也许有人已经注意到了，该结构与第 4 章中介绍的多态具有同样的效果（图 13-3）。函数式语言中可以替换函数，而面向对象语言中使用多态可以替换对象。这两种结构都可以替换一部分处理内容，这一目的是相同的。

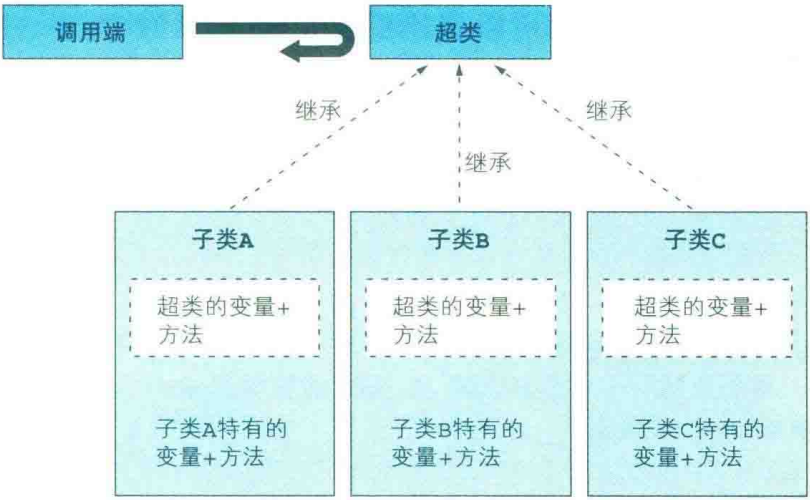

图 13-3　多态

不过，通过对比图 13-2 和图 13-3 可以看出，函数式语言的结构更加简单。面向对象语言中的基本构件是类，因此，即使只想替换一个方法，也需要逐个定义超类和子类群。而在函数式语言中，只需简单地将函数作为参数进行传递即可。

> 通过将函数作为参数进行传递，可以轻松实现与多态相同的结构。

在函数式语言中，我们还可以将函数作为返回值进行返回。根据该结构以及后面将要介绍的部分应用和函数组合等，就可以由函数生成其他函数。该结构是使用函数式语言创建通用框架和库时的强有力的工具。

像这样，接收函数作为参数，或者将函数作为返回值进行返回的函数，称为**高阶函数**（higher order function）。

> 接收函数作为参数，或者将函数作为返回值进行返回的函数，称为高阶函数。

13.6　特征 4：可以灵活组合函数和参数

在函数式语言中，我们可以灵活组合既有的函数和参数，来创建其他函数。关于这一点，存在部分应用和函数组合 2 种结构。

首先来看一下**部分应用**（partial application）。部分应用是针对拥有 2 个以上的参数的函数，仅应用一部分参数来创建其他函数的结构。

我们来举例介绍一下。使用 Haskell 编写的对 x 和 y 这 2 个参数进行加法运算的 add 函数如下所示。

代码清单13.5　add函数

```
add x y = x + y
```

这里简单介绍一下代码。左边的 add 表示函数名，后面紧跟着的 x 和 y 表示该函数接收 2 个参数。右边的 x + y 是函数的逻辑，表示对参数 x 和 y 进行加法运算。

如果对 2 和 3 应用该函数，则返回 5。

代码清单13.6　add函数的运行结果（之1）

```
add 2 3 ⇨ 5
```

对于 add 函数，如果像下面这样，只应用 1 个参数，结果会怎样呢？

代码清单13.7　只应用部分参数的add函数

```
add 2
```

如果是命令式语言，那么在编译或者运行时，应该会发生“缺少参数”的错误。而在函数式语言的情况下，这并不会发生错误。

运行代码清单 13.7 的表达式，就会返回“仅向 add 函数传递了第 1 个参数的函数”，写得更详细一点，就是“将 2 赋给函数 add x y 的第 1 个参数，仅将剩下的 y 作为参数的函数”。针对这里返回的函数，如果像下面这样只赋给它 1 个参数，就可以得到计算结果。

代码清单13.8 add函数的运行结果(之2)

```
(add 2) 3 ⇨ 5
```

我们再来看一下上面的代码。对于通过仅将 1 个参数 2 赋给 add 函数而得到的函数，Haskell 中记为 add 2。如果将参数 3 赋给该函数，就会返回计算结果 5。

