4.11 通过嵌入类型使程序员的工作变轻松

除了上述的 OOP 三大要素之外，我们还有一个重要话题，那就是“通过嵌入类型使工作变轻松的结构”。虽然这主要是类的作用，但也与三大要素有关，所以我们在这里介绍一下。

可能有人会对“通过嵌入类型使工作变轻松”的说法产生怀疑，因为“嵌入类型”给人一种比较死板的感觉。而在编程语言的情况下，嵌入类型确实是可以让程序员的工作变轻松的（图 4-10）。

图 4-10 嵌入类型可以让程序员的工作变轻松

在程序中定义存储值的变量时，我们会指定整型、浮点型、字符型和数组型等“类型”。

为什么要给变量指定类型呢？对有经验的程序员来说，给变量指定类型可能已经成了他们的一种编程习惯，没有必要再重新考虑类型的含义等。

指定类型的原因有如下两个。

首先是为了告诉编译器内存区域的大小。变量所需的内存区域会根据类型自动确定，比如整型是 32 位，浮点型是 64 位（实际上，位数会根据硬件、操作系统及编译器的不同而不同）。因此，通过声明变量的类型，编

译器就可以计算出在内存中保持该变量所需的内存空间。

其次是为了防止程序发生错误。当写出整数与字符相乘或用数组减去浮点数等比较奇怪的逻辑的情况下，在编译或运行程序时就会发生显式的错误。

4.12　将类作为类型使用

OOP 中进一步推行了这种类型结构，程序员也可以将自己定义的类作为类型使用。

> OOP 中可以将类作为类型进行处理。

作为类型的类与数值型、字符串型一样，可以在变量定义、方法的参数以及返回值声明等多处进行指定（代码清单 4.11）。

代码清单4.11　使用类的类型声明

```
// 将变量的类型指定为类
TextFileReader reader;

// 将方法的参数类型指定为类
int getCount(TextReader reader) { /* 省略逻辑处理 */ }

// 将方法的返回值类型指定为类
TextReader getDefalutReader() { /* 省略逻辑处理 */ }
```

对于类型指定为类的变量、参数和返回值，如果要存储该类（及其子类）之外的实例，那么在编译和运行程序时就会发生错误[1]。

例如，在使用 Java 编写如下逻辑的情况下，在编译时就会发生错误（代码清单 4.12）。

① 在 Smalltalk 和 Ruby 等弱类型语言中，由于在变量中可以存储所有对象，所以并不会在编译时发生错误，而会在对所存储的实例进行方法调用时发生错误。

代码清单4.12 对变量赋值的类型检查

```
TextFileReader reader; ————(1)
reader = 100;  // <- 编译错误
reader = new NetworkReader();  // <- 编译错误          } (2)
reader = new TextFileReader();  // <- 编译通过 ————(3)
```

代码中的(1)处将reader变量声明为TextFileReader类型，因此reader变量中只可以存储从TextFileReader类创建的实例[1]。(2)处要将数值和其他类的实例存储到该reader变量中，因此会发生编译错误。(3)处存储了TextFileReader类的实例，因此编译通过。

同样地，关于方法的参数和返回值类型，如果指定了错误的实例，也会发生错误(代码清单4.13)。

代码清单4.13 对方法的参数和返回值的类型检查

```
obj.getCount(new JLabel()); // <- 编译错误（参数类型错误）

TextReader getDefaultReader() {
    return new JButton(); //  <- 编译错误（返回值类型错误）
}
```

在机器语言和汇编语言时代，这种**类型检查**结构几乎是不存在的。高级语言和结构化语言中导入了一些结构来检查编程语言自带的数据类型和结构体[2]的使用方法。OOP中则更进一步，通过将汇总变量和方法的类定义为类型，从而将类型检查作为一种程序规则强制要求。

像这样，编译器和运行环境会匹配类型来检查逻辑，因此程序员的工作就会轻松许多。

另外，这种类型检查结构会根据编程语言的种类的不同而分为强类型

① 还可以存储TextFileReader子类的实例。

② 这是指能够集中持有多个值的数据类型。C语言中使用struct关键字进行定义。

和弱类型两种。**强类型**方式在程序编译时检查错误，Java 和 C# 等就采用这种方式。**弱类型**方式在程序运行时检查错误，Ruby 和 Smalltalk 等采用的就是这种方式。

