# JAVA编程思想

# 第2章 一切都是对象

“尽管以C++为基础，但Java是一种更纯粹的面向对象程序设计语言”。

无论C++还是Java都属于杂合语言。但在Java中，设计者觉得这种杂合并不象在C++里那么重要。杂合语言允许采用多种编程风格；之所以说C++是一种杂合语言，是因为它支持与C语言的向后兼容能力。由于C++是C的一个超集，所以包含的许多特性都是后者不具备的，这些特性使C++在某些地方显得过于复杂。

Java语言首先便假定了我们只希望进行面向对象的程序设计。也就是说，正式用它设计之前，必须先将自己的思想转入一个面向对象的世界（除非早已习惯了这个世界的思维方式）。只有做好这个准备工作，与其他OOP语言相比，才能体会到Java的易学易用。在本章，我们将探讨Java程序的基本组件，并体会为什么说Java乃至Java程序内的一切都是对象。

## 2.1 用句柄操纵对象

每种编程语言都有自己的数据处理方式。有些时候，程序员必须时刻留意准备处理的是什么类型。您曾利用一些特殊语法直接操作过对象，或处理过一些间接表示的对象吗（C或C++里的指针）？

所有这些在Java里都得到了简化，任何东西都可看作对象。因此，我们可采用一种统一的语法，任何地方均可照搬不误。但要注意，尽管将一切都“看作”对象，但操纵的标识符实际是指向一个对象的“句柄”（Handle）。在其他Java参考书里，还可看到有的人将其称作一个“引用”，甚至一个“指针”。可将这一情形想象成用遥控板（句柄）操纵电视机（对象）。只要握住这个遥控板，就相当于掌握了与电视机连接的通道。但一旦需要“换频道”或者“关小声音”，我们实际操纵的是遥控板（句柄），再由遥控板自己操纵电视机（对象）。如果要在房间里四处走走，并想保持对电视机的控制，那么手上拿着的是遥控板，而非电视机。

此外，即使没有电视机，遥控板亦可独立存在。也就是说，只是由于拥有一个句柄，并不表示必须有一个对象同它连接。所以如果想容纳一个词或句子，可创建一个String句柄：

String s;

但这里创建的只是句柄，并不是对象。若此时向s发送一条消息，就会获得一个错误（运行期）。这是由于s实际并未与任何东西连接（即“没有电视机”）。因此，一种更安全的做法是：创建一个句柄时，记住无论如何都进行初始化：

String s = "asdf";

然而，这里采用的是一种特殊类型：字串可用加引号的文字初始化。通常，必须为对象使用一种更通用的初始化类型。

## 2.2 所有对象都必须创建

创建句柄时，我们希望它同一个新对象连接。通常用new关键字达到这一目的。new的意思是：“把我变成这些对象的一种新类型”。所以在上面的例子中，可以说：

String s = new String("asdf");

它不仅指出“将我变成一个新字串”，也通过提供一个初始字串，指出了“如何生成这个新字串”。

当然，字串（String）并非唯一的类型。Java配套提供了数量众多的现成类型。对我们来讲，最重要的就是记住能自行创建类型。事实上，这应是Java程序设计的一项基本操作，是继续本书后余部分学习的基础。

### 2.2.1 保存到什么地方

程序运行时，我们最好对数据保存到什么地方做到心中有数。特别要注意的是内存的分配。有六个地方都可以保存数据：

(1) 寄存器。这是最快的保存区域，因为它位于和其他所有保存方式不同的地方：处理器内部。然而，寄存器的数量十分有限，所以寄存器是根据需要由编译器分配。我们对此没有直接的控制权，也不可能在自己的程序里找到寄存器存在的任何踪迹。

(2) 堆栈。驻留于常规RAM（随机访问存储器）区域，但可通过它的“堆栈指针”获得处理的直接支持。堆栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些Java数据要保存在堆栈里——特别是对象句柄，但Java对象并不放到其中。

(3) 堆。一种常规用途的内存池（也在RAM区域），其中保存了Java对象。和堆栈不同，“内存堆”或“堆”（Heap）最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用new命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价：在堆里分配存储空间时会花掉更长的时间！

(4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管也在RAM里）。程序运行期间，静态存储的数据将随时等候调用。可用static关键字指出一个对象的特定元素是静态的。但Java对象本身永远都不会置入静态存储空间。