这里的重点是，如果仅将 1 个参数赋给接收 2 个参数的 add 函数，结果就会创建 1 个别的函数来接收剩下的 1 个参数。同样，即使是接收 3 个以上的参数的函数，如果只赋给一部分参数，那么也会创建 1 个别的函数来接收剩余的参数。该结构中仅应用一部分参数，因此称为部分应用。部分应用结构的图形表示如图 13-4 所示。

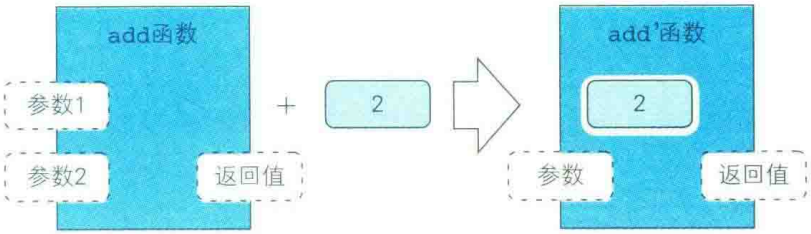

图 13-4 部分应用

> 如果对拥有 2 个以上的参数的函数赋一部分参数，结果就会创建 1 个拥有剩余参数的其他函数。该结构称为部分应用。

即使是拥有 3 个以上的参数的函数，如果逐个应用参数，就也可以表示为仅接收 1 个参数的函数。像这样，我们将“接收多个参数的函数”表示为“接收第 1 个参数，返回‘接收第 2 个及之后的参数的函数’的函数”，称为**柯里化**[1][2]（图 13-5）。

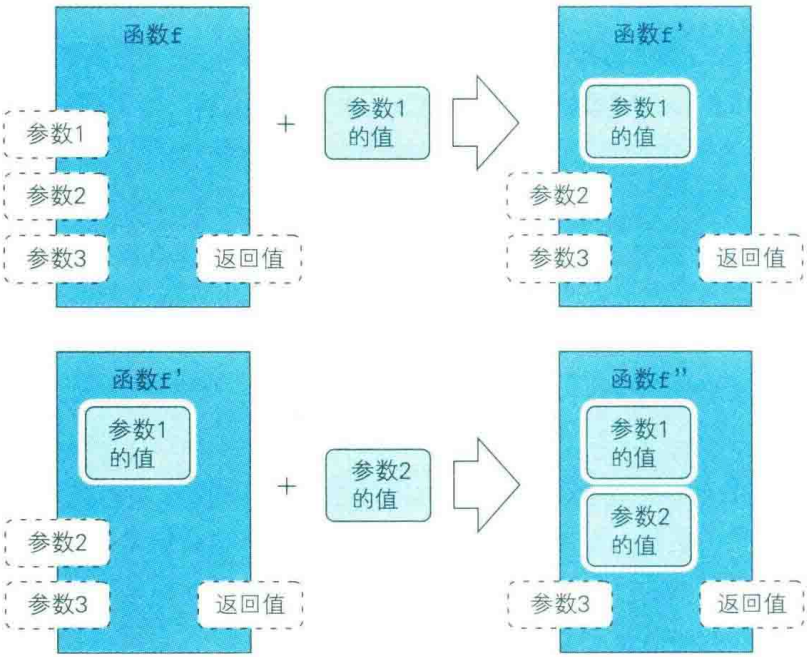

图 13–5　柯里化

> 使用只接收 1 个参数的函数，来依次表示接收多个参数的函数，称为柯里化。

① 柯里化是以 20 世纪的逻辑学家哈斯凯尔·柯里（Haskell Curry）的名字命名的。函数式语言 Haskell 也是以他的名字命名的。

② Haskell 中的所有函数都定义为只接收 1 个参数。不过，在使用单一参数的函数来表示接收多个参数的函数的情况下，需要使用特征 2 中介绍的 λ 表达式，这不便于直观理解。因此，为了表示接收多个参数的函数，Haskell 中提供了语法糖（容易理解的语法）。

接下来，我们介绍一下**函数组合**（function composition）。函数组合是将多个函数进行汇总来创建其他函数的结构。这里通过代码清单 13.9 的简单例子来看一下。代码清单 13.9 中有 2 个函数。第 1 个 square 函数用来计算参数的平方，第 2 个 increment 函数将参数加 1。

代码清单13.9　square函数和increment函数

```
square x    = x * x
increment x = x + 1
```

在函数式语言中，我们可以将这 2 个函数组合成 1 个新函数。Haskell 中通过加上 "." 符号来表示函数组合，如下所示。

代码清单13.10　函数组合

```
square . increment
```

组合成的函数会对参数连续应用 increment 和 square 这 2 个函数。具体来说，就是先对参数应用 increment 函数，进行加 1 计算，然后应用 square 函数，计算平方。该组合成的函数的运行结果如下所示。

