Chp4 函数

函数是计算机编程中非常重要的部分，是编程中最基本的元素之一。函数表示的是一种通用的过程，这种过程能够对外界提供服务。例如，现实生活中，ATM 取款机上有不同的功能，我们可以理解为 ATM 机上具有不同的函数可以调用；我们在 ATM 机上取钱，就可以理解为我们在 ATM 机上调用了“取钱”函数。在这种关系中，我们是“取钱”函数的调用者，“取钱”函数为我们提供服务。

1 函数的基本使用

1.1 函数的三要素

对于函数而言，最重要的部分就是函数的三要素：返回值、函数名、参数表。

返回值，这个概念表示调用函数之后，函数会返回什么数据给调用者；

函数名，顾名思义，这表示函数的名字；

参数表，表示调用函数时所给的参数是什么，也就是说，调用函数时需要给函数哪些“输入”。

以“取钱”函数为例，函数的返回值为“钱”，我们作为调用者调用“取钱”函数，目的就是获得这个函数的返回值“钱”；这个函数的参数表表示我们对调用“取钱”时应该给这个函数传递的参数，取钱时需要“银行卡、密码、取款金额”等一系列参数。

在 Java 中，函数（Function）也被称之为方法（Method）。Java 中并不区分这两个概念，因此，本书中“函数”和“方法”指的是同一个意思。

后面我们将详细介绍 Java 中的函数。

1.2 函数的定义

首先我们介绍 Java 中函数的定义。

在 Java 中，函数定义的位置为：类的里面，其他函数的外面。例如下面的代码：

```
   //1
public class TestFunction{
         //2
        public static void main(String args[]){
               //3
         }
         //4
   }
```

对于上面//1、//2、//3、//4 四个位置而言，只有//2 和//4 的位置能够定义函数。另外，定义了函数可以在主函数中调用。而不论这个函数是定义在主函数之前，还是定义在主函数之后，都能够进行调用。也就是说，一个函数定义在//2 的位置，或者定义在//4 的位置，在函数的定义和使用上，没有任何区别。对于函数来说，只要满足“类的里面，其他函数的外面”这个要求，在定义的顺序方面是没有要求的。

在 Java 中定义一个函数时，首先可以先写两个单词：public static。这两个单词为 Java 中的修饰符。加上这两个修饰符是为了能在主函数中正常的调用。至于这两个修饰符在修饰函数时是什么含义，会在后面的课程中详细为大家阐述。

在 public static 之后，就是函数的三要素。假设我们要写一个 add 函数，该函数接受两个 int 类型作为参数，并且返回这两个参数的和。这样，可以定义 add 函数如下：

```
public static int add(int a, int b)
```

第一个 int 为返回值类型，表示 add 函数返回一个 int 值。add 是函数名，add 后面的圆括号是参数表。

参数表中，可以定义 0 个或多个参数。在函数参数表中定义的参数，被称为“形式参数”，简称形参。从语法上说，形参是特殊的局部变量。一方面，在参数表中定义形参，就好像定义局部变量一样，应当写出变量的类型和变量名。另一方面，在形参也有其作用范围，形参的作用范围就是函数的内部。例如，在上面的代码中，我们定义了 a 和 b 两个形参，这两个形参的作用范围就是 add 函数内部。

写参数表的时候还要注意，如果这个函数接受多个参数，则多个参数之间用逗号隔开。例如上面 add 函数的例子，参数表就写成：int a, int b。需要注意的是，虽然函数的这两个形参类型一致，但是不能写成 int a,b。在定义多个形参的时候，每个形参的类型和参数名都应当完整的列出来。

如果调用一个函数时不需要参数，则参数表为空，在圆括号中什么内容都不写即可。例如，如果我们要写一个函数 time，用来表示当前是几点。这个函数不需要任何的参数，因此可以写成：

```
public static int getCurrentHour()
```

这个函数没有任何参数，因此，其参数表为空。

定义完函数之后，需要在函数后面紧跟一个代码块，这个代码块称为函数的实现。函数的定义，表明的是函数应该如何使用。而函数的实现，则表明的是函数中真正执行的内容。

目前这一阶段，我们接触的所有函数里，函数的定义和函数的实现都是无法分离的。在函数定义完之后，必须加上一对花括号，在花括号中写上函数要执行的内容。完整的 add 函数如下：

```
public static int add(int a, int b){
      int c = a + b;
      return c;
}
```

