间处理的不足。Java 8 吸取了 Joda-Time 库（一个被广泛使用的日期、时间库）的经验，提供了一套全新的日期时间库。

7.4.1 Date 类

Java 提供了 Date 类来处理日期、时间（此处的 Date 是指 java.util 包下的 Date 类，而不是 java.sql 包下的 Date 类），Date 对象既包含日期，也包含时间。Date 类从 JDK 1.0 起就开始存在了，但正因为它历史悠久，所以它的大部分构造器、方法都已经过时，不再推荐使用了。

Date 类提供了 6 个构造器，其中 4 个已经 Deprecated（Java 不再推荐使用，使用不再推荐的构造器时编译器会提出警告信息，并导致程序性能、安全性等方面的问题），剩下的两个构造器如下。

- Date()：生成一个代表当前日期时间的 Date 对象。该构造器在底层调用 System.currentTimeMillis() 获得 long 整数作为日期参数。
- Date(long date)：根据指定的 long 型整数来生成一个 Date 对象。该构造器的参数表示创建的 Date 对象和 GMT 1970 年 1 月 1 日 00:00:00 之间的时间差，以毫秒作为计时单位。

与 Date 构造器相同的是，Date 对象的大部分方法也 Deprecated 了，剩下为数不多的几个方法。

- boolean after(Date when)：测试该日期是否在指定日期 when 之后。
- boolean before(Date when)：测试该日期是否在指定日期 when 之前。
- long getTime()：返回该时间对应的 long 型整数，即从 GMT 1970-01-01 00:00:00 到该 Date 对象之间的时间差，以毫秒作为计时单位。

➢ void setTime(long time)：设置该 Date 对象的时间。

下面程序示范了 Date 类的用法。

程序清单：codes\07\7.4\DateTest.java

```
public class DateTest
{
    public static void main(String[] args)
    {
        Date d1 = new Date();
        // 获取当前时间之后 100ms 的时间
        Date d2 = new Date(System.currentTimeMillis() + 100);
        System.out.println(d2);
        System.out.println(d1.compareTo(d2));
        System.out.println(d1.before(d2));
    }
}
```

总体来说，Date 是一个设计相当糟糕的类，因此 Java 官方推荐尽量少用 Date 的构造器和方法。如果需要对日期、时间进行加减运算，或获取指定时间的年、月、日、时、分、秒信息，可使用 Calendar 工具类。

7.4.2 Calendar 类

因为 Date 类在设计上存在一些缺陷，所以 Java 提供了 Calendar 类来更好地处理日期和时间。Calendar 是一个抽象类，它用于表示日历。

历史上有着许多种纪年方法，它们的差异实在太大了，比如说一个人的生日是“七月七日”，那么一种可能是阳（公）历的七月七日，但也可以是阴（农）历的日期。为了统一计时，全世界通常选择最普及、最通用的日历：Gregorian Calendar，也就是日常介绍年份时常用的“公元几几年”。

Calendar 类本身是一个抽象类，它是所有日历类的模板，并提供了一些所有日历通用的方法；但它本身不能直接实例化，程序只能创建 Calendar 子类的实例，Java 本身提供了一个 GregorianCalendar 类，一个代表格里高利日历的子类，它代表了通常所说的公历。

当然，也可以创建自己的 Calendar 子类，然后将它作为 Calendar 对象使用（这就是多态）。在 IBM 的 alphaWorks 站点（http://www.alphaworks.ibm.com/tech/calendars）上，IBM 的开发人员实现了多种日历。在 Internet 上，也有对中国农历的实现。因为篇幅关系，本章不会详细介绍如何扩展 Calendar 子类，读者可以查看上述 Calendar 的源码来学习。

Calendar 类是一个抽象类，所以不能使用构造器来创建 Calendar 对象。但它提供了几个静态 getInstance()方法来获取 Calendar 对象，这些方法根据 TimeZone，Locale 类来获取特定的 Calendar，如果不指定 TimeZone、Locale，则使用默认的 TimeZone、Locale 来创建 Calendar。



提示：

关于 TimeZone、Locale 的介绍请参考本章后面知识。

Calendar 与 Date 都是表示日期的工具类，它们直接可以自由转换，如下代码所示。

```
// 创建一个默认的 Calendar 对象
Calendar calendar = Calendar.getInstance();
// 从 Calendar 对象中取出 Date 对象
Date date = calendar.getTime();
// 通过 Date 对象获得对应的 Calendar 对象
// 因为 Calendar/GregorianCalendar 没有构造函数可以接收 Date 对象
// 所以必须先获得一个 Calendar 实例，然后调用其 setTime()方法
Calendar calendar2 = Calendar.getInstance();
calendar2.setTime(date);
```

Calendar 类提供了大量访问、修改日期时间的方法，常用方法如下。

➢ void add(int field, int amount)：根据日历的规则，为给定的日历字段添加或减去指定的时间量。

➢ int get(int field)：返回指定日历字段的值。

➢ int getActualMaximum(int field)：返回指定日历字段可能拥有的最大值。例如月，最大值为 11。

- int getActualMinimum(int field)：返回指定日历字段可能拥有的最小值。例如月，最小值为 0。
- void roll(int field, int amount)：与 add()方法类似，区别在于加上 amount 后超过了该字段所能表示的最大范围时，也不会向上一个字段进位。
- void set(int field, int value)：将给定的日历字段设置为给定值。
- void set(int year, int month, int date)：设置 Calendar 对象的年、月、日三个字段的值。
- void set(int year, int month, int date, int hourOfDay, int minute, int second)：设置 Calendar 对象的年、月、日、时、分、秒 6 个字段的值。

上面的很多方法都需要一个 int 类型的 field 参数，field 是 Calendar 类的类变量，如 Calendar.YEAR、Calendar.MONTH 等分别代表了年、月、日、小时、分钟、秒等时间字段。需要指出的是，Calendar.MONTH 字段代表月份，月份的起始值不是 1，而是 0，所以要设置 8 月时，用 7 而不是 8。如下程序示范了 Calendar 类的常规用法。

程序清单：codes\07\7.4\CalendarTest.java

```
public class CalendarTest
{
    public static void main(String[] args)
    {
        Calendar c = Calendar.getInstance();
        // 取出年
        System.out.println(c.get(YEAR));
        // 取出月份
        System.out.println(c.get(MONTH));
        // 取出日
        System.out.println(c.get(DATE));
        // 分别设置年、月、日、小时、分钟、秒
        c.set(2003 , 10 , 23 , 12, 32, 23); // 2003-11-23 12:32:23
        System.out.println(c.getTime());
        // 将 Calendar 的年前推 1 年
        c.add(YEAR , -1); // 2002-11-23 12:32:23
        System.out.println(c.getTime());
        // 将 Calendar 的月前推 8 个月
        c.roll(MONTH , -8); // 2002-03-23 12:32:23
        System.out.println(c.getTime());
    }
}
```

上面程序中粗体字代码示范了 Calendar 类的用法，Calendar 可以很灵活地改变它对应的日期。



提示：

上面程序使用了静态导入，它导入了 Calendar 类里的所有类变量，所以上面程序可以直接使用 Calendar 类的 YEAR、MONTH、DATE 等类变量。

Calendar 类还有如下几个注意点。

1. add 与 roll 的区别

add(int field, int amount)的功能非常强大，add 主要用于改变 Calendar 的特定字段的值。如果需要增加某字段的值，则让 amount 为正数；如果需要减少某字段的值，则让 amount 为负数即可。

add(int field, int amount)有如下两条规则。

- 当被修改的字段超出它允许的范围时，会发生进位，即上一级字段也会增大。例如：

```
Calendar cal1 = Calendar.getInstance();
cal1.set(2003, 7, 23, 0, 0 , 0); // 2003-8-23
cal1.add(MONTH, 6); // 2003-8-23 => 2004-2-23
```

- 如果下一级字段也需要改变，那么该字段会修正到变化最小的值。例如：

```
Calendar cal2 = Calendar.getInstance();
cal2.set(2003, 7, 31, 0, 0 , 0); // 2003-8-31
// 因为进位后月份改为 2 月，2 月没有 31 日，自动变成 29 日
cal2.add(MONTH, 6); // 2003-8-31 => 2004-2-29
```

对于上面的例子，8-31 就会变成 2-29。因为 MONTH 的下一级字段是 DATE，从 31 到 29 改变最小。所以上面 2003-8-31 的 MONTH 字段增加 6 后，不是变成 2004-3-2，而是变成 2004-2-29。

roll()的规则与 add()的处理规则不同：当被修改的字段超出它允许的范围时，上一级字段不会增大。

```
Calendar cal3 = Calendar.getInstance();
cal3.set(2003, 7, 23, 0, 0 , 0); // 2003-8-23
// MONTH 字段“进位”，但 YEAR 字段并不增加
cal3.roll(MONTH, 6); // 2003-8-23 => 2003-2-23
```

下一级字段的处理规则与 add()相似：

```
Calendar cal4 = Calendar.getInstance();
cal4.set(2003, 7, 31, 0, 0 , 0); // 2003-8-31
// MONTH 字段“进位”后变成 2，2 月没有 31 日
// YEAR 字段不会改变，2003 年 2 月只有 28 天
cal4.roll(MONTH, 6); // 2003-8-31 => 2003-2-28
```

2. 设置 Calendar 的容错性

调用 Calendar 对象的 set()方法来改变指定时间字段的值时，有可能传入一个不合法的参数，例如为 MONTH 字段设置 13，这将会导致怎样的后果呢？看如下程序。

程序清单：codes\07\7.4\LenientTest.java

```
public class LenientTest
{
    public static void main(String[] args)
    {
        Calendar cal = Calendar.getInstance();
        // 结果是 YEAR 字段加 1，MONTH 字段为 1（2 月）
        cal.set(MONTH , 13);   // ①
        System.out.println(cal.getTime());
        // 关闭容错性
        cal.setLenient(false);
        // 导致运行时异常
        cal.set(MONTH , 13);   // ②
        System.out.println(cal.getTime());
    }
}
```

上面程序①②两处的代码完全相似，但它们运行的结果不一样：①处代码可以正常运行，因为设置 MONTH 字段的值为 13，将会导致 YEAR 字段加 1；②处代码将会导致运行时异常，因为设置的 MONTH 字段值超出了 MONTH 字段允许的范围。关键在于程序中粗体字代码行，Calendar 提供了一个 setLenient()用于设置它的容错性，Calendar 默认支持较好的容错性，通过 setLenient(false)可以关闭 Calendar 的容错性，让它进行严格的参数检查。

Calendar 有两种解释日历字段的模式：lenient 模式和 non-lenient 模式。当 Calendar 处于 lenient 模式时，每个时间字段可接受超出它允许范围的值；当 Calendar 处于 non-lenient 模式时，如果为某个时间字段设置的值超出了它允许的取值范围，程序将会抛出异常。

