到这里并没有问题。然后，在 (2) 处将变量 musician 的内容赋给变量 john，在 (3) 处将姓名设置为“John”。此时的内存状态如图 5-11 所示。

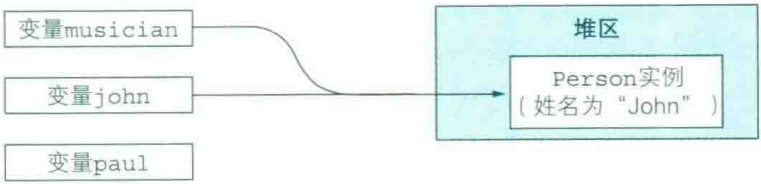

图 5-11　代码清单 5.3 中 (3) 处的堆区

这里需要注意的是 (2) 处的赋值处理。从图 5-11 可以看出，这里只是复制了 Person 实例的指针，在堆区中配置的 Person 实例依然只有一个。关于这一点，即使查看代码清单 5.3 的代码，可能也很难立马发现。不过，在 Java 中，仅当执行 new 命令时才会在堆区新创建实例，像 (2) 处这样对存储实例的变量进行赋值时，只是复制了指针而已。

在这种状态下，即使在 (4) 处再次将变量 musician 的内容赋给变量 paul，由于 Person 实例只有一个，所以变量 paul 与变量 john 指向的内容也相同。在 (5) 处将姓名设为“Paul”之后，状态如图 5-12 所示。

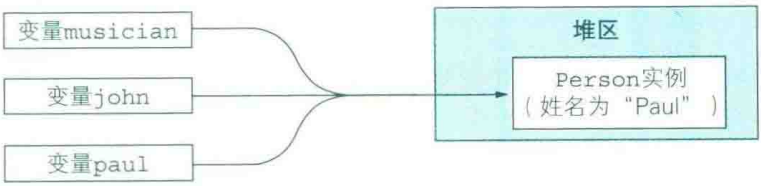

图 5-12　代码清单 5.3 中 (5) 处的堆区

在这种状态下，如果输出的是变量 john 和 paul 指向的实例的姓名，结果都会得到“Paul”。这是因为三个变量都指向同一个实例，所以产生这样的结果也是理所当然的。

需要注意的是，在代码清单 5.3 的 (2) 处和 (4) 处，无论复制多少个变量，复制的都只是存储实例的指针（表示位置的信息）。实例本身一直都只有一个，并不会改变。

在Java中，为了在堆区创建实例，需要使用new命令。这里我们来参考一下正确地设置了Person实例的姓名的代码，如代码清单5.6所示。

代码清单5.6 正确地设置了Person的姓名的代码

```
Person john = new Person(); // 创建 Person 实例
john.setName("John");       // 给变量 john 设置姓名
Person paul = new Person(); // 创建 Person 实例
paul.setName("Paul");       // 给变量 paul 设置姓名
```

如果像这样分别对两个变量使用new命令来创建实例，则内存的状态如图5-13所示。这样一来，我们就可以对两个变量分别指向的实例设置不同的姓名。

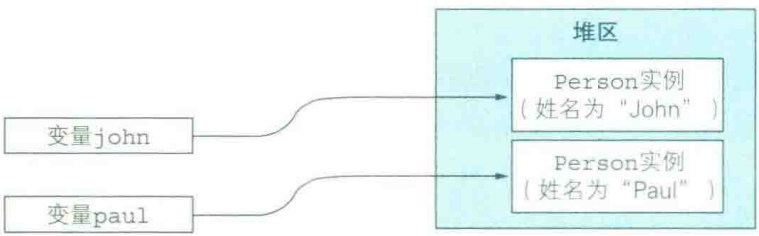

图5-13 执行代码清单5.6后的堆区

在Java中，为了简化语言规范，我们无法在程序中显式地操作表示实例位置的指针。虽然有人说Java中没有指针，但真实情况绝非如此。变量中存储的是指针，当在方法的参数和返回值中指定对象时，实际传递的也是指针。程序员需要充分理解这些内容。