代码清单13.11　组合成的函数的运行结果

```
square . increment 2 ⇨ (2 + 1) * (2 + 1) = 9
square . increment 3 ⇨ (3 + 1) * (3 + 1) = 16
```

像这样，将既有的函数汇总成新的函数的结构称为函数组合。函数组合的图形表示如图 13-6 所示[1]。

① 在 Haskell 中，组合成的函数会从右开始依次进行求值。图 13-6 中将参数写在了左边，将返回值写在了右边，因此，先进行求值的函数 g 配置在左边，函数 f 配置在右边。

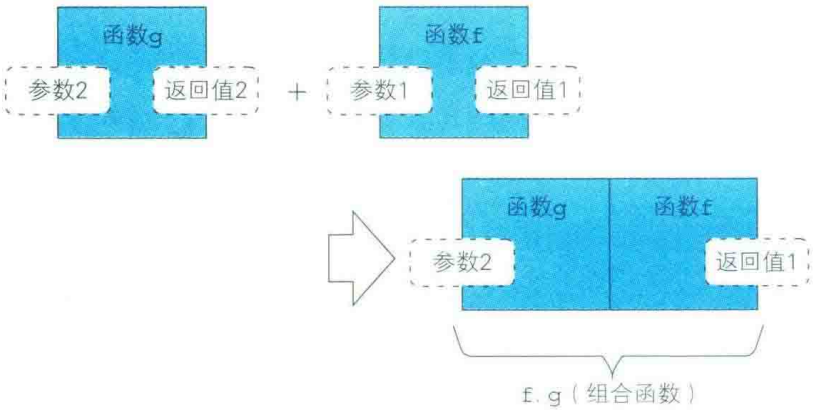

图 13–6　函数组合

除了既有的函数之外，函数组合结构还可以对部分应用的函数进行组合。组合成的函数也可以作为其他函数的参数或返回值。像这样，函数式语言中可以灵活组合既有的函数和参数来创建新函数。

> 通过函数组合，可以汇总多个函数来创建其他函数。

13.7　特征 5：没有副作用

第 5 个特征是没有副作用。“副作用”（side effect）一词在日常生活中经常用来表示药物的不良影响，在编程领域并不常见。

在函数式语言中，**副作用**是指“根据参数计算返回值之外的作业”。具体来说，就是修改变量和外部输入输出。因此，所谓“没有副作用的程序”，就是指不修改变量，完全不执行画面、网络、数据库和文件等的外部输入输出的程序。

看到这里，可能有的读者会怀疑到底有没有这样的程序。实际上，基本上所有的程序都会接收来自键盘的输入、在画面上显示信息、读写数据库或者文件。因此，无论哪种函数式语言，都会使用某种方法实现副作用。

关于这一点我们暂不讨论，这里先来看一下没有副作用的程序结构。

不接受副作用的函数式语言称为**纯函数式语言**。在纯函数式语言中，不管是全局变量还是局部变量，都不可以修改。在设置一次变量的值之后，就不可以再修改了。这相当于只使用 Java 中的 `final` 变量、C 语言和 C++ 中的 `const` 变量来编写程序[1]。

> 在纯函数式语言中，变量的内容不可以修改。

另外，在函数式语言中，这是最基本的结构，因此，给变量设置值或表达式的操作不叫**替换**（substitution），而称为**绑定**（binding），包含设置之后就不可以改变的含义。我们也可以认为，在函数式语言中，变量不是给存储值的内存区域命名，而是给值或表达式本身命名。

> 在函数式语言中，给变量设置值或表达式的操作称为绑定。

是否存在副作用会对函数的动作造成很大影响。如果以存在副作用为前提，那么全局变量在程序运行过程中就有可能改变。因此，即使参数相同，根据运行时间点的不同，引用全局变量的函数的处理结果也可能会不同[2]。而如果没有副作用，函数引用的变量就一直不会改变，因此，函数的处理结果就仅依赖于参数。也就是说，在没有副作用的情况下，如果参数相同，那么不管求值多少次，函数的返回值一定都是一样的。这种性质称为**引用透明性**（referential transparency）。

---

① 可能有人会说："既然不能修改内容，为什么还叫变量呢？"这也许是因为最早的函数式语言 Lisp 能够修改变量，所以之后提出来的纯函数式语言也继续使用了"变量"这个名称。