3. set()方法延迟修改

set(f, value)方法将日历字段 f 更改为 value，此外它还设置了一个内部成员变量，以指示日历字段 f 已经被更改。尽管日历字段 f 是立即更改的，但该 Calendar 所代表的时间却不会立即修改，直到下次调用 get()、getTime()、getTimeInMillis()、add() 或 roll()时才会重新计算日历的时间。这被称为 set()方法的延迟修改，采用延迟修改的优势是多次调用 set()不会触发多次不必要的计算（需要计算出一个代表实际时间的 long 型整数）。

下面程序演示了 set()方法延迟修改的效果。

程序清单：codes\07\7.4\LazyTest.java

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

```
        Calendar cal = Calendar.getInstance();
        cal.set(2003 , 7 , 31);  // 2003-8-31
        // 将月份设为9，但9月31日不存在
        // 如果立即修改，系统将会把cal自动调整到10月1日
        cal.set(MONTH , 8);
        // 下面代码输出10月1日
        // System.out.println(cal.getTime());    // ①
        // 设置DATE字段为5
        cal.set(DATE , 5);    // ②
        System.out.println(cal.getTime());    // ③
    }
}
```

上面程序中创建了代表2003-8-31的Calendar对象，当把这个对象的MONTH字段加1后应该得到2003-10-1（因为9月没有31日），如果程序在①号代码处输出当前Calendar里的日期，也会看到输出2003-10-1，③号代码处将输出2003-10-5。

如果程序将①处代码注释起来，因为Calendar的set()方法具有延迟修改的特性，即调用set()方法后Calendar实际上并未计算真实的日期，它只是使用内部成员变量表记录MONTH字段被修改为8，接着程序设置DATE字段值为5，程序内部再次记录DATE字段为5——就是9月5日，因此看到③处输出2003-9-5。

7.4.3 Java 8新增的日期、时间包

Java 8专门新增了一个java.time包，该包下包含了如下常用的类。

- Clock：该类用于获取指定时区的当前日期、时间。该类可取代System类的currentTimeMillis()方法，而且提供了更多方法来获取当前日期、时间。该类提供了大量静态方法来获取Clock对象。
- Duration：该类代表持续时间。该类可以非常方便地获取一段时间。
- Instant：代表一个具体的时刻，可以精确到纳秒。该类提供了静态的now()方法来获取当前时刻，也提供了静态的now(Clock clock)方法来获取clock对应的时刻。除此之外，它还提供了一系列minusXxx()方法在当前时刻基础上减去一段时间，也提供了plusXxx()方法在当前时刻基础上加上一段时间。
- LocalDate：该类代表不带时区的日期，例如2007-12-03。该类提供了静态的now()方法来获取当前日期，也提供了静态的now(Clock clock)方法来获取clock对应的日期。除此之外，它还提供了minusXxx()方法在当前年份基础上减去几年、几月、几周或几日等，也提供了plusXxx()方法在当前年份基础上加上几年、几月、几周或几日等。
- LocalTime：该类代表不带时区的时间，例如10:15:30。该类提供了静态的now()方法来获取当前时间，也提供了静态的now(Clock clock)方法来获取clock对应的时间。除此之外，它还提供了minusXxx()方法在当前年份基础上减去几小时、几分、几秒等，也提供了plusXxx()方法在当前年份基础上加上几小时、几分、几秒等。
- LocalDateTime：该类代表不带时区的日期、时间，例如2007-12-03T10:15:30。该类提供了静态的now()方法来获取当前日期、时间，也提供了静态的now(Clock clock)方法来获取clock对应的日期、时间。除此之外，它还提供了minusXxx()方法在当前年份基础上减去几年、几月、几日、几小时、几分、几秒等，也提供了plusXxx()方法在当前年份基础上加上几年、几月、几日、几小时、几分、几秒等。
- MonthDay：该类仅代表月日，例如--04-12。该类提供了静态的now()方法来获取当前月日，也提供了静态的now(Clock clock)方法来获取clock对应的月日。
- Year：该类仅代表年，例如2014。该类提供了静态的now()方法来获取当前年份，也提供了静态的now(Clock clock)方法来获取clock对应的年份。除此之外，它还提供了minusYears()方法在当前年份基础上减去几年，也提供了plusYears()方法在当前年份基础上加上几年。
- YearMonth：该类仅代表年月，例如2014-04。该类提供了静态的now()方法来获取当前年月，也提供了静态的now(Clock clock)方法来获取clock对应的年月。除此之外，它还提供了

minusXxx()方法在当前年月基础上减去几年、几月，也提供了 plusXxx()方法在当前年月基础上加上几年、几月。

- ZonedDateTime：该类代表一个时区化的日期、时间。
- ZoneId：该类代表一个时区。
- DayOfWeek：这是一个枚举类，定义了周日到周六的枚举值。
- Month：这也是一个枚举类，定义了一月到十二月的枚举值。

下面通过一个简单的程序来示范这些类的用法。

程序清单：codes\07\7.4\NewDatePackageTest.java

```
public class NewDatePackageTest
{
    public static void main(String[] args)
    {
        // -----下面是关于 Clock 的用法-----
        // 获取当前 Clock
        Clock clock = Clock.systemUTC();
        // 通过 Clock 获取当前时刻
        System.out.println("当前时刻为：" + clock.instant());
        // 获取 clock 对应的毫秒数，与 System.currentTimeMillis()输出相同
        System.out.println(clock.millis());
        System.out.println(System.currentTimeMillis());
        // -----下面是关于 Duration 的用法-----
        Duration d = Duration.ofSeconds(6000);
        System.out.println("6000 秒相当于" + d.toMinutes() + "分");
        System.out.println("6000 秒相当于" + d.toHours() + "小时");
        System.out.println("6000 秒相当于" + d.toDays() + "天");
        // 在 clock 基础上增加 6000 秒，返回新的 Clock
        Clock clock2 = Clock.offset(clock, d);
        // 可以看到 clock2 与 clock1 相差 1 小时 40 分
        System.out.println("当前时刻加 6000 秒为：" +clock2.instant());
        // -----下面是关于 Instant 的用法-----
        // 获取当前时间
        Instant instant = Instant.now();
        System.out.println(instant);
        // instant 添加 6000 秒（即 100 分钟），返回新的 Instant
        Instant instant2 = instant.plusSeconds(6000);
        System.out.println(instant2);
        // 根据字符串解析 Instant 对象
        Instant instant3 = Instant.parse("2014-02-23T10:12:35.342Z");
        System.out.println(instant3);
        // 在 instant3 的基础上添加 5 小时 4 分钟
        Instant instant4 = instant3.plus(Duration
            .ofHours(5).plusMinutes(4));
        System.out.println(instant4);
        // 获取 instant4 的 5 天以前的时刻
        Instant instant5 = instant4.minus(Duration.ofDays(5));
        System.out.println(instant5);
        // -----下面是关于 LocalDate 的用法-----
        LocalDate localDate = LocalDate.now();
        System.out.println(localDate);
        // 获得 2014 年的第 146 天
        localDate = LocalDate.ofYearDay(2014, 146);
        System.out.println(localDate); // 2014-05-26
        // 设置为 2014 年 5 月 21 日
        localDate = LocalDate.of(2014, Month.MAY, 21);
        System.out.println(localDate); // 2014-05-21
        // -----下面是关于 LocalTime 的用法-----
        // 获取当前时间
        LocalTime localTime = LocalTime.now();
        // 设置为 22 点 33 分
        localTime = LocalTime.of(22, 33);
        System.out.println(localTime); // 22:33
        // 返回一天中的第 5503 秒
```

```
        localTime = LocalTime.ofSecondOfDay(5503);
        System.out.println(localTime); // 01:31:43
        // -----下面是关于 localDateTime 的用法-----
        // 获取当前日期、时间
        LocalDateTime localDateTime = LocalDateTime.now();
        // 当前日期、时间加上 25 小时 3 分钟
        LocalDateTime future = localDateTime.plusHours(25).plusMinutes(3);
        System.out.println("当前日期、时间的 25 小时 3 分之后：" + future);
        // -----下面是关于 Year、YearMonth、MonthDay 的用法示例-----
        Year year = Year.now(); // 获取当前的年份
        System.out.println("当前年份：" + year); // 输出当前年份
        year = year.plusYears(5); // 当前年份再加 5 年
        System.out.println("当前年份再过 5 年：" + year);
        // 根据指定月份获取 YearMonth
        YearMonth ym = year.atMonth(10);
        System.out.println("year 年 10 月：" + ym); // 输出 XXXX-10，XXXX 代表当前年份
        // 当前年月再加 5 年、减 3 个月
        ym = ym.plusYears(5).minusMonths(3);
        System.out.println("year 年 10 月再加 5 年、减 3 个月：" + ym);
        MonthDay md = MonthDay.now();
        System.out.println("当前月日：" + md); // 输出--XX-XX，代表几月几日
        // 设置为 5 月 23 日
        MonthDay md2 = md.with(Month.MAY).withDayOfMonth(23);
        System.out.println("5 月 23 日为：" + md2); // 输出--05-23
    }
}
```

该程序就是这些常见类的用法示例，这些 API 和它们的方法都非常简单，而且程序中注释也很清楚，此处不再赘述。

7.5 正则表达式

正则表达式是一个强大的字符串处理工具，可以对字符串进行查找、提取、分割、替换等操作。String 类里也提供了如下几个特殊的方法。

- boolean matches(String regex)：判断该字符串是否匹配指定的正则表达式。
- String replaceAll(String regex, String replacement)：将该字符串中所有匹配 regex 的子串替换成 replacement。
- String replaceFirst(String regex, String replacement)：将该字符串中第一个匹配 regex 的子串替换成 replacement。
- String[] split(String regex)：以 regex 作为分隔符，把该字符串分割成多个子串。

上面这些特殊的方法都依赖于 Java 提供的正则表达式支持，除此之外，Java 还提供了 Pattern 和 Matcher 两个类专门用于提供正则表达式支持。

很多读者都会觉得正则表达式是一个非常神奇、高级的知识，其实正则表达式是一种非常简单而且非常实用的工具。正则表达式是一个用于匹配字符串的模板。实际上，任意字符串都可以当成正则表达式使用，例如"abc"，它也是一个正则表达式，只是它只能匹配"abc"字符串。

如果正则表达式仅能匹配"abc"这样的字符串，那么正则表达式也就不值得学习了。下面开始学习如何创建正则表达式。

7.5.1 创建正则表达式

前面已经介绍了，正则表达式就是一个用于匹配字符串的模板，可以匹配一批字符串，所以创建正则表达式就是创建一个特殊的字符串。正则表达式所支持的合法字符如表 7.1 所示。

表 7.1 正则表达式所支持的合法字符

字 符	解 释
x	字符 x（x 可代表任何合法的字符）
\0mnn	八进制数 0mnn 所表示的字符

续表

字　符	解　释
\xhh	十六进制值 0xhh 所表示的字符
\uhhhh	十六进制值 0xhhhh 所表示的 Unicode 字符
\t	制表符（'\u0009'）
\n	新行（换行）符（'\u000A'）
\r	回车符（'\u000D'）
\f	换页符（'\u000C'）
\a	报警（bell）符（'\u0007'）
\e	Escape 符（'\u001B'）
\cx	x 对应的的控制符。例如，\cM 匹配 Ctrl-M。x 值必须为 A~Z 或 a~z 之一。