最后，我们再来汇总一下需要注意的地方。

> 变量中存储的并不是实例本身，而是实例的指针（表示位置的信息）。
>
> 当将存储实例的变量赋给其他变量时，只是复制指针，堆区中的实例本身并不会发生变化。

5.11 多态让不同的类看起来一样

我们在第 4 章中介绍过，多态是创建公用主程序的结构。这种结构的关键在于，即使替换被调用端的类（相当于子程序），也不会影响调用端的类（相当于主程序）。

具体的实现方法有很多种，这里我们以最典型的**方法表**[1]方式为例进行介绍。

首先在各个类中准备一个方法表。该方法表中依次存储着各个类定义的方法在内存中展开的位置，即方法的指针[2]，具体情形如图 5-14 所示。

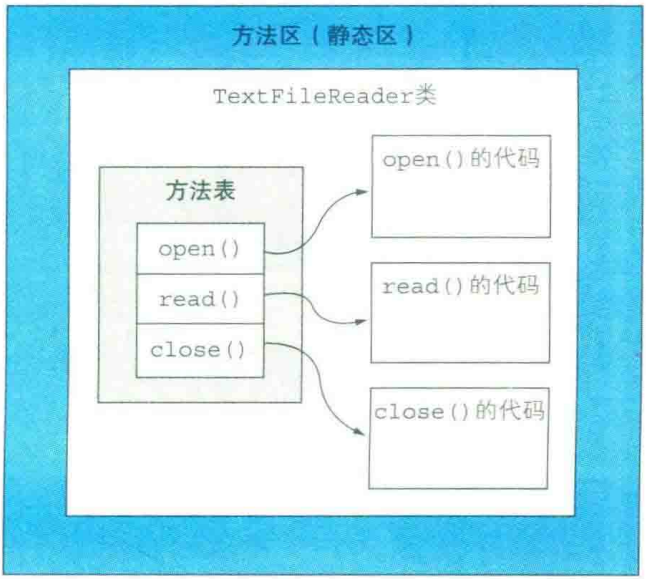

图 5-14 方法表

多态中需要将对象类的方法调用方式全部统一，即让被调用的类“看起来都一样”。

① 也称为虚函数表。

② 在 C 语言和 C++ 中称为函数指针。

方法表就是让不同的类看起来都一样的方法。我们创建一个方法表来汇集指向方法存储位置的指针，将对象类统一为该形式，这样就完成了准备工作。

当调用方法时，编译器会通过该方法表找到目标方法进行执行。这样一来，即使方法中编写的代码不同，也可以统一调用方法。这里我们对多态结构加以整理，如图 5-15 所示。前面我们说方法表是让不同的类看起来都一样的方法，其实更准确的说法应该是，方法表是让不同的类都戴上相同面具的方法。不同的类的方法各不相同，但通过戴上同一个面具——方法表，在调用端看来它们就都一样了。