② 在面向对象语言中，方法返回值的内容除了会受到参数的影响，还会受到实例变量和类变量（`static` 变量）的影响。

对于没有副作用的函数而言，如果参数相同，那么返回值也一定相同。我们将这种性质称为引用透明性。

没有副作用能为软件开发和维护带来诸多益处。

首先，测试会变轻松。如果函数的动作只依赖于参数，那么测试用例只考虑参数的各种情况就可以了。另外，在进行测试时，也无须根据测试条件来设置函数引用的变量的状态。

其次，软件也会更加容易理解。正如我们在第 3 章中介绍的那样，面向对象出现之前的编程语言的一个重大课题就是全局变量问题。由于程序的任意位置都可以访问全局变量，所以在发生错误时调查起来会很费事，而且与全局变量相关的规格修改的影响范围会涉及整个程序。而如果变量内容没有变化，那么对变量进行调查以及确认规格修改的影响也基本不费什么工夫。

最后，没有副作用也易于提高构件的独立性和可重用性。函数仅具有将参数转换为返回值的作用，这样一来就易于创建不依赖于特定情况的通用构件，进而有利于重用。

软件不引起副作用会便于进行测试和维护，易于创建可重用的构件。

接下来我们换个话题，介绍一下延迟求值的相关内容。**延迟求值**（lazy evaluation）是程序运行时的结构，该结构不是从头开始依次进行求值，而是在实际需要的时间点对各个表达式进行求值。

下面我们以稍微复杂一点的应用程序处理为例进行说明，这里不再使用 Haskell，而是使用 Java 的示例代码。实际上，Java 并不支持延迟求值，方便起见，我们暂且假定它支持。

代码清单13.12 getSalary函数

```
/**
 * @param employee 员工
 * @param workHours 实际出勤时间
 * @return 实发工资
 */
int getSalary(Employee employee, int workHours) {
    // 当员工为普通员工时，根据实际出勤时间计算加班费，加到固定工资上
    ~
}
```

代码清单13.12是计算实发工资的getSalary函数。该函数的第1个参数employee是员工，第2个参数workHoures是当月的实际出勤时间，该函数会计算包含加班费在内的实发工资。如果是管理层人员，则实发工资为固定工资，而如果是普通员工，则实发工资根据实际出勤时间计算，包含加班费。

使用getSalary函数的代码如下所示。传递给第2个参数的实际出勤时间是调用其他的getWorkHours函数计算出来的。

代码清单13.13 调用getSalary函数

```
getSalary(employee, getWorkHours(employee));
```

根据执行方式是否是延迟求值，该函数的动作也会不同。

我们先来介绍一下一般的执行方式[1]。在一般的执行方式的情况下，getSalary函数执行前会调用getWorkHours函数，来计算当月的实际出勤时间（图13-7中的左图）。在这种情况下，不管是普通员工，还是管理层人员，都会调用getWorkHours。

反之，在延迟求值方式的情况下，getSalary函数执行前不会对第2

① 与延迟求值相对的是“预先求值”或“正规求值”。

个参数 getWorkHours 函数进行求值。只有在需要实际出勤时间时，才对 getWorkHours 函数进行求值。在该示例中，仅当需要计算普通员工的加班费时，才会使用实际出勤时间，因此，在管理层人员的情况下，并不会执行 getWorkHours 函数（图 13-7 中的右图）。