除此之外，正则表达式中有一些特殊字符，这些特殊字符在正则表达式中有其特殊的用途，比如前面介绍的反斜线（\）。如果需要匹配这些特殊字符，就必须首先将这些字符转义，也就是在前面添加一个反斜线（\）。正则表达式中的特殊字符如表 7.2 所示。

表 7.2　正则表达式中的特殊字符

特殊字符	说　明
$	匹配一行的结尾。要匹配 $ 字符本身，请使用 \$
^	匹配一行的开头。要匹配 ^ 字符本身，请使用 \^
()	标记子表达式的开始和结束位置。要匹配这些字符，请使用 \(和 \)
[]	用于确定中括号表达式的开始和结束位置。要匹配这些字符，请使用 \[和 \]
{}	用于标记前面子表达式的出现频度。要匹配这些字符，请使用 \{ 和 \}
*	指定前面子表达式可以出现零次或多次。要匹配 * 字符本身，请使用 *
+	指定前面子表达式可以出现一次或多次。要匹配 + 字符本身，请使用 \+
?	指定前面子表达式可以出现零次或一次。要匹配 ? 字符本身，请使用 \?
.	匹配除换行符 \n 之外的任何单字符。要匹配 . 字符本身，请使用 \.
\	用于转义下一个字符，或指定八进制、十六进制字符。如果需匹配 \ 字符，请用\\
|	指定两项之间任选一项。如果要匹配 | 字符本身，请使用 |

将上面多个字符拼起来，就可以创建一个正则表达式。例如：

```
"\u0041\\\\"  // 匹配 A\
"\u0061\t"    // 匹配 a<制表符>
"\\?\\["      // 匹配?[
```

注意：

可能有读者觉得第一个正则表达式中怎么有那么多反斜杠啊？这是由于 Java 字符串中反斜杠本身需要转义，因此两个反斜杠（\\）实际上相当于一个（前一个用于转义）。



上面的正则表达式依然只能匹配单个字符，这是因为还未在正则表达式中使用“通配符”，“通配符”是可以匹配多个字符的特殊字符。正则表达式中的“通配符”远远超出了普通通配符的功能，它被称为预定义字符，正则表达式支持如表 7.3 所示的预定义字符。

表 7.3　预定义字符

预定义字符	说　明
.	可以匹配任何字符
\d	匹配 0~9 的所有数字
\D	匹配非数字
\s	匹配所有的空白字符，包括空格、制表符、回车符、换页符、换行符等
\S	匹配所有的非空白字符
\w	匹配所有的单词字符，包括 0~9 所有数字、26 个英文字母和下画线（_）
\W	匹配所有的非单词字符



提示： 上面的7个预定义字符其实很容易记忆——d是digit的意思，代表数字；s是space的意思，代表空白；w是word的意思，代表单词。d、s、w的大写形式恰好匹配与之相反的字符。

有了上面的预定义字符后，接下来就可以创建更强大的正则表达式了。例如：

```
c\\wt  //可以匹配cat、cbt、cct、c0t、c9t等一批字符串
\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d  //匹配如000-000-0000形式的电话号码
```

在一些特殊情况下，例如，若只想匹配a～f的字母，或者匹配除了ab之外的所有小写字母，或者匹配中文字符，上面这些预定义字符就无能为力了，此时就需要使用方括号表达式，方括号表达式有如表7.4所示的几种形式。

表7.4 方括号表达式

方括号表达式	说 明
表示枚举	例如[abc]，表示a、b、c其中任意一个字符；[gz]，表示g、z其中任意一个字符
表示范围：-	例如[a-f]，表示a~f范围内的任意字符；[\\u0041-\\u0056]，表示十六进制字符\u0041到\u0056范围的字符。范围可以和枚举结合使用，如[a-cx-z]，表示a~c、x~z范围内的任意字符
表示求否：^	例如[^abc]，表示非a、b、c的任意字符；[^a-f]，表示不是a~f范围内的任意字符
表示“与”运算：&&	例如[a-z&&[def]]，求a~z和[def]的交集，表示d、e或f [a-z&&[^bc]]，a~z范围内的所有字符，除了b和c之外，即[ad-z] [a-z&&[^m-p]]，a~z范围内的所有字符，除了m~p范围之外的字符，即[a-lq-z]
表示“并”运算	并运算与前面的枚举类似。例如[a-d[m-p]]，表示[a-dm-p]



提示： 方括号表达式比前面的预定义字符灵活多了，几乎可以匹配任何字符。例如，若需要匹配所有的中文字符，就可以利用[\\u0041-\\u0056]形式——因为所有中文字符的Unicode值是连续的，只要找出所有中文字符中最小、最大的Unicode值，就可以利用上面形式来匹配所有的中文字符。

正则表示还支持圆括号表达式，用于将多个表达式组成一个子表达式，圆括号中可以使用或运算符(|)。例如，正则表达式"((public)|(protected)|(private))"用于匹配Java的三个访问控制符其中之一。

除此之外，Java正则表达式还支持如表7.5所示的几个边界匹配符。

表7.5 边界匹配符

边界匹配符	说 明
^	行的开头
$	行的结尾
\b	单词的边界
\B	非单词的边界
\A	输入的开头
\G	前一个匹配的结尾
\Z	输入的结尾，仅用于最后的结束符
\z	输入的结尾

前面例子中需要建立一个匹配000-000-0000形式的电话号码时，使用了\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d正则表达式，这看起来比较烦琐。实际上，正则表达式还提供了数量标识符，正则表达式支持的数量标识符有如下几种模式。

- Greedy（贪婪模式）：数量表示符默认采用贪婪模式，除非另有表示。贪婪模式的表达式会一直匹配下去，直到无法匹配为止。如果你发现表达式匹配的结果与预期的不符，很有可能是因为

——你以为表达式只会匹配前面几个字符，而实际上它是贪婪模式，所以会一直匹配下去。

- Reluctant（勉强模式）：用问号后缀（?）表示，它只会匹配最少的字符。也称为最小匹配模式。
- Possessive（占有模式）：用加号后缀（+）表示，目前只有 Java 支持占有模式，通常比较少用。

三种模式的数量表示符如表 7.6 所示。

表 7.6　三种模式的数量表示符

贪婪模式	勉强模式	占用模式	说　明
X?	X??	X??	X 表达式出现零次或一次
X*	X*?	X*?	X 表达式出现零次或多次
X+	X+?	X+?	X 表达式出现一次或多次
X{n}	X{n}?	X{n}?	X 表达式出现 *n* 次
X{n,}	X{n,}?	X{n,}?	X 表达式最少出现 *n* 次
X{n,m}	X{n,m}?	X{n,m}?	X 表达式最少出现 *n* 次，最多出现 *m* 次

关于贪婪模式和勉强模式的对比，看如下代码：

```
String str = "hello , java!";
// 贪婪模式的正则表达式
System.out.println(str.replaceFirst("\\w*" , "■"));       //输出■ , java!
// 勉强模式的正则表达式
System.out.println(str.replaceFirst("\\w*?" , "■"));      //输出■hello , java!
```

当从"hello , java!"字符串中查找匹配"\\w*"子串时，因为"\w*"使用了贪婪模式，数量表示符（*）会一直匹配下去，所以该字符串前面的所有单词字符都被它匹配到，直到遇到空格，所以替换后的效果是“■, java!”；如果使用勉强模式，数量表示符（*）会尽量匹配最少字符，即匹配 0 个字符，所以替换后的结果是“■hello , java!”。

7.5.2　使用正则表达式

一旦在程序中定义了正则表达式，就可以使用 Pattern 和 Matcher 来使用正则表达式。

Pattern 对象是正则表达式编译后在内存中的表示形式，因此，正则表达式字符串必须先被编译为 Pattern 对象，然后再利用该 Pattern 对象创建对应的 Matcher 对象。执行匹配所涉及的状态保留在 Matcher 对象中，多个 Matcher 对象可共享同一个 Pattern 对象。

因此，典型的调用顺序如下：

```
// 将一个字符串编译成 Pattern 对象
Pattern p = Pattern.compile("a*b");
// 使用 Pattern 对象创建 Matcher 对象
Matcher m = p.matcher("aaaaab");
boolean b = m.matches(); // 返回 true
```

上面定义的 Pattern 对象可以多次重复使用。如果某个正则表达式仅需一次使用，则可直接使用 Pattern 类的静态 matches()方法，此方法自动把指定字符串编译成匿名的 Pattern 对象，并执行匹配，如下所示。

```
boolean b = Pattern.matches("a*b", "aaaaab");  // 返回 true
```

上面语句等效于前面的三条语句。但采用这种语句每次都需要重新编译新的 Pattern 对象，不能重复利用已编译的 Pattern 对象，所以效率不高。

Pattern 是不可变类，可供多个并发线程安全使用。

Matcher 类提供了如下几个常用方法。

- find()：返回目标字符串中是否包含与 Pattern 匹配的子串。
- group()：返回上一次与 Pattern 匹配的子串。
- start()：返回上一次与 Pattern 匹配的子串在目标字符串中的开始位置。
- end()：返回上一次与 Pattern 匹配的子串在目标字符串中的结束位置加 1。
- lookingAt()：返回目标字符串前面部分与 Pattern 是否匹配。

- matches()：返回整个目标字符串与 Pattern 是否匹配。
- reset()，将现有的 Matcher 对象应用于一个新的字符序列。

注意：

在 Pattern、Matcher 类的介绍中经常会看到一个 CharSequence 接口，该接口代表一个字符序列，其中 CharBuffer、String、StringBuffer、StringBuilder 都是它的实现类。简单地说，CharSequence 代表一个各种表示形式的字符串。



通过 Matcher 类的 find()和 group()方法可以从目标字符串中依次取出特定子串（匹配正则表达式的子串），例如互联网的网络爬虫，它们可以自动从网页中识别出所有的电话号码。下面程序示范了如何从大段的字符串中找出电话号码。

程序清单：codes\07\7.5\FindGroup.java

```
public class FindGroup
{
    public static void main(String[] args)
    {
        // 使用字符串模拟从网络上得到的网页源码
        String str = "我想求购一本《疯狂 Java 讲义》，尽快联系我 13500006666"
            + "交朋友，电话号码是 13611125565"
            + "出售二手电脑，联系方式 15899903312";
        // 创建一个 Pattern 对象，并用它建立一个 Matcher 对象
        // 该正则表达式只抓取 13X 和 15X 段的手机号
        // 实际要抓取哪些电话号码，只要修改正则表达式即可
        Matcher m = Pattern.compile("((13\\d)|(15\\d))\\d{8}")
            .matcher(str);
        // 将所有符合正则表达式的子串（电话号码）全部输出
        while(m.find())
        {
            System.out.println(m.group());
        }
    }
}
```