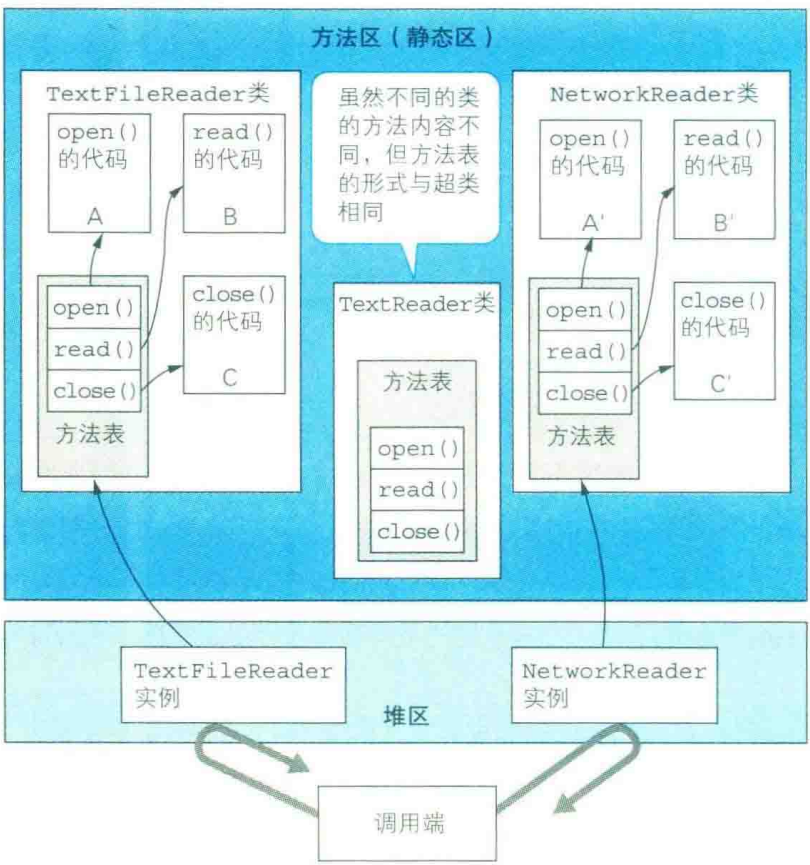

图 5-15　实现多态的内存结构

图 5-15 中有 TextFileReader 和 NetworkReader 两个实例，由于方法表的形式与超类 TextReader 相同，所以我们可以采用相同的方式对这两个实例进行方法调用。

这里介绍的多态结构都是由编译器和运行环境提供的，因此程序员无须关注该结构。从运行效率来看，采用这种结构的做法比单纯调用方法的做法效率低，但就现在的机器性能而言，差别其实并不大。

5.12 根据继承的信息类型的不同，内存配置也不同

接下来介绍继承。

我们在第 4 章中介绍过，继承是将类的共同部分汇总到其他类中的结构，这里的“共同部分”具体是指共同的方法和实例变量。使用继承结构，超类的定义信息就可以直接应用到子类中。

不过，即使继承的信息一样，从内存配置的观点来看，方法和实例变量也是完全不同的。下面我们来介绍一下继承的信息在内存中是如何配置的。

以继承了 Person 类的 Employee（员工）类为例。Employee 类非常简单，它持有员工编号 emplyeeNO 的实例变量，以及获取和设置员工编号的方法，具体代码如代码清单 5.7 所示。

代码清单5.7 Employee类

```
class Employee extends Person {   // Employee 类（继承 Person 类）
  private int employeeNO;   // 持有员工编号的实例变量
  public void setEmployeeNO(int empNO) { /* 省略逻辑处理 */ }
  public int getEmployeeNO() { /* 省略逻辑处理 */ }
}
```

