第 11 章 运行期类型鉴定

运行期类型鉴定（RTTI）的概念初看非常简单——手上只有基础类型的一个句柄时，利用它判断一个对象的正确类型。

然而，对 RTTI 的需要暴露出了面向对象设计许多有趣（而且经常是令人困惑的）的问题，并把程序的构造问题正式摆上了桌面。

本章将讨论如何利用 Java 在运行期间查找对象和类信息。这主要采取两种形式：一种是“传统”RTTI，它假定我们已在编译和运行期拥有所有类型；另一种是 Java1.1 特有的“反射”机制，利用它可在运行期独立查找类信息。首先讨论“传统”的 RTTI，再讨论反射问题。

11.1 对 RTTI 的需要

请考虑下面这个熟悉的类结构例子，它利用了多形性。常规类型是 Shape 类，而特别衍生出来的类型是 Circle，Square 和 Triangle。

516 页图

这是一个典型的类结构示意图，基础类位于顶部，衍生类向下延展。面向对象编程的基本目标是用大量代码控制基础类型（这里是 Shape）的句柄，所以假如决定添加一个新类（比如 Rhomboid，从 Shape 衍生），从而对程序进行扩展，那么不会影响到原来的代码。在这个例子中，Shape 接口中的动态绑定方法是 draw()，所以客户程序员要做的是通过一个普通 Shape 句柄调用 draw()。draw() 在所有衍生类里都会被覆盖。而且由于它是一个动态绑定方法，所以即使通过一个普通的 Shape 句柄调用它，也有表现出正确的行为。这正是多形性的作用。

所以，我们一般创建一个特定的对象（Circle，Square，或者 Triangle），把它上溯造型到一个 Shape（忽略对象的特殊类型），以后便在程序的剩余部分使用匿名 Shape 句柄。

作为对多形性和上溯造型的一个简要回顾，可以象下面这样为上述例子编码（若执行这个程序时出现困难，请参考第 3 章 3.1.2 小节“赋值”）：

516-517 页程序

基础类可编码成一个 interface（接口）、一个 abstract（抽象）类或者一个普通类。由于 Shape 没有真正的成员（亦即有定义的成员），而且并不在意我们创建了一个纯粹的 Shape 对象，所以最适合和最灵活的表达方式便是用一个接口。而且由于不必设置所有那些 abstract 关键字，所以整个代码也显得更为清爽。

每个衍生类都覆盖了基础类 draw 方法，所以具有不同的行为。在 main()中创建了特定类型的 Shape，然后将其添加到一个 Vector。这里正是上溯造型发生的地方，因为 Vector 只容纳了对象。由于 Java 中的所有东西（除基本数据类型外）都是对象，所以 Vector 也能容纳 Shape 对象。但在上溯造型至 Object 的过程中，任何特殊的信息都会丢失，其中甚至包括对象是几何形状这一事实。对 Vector 来说，它们只是 Object。

用 nextElement()将一个元素从 Vector 提取出来的时候，情况变得稍微有些复杂。由于 Vector 只容纳 Object，所以 nextElement()会自然地产生一个 Object 句柄。但我们知道它实际是个 Shape 句柄，而且希望将 Shape 消息发给那个对象。所以需要用传统的"(Shape)"方式造型成一个 Shape。这是 RTTI 最基本的形式，因为在 Java 中，所有造型都会在运行期间得到检查，以确保其正确性。那正是 RTTI 的意义所在：在运行期，对象的类型会得到鉴定。

在目前这种情况下，RTTI 造型只实现了一部分：Object 造型成 Shape，而不是造型成 Circle，Square 或者 Triangle。那是由于我们目前能够肯定的唯一事实就是 Vector 里充斥着几何形状，而不知它们的具体类别。在编译期间，我们肯定的依据是我们自己的规则；而在编译期间，却是通过造型来肯定这一点。

现在的局面会由多形性控制，而且会为 Shape 调用适当的方法，以便判断句柄到底是提供 Circle，Square，还是提供给 Triangle。而且在一般情况下，必须保证采用多形性方案。因为我们希望自己的代码尽可能少知道一些与对象的具体类型有关的情况，只将注意力放在某一类对象（这里是 Shape）的常规信息上。只有这样，我们的代码才更易实现、理解以及修改。所以说多形性是面向对象程序设计的一个常规目标。

然而，若碰到一个特殊的程序设计问题，只有在知道常规句柄的确切类型后，才能最容易地解决这个问题，这个时候又该怎么办呢？举个例子来说，我们有时候想让自己的用户将某一具体类型的几何形状（如三角形）全都变成紫色，以便突出显示它们，并快速找出这一类型的所有形状。此时便要用到 RTTI 技术，用它查询某个 Shape 句柄引用的准确类型是什么。

