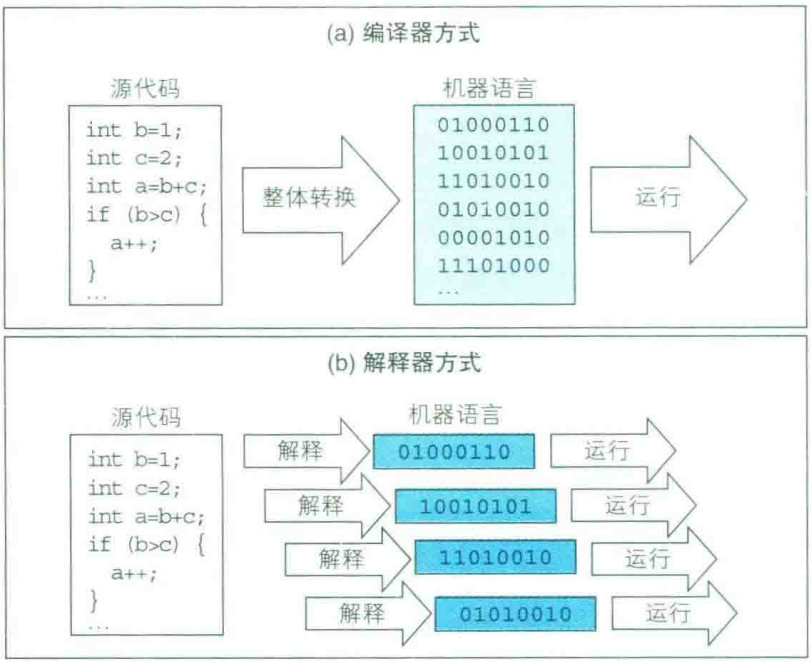

图 5-1 编译器方式和解释器方式

编译器方式是将程序中编写的命令转换为计算机能够理解的机器语言之后再运行的方式。将命令转换为机器语言的程序称为编译器。

解释器方式则是将源代码中编写的程序命令边解释边运行的方式。这种方式能读取源代码并立即运行，因此不需要编译器。如果程序有语法错误，运行时就会发生错误。

这两种方式各有优缺点。

编译器方式的优点是运行效率高。计算机直接读取机器语言进行动作，没有解释程序命令的多余动作，因此运行速度快。而缺点是运行前会耗费一些时间。这是因为程序无法立即运行，需要先进行编译。另外，在发现错误的情况下，还需要将错误修正后才可以运行。

解释器方式的优点是可以立即运行。在使用这种方式的情况下，编写完程序后就可以立即运行一下查看结果。另外，该方式还有一个优点，就是可以确保不同平台（机器、操作系统）之间的兼容性。如果将机器语言的

代码发布到其他环境的硬件中，通常是无法运行的[1]。不过，解释器方式会匹配机器环境进行解释、运行，因此无须对程序进行修改，就可以在多种环境下运行。该方式的缺点是运行速度慢，与编译器方式的优点正好相反。

> 程序的运行方式分为编译器方式和解释器方式两种。编译器方式的运行效率高，而解释器方式能使同一个程序在不同的环境中运行。

这两种方式各有优劣，我们通常会根据具体情况进行选择。在特定的机器环境下，如果应用程序要求较高的处理性能，那么通常会采用编译器方式。政府和银行系统、企业的基础系统等大多采用编译器方式。

对于经由互联网下载到各种机器中运行的软件，解释器方式更能发挥优势。其中，为了提高在 Web 浏览器上显示的画面的操作性而使用的脚本语言等就是典型的例子。另外，即使最终采用编译器方式运行，为了节省编译操作的时间，在有些开发环境中也会使用解释器来执行从编码到调试的工作。