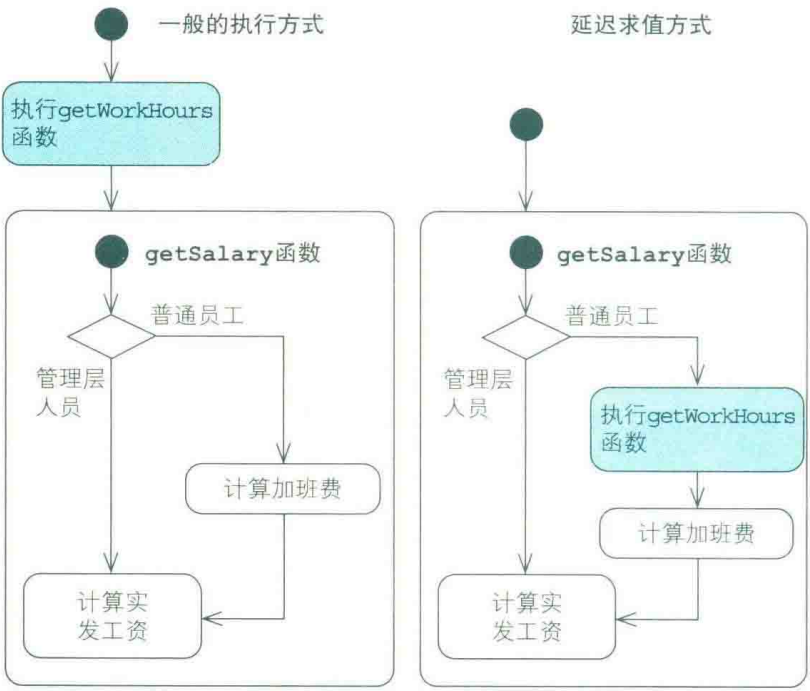

图 13-7　一般的执行方式和延迟求值方式

像这样，在延迟求值方式中，并不提前对所有参数进行求值，而是在函数处理过程中需要时才对参数进行求值。如上述例子所示，当根据条件不会用到某些参数时，执行效率会更高。

延迟求值的对象并不仅限于参数的求值。在根据布尔值进行条件判断时，对于不符合条件的表达式，也不会进行求值。另外，在数据声明中，在实际需要该数据之前，并不会进行内存分配[1]。

① 在采用延迟求值方式的 Haskell 中，利用该性质可以定义“1 到 ∞（无穷）”的列表。

在延迟求值方式中，只有在需要时才对函数或表达式进行求值。

延迟求值与前面介绍的副作用紧密相关。当没有副作用时，如果参数相同，那么不管什么时候对函数进行求值，都一定会得到相同的结果。这样一来，不管从什么地方、以什么顺序对程序中的函数或表达式进行求值，整体的结果都一定是相同的，因此，使用延迟求值方式才成为可能。

在没有副作用的情况下，可以实现延迟求值方式。

13.8 特征 6：使用分类和递归来编写循环处理

在使用函数式语言编写程序的情况下，通常不会使用 `for` 语句和 `while` 语句等循环命令。其中一个原因是，在没有副作用的纯函数式语言中，有时无法定义控制循环的计数器和判断结束的变量。另外，由于指示计算机执行循环处理的 `for` 语句和 `while` 语句，与以“返回值的表达式”为基础的函数式语言不能很好地兼容，所以有时并不会提供该语法。

因此，在使用函数式语言编写循环处理的情况下，通常会使用模式匹配和递归。**递归**是在函数中调用该函数本身的结构，C 语言和 Java 等许多命令式语言中也都提供了该结构。模式匹配是函数式语言中特有的结构，因此，下面我们先来介绍一下模式匹配相关的内容。

模式匹配（pattern matching）是根据参数的值来分情况定义函数的结构。我们看一下下面的 Haskell 的代码示例。

代码清单13.14　使用模式配的函数定义

```
convertTab '\t' = " "  ①
convertTab c    = [c]  ②
```

这里使用模式匹配定义了 convertTab 函数。①处和②处都声明了 convertTab 函数，这是因为根据函数的值而分了不同情况。

①中定义了参数值为 '\t' 字符（制表符）时的处理。这里将制表符转换为半角空格。

②中定义了参数值不为 '\t' 字符（制表符）时的处理[1]。这里并未对参数的字符进行特殊处理，只是转换为了字符串型[2]。

该 convertTab 函数的规格说明如表 13-1 所示。

表 13–1　convertTab 函数的规格说明

	参数的值	处　理
①	'\t' 的情况下	将制表符转换为半角空格
②	其他情况下	将输入的字符直接转换为字符串型

像这样，在函数式语言中，我们可以根据参数的值对函数的逻辑分情况进行定义。使用模式匹配，可以通过函数声明明确表示对应于不同参数值的分情况处理，而不必使用 if 语句和 case 语句。这样一来，逻辑就会变简单，也易于将分情况的条件传达给阅读代码的人。用 Java 重写前面的代码，如下所示（代码清单 13.15）。

代码清单13.15　Java中相当于模式匹配的逻辑定义

```
String convertTab(char c) {
    if (c == '\t')) {
        return " ";
    } else {
```

① 使用模式匹配定义的多个函数被从头开始依次进行检查。在代码清单 13.14 中，①处定义了参数为制表符时的情况。②处的参数 c 表示变量，定义了参数不为制表符时的情况。

② 在 Haskell 中，[] 是列表（集合）的意思，字符列表就是字符串。而根据类型推断，②处的代码对字符以外的参数也可以正常动作。如果想将参数限制为字符型（char），则需要明确表示 escapeChar 函数的类型为 Char -> String。

```
            return String.valueOf(c);
        }
    }
```