运行上面程序，看到如下运行结果：

```
13500006666
13611125565
15899903312
```

从上面运行结果可以看出，find()方法依次查找字符串中与 Pattern 匹配的子串，一旦找到对应的子串，下次调用 find()方法时将接着向下查找。



提示：

通过程序运行结果可以看出，使用正则表达式可以提取网页上的电话号码，也可以提取邮件地址等信息。如果程序再进一步，可以从网页上提取超链接信息，再根据超链接打开其他网页，然后在其他网页上重复这个过程就可以实现简单的网络爬虫了。

find()方法还可以传入一个 int 类型的参数，带 int 参数的 find()方法将从该 int 索引处向下搜索。start()和 end()方法主要用于确定子串在目标字符串中的位置，如下程序所示。

程序清单：codes\07\7.5\StartEnd.java

```
public class StartEnd
{
    public static void main(String[] args)
    {
        // 创建一个 Pattern 对象，并用它建立一个 Matcher 对象
        String regStr = "Java is very easy!";
        System.out.println("目标字符串是：" + regStr);
        Matcher m = Pattern.compile("\\w+")
            .matcher(regStr);
```

```
        while(m.find())
        {
            System.out.println(m.group() + "子串的起始位置："
                + m.start() + "，其结束位置：" + m.end());
        }
    }
}
```

上面程序使用 find()、group()方法逐项取出目标字符串中与指定正则表达式匹配的子串，并使用 start()、end()方法返回子串在目标字符串中的位置。运行上面程序，看到如下运行结果：

```
目标字符串是：Java is very easy!
Java 子串的起始位置：0，其结束位置：4
is 子串的起始位置：5，其结束位置：7
very 子串的起始位置：8，其结束位置：12
easy 子串的起始位置：13，其结束位置：17
```

matches()和 lookingAt()方法有点相似，只是 matches()方法要求整个字符串和 Pattern 完全匹配时才返回 true，而 lookingAt()只要字符串以 Pattern 开头就会返回 true。reset()方法可将现有的 Matcher 对象应用于新的字符序列。看如下例子程序。

程序清单：codes\07\7.5\MatchesTest.java

```
public class MatchesTest
{
    public static void main(String[] args)
    {
        String[] mails =
        {
            "kongyeeku@163.com" ,
            "kongyeeku@gmail.com",
            "ligang@crazyit.org",
            "wawa@abc.xx"
        };
        String mailRegEx = "\\w{3,20}@\\w+\\.(com|org|cn|net|gov)";
        Pattern mailPattern = Pattern.compile(mailRegEx);
        Matcher matcher = null;
        for (String mail : mails)
        {
            if (matcher == null)
            {
                matcher = mailPattern.matcher(mail);
            }
            else
            {
                matcher.reset(mail);
            }
            String result = mail + (matcher.matches() ? "是" : "不是")
                + "一个有效的邮件地址！";
            System.out.println(result);
        }
    }
}
```

上面程序创建了一个邮件地址的 Pattern，接着用这个 Pattern 与多个邮件地址进行匹配。当程序中的 Matcher 为 null 时，程序调用 matcher()方法来创建一个 Matcher 对象，一旦 Matcher 对象被创建，程序就调用 Matcher 的 reset()方法将该 Matcher 应用于新的字符序列。

从某个角度来看，Matcher 的 matches()、lookingAt()和 String 类的 equals()、startsWith()有点相似。区别是 String 类的 equals()和 startsWith()都是与字符串进行比较，而 Matcher 的 matches()和 lookingAt()则是与正则表达式进行匹配。

事实上，String 类里也提供了 matches()方法，该方法返回该字符串是否匹配指定的正则表达式。例如：

```
"kongyeeku@163.com".matches("\\w{3,20}@\\w+\\.(com|org|cn|net|gov)"); // 返回 true
```

除此之外，还可以利用正则表达式对目标字符串进行分割、查找、替换等操作，看如下例子程序。

程序清单：codes\07\7.5\ReplaceTest.java

```
public class ReplaceTest
{
    public static void main(String[] args)
    {
        String[] msgs =
        {
            "Java has regular expressions in 1.4",
            "regular expressions now expressing in Java",
            "Java represses oracular expressions"
        };
        Pattern p = Pattern.compile("re\\w*");
        Matcher matcher = null;
        for (int i = 0 ; i < msgs.length ; i++)
        {
            if (matcher == null)
            {
                matcher = p.matcher(msgs[i]);
            }
            else
            {
                matcher.reset(msgs[i]);
            }
            System.out.println(matcher.replaceAll("哈哈:)"));
        }
    }
}
```

上面程序使用了 Matcher 类提供的 replaceAll()把字符串中所有与正则表达式匹配的子串替换成"哈哈:)"，实际上，Matcher 类还提供了一个 replaceFirst()，该方法只替换第一个匹配的子串。运行上面程序，会看到字符串中所有以“re”开头的单词都会被替换成“哈哈:)”。

实际上，String 类中也提供了 replaceAll()、replaceFirst()、split()等方法。下面的例子程序直接使用 String 类提供的正则表达式功能来进行替换和分割。

程序清单：codes\07\7.5\StringReg.java

```
public class StringReg
{
    public static void main(String[] args)
    {
        String[] msgs =
        {
            "Java has regular expressions in 1.4",
            "regular expressions now expressing in Java",
            "Java represses oracular expressions"
        };
        for (String msg : msgs)
        {
            System.out.println(msg.replaceFirst("re\\w*", "哈哈:)"));
            System.out.println(Arrays.toString(msg.split(" ")));
        }
    }
}
```

上面程序只使用 String 类的 replaceFirst()和 split()方法对目标字符串进行了一次替换和分割。运行上面程序，会看到如图 7.6 所示的运行效果。

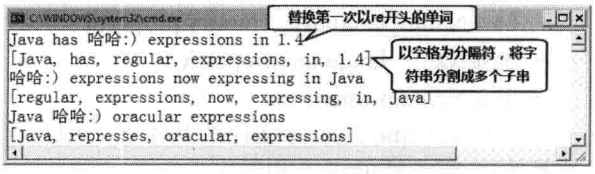


图 7.6 直接使用 String 类提供的正则表达式支持

正则表达式是一个功能非常灵活的文本处理工具，增加了正则表达式支持后的 Java，可以不再使用 StringTokenizer 类（也是一个处理字符串的工具，但功能远不如正则表达式强大）即可进行复杂的字符串处理。

7.6 国际化与格式化

全球化的 Internet 需要全球化的软件。全球化软件，意味着同一种版本的产品能够容易地适用于不同地区的市场，软件的全球化意味着国际化和本地化。当一个应用需要在全球范围使用时，就必须考虑在不同的地域和语言环境下的使用情况，最简单的要求就是用户界面上的信息可以用本地化语言来显示。

国际化是指应用程序运行时，可根据客户端请求来自的国家/地区、语言的不同而显示不同的界面。例如，如果请求来自于中文操作系统的客户端，则应用程序中的各种提示信息错误和帮助等都使用中文文字；如果客户端使用英文操作系统，则应用程序能自动识别，并做出英文的响应。

引入国际化的目的是为了提供自适应、更友好的用户界面，并不需要改变程序的逻辑功能。国际化的英文单词是 Internationalization，因为这个单词太长了，有时也简称 I18N，其中 I 是这个单词的第一个字母，18 表示中间省略的字母个数，而 N 代表这个单词的最后一个字母。

一个国际化支持很好的应用，在不同的区域使用时，会呈现出本地语言的提示。这个过程也被称为 Localization，即本地化。类似于国际化可以称为 I18N，本地化也可以称为 L10N。

Java 8 国际化支持升级到了 Unicode 6.2.0 字符集，因此提供了对不同国家、不同语言的支持，它已经具有了国际化和本地化的特征及 API，因此 Java 程序的国际化相对比较简单。尽管 Java 开发工具为国际化和本地化的工作提供了一些基本的类，但还是有一些对于 Java 应用程序的本地化和国际化来说较困难的工作，例如：消息获取，编码转换，显示布局和数字、日期、货币的格式等。

当然，一个优秀的全球化软件产品，对国际化和本地化的要求远远不止于此，甚至还包括用户提交数据的国际化和本地化。

7.6.1 Java 国际化的思路

Java 程序的国际化思路是将程序中的标签、提示等信息放在资源文件中，程序需要支持哪些国家、语言环境，就对应提供相应的资源文件。资源文件是 key-value 对，每个资源文件中的 key 是不变的，但 value 则随不同的国家、语言而改变。图 7.7 显示了 Java 程序国际化的思路。

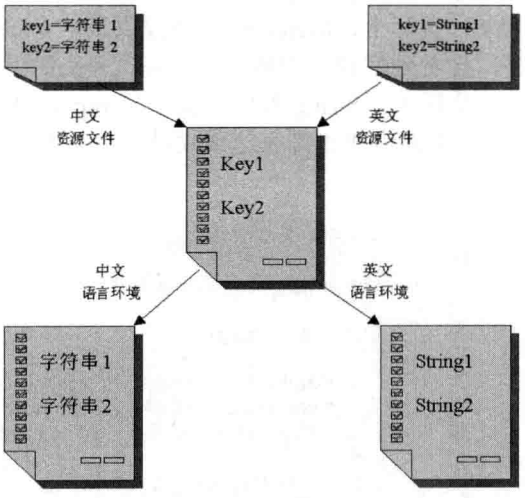


图 7.7　Java 程序国际化的思路

Java 程序的国际化主要通过如下三个类完成。

- java.util.ResourceBundle：用于加载国家、语言资源包。
- java.util.Locale：用于封装特定的国家/区域、语言环境。
- java.text.MessageFormat：用于格式化带占位符的字符串。

为了实现程序的国际化，必须先提供程序所需要的资源文件。资源文件的内容是很多 key-value 对，其中 key 是程序使用的部分，而 value 则是程序界面的显示字符串。

资源文件的命名可以有如下三种形式。

- baseName_language_country.properties
- baseName_language.properties
- baseName.properties

其中 baseName 是资源文件的基本名，用户可随意指定；而 language 和 country 都不可随意变化，必须是 Java 所支持的语言和国家。

7.6.2 Java 支持的国家和语言

事实上，Java 不可能支持所有的国家和语言，如果需要获取 Java 所支持的国家和语言，则可调用 Locale 类的 getAvailableLocales()方法，该方法返回一个 Locale 数组，该数组里包含了 Java 所支持的国家和语言。

下面的程序简单地示范了如何获取 Java 所支持的国家和语言。

程序清单：codes\07\7.6\LocaleList.java

```
public class LocaleList
{
    public static void main(String[] args)
    {
        // 返回 Java 所支持的全部国家和语言的数组
        Locale[] localeList = Locale.getAvailableLocales();
        // 遍历数组的每个元素，依次获取所支持的国家和语言
        for (int i = 0; i < localeList.length ; i++ )
        {
            // 输出所支持的国家和语言
            System.out.println(localeList[i].getDisplayCountry()
                + "=" + localeList[i].getCountry()+ " "
                + localeList[i].getDisplayLanguage()
                + "=" + localeList[i].getLanguage());
        }
    }
}
```