> 类型检查分为强类型和弱类型两种方式。

4.13　编程语言“退化”了吗

下面我们暂时换一个话题。关于预防程序错误，在编程语言的规范上也发生了与强化类型检查目的相同的变化。

比如，比较新的编程语言 Java 并不支持 GOTO 语句（该语句导致了面条式代码的产生）。Java 沿用了 C 语言和 C++ 的基本语言规范，并摒弃了一些功能，包括 GOTO 语句、显式指针、结构体、全局变量和宏等。Java 开发者认为这些功能会让程序变得难懂，或者容易出错，所以最好一开始就不提供。

也就是说，随着编程语言的进化，其功能并不是在一味地增加，还会被删除，使编程语言朝着看似“退化”的方向发展。人类的进化过程也同样如此。在脑容量变大的同时，人类不使用的尾巴和肯肠则逐渐退化了。

4.14　更先进的 OOP 结构

到这里为止，我们介绍了 OOP 的三大要素——类、多态和继承，以及类型检查和语言规范的变化。

不过，Java 和 C# 等比较新的编程语言提供了更先进的功能[1]，其中比较典型的有包、异常和垃圾回收。设计这些功能是为了促进重用、减少错误等。下面我们就来简单地了解一下这些功能。

① Smalltalk 和 Objective-C 等早期语言也随着语言规范和类库的扩展而变得逐渐支持这些功能。

4.15 进化的 OOP 结构之一：包

首先来介绍一下**包**。

前面我们介绍了具有汇总功能的类结构，而包是进一步对类进行汇总的结构（图 4-11）。

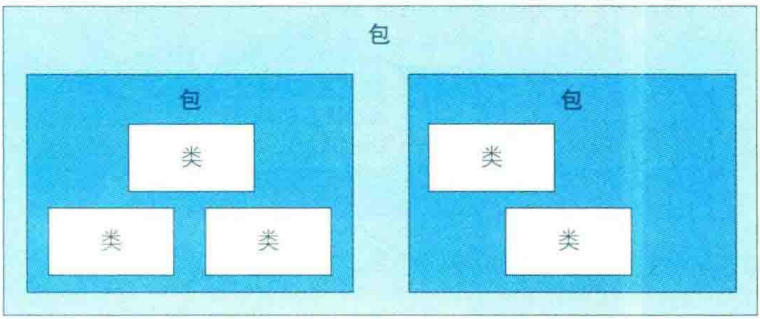


图 4–11 包的结构

包只是进行汇总的容器，它不同于类，不能定义方法和实例变量。有的读者可能会想这种结构有什么用，为了回答这个问题，我们来联想一下文件系统中的目录（文件夹）。虽然目录只是存储文件的容器，但是通过给目录命名，并在其下存储文件，文件管理就会变得非常轻松。反之，我们再想象一下没有目录、所有文件都存储在根目录下的状态。如果文件系统是这样的结构，那么使用起来一定很不方便[1]。包的作用也是如此。它与目录一样，除了类之外，还可以存储其他包，从而创建层次结构。

采用这种结构，即使是代码行数达到几十万、几百万行的大型应用程序，也可以全部放到几十个包中。通过确定包的作用，并将作用相关的类汇集在一起，使用起来就会非常方便。

包还具有防止类名重复的重要作用（图 4-12）。比如，Java 采用类似于

① 笔者在刚参加工作时使用的 1.0 版本的 MS-DOS（Windows 的前身）实际上并不支持层次化目录。

网络域名的形式来命名包，首先是国家名称，然后是组织类型（公司、学校、政府等），接下来是组织名称，这是基本的命名规则。例如，日经 BP 社就是 jp.co.nikkeibp。只要遵循该规则，无论其他组织或者国家编写什么类，都无须关心类名是否重复，从而实现重用。

图 4-12　使用包来避免类名冲突

4.16　进化的 OOP 结构之二：异常

接下来介绍异常。

如果用一句话来概括异常，那就是：采用与返回值不同的形式，从方法返回特殊错误的结构。

像网络通信故障、硬盘访问故障或者数据库死锁等，都属于“特殊错误”。除了故障之外，也存在无法返回正常的返回值的情况，比如文件读取处理中返回 EOF（End Of File，文件结束符）。在传统的子程序结构中，通常使用错误码来处理这种情况。具体来说，就是确定值的含义，并将其作为子程序的返回值返回，例如错误码为 1 时表示死锁、为 2 时表示通信故障、为 3 时表示其他致命错误等。