我们将其与代码清单 13.14 中的 Haskell 代码比较一下。可以发现，与使用 `if` 语句的 Java 代码相比，使用模式匹配的 Haskell 代码能够更加简洁明了地表示处理内容。

该模式匹配结构也可以认为是对编程语言提供的函数定义进行了精妙的处理。不过，该结构是函数式语言所特有的，命令式语言基本上都不提供。这是因为，命令式语言是以机器语言的命令的抽象化为基础的，一个函数（或者方法）的定义对应于在内存中的特定位置展开的一连串的代码。

> 通过模式匹配，可以明确表示根据参数值进行分情况处理。

在介绍完模式匹配之后，我们接着来介绍一下循环处理。正如本节开头所说的那样，在使用函数式语言的情况下，通常都使用模式匹配和递归来编写**循环处理**。

我们以计算自然数 n 的阶乘的 `factorial` 函数为例进行说明[1]。所谓阶乘，是指从 1 到 n 为止的所有数字相乘。阶乘一般使用“!”符号表示，例如，1 到 4 的阶乘的计算结果如下所示。

$1! = 1$

$2! = 2 \times 1 = 2$

$3! = 3 \times 2 \times 1 = 6$

$4! = 4 \times 3 \times 2 \times 1 = 24$

在使用传统的命令式语言来编写计算阶乘的程序时，通常都会使用循环处理。例如，使用 Java 编写的代码如下所示（代码清单 13.16）。

① 这里将自然数定义为“（不包含 0 的）大于等于 1 的整数”。

代码清单13.16　使用Java编写的factorial函数

```
int factorial(int n) {
  int result = 1;     // 保存计算结果的变量
  for (int i=n; i>0; i--) {
    result = result * i;
  }
  return result;
}
```

而如果使用 Haskell 编写计算阶乘的函数，则代码如下所示（代码清单 13.17）。

代码清单13.17　计算阶乘的函数

```
factorial 1 = 1                          ①
factorial n = n * factorial (n - 1)  ②
```

我们来介绍一下该代码。这里也使用模式匹配，分别定义了 2 个函数。

①是参数为 1 时的逻辑。1 的阶乘为 1，所以结果只是返回 1。

②是除 1 之外的自然数的逻辑[1]。这里我们来看一下右边的 `n * factorial (n - 1)`。`factorial (n - 1)` 是“$(n-1)$ 的阶乘”的意思，如果参数为 4，那么②的表达式右边就是“4 × 3!”。另外，在计算“3!”时，根据递归处理，会再次执行 factorial 函数，即“3 × 2!”，同样地，接下来是“2 × 1!”。最后，当参数为 1 时，再次执行 factorial 函数，根据模式匹配，应用①，结果返回 1。将这一连串的处理依次写下来，如下所示。

① 代码清单 13.16 中并未考虑参数小于等于 0 时的情况。如果考虑参数小于等于 0 的情况，就需要使用保护或者 case 表达式来添加判断逻辑。

$4! = 4 \times 3!$ ← 应用②
$3! = 3 \times 2!$ ← 应用②
$2! = 2 \times 1!$ ← 应用②
$1! = 1$ ← 应用①

即：

$4! = 4 \times (3 \times (2 \times (1))) = 24$

像这样，在函数式语言中，我们可以使用模式匹配和递归来编写循环处理。基本形式与代码清单 13.17 一样，使用模式匹配来分情况定义结束条件和循环条件，循环条件中使用递归来调用该函数本身。

我们将代码清单 13.17 的处理内容汇总在表 13-2 中。

表 13–2 factorial 函数的规格说明

	参数的值	处　理
①	1 的情况下	返回 1
②	n 的情况下	返回 n * ((n–1) 的阶乘)

我们再来看一下表 13-2 和代码清单 13.17。可以发现，表 13-2 中简洁地表示了计算自然数阶乘的方法，并直接表示为代码清单 13.17 的代码。像这样，使用模式匹配和递归，相比命令式语言的循环处理，可以更加简洁明了地编写逻辑。

这种使用模式匹配和递归进行的编程，与命令式语言中使用循环处理进行的编程有着很大的不同。这一结构要求具有与传统不同的思想，最终能够简洁明了地完成程序，可以说是函数式语言的巨大魅力。

> 通过使用模式匹配和递归，可以简洁明了地编写循环处理。