注意，在 add 函数中，包括一个“return c”的语句。这个语句是 return 语句，表示函数的返回值是什么。在 add 函数中，return c 表示返回值为 c 变量的值。

return 语句除了表示函数返回值之外，同样可以表示函数的流程跳转。这一部分内容留在下一小节中阐述。

1.3 函数的调用

写完 add 函数之后，就能够在主函数里面对其进行调用。例如：

```
01: public static void main(String args[]){
02:    int m = 10, n = 20;
03:    int result = add(m, n);
04:    System.out.println(result);
05:    add(30, 40);
06: }
```

这样，就在主函数中，调用了 add 函数。要注意的是：

第 03 行。在这一行中，我们把 add 函数作为赋值语句的一部分，因此会先调用 add 函数，然后把 add 函数的返回值赋值给 result 变量。在调用函数时，需要给出函数名和参数表。在上面的代码中，首先，明确的写出函数名 add；其次，在函数名后面写一对圆括号，在括号中给出调用函数时需要的参数 m 和 n。

这里，m 和 n 是传递给 add 函数的参数，被成为实际参数，简称“实参”。实参指的是，在调用函数的时候，为函数指定的参数。在函数调用的过程中，会把实参的值传递给形参。例如上面的代码中，m、n 就是调用函数时的实参，而 a、b 就是定义函数时的形参。在函数调用的时候，会把 m 变量的值传递给形参 a，而把 n 变量的值传递给形参 b。这样，在 add 函数内部进行计算的时候，两个形参 a、b 的值，就是调用方法时两个实参 m、n 的值。

最后，函数返回时，将计算所得的值返回。这个返回值又被赋值给 result 变量。这样就完成了一次函数的调用。

第 05 行，在这一行中，我们又一次调用了 add 函数，并且传入了不同的参数。参数除了可以用变量之外，同样可以使用字面值。另外，需要注意的，调用函数之后，函数的返回值没有被赋值给任何变量，因此函数的返回值没有被保存下来。

完整代码如下：

```
public class TestFunction{
   public static void main(String args[]){
         int m = 10, n = 20;
         int result = add(m, n);
         System.out.println(result);
         add(30, 40);
   }

   public static int add(int a, int b){
         int c = a + b;
         return c;
   }

}
```

运行结果如下：

```
D:\Book\chp4>javac TestFunction.java

D:\Book\chp4>java TestFunction
30

D:\Book\chp4>
```

注意到，由于第二次调用 add 函数时，没有保存其返回值，也没有把它的值输出。

1.3.1 用 return 语句返回值

下面，我们更加详细的来介绍一下 return 语句。return 语句有两层含义，一个含义就是我们之前提到的，return 语句表示返回一个值。

在函数的定义中，如果给出了返回值类型，则必须要返回一个相应类型的值。例如，由于在函数的定义中，add 函数返回值类型为 int。因此，在 add 函数的中必须要返回一个

int 类型的值。

例如，有下面的函数定义：

```
public static int m()
```

这个函数的返回值为 int 类型，表明这个函数必须要返回一个整数值。如果在这个函数中没有 return 语句，则编译会出错。例如，假设代码如下：

```
public static int m(){
   System.out.println("m()");
}
```

则编译时的结果如下：

```
D:\Book\chp4>javac TestReturn.java
TestReturn.java:4: missing return statement
        }
        ^
1 error

D:\Book\chp4>
```

编译器提示，在代码中缺少返回语句。

而且，return 语句返回的值，如果与函数定义中的返回值类型不同，也有可能出错。例如，在 m 方法中如果返回一个 double 类型的值，代码如下：

```
public static int m(){
   System.out.println("m()");
   return 1.5;
}
```

则编译时的结果如下：

```
D:\Book\chp4>javac TestReturn.java
TestReturn.java:4: possible loss of precision
found   : double
required: int
                return 1.5;
                       ^
1 error

D:\Book\chp4>
```

编译器提示，可能损失精度。

怎么来理解这个过程呢？我们可以结合 add 函数来理解。add 函数的代码片段如下：

```
   public static void main(String args[]){
         …
         int result = add(m, n);
         …
   }
   public static int add(int a, int b){
         int c = a + b;
         return c;
   }
```

在 add 函数的内部，定义了一个变量 c。这个变量是一个局部变量，c 的作用范围是 add 函数的内部。然后，return 语句中，返回了 c 的值，并在主方法中把这个返回值赋值给 result。然而，主方法并不在 c 变量的作用范围之内，因此，不能直接在主方法中输出 c。

那 c 的值是怎么返回的呢？可以这么来理解：在调用 add 函数的时候，由于 add 函数的签名中，说明这个 add 函数会返回一个 int 类型的值，因此，Java 会为 add 函数准备一个临时变量，变量的类型是 int，用这个临时变量来保存 add 函数的返回值。当执行到 return c 的时候，会把 c 变量的值赋值给这个临时变量。然后，在主方法的复制语句中，int result = add(m,n)，这就意味着把 add 函数的返回值赋值给 result，也就是把那个临时空间的值赋值给 result。