程序的运行结果如图 7.8 所示。

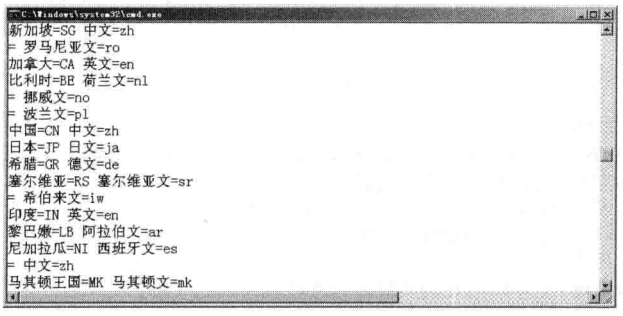


图 7.8　Java 所支持的国家和语言

通过该程序就可获得 Java 所支持的国家/语言环境。



提示：

虽然可以通过查阅相关资料来获取 Java 语言所支持的国家/语言环境，但如果这些资料不能随手可得，则可以通过上面程序来获得 Java 语言所支持的国家/语言环境。

7.6.3　完成程序国际化

对于如下最简单的程序：

```
public class RawHello
{
    public static void main(String[] args)
    {
        System.out.println("Hello World");
    }
}
```

这个程序的执行结果也很简单——肯定是打印出简单的“Hello World”字符串，不管在哪里执行都不会有任何改变！为了让该程序支持国际化，肯定不能让程序直接输出“Hello World”字符串，这种写法直接输出一个字符串常量，永远不会有任何改变。为了让程序可以输出不同的字符串，此处绝不可使用该字符串常量。

为了让上面输出的字符串常量可以改变，可以将需要输出的各种字符串（不同的国家/语言环境对应不同的字符串）定义在资源包中。

为上面程序提供如下两个文件。

第一个文件：mess.properties，该文件的内容为：

```
#资源文件的内容是 key-value 对
hello=你好！
```

第二个文件：mess_en_US.properties，该文件的内容为：

```
#资源文件的内容是 key-value 对
hello=Welcome You!
```

对于包含非西欧字符的资源文件，Java 提供了一个工具来处理该文件：native2ascii，这个工具可以在%JAVA_HOME%/bin 路径下找到。使用该工具的语法格式如下：

```
native2ascii 源资源文件 目的资源文件
```

在命令窗口输入如下命令：

```
native2ascii mess.properties mess_zh_CN.properties
```

上面的命令将生成一个 mess_zh_CN.properties 文件，该文件才是程序需要的资源文件，该文件看上去包含很多乱码，其实是非西欧字符的 Unicode 编码方式。

看到这两份文件文件名的 baseName 是相同的：mess。前面已经介绍了资源文件的三种命名方式，其中 baseName 后面的国家、语言必须是 Java 所支持的国家、语言组合。将上面的 Java 程序修改成如下形式。

程序清单：codes\07\7.6\Hello.java

```
public class Hello
{
    public static void main(String[] args)
    {
        // 取得系统默认的国家/语言环境
        Locale myLocale = Locale.getDefault(Locale.Category.FORMAT);
        // 根据指定的国家/语言环境加载资源文件
        ResourceBundle bundle = ResourceBundle
            .getBundle("mess" , myLocale);
        // 打印从资源文件中取得的消息
        System.out.println(bundle.getString("hello"));
    }
}
```

上面程序中的打印语句不再是直接打印“Hello World”字符串，而是打印从资源包中读取的信息。如果在中文环境下运行该程序，将打印“你好!”；如果在“控制面板”中将机器的语言环境设置成美国，然后再次运行该程序，将打印“Welcome You!”字符串。

从上面程序可以看出，如果希望程序完成国际化，只需要将不同的国家/语言（Locale）的提示信息分别以不同的文件存放即可。例如，简体中文的语言资源文件就是 Xxx_zh_CN.properties 文件，而美国英语的语言资源文件就是 Xxx_en_US.properties 文件。

Java 程序国际化的关键类是 ResourceBundle，它有一个静态方法：getBundle(String baseName，Locale locale)，该方法将根据 Locale 加载资源文件，而 Locale 封装了一个国家、语言，例如，简体中文环境可以用简体中文的 Locale 代表，美国英语环境可以用美国英语的 Locale 代表。

从上面资源文件的命名中可以看出，不同国家、语言环境的资源文件的 baseName 是相同的，即 baseName 为 mess 的资源文件有很多个，不同的国家、语言环境对应不同的资源文件。

例如，通过如下代码来加载资源文件。

```
// 根据指定的国家/语言环境加载资源文件
ResourceBundle bundle = ResourceBundle.getBundle("mess" , myLocale);
```

上面代码将会加载 baseName 为 mess 的系列资源文件之一，到底加载其中的哪个资源文件，则取决于 myLocale；对于简体中文的 Locale，则加载 mess_zh_CN.properties 文件。

一旦加载了该文件后，该资源文件的内容就是多个 key-value 对，程序就根据 key 来获取指定的信息，例如获取 key 为 hello 的消息，该消息是“你好!”——这就是 Java 程序国际化的过程。

如果对于美国英语的 Locale，则加载 mess_en_US.properties 文件，该文件中 key 为 hello 的消息是“Welcome You!”。

Java 程序国际化的关键类是 ResourceBundle 和 Locale，ResourceBundle 根据不同的 Locale 加载语言资源文件，再根据指定的 key 取得已加载语言资源文件中的字符串。

7.6.4 使用 MessageFormat 处理包含占位符的字符串

上面程序中输出的消息是一个简单消息，如果需要输出的消息中必须包含动态的内容，例如，这些内容必须是从程序中取得的。比如如下字符串：

```
你好，yeeku！今天是2014-5-30 下午11:55。
```

在上面的输出字符串中，yeeku 是浏览者的名字，必须动态改变，后面的时间也必须动态改变。在这种情况下，可以使用带占位符的消息。例如，提供一个 myMess_en_US.properties 文件，该文件的内容如下：

```
msg=Hello,{0}!Today is {1}.
```

提供一个 myMess.properties 文件，该文件的内容如下：

```
msg=你好，{0}！今天是{1}。
```

注意：

上面的资源文件必须使用 native2ascii 命令来处理，将处理后的文件保存为 myMess_zh_CN.properties。



当程序直接使用 ResourceBundle 的 getString()方法来取出 msg 对应的字符串时，在简体中文环境下得到“你好，{0}！今天是{1}。”字符串，这显然不是需要的结果，程序还需要为{0}和{1}两个占位符赋值。此时需要使用 MessageFormat 类，该类包含一个有用的静态方法。

- format(String pattern , Object... values)：返回后面的多个参数值填充前面的 pattern 字符串，其中 pattern 字符串不是正则表达式，而是一个带占位符的字符串。

借助于上面的 MessageFormat 类的帮助，将国际化程序修改成如下形式。

程序清单：codes\07\7.6\HelloArg.java

```
public class HelloArg
{
    public static void main(String[] args)
    {
        // 定义一个 Locale 变量
        Locale currentLocale = null;
        // 如果运行程序指定了两个参数
        if (args.length == 2)
        {
            // 使用运行程序的两个参数构造 Locale 实例
            currentLocale = new Locale(args[0] , args[1]);
        }
        else
        {
            // 否则直接使用系统默认的 Locale
            currentLocale = Locale.getDefault(Locale.Category.FORMAT);
        }
        // 根据 Locale 加载语言资源
        ResourceBundle bundle = ResourceBundle
            .getBundle("myMess" , currentLocale);
        // 取得已加载的语言资源文件中 msg 对应消息
        String msg = bundle.getString("msg");
        // 使用 MessageFormat 为带占位符的字符串传入参数
        System.out.println(MessageFormat.format(msg
            , "yeeku" , new Date()));
    }
}
```

从上面的程序中可以看出，对于带占位符的消息字符串，只需要使用 MessageFormat 类的 format()方法为消息中的占位符指定参数即可。

7.6.5 使用类文件代替资源文件

除了使用属性文件作为资源文件外，Java 也允许使用类文件代替资源文件，即将所有的 key-value 对存入 class 文件，而不是属性文件。

使用类文件来代替资源文件必须满足如下条件。

- 该类的类名必须是 baseName_language_country，这与属性文件的命名相似。
- 该类必须继承 ListResourceBundle，并重写 getContents()方法，该方法返回 Object 数组，该数组的每一项都是 key-value 对。

下面的类文件可以代替上面的属性文件。

程序清单：codes\07\7.6\myMess_zh_CN.java

```
public class myMess_zh_CN extends ListResourceBundle
{
    // 定义资源
    private final Object myData[][]=
    {
        {"msg","{0}，你好！今天的日期是{1}"}
    };
    // 重写 getContents()方法
    public Object[][] getContents()
    {
        // 该方法返回资源的 key-value 对
        return myData;
    }
}
```

上面文件是一个简体中文语言环境的资源文件，该文件可以代替 myMess_zh_CN.properties 文件；如果需要代替美国英语语言环境的资源文件，则还应该提供一个 myMess_en_US 类。

如果系统同时存在资源文件、类文件，系统将以类文件为主，而不会调用资源文件。对于简体中文的 Locale，ResourceBundle 搜索资源文件的顺序是：

（1）baseName_zh_CN.class
（2）baseName_zh_CN.properties
（3）baseName_zh.class
（4）baseName_zh.properties
（5）baseName.class
（6）baseName.properties

系统按上面的顺序搜索资源文件，如果前面的文件不存在，才会使用下一个文件。如果一直找不到对应的文件，系统将抛出异常。

7.6.6 使用 NumberFormat 格式化数字

MessageFormat 是抽象类 Format 的子类，Format 抽象类还有两个子类：NumberFormat 和 DateFormat，它们分别用以实现数值、日期的格式化。NumberFormat、DateFormat 可以将数值、日期转换成字符串，也可以将字符串转换成数值、日期。图 7.9 显示了 NumberFormat 和 DateFormat 的主要功能。

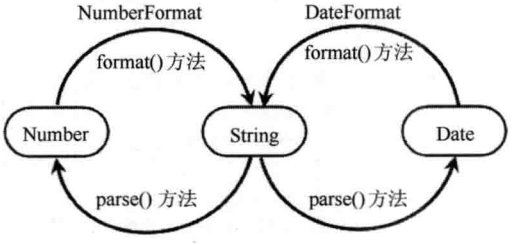


图 7.9 NumberFormat 和 DateFormat 的主要功能

NumberFormat 和 DateFormat 都包含了 format()和 parse()方法，其中 format()用于将数值、日期格式化成字符串，parse()用于将字符串解析成数值、日期。

NumberFormat 也是一个抽象基类，所以无法通过它的构造器来创建 NumberFormat 对象，它提供了如下几个类方法来得到 NumberFormat 对象。

- getCurrencyInstance()：返回默认 Locale 的货币格式器。也可以在调用该方法时传入指定的 Locale，则获取指定 Locale 的货币格式器。

- getIntegerInstance()：返回默认 Locale 的整数格式器。也可以在调用该方法时传入指定的 Locale，则获取指定 Locale 的整数格式器。
- getNumberInstance()：返回默认 Locale 的通用数值格式器。也可以在调用该方法时传入指定的 Locale，则获取指定 Locale 的通用数值格式器。
- getPercentInstance()：返回默认 Locale 的百分数格式器。也可以在调用该方法时传入指定的 Locale，则获取指定 Locale 的百分数格式器。