(5) 常数存储。常数值通常直接置于程序代码内部。这样做是安全的，因为它们永远都不会改变。有的常数需要严格地保护，所以可考虑将它们置入只读存储器（ROM）。

(6) 非RAM存储。若数据完全独立于一个程序之外，则程序不运行时仍可存在，并在程序的控制范围之外。其中两个最主要的例子便是“流式对象”和“固定对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，对象保存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普通的、基于RAM的对象。Java 1.1提供了对Lightweight persistence的支持。未来的版本甚至可能提供更完整的方案。

### 2.2.2 特殊情况：主要类型

有一系列类需特别对待；可将它们想象成“基本”、“主要”或者“主”（Primitive）类型，进行程序设计时要频繁用到它们。之所以要特别对待，是由于用new创建对象（特别是小的、简单的变量）并不是非常有效，因为new将对象置于“堆”里。对于这些类型，Java采纳了与C和C++相同的方法。也就是说，不是用new创建变量，而是创建一个并非句柄的“自动”变量。这个变量容纳了具体的值，并置于堆栈中，能够更高效地存取。

Java决定了每种主要类型的大小。就象在大多数语言里那样，这些大小并不随着机器结构的变化而变化。这种大小的不可更改正是Java程序具有很强移植能力的原因之一。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **主类型** | **大小** | **最小值** | **最大值** | **封装器类型** |
| boolean | 1-bit | – | – | **Boolean** | |
| char | 16-bit | Unicode 0 | Unicode 216- 1 | **Character** | |
| byte | 8-bit | -128 | +127 | **Byte**[[11]](mk:@MSITStore:C:\\Users\\茜茜\\Desktop\\JAVA编程思想（完整版）.CHM::/chapter/chapter2.htm" \l "fn11) | |
| short | 16-bit | -215 | +215 – 1 | **Short1** | |
| int | 32-bit | -231 | +231 – 1 | **Integer** | |
| long | 64-bit | -263 | +263 – 1 | **Long** | |
| float | 32-bit | IEEE754 | IEEE754 | **Float** | |
| double | 64-bit | IEEE754 | IEEE754 | **Double** | |
| void | – | – | – | **Void**1 | |

①：到Java 1.1才有，1.0版没有。

数值类型全都是有符号（正负号）的，所以不必费劲寻找没有符号的类型。

主数据类型也拥有自己的“封装器”（wrapper）类。这意味着假如想让堆内一个非主要对象表示那个主类型，就要使用对应的封装器。例如：

char c = 'x';

Character C = new Character('c');

也可以直接使用：

Character C = new Character('x');

这样做的原因将在以后的章节里解释。

1. 高精度数字

Java 1.1增加了两个类，用于进行高精度的计算：BigInteger和BigDecimal。尽管它们大致可以划分为“封装器”类型，但两者都没有对应的“主类型”。

这两个类都有自己特殊的“方法”，对应于我们针对主类型执行的操作。也就是说，能对int或float做的事情，对BigInteger和BigDecimal一样可以做。只是必须使用方法调用，不能使用运算符。此外，由于牵涉更多，所以运算速度会慢一些。我们牺牲了速度，但换来了精度。

BigInteger支持任意精度的整数。也就是说，我们可精确表示任意大小的整数值，同时在运算过程中不会丢失任何信息。

BigDecimal支持任意精度的定点数字。例如，可用它进行精确的币值计算。

至于调用这两个类时可选用的构建器和方法，请自行参考联机帮助文档。

### 2.2.3 Java的数组

几乎所有程序设计语言都支持数组。在C和C++里使用数组是非常危险的，因为那些数组只是内存块。若程序访问自己内存块以外的数组，或者在初始化之前使用内存（属于常规编程错误），会产生不可预测的后果（注释②）。

②：在C++里，应尽量不要使用数组，换用标准模板库（Standard TemplateLibrary）里更安全的容器。

Java的一项主要设计目标就是安全性。所以在C和C++里困扰程序员的许多问题都未在Java里重复。一个Java可以保证被初始化，而且不可在它的范围之外访问。由于系统自动进行范围检查，所以必然要付出一些代价：针对每个数组，以及在运行期间对索引的校验，都会造成少量的内存开销。但由此换回的是更高的安全性，以及更高的工作效率。为此付出少许代价是值得的。

创建对象数组时，实际创建的是一个句柄数组。而且每个句柄都会自动初始化成一个特殊值，并带有自己的关键字：null（空）。一旦Java看到null，就知道该句柄并未指向一个对象。正式使用前，必须为每个句柄都分配一个对象。若试图使用依然为null的一个句柄，就会在运行期报告问题。因此，典型的数组错误在Java里就得到了避免。