但是这种方法存在两个问题。

第一个问题是需要在应用程序中执行错误码的判断处理。如果忘记编写判断处理，或者弄错值，那么在发生故障时就很难确定具体原因。另外，在添加、删除错误码的值的情况下，程序员需要亲自确认所有相关的子程序来改写。

第二个问题是判断错误码的相同逻辑在子程序之间是连锁的。通常在调用端的子程序中必须编写判断错误码的值的逻辑。另外，当调用端的子程序中无法执行错误的后续处理时，就会返回同样的错误码。像这样，如果错误码的判断处理在整个应用程序中连锁，那么程序逻辑就会变得很冗长（图 4-13）。

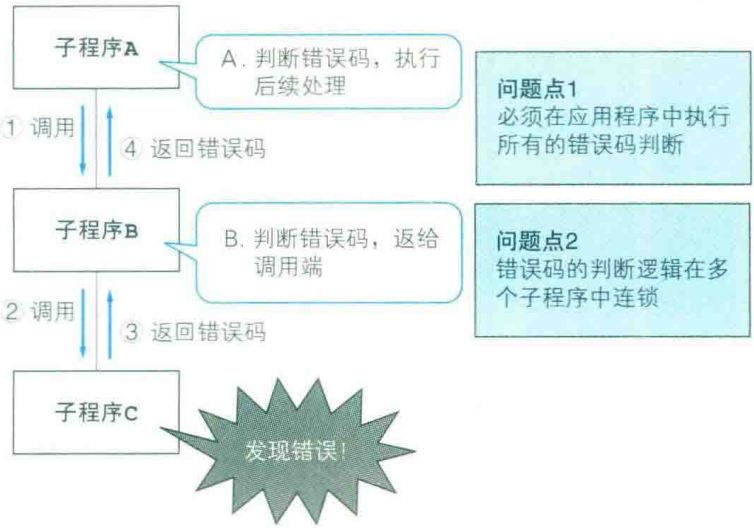


图 4-13　基于错误码方式的错误处理的连锁

异常就是用于解决以上问题的结构。

异常结构会在方法中声明可能会返回特殊错误。这种特殊错误的返回方式就是异常，其语法不同于子程序的返回值。

在声明异常的方法的调用端，如果编写的异常处理逻辑不正确，程序就会发生错误[①]，这样就解决了第一个问题。

另外，在声明异常的方法的调用端，有时在发生错误时并不执行特殊处理，而是将错误传递给上位方法。在这种情况下，只需在方法中声明异

① 在 Java 等强类型语言中会发生编译错误，而在 Ruby 等弱类型语言中则会发生运行时错误。

常，没有必要编写错误处理，这样就解决了第二个问题（图 4-14）。

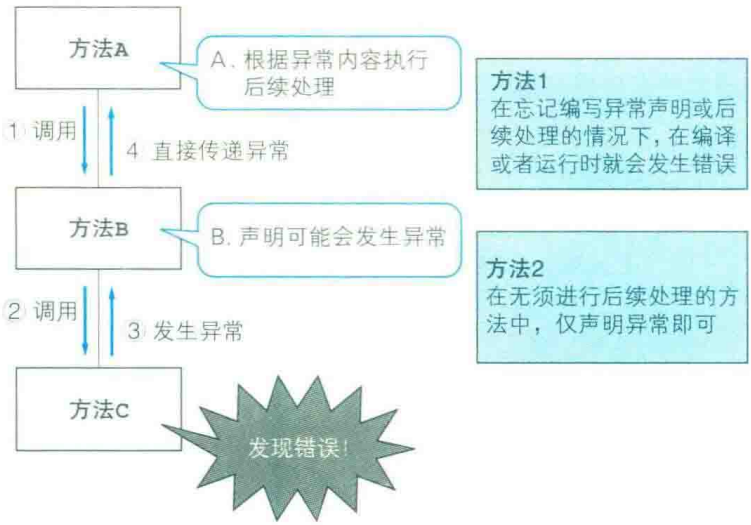


图 4-14　基于异常结构的错误处理

这种结构可以将重复的错误处理汇总到一处，并且当忘记编写必要的错误处理时，编译器和运行环境会进行提醒，非常方便。这种结构可以达到去除冗余、防止错误的效果。

4.17　进化的 OOP 结构之三：垃圾回收

我们在前面介绍类的“创建很多个”的功能时，提到过在运行时创建实例的话题，但并未涉及如何删除实例的相关内容。当创建实例时，实例变量的内存区域就会被确保。当采用 OOP 编写的应用程序运行时，为了从类创建实例并进行动作，根据应用程序的不同，有时可能会在运行时创建很多实例。