而在刚刚的 m 方法中，我们看到

```
public static int m(){
   System.out.println("m()");
   return 1.5;
}
```

在方法中返回一个 double 类型的值。m 方法的定义中，说明返回值是一个 int 类型，因此编译器为 m 函数分配一个 int 类型的临时变量，用来保存返回值。但是，在调用 return 1.5 的时候，程序会试图把一个 double 类型的 1.5 赋值给一个 int 类型的临时变量，这样就会产生一个错误。

我们继续修改一下 m 函数：

```
public static int m(int arg){
   System.out.println("m()");
   if (arg == 10) return 0;
}
```

这段代码依然有问题，编译时会产生一个编译时错误，错误信息如下：

```
D:\Book\chp4>javac TestReturn.java
TestReturn.java:5: missing return statement
        }
        ^
1 error

D:\Book\chp4>_
```

为什么会产生这个问题呢？原因在于，当我们定义了一个函数并指明其返回值为 int 类型之后，就要保证，无论使用什么参数调用这个函数，函数都能够返回一个 int 值。而上面的代码中，我们对参数 arg 进行了判断，如果这个参数的值为 10，则返回 0。那如果这个参数的值不为 10 呢？在这个函数的实现中没有说明。由于调用这个函数有可能没有返回值，因此，编译时会产生一个编译时错误。

为了解决这个错误，可以为 if 语句增加一个 else 代码块，并进行 return。修改后的代码如下：

```
public static int m(int arg){
   System.out.println("m()");
   if (arg == 10) return 0;
   else return 1;
}
```

1.3.2 流程跳转以及 void

上一小节我们介绍了 return 语句返回值。return 语句除了能够返回值之外，还能够控制流程的跳转。具体的说，在执行 return 语句的时候，被调用的函数会终止执行，并返回到函数的调用点上。例如，看下面的例子，我们修改 TestFunction 的代码如下：

```
01: public class TestFunction{
02:   public static void main(String args[]){
03:         System.out.println(“Line 3”);
04:         int m = 10, n = 20;
05:         int result = add(m, n);
06:         System.out.println(result);
07:         System.out.println(“Line 7”);
08:         add(10, 20);
09:         System.out.println(“Line 9”);
10:   }
11:   public static int add(int a, int b){
12:         System.out.println(“Line 12”);
13:         int c = a + b;
14:         System.out.println(“Line 14”);
15:         return c;
16:   }
17: }
```

在这个程序中，作为程序执行的入口，从主函数开始执行。首先输出 Line3。

之后，在第 5 行，程序调用 add 函数，因此程序从第 5 行跳转到 11 行，然后继续往下执行。依次执行的结果，输出为 Line12、Line14。

在 15 行，程序返回。这里，return 语句包含两层不同的含义：一表示返回值为 c 变量的值，二表示程序流程返回。所谓的流程返回，指的是函数返回到调用点上，即程序从 15 行跳转回第 5 行。

要注意的是，在 add 函数中定义了 c 变量，这是一个局部变量。同时，add 函数的两个形参 a 和 b 也相当于 add 函数的局部变量。在程序跳转之后，程序就在 a、b、c 这三个局部变量的作用范围之外，无法使用 a、b、c 的值。

此外，因为 return 语句具有流程返回的特性，因此，如果在 15 行之后增加一句：

```
System.out.println(“come here”);
```

由于这句话写在 return 语句之后，而永远不会被执行，因此这个语句会引起一个编译错误。

另外，有一类函数不需要返回任何值。例如，我们写一个函数，这个函数接受一个参数 n，根据 n 的值不同，来打印 n 个 Hello World。

很明显，这个函数需要一个参数 n。但是，这个函数由于只是完成打印功能，我们在调用这个函数的时候，不需要这个函数返回任何值。因此，在定义这个函数的时候，我们要告诉编译器，这个函数不需要返回值。为此，我们可以把这个函数的返回值类型写成“void”类型。

代码如下：

```
public static void printHelloWorld(int n){
```

```
  for(int i = 0; i<n; i++){
        System.out.println("Hello World");
  }
}
```

注意，上面的代码中，在定义函数时，把函数的返回值定义为 void。这意味着这个函数不返回任何值。

因为 void 函数不返回值，因此，在 void 函数中，可以不包含 return 语句。例如上面的例子，在 printHelloWorld 这个函数中，没有 return 语句出现。当函数中所有的代码都执行结束时，函数自然会结束，然后就会返回函数的调用点上。