也可以创建主类型数组。同样地，编译器能够担保对它的初始化，因为会将那个数组的内存划分成零。

数组问题将在以后的章节里详细讨论。

## 2.3 绝对不要清除对象

在大多数程序设计语言中，变量的“存在时间”（Lifetime）一直是程序员需要着重考虑的问题。变量应持续多长的时间？如果想清除它，那么何时进行？在变量存在时间上纠缠不清会造成大量的程序错误。在下面的小节里，将阐示Java如何帮助我们完成所有清除工作，从而极大了简化了这个问题。

### 2.3.1 作用域

大多数程序设计语言都提供了“作用域”（Scope）的概念。对于在作用域里定义的名字，作用域同时决定了它的“可见性”以及“存在时间”。在C，C++和Java里，作用域是由花括号的位置决定的。参考下面这个例子：

{

int x = 12;

/\* only x available \*/

{

int q = 96;

/\* both x & q available \*/

}

/\* only x available \*/

/\* q “out of scope” \*/

}

作为在作用域里定义的一个变量，它只有在那个作用域结束之前才可使用。

在上面的例子中，缩进排版使Java代码更易辨读。由于Java是一种形式自由的语言，所以额外的空格、制表位以及回车都不会对结果程序造成影响。

注意尽管在C和C++里是合法的，但在Java里不能象下面这样书写代码：

{

int x = 12;

{

int x = 96; /\* illegal \*/

}

}

编译器会认为变量x已被定义。所以C和C++能将一个变量“隐藏”在一个更大的作用域里。但这种做法在Java里是不允许的，因为Java的设计者认为这样做使程序产生了混淆。

### 2.3.2 对象的作用域

Java对象不具备与主类型一样的存在时间。用new关键字创建一个Java对象的时候，它会超出作用域的范围之外。所以假若使用下面这段代码：

{

String s = new String("a string");

} /\* 作用域的终点 \*/

那么句柄s会在作用域的终点处消失。然而，s指向的String对象依然占据着内存空间。在上面这段代码里，我们没有办法访问对象，因为指向它的唯一一个句柄已超出了作用域的边界。在后面的章节里，大家还会继续学习如何在程序运行期间传递和复制对象句柄。

这样造成的结果便是：对于用new创建的对象，只要我们愿意，它们就会一直保留下去。这个编程问题在C和C++里特别突出。看来在C++里遇到的麻烦最大：由于不能从语言获得任何帮助，所以在需要对象的时候，根本无法确定它们是否可用。而且更麻烦的是，在C++里，一旦工作完成，必须保证将对象清除。

这样便带来了一个有趣的问题。假如Java让对象依然故我，怎样才能防止它们大量充斥内存，并最终造成程序的“凝固”呢。在C++里，这个问题最令程序员头痛。但Java以后，情况却发生了改观。Java有一个特别的“垃圾收集器”，它会查找用new创建的所有对象，并辨别其中哪些不再被引用。随后，它会自动释放由那些闲置对象占据的内存，以便能由新对象使用。这意味着我们根本不必操心内存的回收问题。只需简单地创建对象，一旦不再需要它们，它们就会自动离去。这样做可防止在C++里很常见的一个编程问题：由于程序员忘记释放内存造成的“内存溢出”。

## 2.4 新建数据类型：类

如果说一切东西都是对象，那么用什么决定一个“类”（Class）的外观与行为呢？换句话说，是什么建立起了一个对象的“类型”（Type）呢？大家可能猜想有一个名为“type”的关键字。但从历史看来，大多数面向对象的语言都用关键字“class”表达这样一个意思：“我准备告诉你对象一种新类型的外观”。class关键字太常用了，以至于本书许多地方并没有用粗体字或双引号加以强调。在这个关键字的后面，应该跟随新数据类型的名称。例如：

class ATypeName {/\*类主体置于这里}

这样就引入了一种新类型，接下来便可用new创建这种类型的一个新对象：

ATypeName a = new ATypeName();

在ATypeName里，类主体只由一条注释构成（星号和斜杠以及其中的内容，本章后面还会详细讲述），所以并不能对它做太多的事情。事实上，除非为其定义了某些方法，否则根本不能指示它做任何事情。