在 C 和 C++ 等之前的编程语言中，需要在应用程序中显式地指示删除不再需要的内存区域。但是，在编写删除实例的处理代码时需要多加注意。如果误删了其他地方仍在使用的实例，当之后使用该实例的逻辑运行时，程

序的动作就会错误。反之，如果忘记删除任何地方都不再使用的实例，不需要的实例就会不断增多，从而占用内存，造成内存泄漏。在 OOP 中，使用“创建很多个”功能，我们可以自由地创建实例，但在删除时需要慎重进行。

Java 和 C# 等很多 OOP 中采用了由系统自动进行删除实例的处理的结构，该结构称为**垃圾回收**（Garbage Collection，GC）。

在这种结构中，删除内存中不再需要的实例是系统提供的专用程序——垃圾回收器的工作。采用这种结构，程序员就不用再编写容易出错的删除实例的处理了（图 4-15）。

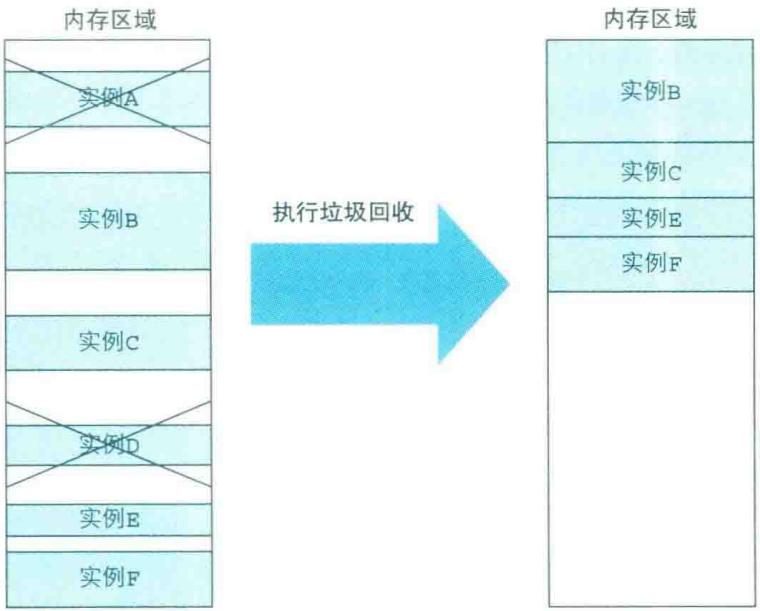


图 4-15　垃圾回收器删除内存中不再需要的实例

这种结构不将容易出错的内存释放处理作为编程语言的语法提供，而是由系统自动执行。正如前面介绍的那样，这也可以看作一种为了让程序员的工作变轻松而“退化”的语言规范。

另外，关于垃圾回收的详细内容，我们将在第 5 章进行介绍。

4.18　对 OOP 进化的总结

本章介绍了 OOP 提供的能让程序员的工作变轻松的功能，包括类、多态、继承、包、异常和垃圾回收，这些功能都有助于编写出高质量的程序。第 3 章中介绍了机器语言到结构化语言的进化，这里我们再整理一下 OOP 中又发生了什么样的进化。

编程语言进化到高级语言时，通过高级命令实现了表现力的提高，使用子程序去除了重复逻辑。

在接下来的结构化语言中，又强化了有助于维护程序的功能，导入了三种基本结构、无 GOTO 编程以及强化子程序独立性的结构。

为了进一步提高程序的可维护性和质量，OOP 中提供了一些通过添加限制来降低复杂度的功能。另外，还大幅强化了构件化、可重用的功能。下面我们对这些内容加以总结，如图 4-16 所示。