由于该 Employee 类继承了 Person 类，所以可以直接使用 Person 类中定义的方法和实例变量。

最开始的内存配置的整体情况如图 5-16 所示。

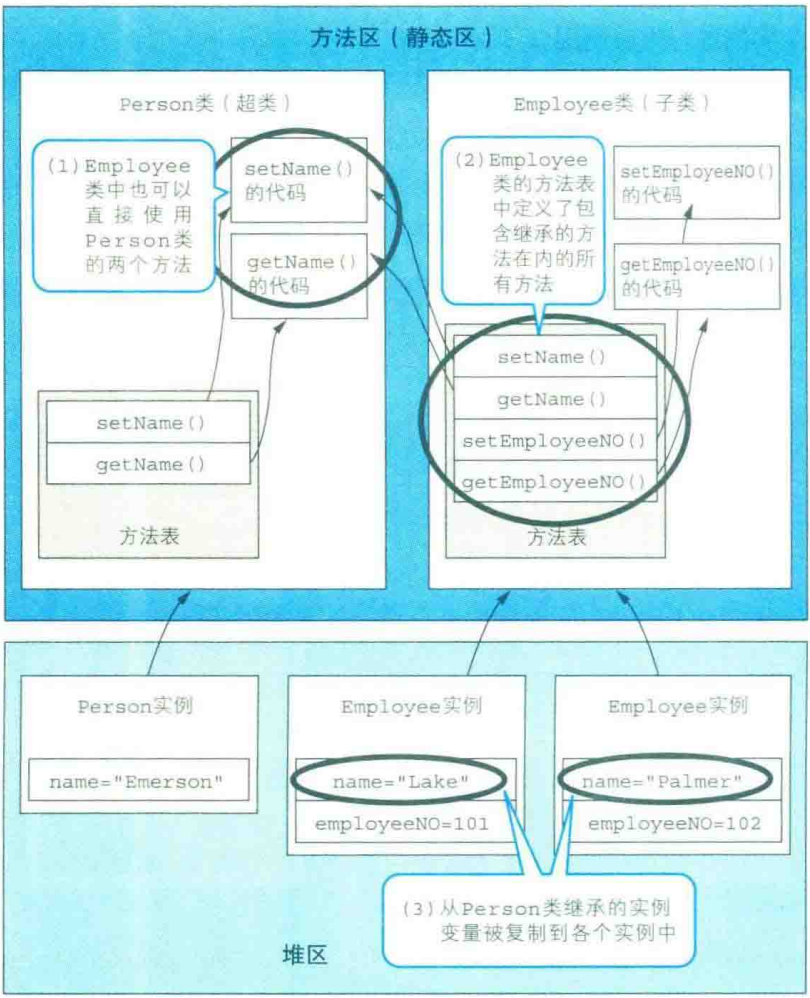

图 5-16 继承的信息的内存配置

我们先从方法开始介绍。

子类中可以直接使用超类中定义的方法。由于方法区中存储的代码信息也可以被直接使用，所以子类中会使用超类的信息，而不将被继承的方法的代码信息在内存中展开（图 5-16 中 (1) 处的说明）。另外，由于能够对

子类的实例调用超类中定义的方法，所以在子类的“脸”——方法表中定义了包含继承的方法在内的所有方法（图 5-16 中 (2) 处的说明）。

也就是说，继承的方法虽然存储在方法表中，但实际的代码信息使用的却是超类的内容。

接着我们再来介绍一下实例变量。通过继承，实例变量的定义也被复制到了子类中，但实际的值却会根据实例的不同而有所不同。因此，堆区中创建的子类的所有实例都会复制并持有超类中定义的实例变量（图 5-16 中 (3) 处的说明）。这样一来，所有的实例都会被分配变量区域，这与使用“隐藏”功能声明为 `private`（私有）的实例变量是一样的，请大家注意。

> 从超类继承的方法和实例变量的内存配置是完全不同的。
> 堆区中的子类的所有实例都会复制并持有超类中定义的实例变量。

5.13 孤立的实例由垃圾回收处理

最后我们来介绍一下 OOP 运行机制中的垃圾回收。正如第 4 章中介绍的那样，垃圾回收会自动删除堆区中残留的不再需要的实例。虽然这项功能对程序员来说非常便捷，但是实现起来却非常复杂。

因此，本书中不会深入介绍垃圾回收的详细结构。不过，理解什么状态的实例是垃圾回收的对象是使用 OOP 的程序员应该具备的基本素养。如果不了解这部分内容，编写出来的代码就可能会残留大量无法删除的实例，这样一来，无论垃圾回收的算法多么优秀也无法防止内存泄漏。因此，接下来将为大家介绍垃圾回收的基本结构，以及什么样的实例是删除对象。

我们先来看一下由谁执行垃圾回收。其实垃圾回收是由一个被称为**垃圾回收器**的专用程序执行的。该程序由编程语言的运行环境（在 Java 中为 Java VM）提供，并作为独立的线程运行。该程序会在适当的时间点确认堆区的状态，当发现空内存区域变少时，就会启动垃圾回收处理。