这两种运行方式与编程语言之间基本上没有什么对应关系。实际上，许多编程语言都既支持编译器方式，又支持解释器方式。

不过在比较新的编程语言 Java 和 .NET 中，情况则稍有不同。有趣的是，这些编程环境中采用的并非这两种方式，而是**中间代码方式**（图 5-2）。

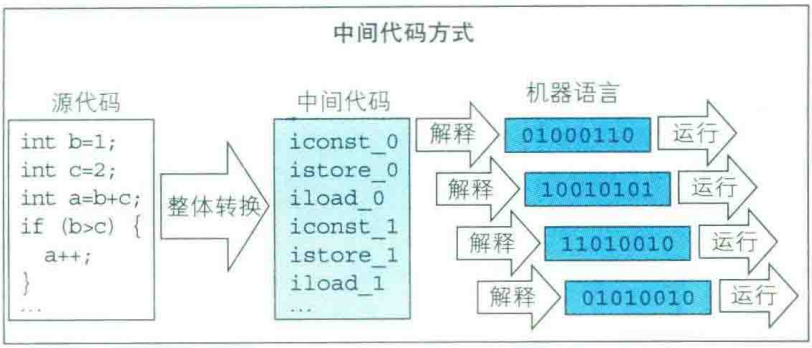

图 5-2　中间代码方式

① 不过，如果使用硬件或者操作系统的虚拟化技术，那么也可以运行不同的机器语言代码。

这种方式首先使用编译器将源代码转换为不依赖于特定机器语言的中间代码，然后使用专门的解释器来解释中间代码并运行。

这样做是为了汲取两种方式的优点。既可以将同一个程序发布到不同的机器上，又可以发扬编译器运行效率高的优点。通常这种贪婪的做法容易导致“鱼与熊掌不可兼得”的结果，但得益于硬件的进步和各种机器共存的互联网环境，这种方式最终得以实现。

> 采用中间代码方式，同一个程序可以在不同的运行环境中高效地运行。

5.3 解释、运行中间代码的虚拟机

下面我们来简单介绍一下实现中间代码方式的结构。由于中间代码的命令是不依赖于特定运行环境的形式，所以 CPU 无法直接读取并运行。因此，我们需要一种解释中间代码并将其转换为 CPU 能够直接运行的机器语言的结构。这种结构一般被称为**虚拟机**（Virtual Machine，VM）[1]。比如 Java 中的 Java VM（Java Virtual Machine，Java 虚拟机）就是虚拟机结构的一个例子。各个平台都有相应的 Java VM，运行时读取 Java 的中间代码——字节码，转换为该平台使用的机器语言，从而运行程序（图 5-3）。

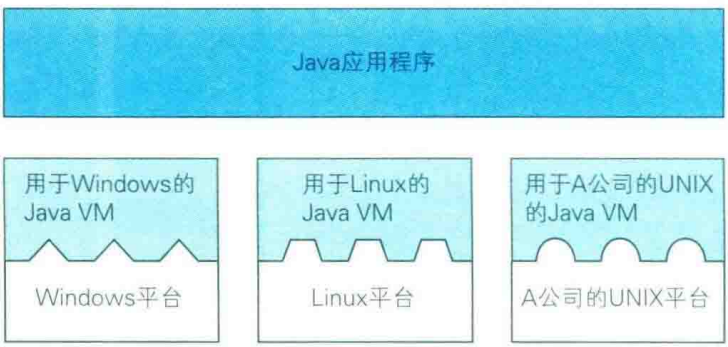

图 5-3 虚拟机吸收平台间的不同

① 这里介绍的是编程语言层面的虚拟机，除此之外还存在硬件、操作系统等层面的虚拟机。

微软开发的 .NET 的结构也是如此。但是，由于在 .NET 中，C# 和 Visual Basic 等各种编程语言使用共同的虚拟机，所以我们称之为公共语言运行时（Common Language Runtime，CLR）。

之所以能进行大规模的重用，即创建类库和框架等可重用构件群，是因为 OOP 提供了类、多态和继承等优秀的编程功能。而之所以能进行大范围的重用，即能在各种平台上使用创建的软件构件，则是因为使用了虚拟机结构。这种虚拟机结构为促进软件重用做出了重要贡献。

5.4 CPU 同时运行多个线程

在介绍了程序的运行机制之后，接下来我们再介绍另外一个重要的概念——**线程**（thread）。

作为一个计算机术语，线程的意思是“程序的运行单位”[1]。英文单词“thread”是“线”的意思，进一步延伸出“生命线”“寿命”的含义，而“生命线”这一含义就能很贴切地表现计算机处理的运行单位。