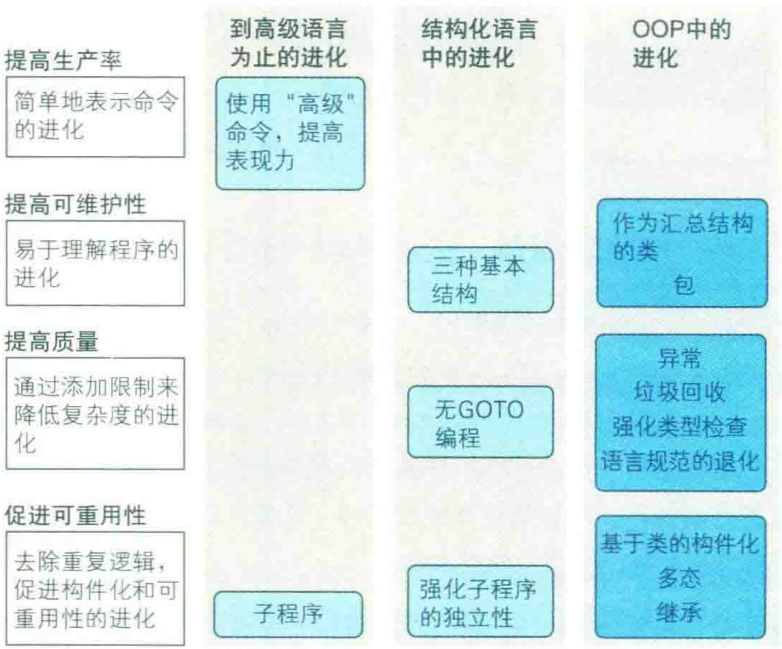


图 4-16　编程语言的进化（之二）

从以上内容可以看出，OOP 绝不是替代了传统的编程技术，而是以之前的编程技术为基础，并针对之前的技术缺点进行了补充。与传统的编程语言相比，OOP 导入了许多变化非常大的独特结构，甚至可以说它是编程语言的突然变异。不过，在编程技术的不断发展中，这些结构也是必然会出现的。

为了写出高质量、可维护性强且易于重用的软件，请大家一定要使用 OOP。这是因为 OOP 是凝聚了前人智慧与研究成果的编程技术。

4.19 决心决定 OOP 的生死

有人说面向对象不是结构的问题，而是一种思想。还有人说在 C++ 和 Java 普及之前，只要有干劲，无论是 C 语言还是 COBOL，都可以实现面向对象编程。笔者认为，从某种意义上来说，这些观点不无道理。

这是因为 OOP 是一种手段，其目的不在于被人们使用，而是提高程序的质量、可维护性和可重用性。

本章开头介绍过，OOP 是去除程序冗余、进行整理的编程技术。我们还打比方说，这就像打扫凌乱的房间需要吸尘器和整理架一样。

当然，仅准备新的吸尘器和使用方便的整理架，房间并不会变整洁。更重要的是要有打扫房间的决心，以及将这种决心转变为行动的执行力。

编程中也是一样。仅使用类、多态和继承等结构，并不能提高程序的可维护性和可重用性。这些结构弄错一个，都会让问题变得很棘手。因此，切不可胡乱使用，否则程序就会变得难以理解。

特别是像 OOP 这样有趣的结构，人们一旦对其有所了解，无论如何都想立即使用，这也是人之常情。如果是出于兴趣而编写的程序，这样做倒没有什么关系，但如果是实际工作中使用的程序，这样就会很麻烦。切记我们的目的是编写出高质量、易于维护和可重用的程序，面向对象只是实现该目的的一个手段而已。

能否充分发挥 OOP 的功能，取决于使用它的程序员。我们首先要思考怎么做才能使程序更容易维护和重用。然后考虑使用三种基本结构和公用子程序来进行实现。如果这样还不够，那就轮到类、多态和继承大显身手了。