看到这里大家可能会产生疑问：垃圾回收器是怎么判断实例不被需要了呢？

答案就是“发现孤立的实例”。

使用 OOP 编写的应用程序通过从类创建实例并对该实例启动方法来运行，并且实例还可以引用其他实例（具体结构是在变量中存储实例的指针，相关内容我们已经在前面介绍过了）。在使用 OOP 编写的应用程序中会创建很多实例，它们互相引用，整体构成一个网络。

这种实例网络并不只在堆区发挥作用，在栈区和方法区（静态区）中也会起到很重要的作用。

我们在前面介绍过，作为一般的程序结构，正在运行的方法的参数和局部变量存储在栈区中。OOP 也是如此。不过在 OOP 中，参数和局部变量可以指定实例。在这种情况下，栈区中存储的是堆区中的实例的指针。方法区也可以引用堆区的实例[1]，这里就不再详细介绍了。

栈区和方法区中存储着应用程序处理所需的信息，因此这里引用的实例不会成为垃圾回收的对象。也就是说，栈区和方法区是网络的“根部”。

脱离网络关系的实例，即从根部无法到达的实例，就是垃圾回收的对象。

讲到这里，我们再来看一下本章开头的热身问答。

< 热身问答 >

请从图 5-17 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）。

① 类变量（static 变量）的内存区域在方法区（静态区）中分配。

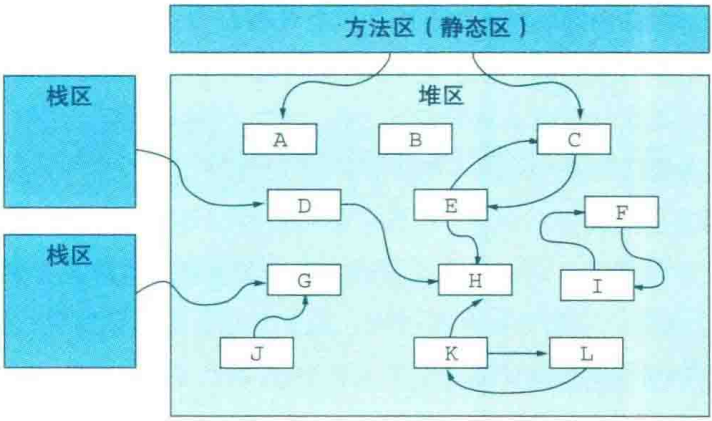

图 5–17　垃圾回收测试

< 问题的答案和讲解 >

正确答案是 B、F、I、J、K 和 L 六个实例（图 5-18）。

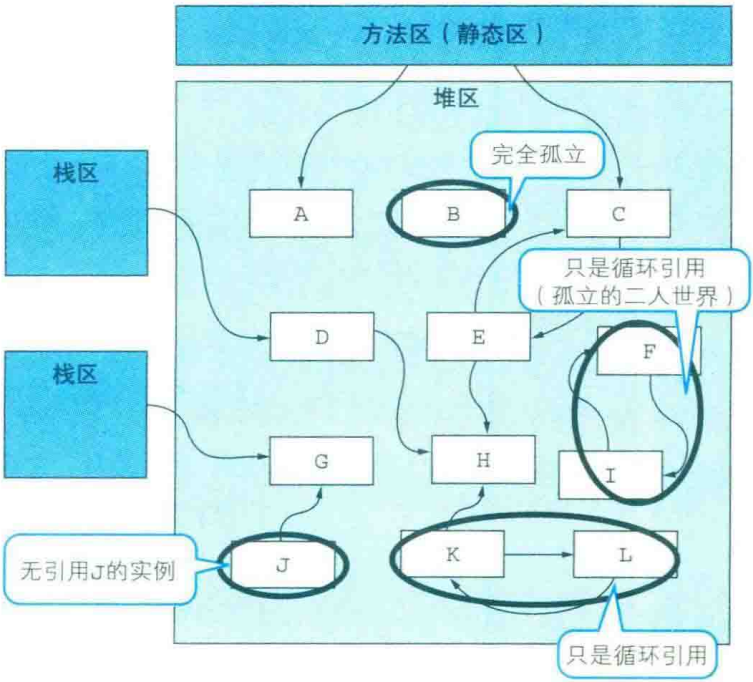

图 5–18　垃圾回收问题的答案

首先，最容易理解的是B。因为没有任何位置引用该实例，该实例本身也没有引用其他位置，处于完全孤立的状态，所以将其删除也不会引发任何问题。其次比较容易理解的是F和I。F和I是互相引用的关系，但都无法从方法区和栈区到达。在这种状态下，应用程序无法对其进行访问，因此这两者也是删除对象。J可能稍微有点难以理解。J引用G，乍一看它好像位于网络中。不过，由于没有实例引用J，所以J也是删除对象。K、L与J一样。它们之间互相引用，虽然K还引用其他实例，但从其他位置是无法访问到K和L的。

各位感觉如何？如果能够理解以上内容，那么就说明你具备垃圾回收相关的基本素养。如果做错了，请一定再试着挑战一下。

在编程时应该注意的是，栈区和方法区不要一直引用不再需要的实例。不过，在实际编程时很难注意到这一点，而且在没有连锁引用大量实例时也无须在意。大家只用记住，在万一程序因为内存不足而异常结束的情况下，这是修改应用程序的一个要点。