11.1.1 Class 对象

为理解 RTTI 在 Java 里如何工作，首先必须了解类型信息在运行期是如何表示的。这时要用到一个名为“Class 对象”的特殊形式的对象，其中包含了与类有关的信息（有时也把它叫作“元类”）。事实上，我们要用 Class 对象创建属于

某个类的全部“常规”或“普通”对象。

对于作为程序一部分的每个类，它们都有一个 Class 对象。换言之，每次写一个新类时，同时也会创建一个 Class 对象（更恰当地说，是保存在一个完全同名的.class 文件中）。在运行期，一旦我们想生成那个类的一个对象，用于执行程序的 Java 虚拟机（JVM）首先就会检查那个类型的 Class 对象是否已经载入。若尚未载入，JVM 就会查找同名的.class 文件，并将其载入。所以 Java 程序启动时并不是完全载入的，这一点与许多传统语言都不同。

一旦那个类型的 Class 对象进入内存，就用它创建那一类型的所有对象。

若这种说法多少让你产生了一点儿迷惑，或者并没有真正理解它，下面这个示范程序或许能提供进一步的帮助：

519-520 页程序

对每个类来说（Candy，Gum 和 Cookie），它们都有一个 static 从句，用于在类首次载入时执行。相应的信息会打印出来，告诉我们载入是什么时候进行的。在 main()中，对象的创建代码位于打印语句之间，以便侦测载入时间。

特别有趣的一行是：

Class.forName("Gum");

该方法是 Class（即全部 Class 所从属的）的一个 static 成员。而 Class 对象和其他任何对象都是类似的，所以能够获取和控制它的一个句柄（装载模块就是干这件事的）。为获得 Class 的一个句柄，一个办法是使用 forName()。它的作用是取得包含了目标类文本名字的一个 String（注意拼写和大小写）。最后返回的是一个 Class 句柄。

该程序在某个 JVM 中的输出如下：

520 页中程序

可以看到，每个 Class 只有在它需要的时候才会载入，而 static 初始化工作是在类载入时执行的。

非常有趣的是，另一个 JVM 的输出变成了另一个样子：

520 页下程序

看来 JVM 通过检查 main()中的代码，已经预测到了对 Candy 和 Cookie 的需要，但却看不到 Gum，因为它是通过对 forName()的一个调用创建的，而不是通过更典型的 new 调用。尽管这个 JVM 也达到了我们希望的效果，因为确实会在我们需要之前载入那些类，但却不能肯定这儿展示的行为百分之百正确。

1. 类标记

在 Java 1.1 中，可以采用第二种方式来产生 Class 对象的句柄：使用“类标记”。对上述程序来说，看起来就象下面这样：

Gum.class;

这样做不仅更加简单，而且更安全，因为它会在编译期间得到检查。由于它取消了对方法调用的需要，所以执行的效率也会更高。

类标记不仅可以应用于普通类，也可以应用于接口、数组以及基本数据类型。除此以外，针对每种基本数据类型的封装器类，它还存在一个名为 TYPE 的标准字段。TYPE 字段的作用是为相关的基本数据类型产生 Class 对象的一个句柄，如下所示：

……等价于……
521 页表格略

11.1.2 造型前的检查

迄今为止，我们已知的 RTTI 形式包括：

(1) 经典造型，如"(Shape)"，它用 RTTI 确保造型的正确性，并在遇到一个失败的造型后产生一个 ClassCastException 违例。

(2) 代表对象类型的 Class 对象。可查询 Class 对象，获取有用的运行期资料。

在 C++中，经典的"(Shape)"造型并不执行 RTTI。它只是简单地告诉编译器将对象当作新类型处理。而 Java 要执行类型检查，这通常叫作“类型安全”的下溯造型。之所以叫“下溯造型”，是由于类分层结构的历史排列方式造成的。若将一个 Circle（圆）造型到一个 Shape（几何形状），就叫做上溯造型，因为圆只是几何形状的一个子集。反之，若将 Shape 造型至 Circle，就叫做下溯造型。然而，尽管我们明确知道 Circle 也是一个 Shape，所以编译器能够自动上溯造型，但却不能保证一个 Shape 肯定是一个 Circle。因此，编译器不允许自动下溯造型，除非明确指定一次这样的造型。

RTTI 在 Java 中存在三种形式。关键字 instanceof 告诉我们对象是不是一个特定类型的实例（Instance 即“实例”）。它会返回一个布尔值，以便以问题的形式使用，就象下面这样：

```
if(x instanceof Dog)
((Dog)x).bark();
```