听到“程序的运行单位”这样的介绍，有的读者可能会联想到在计算机上独立运行的应用程序，比如电子邮件软件、Web 浏览器和电子表格软件等。它们确实也是计算机中的一种运行单位，但这一单位通常被称为**进程**（process）[2]。

进程表示的单位比线程大，一个进程可以包含多个线程。实际上，许多应用程序都是使用多个线程实现的，比较常见的例子有文字处理软件中的文本创建和打印处理、Web 浏览器中的请求处理和“停止”按钮的处理等。这些独立的处理是通过一个应用程序中作用不同的多个线程同时并发运行来实现的（图 5-4）。

① 在大型机盛行的时代，这称为“任务”（task）。而在如今的 UNIX 和 Windows 等开放平台中，通常都使用“线程”一词。

② 在大型机中一般称为“job”。

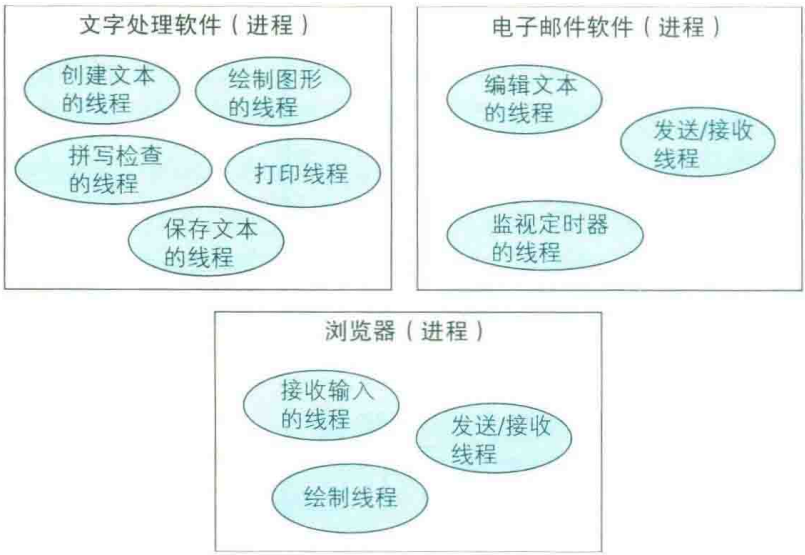

图 5–4　进程和线程的示例

虽然我们在这里采用了“多个线程同时并发运行”的说法，但是严格来说，这样的表述并不准确。实际上，计算机的心脏——CPU 在某个时刻只能执行一个处理。那么，我们为什么说并发处理能够实现呢？这是因为 CPU 会依次循环执行多个线程的处理。当 CPU 执行线程的处理时，并不是将一个线程从开始一直执行到结束，而是仅在非常短的规定时间（通常以毫秒为单位）内执行。在执行完这段规定时间后，即使该线程的处理还未完成，计算机也会暂时中断处理，转而处理下一个线程。下一个线程也是如此，仅执行规定的时间，然后马上跳转到下一个线程。虽然 CPU 实际上是在交替执行多个线程的处理，但是由于其处理速度非常快，所以在计算机的使用者看来就像在同时执行多个作业一样[1]（图 5-5）。这种能够同时运行多个线程的环境称为**多线程环境**。这种多线程功能基本上都是作为操作系统的功能提供的。

① 不过，在一台计算机中搭载多个 CPU 的多处理器结构的情况下，有多少个 CPU，就可以并发执行多少个处理。

图 5-5 依次执行多个作业的多线程功能

为什么要进行这么复杂的操作呢？原因就在于这样可以减少等待时间，提高整体的处理效率。

计算机的工作并不只是执行机器语言的命令，还有读写硬盘、使用打印机打印、与其他联网的计算机进行通信、等待来自鼠标和键盘的输入，等等，需要与外部进行很多交互。这种与外部的交互对人类而言可能只是一瞬间的事，但对以微秒和毫秒为单位进行作业的 CPU 来说却是很长的等待时间。因此，如果在此期间只是默默等待，那么 CPU 不执行作业的空闲时间就会变得非常多。