最后让我们来打个比方：垃圾回收是一种“垃圾回收器大魔王找到与谁都没有牵手的实例，并将其当场杀掉”的结构。

到目前为止，本书一直在主张OOP结构不是直接表示现实世界的技术，如果硬要用现实世界的事物进行比喻，结果会非常糟糕。哪怕松开拉着孩子的手一瞬间，最后也不是再也见不到那么简单了，大魔王会当场下手。真是一个可怕的世界！

深入学习的参考书籍

第 4 章和第 5 章相关的参考书籍如下所示。

[1] 小森裕介. なぜ、あなたは Java でオブジェクト指向開発ができないのか——Java の壁を克服する実践トレーニング [M]. 东京：技術評論社，2004.

☆☆☆

该书通过猜拳、抽乌龟和憋七等常见的简单游戏程序，详细地介绍了 Java 中的 OOP 结构、使用 UML 的类设计，以及可重用框架的创建方法等。特别推荐给程序员新手及熟悉 COBOL 和 C 语言的程序员阅读。

[2] 结城浩. 改訂第 2 版 Java 言語プログラミングレッスン（下）——オブジェクト指向を始めよう [M]. 东京：SoftBank Creative，2005.

☆☆☆

承接上卷对控制语句、变量定义等 Java 基本语法的讲解，该下卷讲解了类和实例、继承和多态、包、异常、垃圾回收，以及 Java 类库中包含的基本类的用法和线程功能等。该书介绍得非常细致，还设置了热身问答、练习题等，以帮助读者加深理解。推荐给想要正式学习 Java 的面向对象功能的读者阅读。

[3] 高桥征义，后藤裕藏. Ruby 基础教程（第 5 版）[M]. 何文斯，译. 北京：人民邮电出版社，2017.

☆☆

该书从体验 Ruby 编程开始，对 Ruby 的语法、基本类的用法和 Ruby 特有的功能等进行了介绍，最后介绍了应用示例，逐步推进，结构非常清晰。该书非常适合 Ruby 编程的初学者阅读。

[4] 梅泽真史. 自由自在 Squeak プログラミング [M]. 东京：SRC，2004.

☆☆

建议想要深入理解面向对象的概念的读者了解一下 Smalltalk。“一切都是对象”（everything is an object）的开发环境将带给你超越本书“OOP 是结构化语言的进化形式”这一主张的灵感。这是一部接近 600 页的大作，但由于作者对 Smalltalk 有深入理解，所以文章内容细致、明快，读起来非常顺畅。

[5] Tucker !. 憂鬱なプログラマのためのオブジェクト指向開発講座——C++ による実践的ソフトウェア構築入門 [M]. 东京：翔泳社，1998.