第5章 理解内存结构：程序员的基本素养

本章的关键词

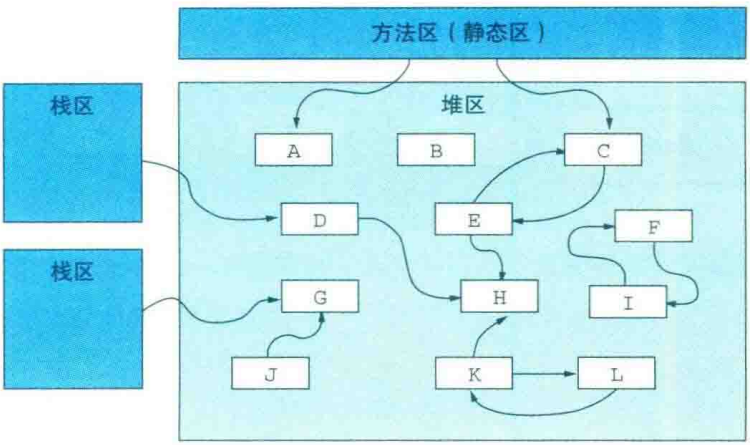
编译器、解释器、虚拟机、线程、静态区、堆区、栈区、指针、方法表

热身问答

在阅读正文之前，请挑战一下下面的问题来热热身吧。

问题

请从下图中选出是垃圾回收对象的实例（A~L 的长方形表示实例，箭头表示引用关系）。

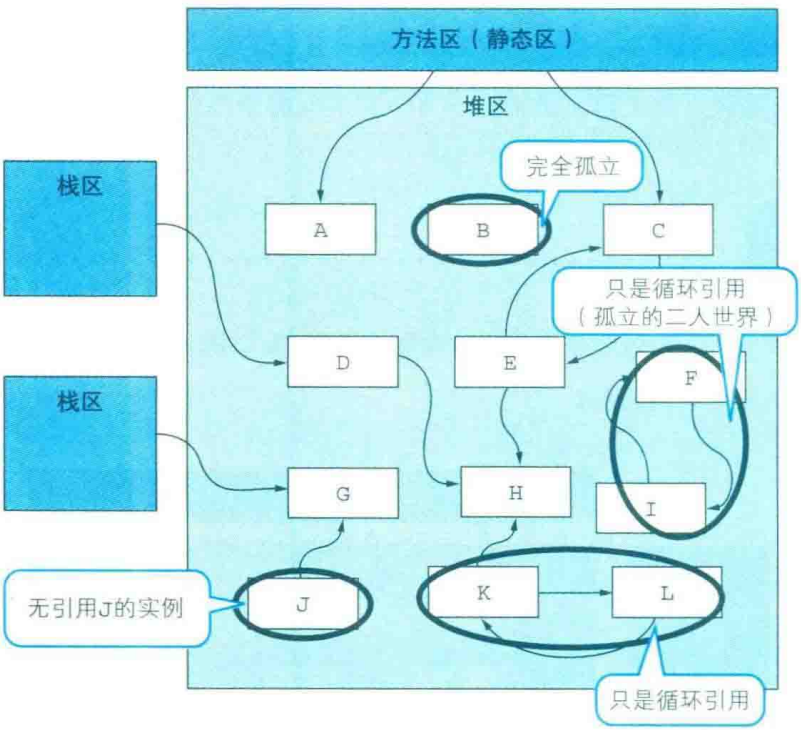


答案

共有 B、F、I、J、K 和 L 六个实例。

解析

从实例网络的根部——栈区和方法区无法到达的实例就是要删除的对象。



本章重点

第 4 章从程序员的视角介绍了 OOP 结构的便捷性。本章将稍微转换一下视角，来介绍使用 OOP 编写的程序在计算机中是怎样运行的。

本章是一个独立的话题。对于使用 OOP 编程的人来说，本章内容是其应该掌握的基本知识。掌握了内部运行机制之后，也能够更深入地理解 OOP 的功能。所以借此机会，希望大家能够将之前不明白的地方也一并掌握。

5.1 理解 OOP 程序的运行机制

在使用 Java、C# 等较新的编程语言时，我们一般并不关心使用这些语言编写的程序实际是如何运行的。使用 OOP 编写的程序的特征在于内存使用方式，但如果大家在编写程序时完全不了解内部运行机制，那么编写的程序可能就会占用过多内存，从而影响机器资源。有时即便在调试时发现了问题，也有可能什么都做不了。

因此，关于自己所编写的程序的运行机制，我们需要了解一些最基本的知识。在汇编语言占据主流的时代，这种最基本的知识就是硬件寄存器的结构，在 C 语言时代则是指针结构。而在编程语言进一步进化的今天，笔者认为最基本的知识则是“理解内存的使用方法”。这也可以说是使用 OOP 的程序员的基本素养。

5.2 两种运行方式：编译器与解释器

我们首先来介绍一下程序的基本运行方式，大致可以分为编译器方式和解释器方式两种（图 5-1）。