为了避免出现这种状态，CPU 不会集中于一个线程的作业，而是会同时执行多个线程的作业。

> 通过并发处理多个线程，可以高效利用 CPU 资源。

5.5 使用静态区、堆区和栈区进行管理

接下来介绍内存使用方式。正如本章开头介绍的那样，OOP 运行环境的特征就在于内存使用方式。不过，OOP 运行环境与使用传统编程语言编写的程序的运行环境也存在许多共同点。因此，这里我们抛开 OOP 特有的部分，先为大家介绍一下程序运行时一般的内存使用方式。

程序的内存区域基本上可以分为静态区[1]、堆区和栈区三部分（图 5-6）。

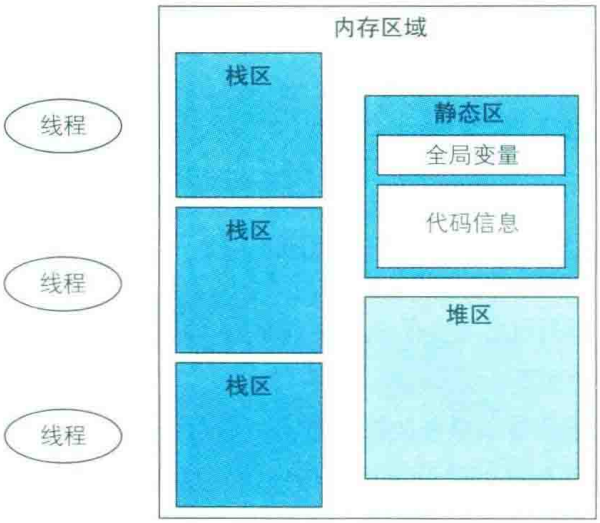

图 5-6　三种内存区域

> 程序的内存区域分为静态区、堆区和栈区三部分。

下面依次对各个区域进行介绍。

静态区从程序开始运行时产生，在程序结束前一直存在。之所以称为“静态”，是因为该区域中存储的信息配置在程序运行时不会发生变化。静态变量，即全局变量和将程序命令转换为可执行形式的代码信息就存储在该区域中。

堆区是程序运行时动态分配的内存区域。“堆”的英文“heap”有“许多”“大量”之意。由于在程序开始运行时预先分配大量的内存区域，所以命名为堆区。

堆区是在程序运行过程中根据应用程序请求的大小进行分配的，当不

① 静态区分为代码区和变量区，也分别称为程序区和静态变量区，本书将二者统称为静态区。

再需要时就将其释放。最好为堆区划分一块较大的空区域，以便有效利用内存。在多个线程同时请求内存时，也需要保持一致性，因此一般由操作系统或虚拟机提供管理功能。实际的分配和释放处理都是通过该管理功能进行的。

栈区是用于线程的控制的内存区域。堆区供多个线程共同使用，而栈区则是为每个线程准备一个。各个线程依次调用子程序（在 OOP 中是方法）进行动作。栈区是用于控制子程序调用的内存区域，存储着子程序的参数、局部变量和返回位置等信息。

栈区这一名称来源于其使用方法。“栈”的英文“stack”有“堆积”的含义。栈区中不断堆积新的信息，使用时从最上面放置的信息开始使用（图 5-7），这种用法称为**后入先出**（Last In First Out，LIFO）。子程序调用是嵌套结构，在调用的子程序的处理结束之前，再次调用子程序。通过这种方式，可以高效地使用内存区域。

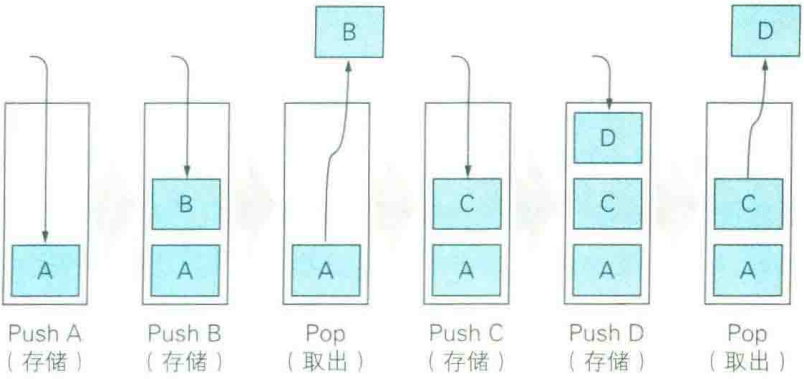

图 5-7　栈区的用法

三种内存区域的特征如表 5-1 所示。

表 5-1 三种内存区域的特征

类　型	静 态 区	堆　区	栈　区
用法	在应用程序开始运行时分配	开始时分配一定的区域，之后会根据需要再为应用程序分配	后入先出方式
存储的信息	全局变量、运行代码	任意（取决于应用程序）	调用的子程序的参数、局部变量和返回位置
分配单位	为整个应用程序分配一个	为一个系统或应用程序分配一个	为每个线程分配一个