☆

这是一部超过 400 页的大作，详细介绍了面向对象编程、分析和设计的相关内容。该书出版于 20 世纪 90 年代，由于当时 Java 和 UML 尚未普及，所以书中的编程语言采用 C++，描述方法采用 OMT（UML 的前身），这在今天看来虽然有些过时，但该书在出版后被传阅至今，可以说是一本经典图书。

[6] 中村成洋，相川光. 垃圾回收的算法与实现 [M]. 丁灵，译. 北京：人民邮电出版社，2016.

☆☆☆

这是一本系统地汇总了垃圾回收相关内容的雄心之作。该书结合大量形象的插图和示例代码，对在应用程序系统中运行的垃圾回收的算法与实现进行了细致的讲解。强烈推荐给有志于创建出一流架构的技术人员阅读。

[7] 米持幸寿. Java でなぜつくるのか——知っておきたい Java プログラミングの基礎知識 [M]. 日经 BP 社，2005.

☆☆

该书全面介绍了 Java 普及的原因、Java 的结构以及 Java 虚拟机和内存管理机制的相关内容。

专栏

编程往事

OOP 中 dump 看起来很费劲？

大家猜笔者在刚知道 OOP 时的感想是什么？

读到这里的读者可能会猜测：

“厉害！这样应该就可以创建独立性很强的软件构件了！”

“使用多态可以创建公用主程序，好方便！”

但其实并不是这样，而是“dump 看起来好像很费劲啊”。

听到“dump”，可能有的读者会想到翻斗车（dump car），这样想的读者应该都是新人培训时学习 Java 的人吧。

“dump”作为计算机术语，当然是指“memory dump”（内存转储）。“dump”是“倾倒”的意思，这里是指将程序异常结束时的内存内容汇总到一起输出的结果。

memory dump 的形式如图 5-a 所示。一般情况下，左侧输出内存地址，中间以十六进制数的形式输出内存内容，右侧输出将内存内容转换为字符编码后的值。

笔者在使用汇编语言和 COBOL 编写程序时，调试时要从多达几十页的 memory dump 中查找要确认的变量，一边添加行标记，一边确认状态，然后确认程序逻辑，找到错误的原因，这是固定步骤。由于从 memory dump 中查找变量位置非常费劲，所以一般会将易于查看的字符串写入程序的变量区域中（比如图 5-a 中第一

```
3E290    2A2A 4441 5441 2A2A    **DATA**
3E298    004E 3C00 3141 205C    .N<.1A \
3E2A0    3030 3132 2000 4B6B    0012 .Kk
```

地址（十六进制数）　内存内容（十六进制数）　内存内容（字符形式）

图 5–a　memory dump 的形式

行输出的“**DATA**”字符串）。

***

现在，使用高性能的调试器，可以在任意位置暂停程序运行，将想要确认的变量显示到窗口中，这种开发环境已经变得很普遍，但在当时来看，这简直就像做梦一样。

因此，在笔者初次听到“在 OOP 中，当创建实例时，存储变量的内存区域会在堆区的某个位置被动态地分配”时，笔者就想：“即便编程变得很轻松，但如果 dump 看起来费劲，调试起来很麻烦，就没有什么用。”

但在如今这个时代，幸运（同时也很遗憾）的是，知道 memory dump 的人已经不多了。

本章的关键词

类库、框架、组件、设计模式

第6章 重用：OOP带来的软件重用和思想重用

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

框架结构被称为“好莱坞原则”（Hollywood Principle），下列哪一项是对此的正确解释？

A. 著名的MVC（Model-View-Controller，模型－视图－控制器）框架实际上是由好莱坞的女演员提出的

B. 正如电影制作相关的思想、技术和人才都汇集于好莱坞一样，框架将软件的控制功能都汇总在一处

C. 好莱坞电影的故事展开基本上都采用相同的轮廓（框架）

D. 应用程序不调用框架，就像好莱坞禁止演员给制作方打电话推销自己一样