一旦取得了 NumberFormat 对象后，就可以调用它的 format()方法来格式化数值，包括整数和浮点数。如下例子程序示范了 NumberFormat 的三种数字格式化器的用法。

程序清单：codes\07\7.6\NumberFormatTest.java

```
public class NumberFormatTest
{
    public static void main(String[] args)
    {
        // 需要被格式化的数字
        double db = 1234000.567;
        // 创建四个 Locale，分别代表中国、日本、德国、美国
        Locale[] locales = {Locale.CHINA, Locale.JAPAN
            , Locale.GERMAN,  Locale.US};
        NumberFormat[] nf = new NumberFormat[12];
        // 为上面四个 Locale 创建 12 个 NumberFormat 对象
        // 每个 Locale 分别有通用数值格式器、百分数格式器、货币格式器
        for (int i = 0 ; i < locales.length ; i++)
        {
            nf[i * 3] = NumberFormat.getNumberInstance(locales[i]);
            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。

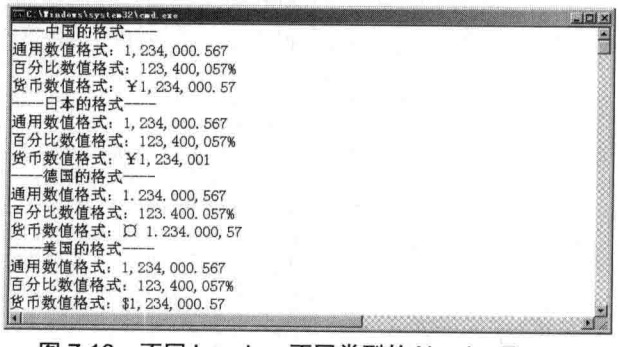


图 7.10　不同 Locale、不同类型的 NumberFormat

7.6.7 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\7.6\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 = "2014-12-12";
String str2 = "2014年12月10日";
// 下面输出 Fri Dec 12 00:00:00 CST 2014
System.out.println(DateFormat.getDateInstance().parse(str1));
// 下面输出 Wed Dec 10 00:00:00 CST 2014
System.out.println(DateFormat.getDateInstance(LONG).parse(str2));
// 下面抛出 ParseException 异常
System.out.println(DateFormat.getDateInstance().parse(str2));
```

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"2014 年 12 月 10 日"是一个 LONG 样式的日期字符串，必须用 LONG 样式的 DateFormat 实例解析，否则将抛出异常。

7.6.8 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\7.6\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 2014 年中第 101 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###三月##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。

7.7 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：

在 DateTimeFormatter 的官方 API 文档中，会看到如下两行示例代码：

```
String text = date.toString(formatter);
LocalDate date = LocalDate.parse(text, formatter);
```

上面第一行代码使用了一个 date 对象，但该 date 对象到底是哪个类的实例？该文档语焉不详，而 java.util.Date、LocalDate、LocalTime、LocalDateTime 等类似乎都没有带参数的 toString()方法，官方文档似乎有错误，读者请小心。



7.7.1 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\7.7\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.ofLocalizedTime(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。

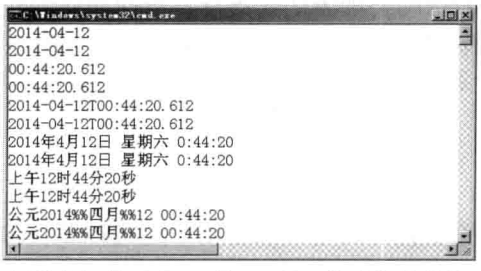


图 7.12 DateTimeFormatter 格式化的效果

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。

7.7.2 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\7.7\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$$$四月$$$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。

7.8 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
- 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。

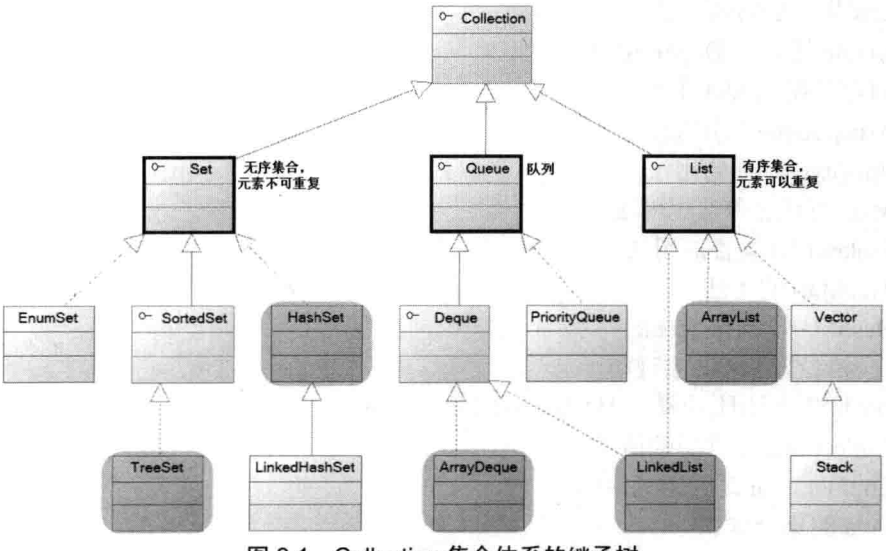


图 8.1　Collection 集合体系的继承树

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。

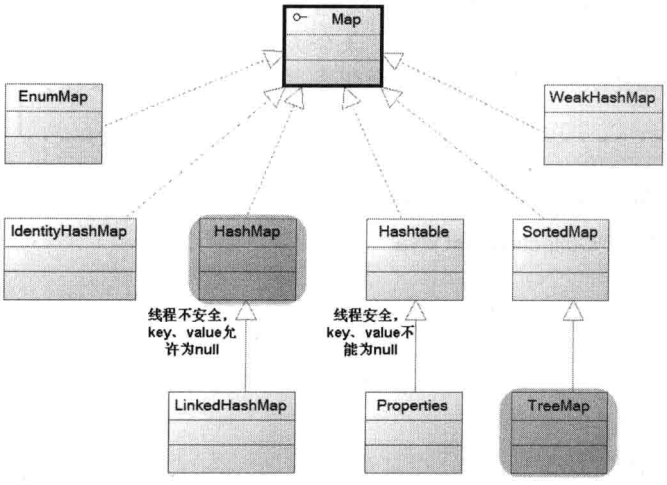


图 8.2 Map 体系的继承树

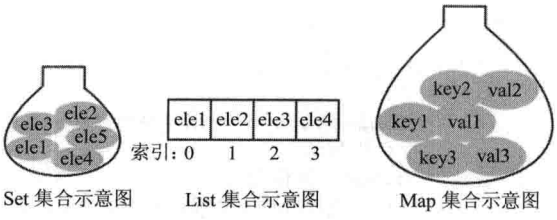


图 8.3 三种集合示意图

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 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

除了可以使用 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 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：

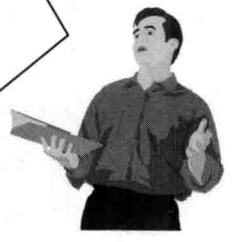
如果需要把某个类的对象保存到 HashSet 集合中，重写这个类的 equals()方法和 hashCode()方法时，应该尽量保证两个对象通过 equals()方法比较返回 true 时，它们的 hashCode()方法返回值也相等。



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。

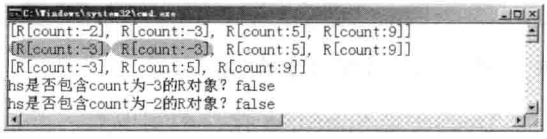


图 8.4 HashSet 集合中出现重复的元素

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。

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());
        ts.add(new Err());  // ①
    }
}
```

上面程序试图向 TreeSet 集合中添加两个 Err 对象，添加第一个对象时，TreeSet 里没有任何元素，所以不会出现任何问题；当添加第二个 Err 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——如果其对应的类没有实现 Comparable 接口，则会引发 ClassCastException 异常。因此，上面程序将会在①代码处引发该异常。

注意：

向 TreeSet 集合中添加元素时，只有第一个元素无须实现 Comparable 接口，后面添加的所有元素都必须实现 Comparable 接口。当然这也不是一种好做法，当试图从 TreeSet 中取出元素时，依然会引发 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。

总结起来一句话：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。

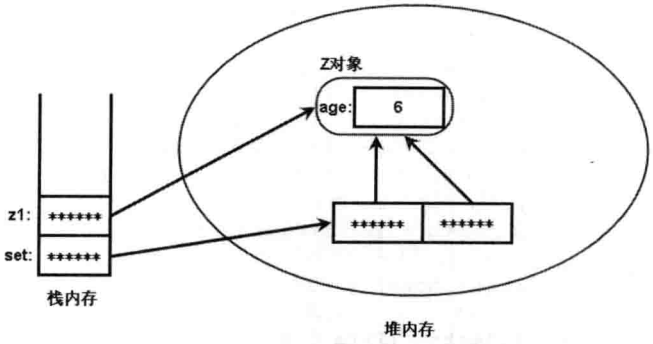


图 8.5 TreeSet 及 Z 对象在内存中的存储示意图

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);
    }
}
```

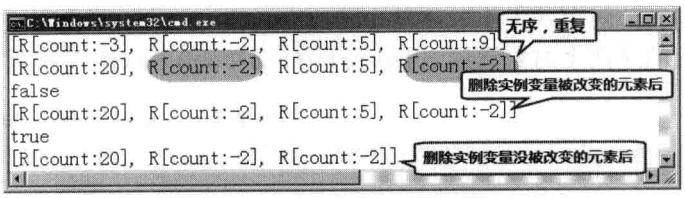


图 8.6　TreeSet 中出现重复元素

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 对象(age:9), M 对象(age:5), M 对象(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。

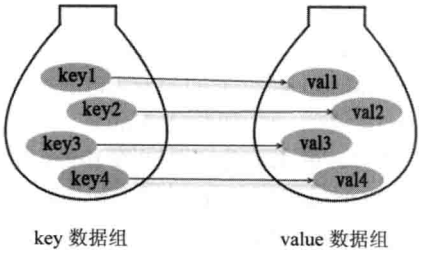


图 8.7 分开看 Map 的 key 组和 value 组

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 除了为 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。



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。

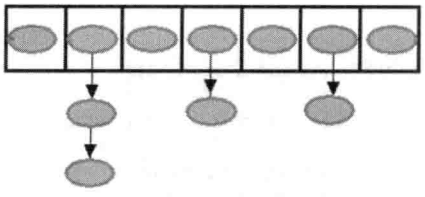


图 8.8 hash 表中存储元素的示意图

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，读者可参考光盘代码。

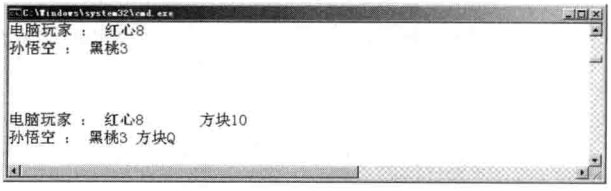


图 8.9 控制台梭哈游戏界面

运行上面程序，即可看到如图 8.9 所示的界面。

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。

8.9 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
- 设定类型形参的上限
- 在方法签名中定义类型形参
- 泛型方法和类型通配符的区别与联系
- 设定类型通配符的下限
- 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 JDK 1.5 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 7 泛型的“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。