当然，void 函数中，也能够出现 return 语句。但是，此时，void 函数的 return 语句不能返回一个值，只能够表示流程的返回。例如，我们可以在 printHelloWorld 中增加如下代码：

```
import java.util.Scanner;
public class TestVoid{
  public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.print("Input a number:");
        int a = sc.nextInt();
        printHelloWorld(a);
  }
  public static void printHelloWorld(int n){
        if (n > 10) return;
        for(int i = 0; i<n; i++){
              System.out.println("Hello World");
        }
  }
}
```

在 printHelloWorld 函数中，在进入 for 循环之前，会进行一次判断，判断参数 n 是否大于 10。如果参数 n 大于 10 的话，则程序会执行 return 语句。这个 return 语句如果被执行的话，则 printHelloWorld 函数立刻返回，不执行后面的 for 循环而直接返回到了主函数中。执行结果如下，当我们输入 15 时，printHelloWorld 中的 return 语句被执行，因此 for 循环不被执行而程序直接返回，输出结果如下：

```
D:\Book\chp4>javac TestVoid.java

D:\Book\chp4>java TestVoid
Input a number:15

D:\Book\chp4>
```

当我们输入 6 时，return 语句不被执行，因此 for 循环被执行，输出 6 个 HelloWorld。执行结果如下：

```
D:\Book\chp4>java TestVoid
Input a number:6
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

D:\Book\chp4>
```

2 实参与形参

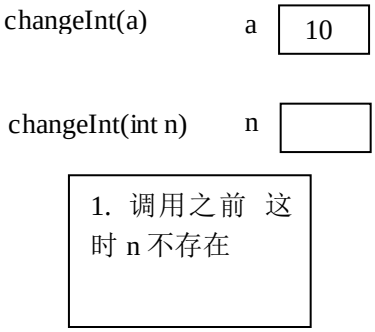
下面我们要为大家介绍的是实参和形参这两个概念。在上面的 TestFunction 程序中，实参就是 m，n，这两个参数是调用 add 函数时实际传递的参数。而在 add 函数的签名处，定义了两个参数(int a, int b)，这两个参数是所谓的形参。从本质上来说，形参相当于特殊的局部变量。例如，对于上面的 a、b 两个参数而言，这两个参数相当于 add 函数中定义的局部变量，它们的作用范围就是在 add 函数内部。在调用 add 函数的时候，会把实参的值传递给形参。

关于参数以及参数的传递，我们看下面这个例子：

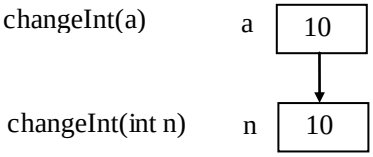
```
public class TestParameter {
   public static void main(String[] args) {
         int a = 10;
         changeInt(a);
         System.out.println(a);
   }

   public static void changeInt(int n){
         n ++;
   }
}
```

编译运行这个程序，输出结果为 10。为什么会这样呢？明明在 changeInt()函数中把参数加 1 了呀？

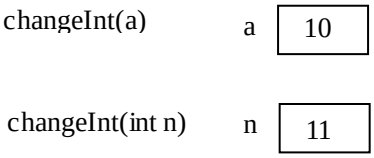
要注意的是，我们在调用函数的时候，实参传递给形参，是把实参的值传递给形参。如下图所示：



1. 调用之前 这时 n 不存在



2. 调用时，实参的值传递给形参



3. 在函数中修改了形参的值，实参不受影响

4. changeInt 函数返回，形参不存在，实参的值没有被改变

3 函数的嵌套调用

前面的内容，我们为大家介绍了函数的一些基本使用。接下来，要介绍的是函数中一些更加灵活的应用。看下面这个代码的例子：

```
public class TestNestedCall{
   public static void main(String args[]){
         System.out.println("main1");
         ma();
         System.out.println("main2");
   }

   public static void ma(){
         System.out.println("ma1");
         mb();
         System.out.println("ma2");
   }

   public static void mb(){
         System.out.println("mb1");
         System.out.println("mb2");
```

```
    }
}
```

我们来看一下上面这段代码的执行过程。在上面这段代码中，首先，是在 main 函数中输出“main1”，然后，在 main 函数中调用了 ma 方法，于是，流程从 main 中跳转到了 ma 函数中。在 ma 函数的第一个语句中，输出“ma1”。