### 2.4.1 字段和方法

定义一个类时（我们在Java里的全部工作就是定义类、制作那些类的对象以及将消息发给那些对象），可在自己的类里设置两种类型的元素：数据成员（有时也叫“字段”）以及成员函数（通常叫“方法”）。其中，数据成员是一种对象（通过它的句柄与其通信），可以为任何类型。它也可以是主类型（并不是句柄）之一。如果是指向对象的一个句柄，则必须初始化那个句柄，用一种名为“构建器”（第4章会对此详述）的特殊函数将其与一个实际对象连接起来（就象早先看到的那样，使用new关键字）。但若是一种主类型，则可在类定义位置直接初始化（正如后面会看到的那样，句柄亦可在定义位置初始化）。

每个对象都为自己的数据成员保有存储空间；数据成员不会在对象之间共享。下面是定义了一些数据成员的类示例：

class DataOnly {

int i;

float f;

boolean b;

}

这个类并没有做任何实质性的事情，但我们可创建一个对象：

DataOnly d = new DataOnly();

可将值赋给数据成员，但首先必须知道如何引用一个对象的成员。为达到引用对象成员的目的，首先要写上对象句柄的名字，再跟随一个点号（句点），再跟随对象内部成员的名字。即“对象句柄.成员”。例如：

d.i = 47;

d.f = 1.1f;

d.b = false;

一个对象也可能包含了另一个对象，而另一个对象里则包含了我们想修改的数据。对于这个问题，只需保持“连接句点”即可。例如：

myPlane.leftTank.capacity = 100;

除容纳数据之外，DataOnly类再也不能做更多的事情，因为它没有成员函数（方法）。为正确理解工作原理，首先必须知道“自变量”和“返回值”的概念。我们马上就会详加解释。

1. 主成员的默认值

若某个主数据类型属于一个类成员，那么即使不明确（显式）进行初始化，也可以保证它们获得一个默认值。

主类型 默认值

Boolean false

Char '\u0000'(null)

byte (byte)0

short (short)0

int 0

long 0L

float 0.0f

double 0.0d

一旦将变量作为类成员使用，就要特别注意由Java分配的默认值。这样做可保证主类型的成员变量肯定得到了初始化（C++不具备这一功能），可有效遏止多种相关的编程错误。

然而，这种保证却并不适用于“局部”变量——那些变量并非一个类的字段。所以，假若在一个函数定义中写入下述代码：

int x;

那么x会得到一些随机值（这与C和C++是一样的），不会自动初始化成零。我们责任是在正式使用x前分配一个适当的值。如果忘记，就会得到一条编译期错误，告诉我们变量可能尚未初始化。这种处理正是Java优于C++的表现之一。许多C++编译器会对变量未初始化发出警告，但在Java里却是错误。

## 2.5 方法、自变量和返回值

迄今为止，我们一直用“函数”（Function）这个词指代一个已命名的子例程。但在Java里，更常用的一个词却是“方法”（Method），代表“完成某事的途径”。尽管它们表达的实际是同一个意思，但从现在开始，本书将一直使用“方法”，而不是“函数”。

Java的“方法”决定了一个对象能够接收的消息。通过本节的学习，大家会知道方法的定义有多么简单！

方法的基本组成部分包括名字、自变量、返回类型以及主体。下面便是它最基本的形式：

返回类型 方法名( /\* 自变量列表\*/ ) {/\* 方法主体 \*/}

返回类型是指调用方法之后返回的数值类型。显然，方法名的作用是对具体的方法进行标识和引用。自变量列表列出了想传递给方法的信息类型和名称。

Java的方法只能作为类的一部分创建。只能针对某个对象调用一个方法（注释③），而且那个对象必须能够执行那个方法调用。若试图为一个对象调用错误的方法，就会在编译期得到一条出错消息。为一个对象调用方法时，需要先列出对象的名字，在后面跟上一个句点，再跟上方法名以及它的参数列表。亦即“对象名.方法名(自变量1，自变量2，自变量3...)。举个例子来说，假设我们有一个方法名叫f()，它没有自变量，返回的是类型为int的一个值。那么，假设有一个名为a的对象，可为其调用方法f()，则代码如下：

int x = a.f();

返回值的类型必须兼容x的类型。

象这样调用一个方法的行动通常叫作“向对象发送一条消息”。在上面的例子中，消息是f()，而对象是a。面向对象的程序设计通常简单地归纳为“向对象发送消息”。

③：正如马上就要学到的那样，“静态”方法可针对类调用，毋需一个对象。