第2章

Lambda表达式

Java 8 的最大变化是引入了 Lambda 表达式——一种紧凑的、传递行为的方式。它也是本书后续章节所述内容的基础，因此，接下来就了解一下什么是 Lambda 表达式。

2.1 第一个Lambda表达式

Swing 是一个与平台无关的 Java 类库，用来编写图形用户界面（GUI）。该类库有一个常见用法：为了响应用户操作，需要注册一个*事件监听器*。用户一输入，监听器就会执行一些操作（见例 2-1）。

例 2-1 使用匿名内部类将行为和按钮单击进行关联

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

在这个例子中，我们创建了一个新对象，它实现了 `ActionListener` 接口。这个接口只有一个方法 `actionPerformed`，当用户点击屏幕上的按钮时，`button` 就会调用这个方法。匿名内部类实现了该方法。在例 2-1 中该方法所执行的只是输出一条信息，表明按钮已被点击。

这实际上是一个*代码即数据*的例子——我们给按钮传递了一个代表某种行为的对象。

设计匿名内部类的目的，就是为了方便 Java 程序员将代码作为数据传递。不过，匿名内部类还是不够简便。为了调用一行重要的逻辑代码，不得不加上 4 行冗繁的样板代码。若把样板代码用其他颜色区分开来，就可一目了然：

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

尽管如此，样板代码并不是唯一的问题：这些代码还相当难读，因为它没有清楚地表达程序员的意图。我们不想传入对象，只想传入行为。在 Java 8 中，上述代码可以写成一个 Lambda 表达式，如例 2-2 所示。

例 2-2　使用 Lambda 表达式将行为和按钮单击进行关联

```
button.addActionListener(event -> System.out.println("button clicked"));
```

和传入一个实现某接口的对象不同，我们传入了一段代码块——一个没有名字的函数。`event` 是参数名，和上面匿名内部类示例中的是同一个参数。`->` 将参数和 Lambda 表达式的主体分开，而主体是用户点击按钮时会运行的一些代码。

和使用匿名内部类的另一处不同在于声明 `event` 参数的方式。使用匿名内部类时需要显式地声明参数类型 `ActionEvent event`，而在 Lambda 表达式中无需指定类型，程序依然可以编译。这是因为 `javac` 根据程序的上下文（`addActionListener` 方法的签名）在后台推断出了参数 `event` 的类型。这意味着如果参数类型不言而明，则无需显式指定。稍后会介绍类型推断的更多细节，现在先来看看编写 Lambda 表达式的各种方式。

尽管与之前相比，Lambda 表达式中的参数需要的样板代码很少，但是 Java 8 仍然是一种静态类型语言。为了增加可读性并迁就我们的习惯，声明参数时也可以包括类型信息，而且有时编译器不一定能根据上下文推断出参数的类型！

2.2　如何辨别Lambda表达式

Lambda 表达式除了基本的形式之外，还有几种变体，如例 2-3 所示。

例 2-3　编写 Lambda 表达式的不同形式

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
```

```
        System.out.print("Hello");
        System.out.println(" World");
    };

    BinaryOperator<Long> add = (x, y) -> x + y;  ❹

    BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶中所示的 Lambda 表达式不包含参数，使用空括号 () 表示没有参数。该 Lambda 表达式实现了 `Runnable` 接口，该接口也只有一个 `run` 方法，没有参数，且返回类型为 `void`。

❷中所示的 Lambda 表达式包含且只包含一个参数，可省略参数的括号，这和例 2-2 中的形式一样。

Lambda 表达式的主体不仅可以是一个表达式，而且也可以是一段代码块，使用大括号（{}）将代码块括起来，如❸所示。该代码块和普通方法遵循的规则别无二致，可以用返回或抛出异常来退出。只有一行代码的 Lambda 表达式也可使用大括号，用以明确 Lambda 表达式从何处开始、到哪里结束。

Lambda 表达式也可以表示包含多个参数的方法，如❹所示。这时就有必要思考怎样去阅读该 Lambda 表达式。这行代码并不是将两个数字相加，而是创建了一个函数，用来计算两个数字相加的结果。变量 `add` 的类型是 `BinaryOperator<Long>`，它不是两个数字的和，而是将两个数字相加的那行代码。

到目前为止，所有 Lambda 表达式中的参数类型都是由编译器推断得出的。这当然不错，但有时最好也可以显式声明参数类型，此时就需要使用小括号将参数括起来，多个参数的情况也是如此。如❺所示。

*目标类型*是指 Lambda 表达式所在上下文环境的类型。比如，将 Lambda 表达式赋值给一个局部变量，或传递给一个方法作为参数，局部变量或方法参数的类型就是 Lambda 表达式的目标类型。

上述例子还隐含了另外一层意思：Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。目标类型也不是一个全新的概念。如例 2-4 所示，Java 中初始化数组时，数组的类型就是根据上下文推断出来的。另一个常见的例子是 `null`，只有将 `null` 赋值给一个变量，才能知道它的类型。

例 2-4　等号右边的代码并没有声明类型，系统根据上下文推断出类型信息

```
    final String[] array = { "hello", "world" };
```

2.3 引用值，而不是变量

如果你曾使用过匿名内部类，也许遇到过这样的情况：需要引用它所在方法里的变量。这时，需要将变量声明为 final，如例 2-5 所示。将变量声明为 final，意味着不能为其重复赋值。同时也意味着在使用 final 变量时，实际上是在使用赋给该变量的一个特定的值。

例 2-5　匿名内部类中使用 final 局部变量

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hi " + name);
    }
});
```

Java 8 虽然放松了这一限制，可以引用非 final 变量，但是该变量在*既成事实上必须是* final。虽然无需将变量声明为 final，但在 Lambda 表达式中，也无法用作非终态变量。如果坚持用作非终态变量，编译器就会报错。

既成事实上的 final 是指只能给该变量赋值一次。换句话说，Lambda 表达式引用的是*值*，而不是变量。在例 2-6 中，name 就是一个既成事实上的 final 变量。

例 2-6　Lambda 表达式中引用既成事实上的 final 变量

```
String name = getUserName();
button.addActionListener(event -> System.out.println("hi " + name));
```

final 就像代码中的线路噪声，省去之后代码更易读。当然，有些情况下，显式地使用 final 代码更易懂。是否使用这种既成事实上的 final 变量，完全取决于个人喜好。

如果你试图给该变量多次赋值，然后在 Lambda 表达式中引用它，编译器就会报错。比如，例 2-7 无法通过编译，并显示出错信息：local variables referenced from a Lambda expression must be final or effectively final[1]。

例 2-7　未使用既成事实上的 final 变量，导致无法通过编译

```
String name = getUserName();
name = formatUserName(name);
button.addActionListener(event -> System.out.println("hi " + name));
```

这种行为也解释了为什么 Lambda 表达式也被称为*闭包*。未赋值的变量与周边环境*隔离*起来，进而被绑定到一个特定的值。在众说纷纭的计算机编程语言圈子里，Java 是否拥有真正的闭包一直备受争议，因为在 Java 中只能引用既成事实上的 final 变量。*名字虽异，功能相同*，就好比把菠萝叫作凤梨，其实都是同一种水果。为了避免无意义的争论，全书将使用“Lambda 表达式”一词。无论名字如何，如前文所述，Lambda 表达式都是静态类型

注 1：Lambda 表达式中引用的局部变量必须是 final 或既成事实上的 final 变量。——译者注