然后，在 ma 方法中，又调用了 mb 方法。我们可以看到，在一个函数中，还可以调用另外一个函数，这种调用方式被称为函数的嵌套调用。在我们的这个例子中，主函数调用 ma 函数，ma 调用 mb 函数，示意图如下：

main → ma → mb

在 ma 调用 mb 函数之后，mb 函数输出“mb1”和“mb2”。然后，mb 函数中所有的代码都执行完了，mb 函数返回其调用点，也就是返回到 ma 方法中。然后，ma 方法继续执行，输出“ma2”。然后，ma 方法的代码也都全部完成，于是 ma 方法也返回调用点，也就是返回到主函数中。最后，主函数输出“main2”，主函数中所有代码都执行完毕，程序终止。

上述代码运行结果如下：

```
D:\Book\chp4>javac TestNestedCall.java

D:\Book\chp4>java TestNestedCall
main1
ma1
mb1
mb2
ma2
main2

D:\Book\chp4>_
```

4 函数的递归调用

函数除了能够调用其他函数之外，还可以调用函数本身。这就是函数的递归调用。

我们来看一个例子。我们要写一个函数，这个函数接受一个整数 n 作为参数，然后输出这个整数的阶乘 n!。我们定义阶乘 n！为：

n! = n * (n-1) * (n-2) … *1

例如，5 的阶乘 5！= 5*4*3*2*1 = 120。

很显然，这个程序可以用循环来完成。在这儿，我们为大家介绍一种不用循环的方式，我们可以使用函数的递归来完成这个程序。

首先，由于我们要写一个计算阶乘的函数，因此，我们定义函数如下：

public static int factorial(int n)

这个函数接受一个参数，用来计算一个整数 n 的阶乘。那我们怎么实现呢？考虑到有下面这个数学规律：

n! = n * [(n-1) * (n-2) * (n-3) … *1] = n * (n-1)!。

也就是说，我们的函数可以写成这样的步骤：

```
public static int factorial(int n){ // 计算 n 的阶乘
    //第一步，计算 (n-1)的阶乘
    //第二步，把上一步计算所得的结果乘以 n 返回
```

```
}
```

注意到，第一步要完成的事情，就是计算 n-1 的阶乘。由于我们这个函数，就是用来计算 n 的阶乘。因此，第一步也可以理解为，以 n-1 为参数，调用 factorial。代码如下：

```
public static int factorial(int n){ // 计算 n 的阶乘
    //第一步，计算 (n-1)的阶乘
    factorial(n-1);
    //第二步，把上一步计算所得的结果乘以 n 返回
}
```

在这个函数中，出现了在 factorial 函数内部调用 factorial 函数的情况。这种函数自身调用自身的情况，我们就把它称为函数的递归调用。

我们完善上面的代码，把第二部也翻译成代码：

```
public static int factorial(int n){ // 计算 n 的阶乘
    //第一步，计算 (n-1)的阶乘
    int result = factorial(n-1);
    //第二步，把上一步计算所得的结果乘以 n 返回
    return result * n;
}
```

这样，我们的函数已经快要写好了。但是，这里还有一个小问题。例如，如果当 n 为 3 时，在 factorial(3)函数的内部，会调用 factorial(2)；在 factorial(2)的内部，会调用 factorial(1)；在 factorial(1)的内部，会调用 factorial(0)；在 factorial(0)的内部，会调用 factorial(-1)……

因此上，如果不加以控制的话，递归调用会无限制的继续下去，直到计算机中所有资源耗尽为止。

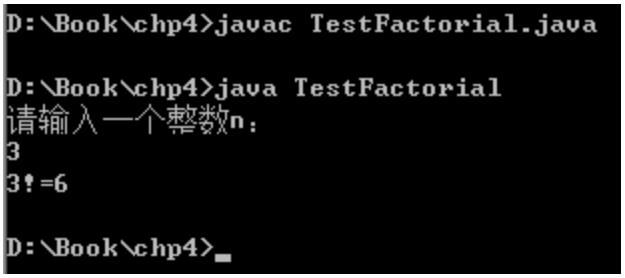
为此，我们需要为递归调用规定一个结束的条件。具体到计算阶乘的这个程序中，当参数 n 的值为 1 时，表示计算 1 的阶乘。到这一步，就没有继续递归调用下去的必要了，因此，当 n 为 1 时，应当直接返回。

完整的代码如下：

```
import java.util.Scanner;
public class TestFactorial{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入一个整数 n：");
        int n = sc.nextInt();
        int a = factorial(n);
        System.out.println(n + "!=" + a);

    }

    public static int factorial(int n){
        if (n == 1) return 1;
        int result = factorial(n-1);
        return n * result;
    }
```

}

运行结果如下：



在这个程序运行的过程中，这个程序首先会读入 n，然后，会把 n 的值传递给 factorial 函数。由于我们输入的是整数 3，因此，会调用 factorial(3)。

在 factorial(3) 函数的内部，由于 3==1 的值为假，因此会执行

int result = factorial(3-1);