将 x 造型至一个 Dog 前，上面的 if 语句会检查对象 x 是否从属于 Dog 类。进行造型前，如果没有其他信息可以告诉自己对象的类型，那么 instanceof 的使用是非常重要的——否则会得到一个 ClassCastException 违例。

我们最一般的做法是查找一种类型（比如要变成紫色的三角形），但下面这个程序却演示了如何用 instanceof 标记出所有对象。

522-524 页程序

在 Java 1.0 中，对 instanceof 有一个比较小的限制：只可将其与一个已命名的类型比较，不能同 Class 对象作对比。在上述例子中，大家可能觉得将所有那些 instanceof 表达式写出来是件很麻烦的事情。实际情况正是这样。但在 Java 1.0 中，没有办法让这一工作自动进行——不能创建 Class 的一个 Vector，再将其与之比较。大家最终会意识到，如编写了数量众多的 instanceof 表达式，整个设计都可能出现问题。

当然，这个例子只是一个构想——最好在每个类型里添加一个 static 数据成员，然后在构建器中令其增值，以便跟踪计数。编写程序时，大家可能想象自己拥有类的源码控制权，能够自由改动它。但由于实际情况并非总是这样，所以

RTTI 显得特别方便。

1. 使用类标记

PetCount.java 示例可用 Java 1.1 的类标记重写一遍。得到的结果显得更加明确易懂：

524-526 页程序

在这里，typenames（类型名）数组已被删除，改为从 Class 对象里获取类型名称。注意为此而额外做的工作：例如，类名不是 Getbil，而是 c11.petcount2.Getbil，其中已包含了包的名字。也要注意系统是能够区分类和接口的。

也可以看到，petTypes 的创建模块不需要用一个 try 块包围起来，因为它会在编译期得到检查，不会象 Class.forName()那样“掷”出任何违例。

Pet 动态创建好以后，可以看到随机数字已得到了限制，位于 1 和 petTypes.length 之间，而且不包括零。那是由于零代表的是 Pet.class，而且一个普通的 Pet 对象可能不会有人感兴趣。然而，由于 Pet.class 是 petTypes 的一部分，所以所有 Pet（宠物）都会算入计数中。

2. 动态的 instanceof

Java 1.1 为 Class 类添加了 isInstance 方法。利用它可以动态调用 instanceof 运算符。而在 Java 1.0 中，只能静态地调用它（就象前面指出的那样）。因此，所有那些烦人的 instanceof 语句都可以从 PetCount 例子中删去了。如下所示：

526-528 页程序

可以看到，Java 1.1 的 isInstance()方法已取消了对 instanceof 表达式的需要。此外，这也意味着一旦要求添加新类型宠物，只需简单地改变 petTypes 数组即可；毋需改动程序剩余的部分（但在使用 instanceof 时却是必需的）。

11.2 RTTI 语法

Java 用 Class 对象实现自己的 RTTI 功能——即便我们要做的只是象造型那样的一些工作。Class 类也提供了其他大量方式，以方便我们使用 RTTI。

首先必须获得指向适当 Class 对象的的一个句柄。就象前例演示的那样，一个办法是用一个字串以及 Class.forName()方法。这是非常方便的，因为不需要那种类型的一个对象来获取 Class 句柄。然而，对于自己感兴趣的类型，如果已有了它的一个对象，那么为了取得 Class 句柄，可调用属于 Object 根类一部分的一个方法：getClass()。它的作用是返回一个特定的 Class 句柄，用来表示对象的实际类型。Class 提供了几个有趣且较为有用的方法，从下例即可看出：

528-529 页程序

从中可以看出，class FancyToy 相当复杂，因为它从 Toy 中继承，并实现了 HasBatteries，Waterproof 以及 ShootsThings 的接口。在 main()中创建了一个 Class 句柄，并用位于相应 try 块内的 forName()初始化成 FancyToy。

Class.getInterfaces 方法会返回 Class 对象的一个数组，用于表示包含在 Class 对象内的接口。

若有一个 Class 对象，也可以用 getSuperclass()查询该对象的直接基础类是什么。当然，这种做会返回一个 Class 句柄，可用它作进一步的查询。这意味着在运行期的时候，完全有机会调查到对象的完整层次结构。