5.6 OOP 的特征在于内存的用法

到这里为止，我们介绍了一般的程序运行环境，包括编译器、解释器、虚拟机、线程和内存管理等。对于程序员来说，这些内容可以说是必须掌握的常识。如果之前读者有什么地方不明白，请一定借此机会牢记。

接下来我们将探讨 OOP 特有的话题。即使是使用 OOP 编写的程序，其基本结构也与之前介绍的一样。不过，使用 OOP 编写的程序的内存用法与之前有很大不同，下面我们就来详细地说明一下。

虽然统称为 OOP，但实际上存在许多编程语言，如 Java、C++、C#、Smalltalk、Ruby、Visual Basic.NET 和 Objective-C 等。这些编程语言都支持类、多态和继承这三大要素的功能，不过语言规范会根据编程语言的不同而稍微存在区别。另外，根据编译器、操作系统等的不同，运行时的机制有时也会不一样。

接下来，我们以 OOP 中最普及的 Java 为例来介绍程序的运行机制。不过，这里的目的是从原理上介绍 OOP 的结构，而不是介绍 Java 的详细结构。因此，如果读者想要知道 Java 等具体编程语言的运行环境的结构，请参考相关图书和产品手册等。

5.7 每个类只加载一个类信息

我们在第 4 章中介绍过，当使用 OOP 编写的程序运行时，会从类创建实例进行动作。而实际上，当程序运行时，在创建实例之前，需要将对应的类信息加载到内存中。

这里所说的类信息，是不依赖于各个实例的类固有的信息。类信息的内容根据编程语言的不同而有所变化，但无论哪种语言，最重要的都是方法中编写的代码信息。即使是从同一个类创建的实例，实例变量的值也各有不同，但方法中编写的代码信息是不会变的。因此，代码信息是类固有的信息，每个类只加载一个。

> 每个类只加载一个方法中编写的代码信息。

加载类信息的时间点大致分为两种方式：一种方式是预先整体加载所有类信息；另一种方式是在需要时依次将类信息加载到内存中。

前者是在应用程序开始执行时将所定义的类全部加载到内存中。这种在最开始就加载所有代码信息的方式是传统编程语言中采用的一般结构。在 OOP 中，考虑到与 C 语言的兼容性而创建的 C++ 也采用该方式。

Java 和 .NET 等则采用后一种方式。在使用该方式的情况下，每当所执行的代码使用新类时，都会从文件中读取对应的类信息并加载到内存中。此时，该信息会与其他已经加载的类信息进行关联。虽然采用此种方式会在每次读取新类时都产生额外开销，导致运行性能变差，但由于实际上只使用运行的代码所占用的内存，所以能够减少整体的内存使用量。另外，使用该方式还可以保证运行时的灵活性，比如在各个网络中分散管理的程序文件在运行时更容易结合起来运行等。

加载类信息的内存区域相当于图 5-6 中的静态区。不过，Java 中采用依次加载所需的类信息的方式，在运行时内存配置会发生变化，因此将加

载类信息的内存区域称为方法区，而不是静态区[1]。本章及之后的讲解都将使用“方法区”这一术语。不过，为了便于与前面的讲解进行对比，我们会根据需要使用“方法区（静态区）”的表述。

5.8 每次创建实例都会使用堆区

接下来介绍实例的结构。

在执行创建实例的命令（Java 中是 new 命令）时，程序会在堆区分配所需大小的内存，用于存储该类的实例变量。这时，为了实现指定实例来调用方法的结构，还需要将实例和方法区中的类信息对应起来。

第 4 章的图 4-4 展示了针对每个实例都有方法和变量在内存中展开的情况，不过它只是一个抽象的示意图。其实在每个类中，方法中编写的代码信息都只存在于一个位置，通过实例指向该位置进行管理。