为了计算这个 result 的值，会先计算 factorial(2)函数的值。因此，factorial(3)函数调用 factorial(2)函数。示意如下：

factorial(3) → factorial(2)

然后，在 factorial(2)函数的内部，由于 2==1 的值为假，因此，会执行：

int result = factorial(1);

因此，会在 factorial(2)函数的内部，调用 factorial(1)函数，示意如下：

factorial(3) → factorial(2) → factorial(1)

然后，调用 factorial(1) 函数。由于此时，参数的值为 1，因此 factorial(1) 函数返回 1，返回给 factorial(2)函数。示意如下：

factorial(3) → factorial(2) ←返回 1—factorial(1)

返回到 factorial(2)之后，此时，要执行 return n * result 语句，因此 factorial(2)会返回 1 * 2，会把计算所得的值返回给 factorial(3)函数。示意如下：

factorial(3) ←返回 2-- factorial(2) ←返回 1—factorial(1)

最后，factorial(3)执行到 return n * result。此时，n 的值为 3，result 的值为 factorial(2)函数的返回值 2，因此 factorial(3)会返回 6。至此，递归调用均完成。

如果需要写递归的话，需要注意一下几个问题：

1、要首先研究出一个推导公式，这个公式要与参数有关，最好这个公式能够用(n-1)调用的结果，来计算调用 n 的结果。例如，上面的 n! = (n-1)! * n 就是一个很好的例子。

2、 要写出递归的收敛条件。例如，上面对 n 是否为 1 的判断，这部分就是在说明递归什么时候终止。

5 函数的作用

那么，为什么我们要写函数呢？为什么不能把所有的步骤都写在主函数中，而要把一些步骤写成函数的形式呢？

我们来看下面的例子。

```
public class TestSeperator{
   public static void main(String args[]){
         System.out.println("Hello World");
         //打印分隔符 --------------------
         for(int i = 0 ; i < 20 ; i++){
               System.out.print("-");
         }
         System.out.println();
         System.out.println("你好，世界");
         //打印分隔符 --------------------
         for(int i = 0 ; i < 20 ; i++){
               System.out.print("-");
         }
         System.out.println();
         System.out.println("Bonjour tout le monde");
         //打印分隔符 --------------------
         for(int i = 0 ; i < 20 ; i++){
               System.out.print("-");
         }
         System.out.println();
         System.out.println("Hallo Welt");

   }
}
```

上面的代码，分别用英语、中文、法语、德语四种语言，输出“Hello World”这句话。在输出这四种语言的过程中，会输出三行“----------”这个分隔符，把不同的语言分隔开。运行结果如下：

```
D:\Book\chp4>javac TestSeperator.java

D:\Book\chp4>java TestSeperator
Hello World
--------------------
你好，世界
--------------------
Bonjour tout le monde
--------------------
Hallo Welt

D:\Book\chp4>
```

我们来分析一下上面的代码。由于要输出三个分隔符，因此，需要有三个一模一样的打印语句。因此，我们可以写一个函数：printSeperator()，这个函数专门用来打印分隔符。修改后的代码如下：

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

```
        System.out.println("Hello World");
        printSeperator();
        System.out.println("你好，世界");
        printSeperator();
        System.out.println("Bonjour tout le monde");
        printSeperator();
        System.out.println("Hallo Welt");
    }

    public static void printSeperator(){
        for(int i = 0 ; i < 20 ; i++){
            System.out.print("-");
        }
        System.out.println();

    }
}
```

我们可以看到，在主函数中，没有出现循环输出分隔符的语句。取而代之的，是对 printSeperator 函数的调用。这样，我们把输出分隔符的逻辑统一写到一个函数中，就能减少在主方法中，重复和类似的冗余代码，使得主方法中的代码更加清晰，从而提高代码的可读性。这就是函数的第一个作用：**减少冗余代码**。

使用函数的第二个作用是，让代码更加便于维护。例如，如果现在需求产生了变化，在输出分隔符时，希望能够输出一行“+”而不是一行“-”，此时，就必须要修改代码。

如果不使用函数的话，则必须要改动主函数中所有的输出分隔符的语句。在这个具体的例子中，在主函数中的代码要修改三处。修改后的代码片段如下：

```
System.out.println("Hello World");
//打印分隔符 ++++++++++++++++++++
for(int i = 0 ; i < 20 ; i++){
    System.out.print("+");
}
System.out.println();
System.out.println("你好，世界");
//打印分隔符 ++++++++++++++++++++
for(int i = 0 ; i < 20 ; i++){
    System.out.print("+");
}
System.out.println();
System.out.println("Bonjour tout le monde");
//打印分隔符 ++++++++++++++++++++
for(int i = 0 ; i < 20 ; i++){
    System.out.print("+");
}
```

```
System.out.println();
System.out.println("Hallo Welt");
```