若从表面看，Class 的 newInstance()方法似乎是克隆（clone()）一个对象的另一种手段。但两者是有区别的。利用 newInstance()，我们可在没有现成对象供“克隆”的情况下新建一个对象。就象上面的程序演示的那样，当时没有 Toy 对象，只有 cy——即 y 的 Class 对象的一个句柄。利用它可以实现“虚拟构建器”。换言之，我们表达：“尽管我不知道你的准确类型是什么，但请你无论如何都正确地创建自己。”在上述例子中，cy 只是一个 Class 句柄，编译期间并不知道进一步的类型信息。一旦新建了一个实例后，可以得到 Object 句柄。但那个句柄指向一个 Toy 对象。当然，如果要将除 Object 能够接收的其他任何消息发出去，首先必须进行一些调查研究，再进行造型。除此以外，用 newInstance()创建的类必须有一个默认构建器。没有办法用 newInstance()创建拥有非默认构建器的对象，所以在 Java 1.0 中可能存在一些限制。然而，Java 1.1 的“反射”API（下一节讨论）却允许我们动态地使用类里的任何构建器。

程序中的最后一个方法是 printInfo()，它取得一个 Class 句柄，通过 getName()获得它的名字，并用 interface()调查它是不是一个接口。

该程序的输出如下：

530 页程序

所以利用 Class 对象，我们几乎能将一个对象的祖宗十八代都调查出来。

11.3 反射：运行期类信息

如果不知道一个对象的准确类型，RTTI 会帮助我们调查。但却有一个限制：类型必须是在编译期间已知的，否则就不能用 RTTI 调查它，进而无法展开下一步的工作。换言之，编译器必须明确知道 RTTI 要处理的所有类。

从表面看，这似乎并不是一个很大的限制，但假若得到的是一个不在自己程序空间内的对象的句柄，这时又会怎样呢？事实上，对象的类即使在编译期间也不可由我们的程序使用。例如，假设我们从磁盘或者网络获得一系列字节，而且被告知那些字节代表一个类。由于编译器在编译代码时并不知道那个类的情况，所以怎样才能顺利地使用这个类呢？

在传统的程序设计环境中，出现这种情况的概率或许很小。但当我们转移到一个规模更大的编程世界中，却必须对这个问题加以高度重视。第一个要注意的是基于组件的程序设计。在这种环境下，我们用“快速应用开发”（RAD）模型来构建程序项目。RAD 一般是在应用程序构建工具中内建的。这是编制程序的一种可视途径（在屏幕上以窗体的形式出现）。可将代表不同组件的图标拖曳到窗体中。随后，通过设定这些组件的属性或者值，进行正确的配置。设计期间的配置要求任何组件都是可以“例示”的（即可以自由获得它们的实例）。这些组件也要揭示出自己的一部分内容，允许程序员读取和设置各种值。此外，用于控制 GUI 事件的组件必须揭示出与相应的方法有关的信息，以便 RAD 环境帮助程序员用自己的代码覆盖这些由事件驱动的方法。“反射”提供了一种特殊的机制，

可以侦测可用的方法，并产生方法名。通过 Java Beans（第 13 章将详细介绍），Java 1.1 为这种基于组件的程序设计提供了一个基础结构。

在运行期查询类信息的另一个原动力是通过网络创建与执行位于远程系统上的对象。这就叫作“远程方法调用”（RMI），它允许 Java 程序（版本 1.1 以上）使用由多台机器发布或分布的对象。这种对象的分布可能是由多方面的原因引起的：可能要做一件计算密集型的工作，想对它进行分割，让处于空闲状态的其他机器分担部分工作，从而加快处理进度。某些情况下，可能需要将用于控制特定类型任务（比如多层客户／服务器架构中的“运作规则”）的代码放置在一台特殊的机器上，使这台机器成为对那些行动进行描述的一个通用储藏所。而且可以方便地修改这个场所，使其对系统内的所有方面产生影响（这是一种特别有用的设计思路，因为机器是独立存在的，所以能轻易修改软件！）。分布式计算也能更充分地发挥某些专用硬件的作用，它们特别擅长执行一些特定的任务——例如矩阵逆转——但对常规编程来说却显得太夸张或者太昂贵了。

在 Java 1.1 中，Class 类（本章前面已有详细论述）得到了扩展，可以支持“反射”的概念。针对 Field，Method 以及 Constructor 类（每个都实现了 Memberinterface——成员接口），它们都新增了一个库：java.lang.reflect。这些类型的对象都是 JVM 在运行期创建的，用于代表未知类里对应的成员。这样便可用构建器创建新对象，用 get()和 set()方法读取和修改与 Field 对象关联的字段，以及用 invoke()方法调用与 Method 对象关联的方法。此外，我们可调用方法 getFields()，getMethods()，getConstructors()，分别返回用于表示字段、方法以及构建器的对象数组（在联机文档中，还可找到与 Class 类有关的更多的资料）。因此，匿名对象的类信息可在运行期被完整的揭露出来，而在编译期间不需要知道任何东西。