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类型形参，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类型形参，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类型形参，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、类时声明类型形参，类型形参在整个接口、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类型形参。

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 类型形参定义实例变量
    private T info;
    public Apple(){}
    // 下面方法中使用 T 类型形参来定义构造器
    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（不要理会这个类型形参是否具有实际意义），使用 Apple<T>类时就可为 T 类型形参传入实际类型，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、类时使用了类型形参，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、父类时不能再包含类型形参。例如，下面代码就是错误的。

```
// 定义类 A 继承 Apple 类，Apple 类不能跟类型形参
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、方法时可以声明类型形参，使用类、接口、方法时应该为类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、接口时也可以不为类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
```

如果从 Apple<String>类派生子类，则在 Apple 类中所有使用 T 类型形参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参数，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）。

不管为泛型的类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类型形参。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类型形参
    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类型形参
    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。

注意：

数组和泛型有所不同，假设 Foo 是 Bar 的一个子类型（子类或者子接口），那么 Foo[]依然是 Bar[]的子类型；但 G<Foo>不是 G<Bar>的子类型。



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。

注意：

上面程序中使用的 List<?>，其实这种写法可以适应于任何支持泛型声明的接口和类，比如写成 Set<?>、Collection<?>、Map<?,?>等。



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<?>，把 Canvas 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
    public void drawAll(List<?> shapes)
    {
        for (Object obj : shapes)
        {
            Shape s = (Shape)obj;
            s.draw(this);
        }
    }
}
```

上面程序使用了通配符来表示所有的类型。上面的 drawAll()方法可以接受 List<Circle>对象作为参数，问题是上面的方法实现体显得极为臃肿而烦琐：使用了泛型还需要进行强制类型转换。

实际上需要一种泛型表示方法，它可以表示所有 Shape 泛型 List 的父类。为了满足这种需求，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：

```
// 它表示所有 Shape 泛型 List 的父类
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。

9.3.3 设定类型形参的上限

Java 泛型不仅允许在使用通配符形参时设定上限，而且可以在定义类型形参时设定上限，用于表示传给该类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 类的类型形参的上限是 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：类型形参 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，程序需要为类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），表明该类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，为类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，如果需要为类型形参指定类上限，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、接口时可以使用类型形参，在该类的方法定义和成员变量定义、接口的方法定义中，这些类型形参可被当成普通类型来用。在另外一些情况下，定义类、接口时没有使用类型形参，但定义方法时想自己定义类型形参，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类型形参。泛型方法的用法格式如下：

```
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类型形参声明，类型形参声明以尖括号括起来，多个类型形参之间以逗号（,）隔开，所有的类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 类型形参
    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 类型形参，这个 T 类型形参就可以在该方法内当成普通类型使用。与接口、类声明中定义的类型形参不同的是，方法声明中定义的形参只能在该方法里使用，而接口、类声明中定义的类型形参则可以在整个接口、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类型形参的数据类型，因为编译器根据实参推断类型形参的值，它通常推断出最直接的类型参数。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 类型形参代表的实际类型是 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 类型形参代表了 Number 类型。

为了让编译器能准确地推断出泛型方法中类型形参的类型，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 类型形参
    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类型形参的类型。

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，这时定义类型形参时设定上限（其中 E 是 Collection 接口里定义的类型形参，在该接口里 E 可当成普通类型使用）。

上面两个方法中类型形参 T 只使用了一次，类型形参 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。

泛型方法允许类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类型形参——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。

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。但注意上面的类型形参 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，那类型形参 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类型形参）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类型形参）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；但泛型方法中的类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类型形参一样，Java 也允许在构造器签名中声明类型形参，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”类型形参的类型，而且程序员也可以显式地为构造器中的类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 参数为 String
        new Foo("疯狂 Java 讲义");
        // 泛型构造器中的 T 参数为 Integer
        new Foo(200);
        // 显式指定泛型构造器中的 T 参数为 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 参数为 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类型形参 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类型形参 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类型形参的实际类型，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类型形参是 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　设定通配符下限

假设自己实现一个工具方法：实现将 src 集合里的元素复制到 dest 集合里的功能，因为 dest 集合可以保存 src 集合里的所有元素，所以 dest 集合元素的类型应该是 src 集合元素类型的父类。为了表示两个参数之间的类型依赖，考虑同时使用通配符、泛型参数来实现该方法。代码如下：

```
public static <T> void copy(Collection<T> dest , Collection<? extends T> src)
{
    for (T ele : src)
    {
        dest.add(ele);
    }
}
```

上面方法实现了前面的功能。现在假设该方法需要一个返回值，返回最后一个被复制的元素，则可以把上面方法改为如下形式：

```
public static <T> T copy(Collection<T> dest , Collection<? extends T> src)
{
    T last = null;
    for (T ele : src)
    {
        last = ele;
        dest.add(ele);
    }
    return last;
}
```

表面上看起来，上面方法实现了这个功能，实际上有一个问题：当遍历 src 集合的元素时，src 元素的类型是不确定的（只可以肯定它是 T 的子类），程序只能用 T 来笼统地表示各种 src 集合的元素类型。例如如下代码：

```
List<Number> ln = new ArrayList<>();
List<Integer> li = new ArrayList<>();
// 下面代码引起编译错误
Integer last = copy(ln , li);
```

上面代码中 ln 的类型是 List<Number>，与 copy()方法签名的形参类型进行对比即得到 T 的实际类型是 Number，而不是 Integer 类型——即 copy()方法的返回值也是 Number 类型，而不是 Integer 类型，但实际上最后一个复制元素的元素类型一定是 Integer。也就是说，程序在复制集合元素的过程中，丢失了 src 集合元素的类型。

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是前者的父类即可。为了表达这种约束关系，Java 允许设定通配符的下限：<? super Type>，这个通配符表示它必须是 Type 本身，或是 Type 的父类。下面程序采用设定通配符下限的方式改写了前面的 copy()方法。

程序清单：codes\09\9.4\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;
            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。

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 类时的类型形参
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。如下程序所示。

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，具体请查阅 Java 的 API 文档。

➤➤ 9.4.5 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 T 类型参数的类型是 Number；也可以匹配②号 copy()方法，此时 T 参数的类型是 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。

9.4.6 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类型参数的目标类型。
- 可在方法调用链中，将推断得到的类型参数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类型参数为 String
        MyUtil<String> ls = MyUtil.nil();
        // 无须使用下面语句在调用 nil()方法时指定类型参数的类型
        MyUtil<String> mu = MyUtil.<String>nil();
        // 可调用 cons()方法所需的参数类型来推断类型参数为 Integer
        MyUtil.cons(42, MyUtil.nil());
        // 无须使用下面语句在调用 nil()方法时指定类型参数的类型
        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()方法时显式指定类型参数为 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类型参数为 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()方法时显式指定类型参数为 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）推断出此处的类型参数为 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()方法时类型参数为 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类型参数，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参数。如果没有为这个泛型类指定实际的类型参数，则该类型参数被称作 raw type（原始类型），默认是声明该类型参数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类型参数的上限（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，其类型形参的上限是 Number，这个类型形参用来定义 Apple 类的 size 变量。程序在①处创建了一个 Apple 对象，该 Apple 对象传入了 Integer 作为类型形参的值，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 的类型形参的上限是 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 对象保留了集合元素的类型信息。当把这个 List 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，将引发运行时异常。

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((Sting)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类型变量或类型形参，除非是无上限的类型通配符。但可以声明元素类型包含类型变量或类型形参的数组。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 List<String>[10];
// 将 lsa 向上转型为 Object[]类型的变量
Object[] oa = (Object[])lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(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：

```
// 下面代码编译时有“[unchecked] 未经检查的转换”警告
List<String>[] lsa = new ArrayList[10];
Object[] oa = lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(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。如下代码所示。

```
List<?>[] lsa = new ArrayList<?>[10];
Object[] oa = lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li;
// 下面代码引发 ClassCastException 异常
String s = (String)lsa[1].get(0);
```

编译上面代码不会发出任何警告，运行上面程序将在粗体字行引发 ClassCastException 异常。因为程序需要将 lsa 的第一个数组元素的第一个集合元素强制类型转换为 String 类型，所以程序应该自己通过 instanceof 运算符来保证它的数据类型。即改为如下形式：

```
List<?>[] lsa = new ArrayList<?>[10];
Object[] oa = lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(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，创建元素类型是类型变量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 JDK 1.5 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 JDK 1.5 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类型形参，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。

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，除了传统的像 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。

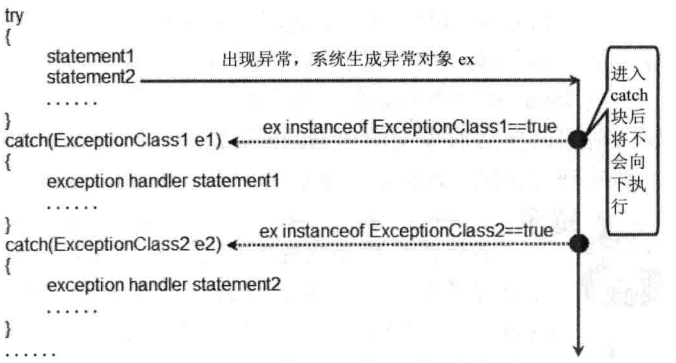


图 10.1　Java 异常捕获流程示意图

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。

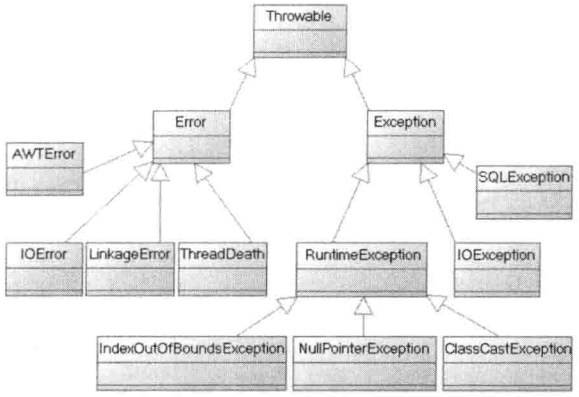


图 10.2 Java 常见的异常类之间的继承关系

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 提供的多异常捕获

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。

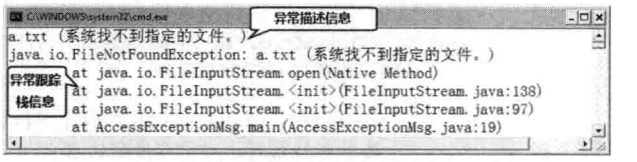


图 10.3　访问异常信息

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。

注意：

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。

注意：

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 7 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。

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。

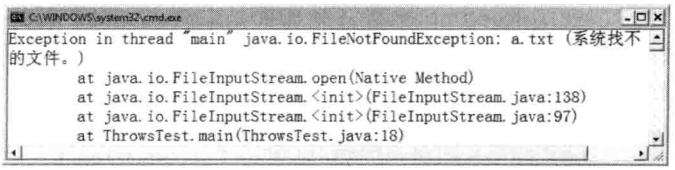


图 10.4 main 方法声明把异常交给 JVM 处理

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");
    }
}
```