而相对应的，如果使用函数的话，由于在主函数中仅仅是对同一个函数的多次调用，因此，让函数实现发生改变的时候，函数的调用不需要改变。我们只需要对函数的实现修改一次，就完成了我们的工作。修改后的代码如下：

```
public static void main(String args[]){
   System.out.println("Hello World");
   printSeperator();
   System.out.println("你好，世界");
   printSeperator();
   System.out.println("Bonjour tout le monde");
   printSeperator();
   System.out.println("Hallo Welt");
}

public static void printSeperator(){
   for(int i = 0 ; i < 20 ; i++){
         System.out.print("+");
   }
   System.out.println();
}
```

这就是函数的第二个作用：**提高代码的可维护性**。

另外，使用函数，利用函数参数的变化，能够让代码更加灵活。例如，假设现在对分隔符的长度有不同的要求，要求第一个分隔符长度为 20，第二个分隔符长度为 25，第三个分割符长度为 30。

这样的需求，如果不适用函数的话，则必须要修改这三个输出的地方，并且每个地方的输出语句都比较复杂。而如果使用函数的话，则可以让函数增加一个参数，这个参数表示分隔符的长度。修改后的 printSeperator 函数如下：

```
public static void printSeperator(int n){
   for (int i = 0; i < n; i++){
         System.out.print("+");
   }
   System.out.println();
}
```

这样，在主函数中，只要对这个函数传入不同的参数进行三次调用即可。修改后完整的主函数如下：

```
public static void main(String args[]){
   System.out.println("Hello World");
   printSeperator(20);
   System.out.println("你好，世界");
   printSeperator(25);
```

```
        System.out.println("Bonjour tout le monde");
        printSeperator(30);
        System.out.println("Hallo Welt");
    }
```

可以看到，这样修改的工作量是比较小的。而如果不使用函数呢？相对而言，工作量就比较大，在此不再具体阐述。

上面我们所说的，是使用函数的另一个好处：**能够让程序更加灵活**。

另外，我们写出的 printSeperator()函数，由于写好了打印分隔符的功能，因此，如果其他的程序员也要使用打印分隔符的功能的话，可以不用自己从头再完成，完全可以调用我们之前写好的 printSeperator()函数。因此，使用函数还能够**提高代码的可重用性**。所谓的可重用性，指的是在完成类似的功能的时候，能够利用已有的代码，从而能够对代码重复利用，减少不必要的重复劳动。

在很多编程语言中，为了能够提高代码的重用性，往往会由语言的开发者以及一些优秀的程序员一起，提供大量的函数。这些函数能够完成各种各样的通用的功能，形成一个“函数库”。而程序员在编程的时候，就是利用函数库中提供的功能，在自己的程序中调用各个函数，最终完成自己需要的逻辑。

6 自顶向下，逐步求精

我们介绍了函数的基本语法，也讲解了函数的一些作用，那么应当如何设计函数呢？在编程中，哪些部分应当写成函数呢？

下面，我们介绍一种编程的思路。使用这种思路，能够设计出比较好的程序结构，并提炼出比较合理的函数调用结构。这种思想，就是“自顶向下，逐步求精”的思想。简单的说，就是在写程序的时候，先列出程序中比较大的步骤，然后，再把每一个步骤细化，最终形成代码。

我们用一个实例来看看应当如何使用这种思想。

我们来看一个程序需求：

验证哥德巴赫猜想：任何一个大于 6 的偶数，都能分解成两个质数的和。质数，指的是除了 1 和本身之外，没有别的因子的数。

要求：输入一个整数，输出这个数能被分解成哪两个质数的和。

例如 : 输入 14

14=3+11

14=7+7

这是一个比较复杂的程序。拿到这个程序的需求之后，首先应该先设计出程序的大体思路。基本思路如下：

1、 读入一个整数 n

2、 把这个整数拆成两个数 a、b 的和

3、 判断 a 是否是质数

4、 判断 b 是否是质数

5、 如果 3、4 两个判断都为真，则输出 a 和 b
6、 如果这个整数还能拆分，则回到第 2 步。否则程序退出

很显然，2~6 步是一个循环，调整一下结构，如下：
读取整数 n
循环（把整数 n 拆成两个整数 a 和 b）{
    判断 a 是否是质数
    判断 b 是否是质数
    如果 a、b 都是质数，则输出 a 和 b
}