以第 4 章中编写的持有三个方法的 TextFileReader 类为例，三者间的关系如图 5-8 所示。

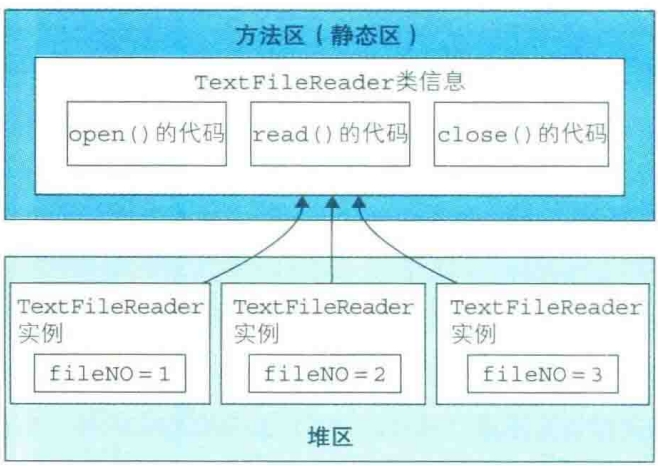

图 5-8 堆区中分配内存的实例

① 在 Java 中，加载的类信息除了方法之外，还包括类本身定义的类变量（static 变量）、常量信息及类名和方法名等符号信息。

OOP 中的内存使用方法的最大特征就是实例的创建方法。

在使用传统编程语言编写的程序中，通过将代码和全局变量配置在静态区，并使用栈区传递子程序的调用信息，几乎可以实现所有处理。堆区在执行分配处理时会耗费额外开销，另外，如果在使用完后忘记释放内存，就会导致不需要的内存一直残留，从而容易造成内存泄漏问题，因此堆区的使用并不广泛。

不过，在 Java 等诸多 OOP 中，创建的实例都被配置在堆区中。程序员必须意识到“使用 OOP 编写的程序会大量使用有限的堆区来运行”这一点。

> 使用 OOP 编写的程序会大量使用有限的堆区来运行。

由于近来硬件的性能得到了极大的提升，从整体来看，从堆区分配内存、释放内存的额外开销已经变得非常小了。另外，得益于垃圾回收功能（后述），我们也几乎不用在意忘记释放不再使用的内存而导致内存泄漏问题了。

不过，虽说内存容量变大了，但同时创建几万、几十万个实例还是会使 CPU 的负荷变大，从而造成内存区域不足，甚至引发系统故障。

因此，当编写的应用程序要一下子读取大量信息并进行处理时，我们必须预先规划该处理会使用多少堆区。

5.9 在变量中存储实例的指针

本节将为大家介绍创建的实例是如何存储到变量中的。首先我们再来看一下第 4 章中从 `TextFileReader` 类创建实例的代码示例。

代码清单5.1 创建TextFileReader的实例

```
TextFileReader reader = new TextFileReader();
```

通过前面的介绍我们已经知道，创建的 `TextFileReader` 类的实例被配置在堆区中。

那么，变量 `reader` 中存储的是什么信息呢？

该变量并不一定在堆区中。如果是方法的参数或局部变量，则配置在栈中，也有可能配置在方法区（静态区）的类信息中。

因此，变量 reader 中存储的并不是 TextFileReader 类的实例本身，而是堆区中创建的实例的**指针**[1]。

> 存储实例的变量中存储的并不是实例本身，而是实例的指针。

听到指针，有的人可能会不由自主地抗拒，特别是有的编程初学者会认为指针非常难。

请大家放心。指针很简单。用一句话来说，指针就是“表示内存区域的位置的信息”。假设堆区中分配的内存区域是土地，那么指针就相当于住址。不管土地多么辽阔，住址的形式都是省、市、区、街道、门牌号。指针也是如此，无论内存区域多大，其表示形式都是固定的。采用该方法，就可以不用在意实例的大小，一直使用相同的形式来管理实例（图 5-9）。