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排除 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。

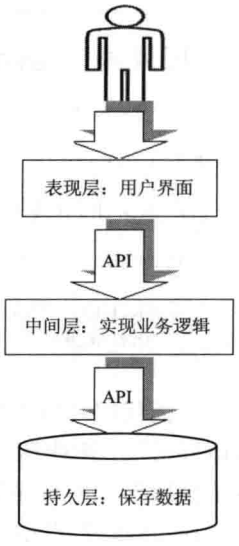


图 10.5　分层结构示意图

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 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 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。

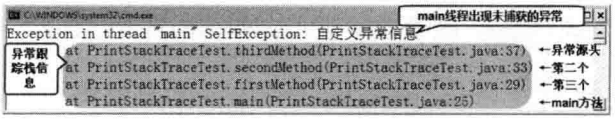


图 10.6　异常的跟踪栈信息

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）。

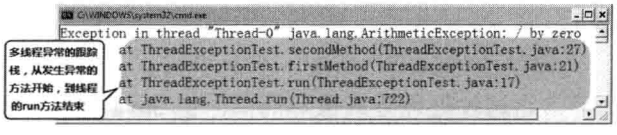


图 10.7 多线程的异常跟踪栈

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 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。



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。

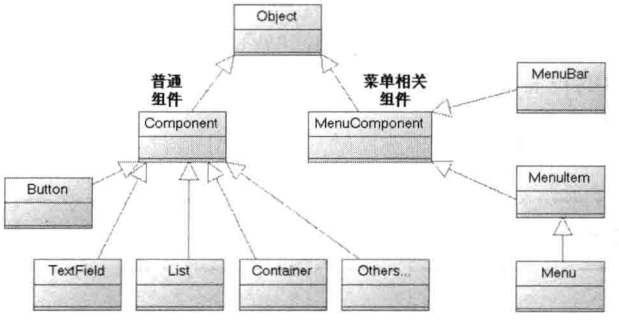


图 11.1　AWT 图形组件之间的继承关系

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。

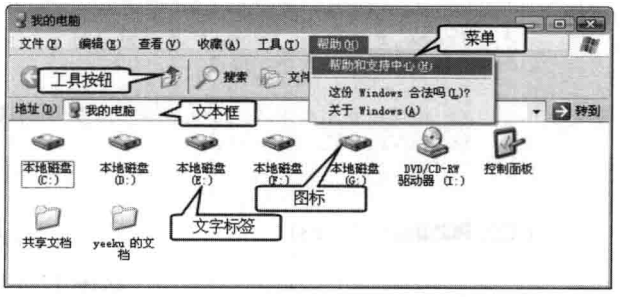


图 11.2　窗口的“分解”

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。

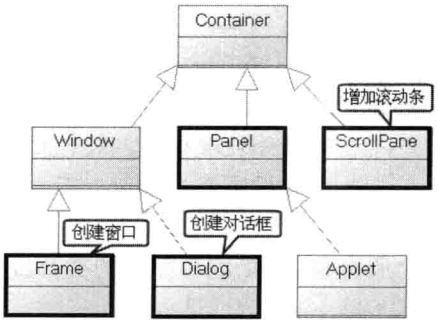


图 11.3　AWT 容器的继承关系

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);
    }
}
```

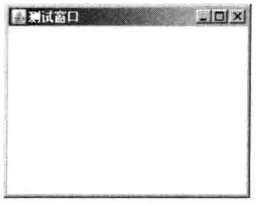


图 11.4　通过 Frame 创建的空白窗口

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。

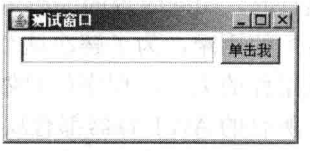


图 11.5 使用 Panel 盛装文本框和按钮

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。

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);
    }
}
```

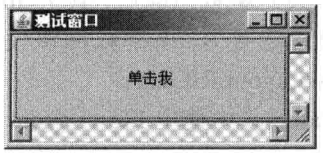


图 11.6 ScrollPane 容器的效果

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);
    }
}
```

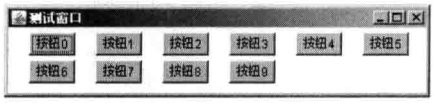


图 11.7 FlowLayout 布局管理器

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。

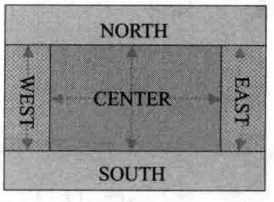


图 11.8 BorderLayout 布局管理器的布局示意图

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。

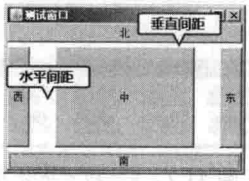
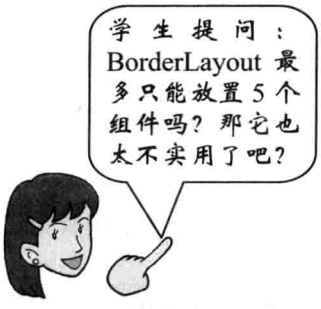


图 11.9　BorderLayout 布局管理器的效果





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。

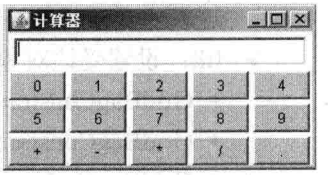


图 11.11　使用 GridLayout 布局管理器的效果



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();
constainer.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。

```
constainer.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）。

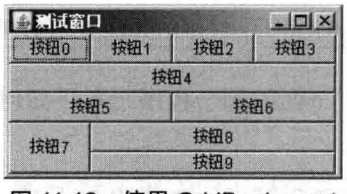


图 11.12　使用 GridBagLayout 布局管理器的效果



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。

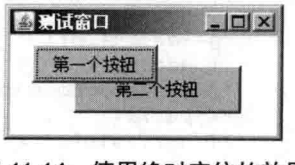


图 11.14　使用绝对定位的效果

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。

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。

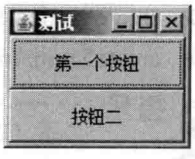


图 11.15 垂直方向的 BoxLayout 布局管理器

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。

- 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。

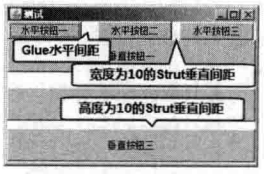


图 11.17 使用间距分隔 Box 器中的按钮效果



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。

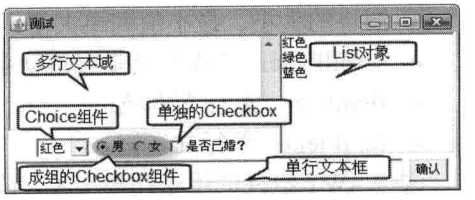


图 11.18　常见的 AWT 组件



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。

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。

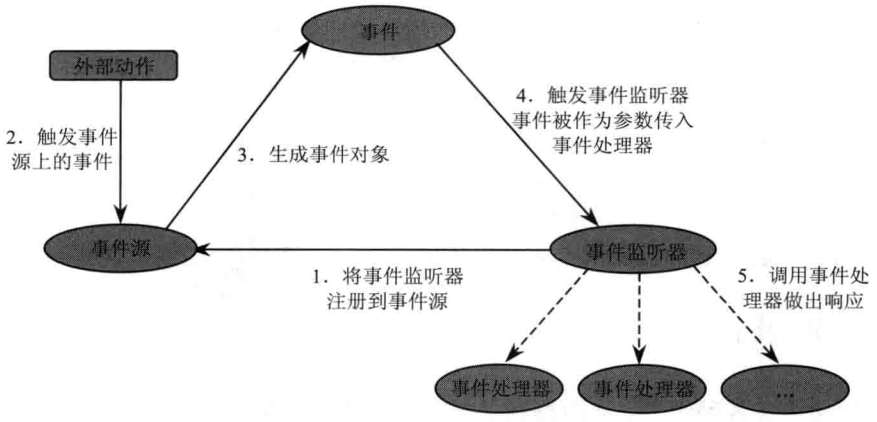


图 11.20　AWT 的事件处理流程示意图

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。

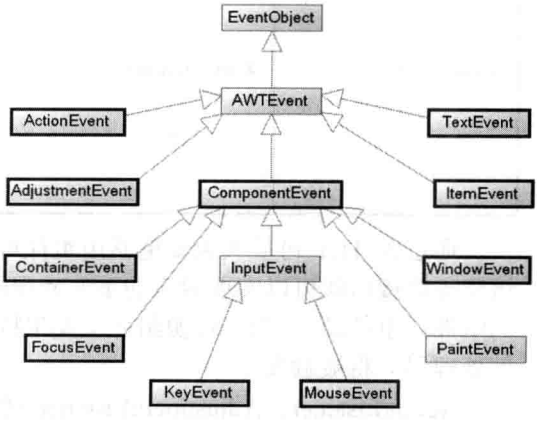


图 11.21 AWT 事件继承层次图

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
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 fl 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。

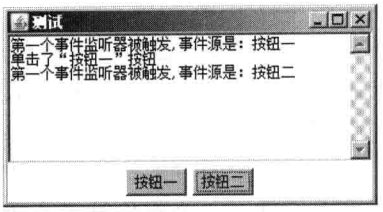


图 11.22　一个按钮被两个监听器监听、一个监听器监听两个按钮

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。

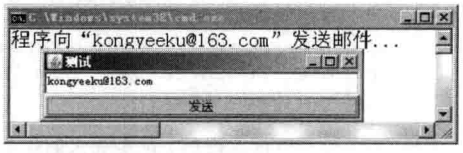


图 11.23 外部类形式的事件监听器类

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。

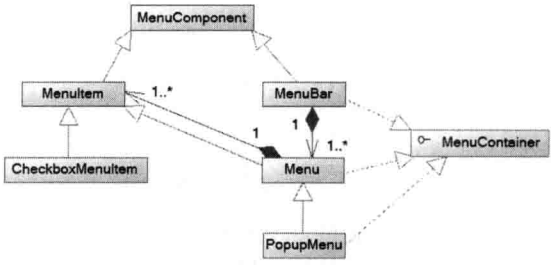


图 11.24　AWT 菜单组件类之间的继承、组合关系

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

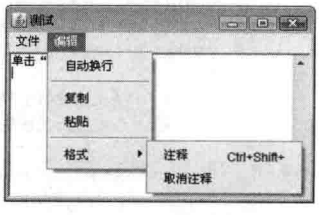


图 11.25　AWT 菜单示例