在上面的基本思路中，我们可以看到，“判断 a 是否是质数”和“判断 b 是否是质数”这两步操作基本一样。因此，很显然，这里我们应该写出一个函数，这个函数能够判断一个整数是否是质数。先定义这个函数如下：

```
public static boolean isPrime(int n){
}
```

经过这几步的分析，我们的思路已经逐渐细化了。现在代码如下：

```
import java.util.Scanner;
public class TestGoldBach{
   public static void main(String args[]){
         //读入整数
         Scanner sc = new Scanner(System.in);
         int n = sc.nextInt();

         循环{
               int a = 第一个整数
               int b = 第二个整数
               if (isPrime(a) && isPrime(b)){
                     System.out.println(n + "=" + a + "+" + b);
               }
         }
   }

   //判断一个整数是否是质数
   public static boolean isPrime(int a){
   }
}
```

继续细化。现在我们关注的重点是两个：1、如何把一个整数 n 拆成两个整数 a 和 b；2、如何判断一个整数是质数。

首先，我们来看拆数的逻辑。如果能够确定一个整数 a，则另外一个整数 b 也就确定了，可以通过 b = n – a 这个式子计算出 b 的值。

那么如果给出一个整数，如何确定 a 的值呢？我们可以先看一个例子。假设 n 为 14，则所有拆数的拆法是：

1 + 13
2 + 12
3 + 11
4 + 10
5 + 9
6 + 8
7 + 7

在往下就是重复的拆法了。这样，我们把第一个数当做 a，则 a 从 1 变化到 7，也就是变化到 14/2。于是，我们拆数的循环就能够分析出来了：

```
for(int i = 1; i<=n/2; i++){
   int a = i;
   int b = n-i;
   if (isPrime(a) && isPrime(b)){
         System.out.println(n + "=" + a + "+" + b);
   }
}
```

至此，主函数全部完成。接下来，完成 isPrime 方法。对于如何判断一个整数是否是质数，我们依然使用自顶向下，逐步求精的方式。由于质数，指的是除了 1 和本身之外，没有其他的因子，因此判断一个整数 a 是否是质数，只要看 2~a-1 的范围内有没有 a 的因子就可以了。

因此，主要思路如下：

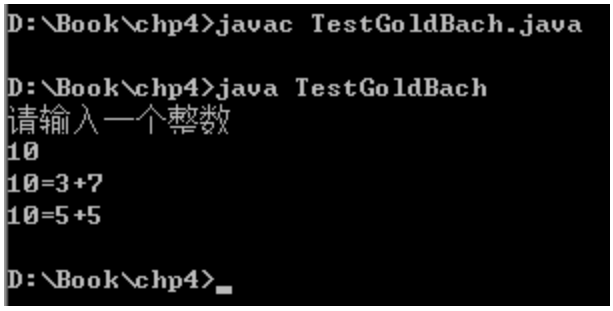
循环 i : 2~a-1{
   如果 i 是 a 的因子，则说明 a 不是质数
}
循环结束，则说明 2~a-1 都不是 a 的因子，因此 a 是质数

至此，相关代码也就呼之欲出了。完整的代码如下：

```
import java.util.Scanner;
public class TestGoldBach{
   public static void main(String args[]){
         //读入整数
         System.out.println("请输入一个整数");
         Scanner sc = new Scanner(System.in);
         int n = sc.nextInt();

         for(int i = 1; i<=n/2; i++){
               int a = i;
               int b = n-i;
               if (isPrime(a) && isPrime(b)){
                     System.out.println(n + "=" + a + "+" + b);
               }
         }
   }
```

```
    //判断一个整数是否是质数
    public static boolean isPrime(int a){
          for(int i=2; i<= a-1; i++){
                if (a % i == 0) return false;
          }
          return true;
    }
}
```

运行结果如下：



需要说明的是，上面给出的 isPrime 函数的代码并不是最好的。读者可以自己再想想，有没有办法能够优化 isPrime 函数的代码，让它的速度更快，更有效率。

上面我们介绍的，就是“自顶向下，逐步求精”的思想。可以看到，在写代码的过程中，这种思想先把解决复杂问题时，先罗列出比较粗略的步骤。在这个过程中，可以从中找到重复、可重用的部分，应该把这一部分提炼成函数。然后，再把每一个步骤细化，最后把代码完成。

这种思想，也可以说是一种比较典型的“面向过程”的思想。这种思想指导我们，在解决计算机问题的时候，应当先考虑解决问题的步骤和过程，然后把每个步骤、过程进行细化，直至最终得到代码。面向过程的思想是编程中最基本的一种思想之一，也是程序员的基本功之一。我们应当在自己的编程和练习中，锻炼自己面向过程的思想，提高自己解决计算机基本问题的能力。