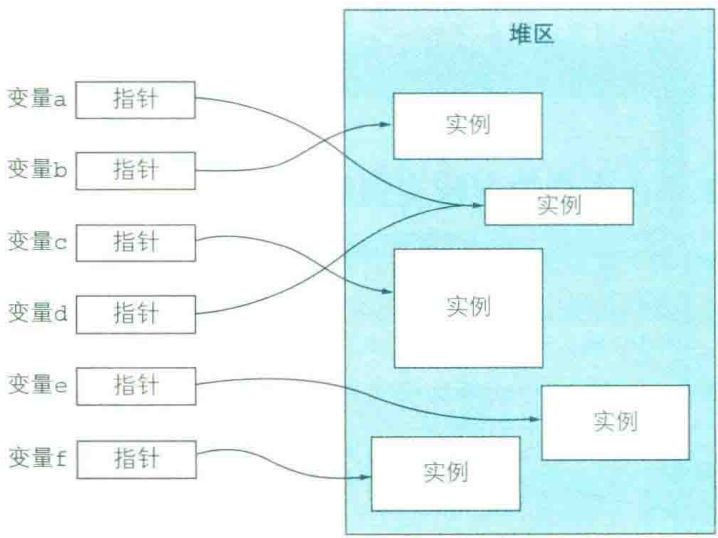

图 5-9 在变量中存储指针

① Java 中将其称为“引用”，而非“指针”。

C++ 中可以指定是在变量中存储实例本身还是指针，而 Java 无法在变量中直接存储实例。随着 Java 的语言规范逐渐变得简单，堆区中存储的实例的内存管理可以通过垃圾回收自动进行，因此，通常在堆区创建实例，并在变量中存储指针。

5.10 复制存储实例的变量时要多加注意

下面我们来介绍一个编程时应该注意的事项。

那就是复制存储实例的变量时的动作。如果在编程时不加以注意，就可能会引起难以察觉的 bug。

接下来，我们使用一段 Java 代码来对此进行说明。

首先定义一个非常简单的 Person 类，该类仅持有一个姓名的实例变量（代码清单 5.2）。

代码清单5.2 简单的Person类

```
class Person {              // Person 类
  private String name;  // 持有姓名的实例变量
  public void setName(String nm) {  // 设置姓名的方法
    this.name = nm;       // 将姓名设置为实例变量
  }
  public String getName() {  // 获取姓名的方法
    return this.name;          // 返回持有的姓名
  }
}
```

然后，我们准备两个变量来存储 Person 类的实例，并分别设置姓名（代码清单 5.3）。

代码清单5.3 给Person设置姓名

```
Person musician = new Person(); // (1) 创建 Person 实例
Person john = musician; // (2) 赋给变量 john
john.setName("John");   // (3) 给变量 john 设置姓名
Person paul = musician; // (4) 赋给变量 paul
```

```
paul.setName("Paul");    // (5) 给变量 paul 设置姓名
```

之后，输出两个变量的姓名（代码清单 5.4）。

代码清单5.4 输出Person的姓名

```
System.out.println(john.getName()); // 输出变量 john 的姓名
System.out.println(paul.getName()); // 输出变量 paul 的姓名
```

结果如下所示（代码清单 5.5）。

代码清单5.5 输出Person姓名的结果

```
Paul
Paul
```

好奇怪！明明将姓名“John”赋给了变量 john，将姓名“Paul”赋给了变量 paul，为什么最终都变为“Paul”了呢？如果你认为这是理所当然的，那么请跳过接下来的讲解，直接进入下一节。

遗憾的是，一定还有读者不太明白，这是因为他们没有充分理解实例和存储实例的变量之间的关系，请不明白的读者务必阅读接下来的内容。

让我们从头开始依次讲解代码清单 5.3 的运行结果。首先从 (1) 处开始，这里在堆区创建 Person 实例，并将该实例的指针（表示位置的信息）存储到变量 musician 中。此时的内存状态如图 5-10 所示。

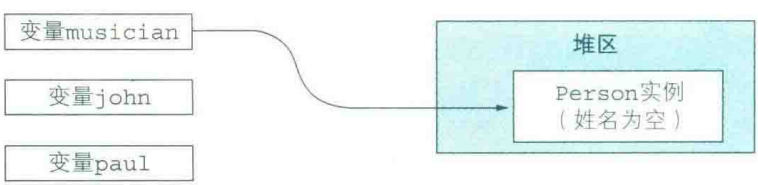

图 5-10 代码清单 5.3 中 (1) 处的堆区