大家要认识的很重要的一点是“反射”并没有什么神奇的地方。通过“反射”同一个未知类型的对象打交道时，JVM 只是简单地检查那个对象，并调查它从属于哪个特定的类（就象以前的 RTTI 那样）。但在这之后，在我们做其他任何事情之前，Class 对象必须载入。因此，用于那种特定类型的.class 文件必须能由 JVM 调用（要么在本地机器内，要么可以通过网络取得）。所以 RTTI 和“反射”之间唯一的区别就是对 RTTI 来说，编译器会在编译期打开和检查.class 文件。换句话说，我们可以用“普通”方式调用一个对象的所有方法；但对“反射”来说，.class 文件在编译期间是不可使用的，而是由运行期环境打开和检查。

11.3.1 一个类方法提取器

很少需要直接使用反射工具；之所以在语言中提供它们，仅仅是为了支持其他 Java 特性，比如对象序列化（第 10 章介绍）、Java Beans 以及 RMI（本章后面介绍）。但是，我们许多时候仍然需要动态提取与一个类有关的资料。其中特别有用的工具便是一个类方法提取器。正如前面指出的那样，若检视类定义源码或者联机文档，只能看到在那个类定义中被定义或覆盖的方法，基础类那里还有大量资料拿不到。幸运的是，“反射”做到了这一点，可用它写一个简单的工具，令其自动展示整个接口。下面便是具体的程序：

533-534 页程序

Class 方法 getMethods() 和 getConstructors() 可以分别返回 Method 和

Constructor 的一个数组。每个类都提供了进一步的方法，可解析出它们所代表的方法的名字、参数以及返回值。但也可以象这样一样只使用 toString()，生成一个含有完整方法签名的字串。代码剩余的部分只是用于提取命令行信息，判断特定的签名是否与我们的目标字串相符（使用 indexOf()），并打印出结果。

这里便用到了“反射”技术，因为由 Class.forName()产生的结果不能在编译期间获知，所以所有方法签名信息都会在运行期间提取。若研究一下联机文档中关于“反射”（Reflection）的那部分文字，就会发现它已提供了足够多的支持，可对一个编译期完全未知的对象进行实际的设置以及发出方法调用。同样地，这也属于几乎完全不用我们操心的一个步骤——Java 自己会利用这种支持，所以程序设计环境能够控制 Java Beans——但它无论如何都是非常有趣的。

一个有趣的试验是运行 java ShowMehods ShowMethods。这样做可得到一个列表，其中包括一个 public 默认构建器，尽管我们在代码中看见并没有定义一个构建器。我们看到的是由编译器自动合成的那一个构建器。如果随之将 ShowMethods 设为一个非 public 类（即换成“友好”类），合成的默认构建器便不会在输出结果中出现。合成的默认构建器会自动获得与类一样的访问权限。

ShowMethods 的输出仍然有些“不爽”。例如，下面是通过调用 java ShowMethods java.lang.String 得到的输出结果的一部分：

534-535 页程序

若能去掉象 java.lang 这样的限定词，结果显然会更令人满意。有鉴于此，可引入上一章介绍的 StreamTokenizer 类，解决这个问题：

535-537 页程序

ShowMethodsClean 方法非常接近前一个 ShowMethods，只是它取得了 Method 和 Constructor 数组，并将它们转换成单个 String 数组。随后，每个这样的 String 对象都在 StripQualifiers.Strip()里“过”一遍，删除所有方法限定词。正如大家看到的那样，此时用到了 StreamTokenizer 和 String 来完成这个工作。

假如记不得一个类是否有一个特定的方法，而且不想在联机文档里逐步检查类结构，或者不知道那个类是否能对某个对象（如 Color 对象）做某件事情，该工具便可节省大量编程时间。

第 17 章提供了这个程序的一个 GUI 版本，可在自己写代码的时候运行它，以便快速查找需要的东西。

11.4 总结

利用 RTTI 可根据一个匿名的基础类句柄调查出类型信息。但正是由于这个原因，新手们极易误用它，因为有些时候多形性方法便足够了。对那些以前习惯程序化编程的人来说，极易将他们的程序组织成一系列 switch 语句。他们可能用 RTTI 做到这一点，从而在代码开发和维护中损失多形性技术的重要价值。Java 的要求是让我们尽可能地采用多形性，只有在极特别的情况下才使用 RTTI。

但为了利用多形性，要求我们拥有对基础类定义的控制权，因为有些时候在程序范围之内，可能发现基础类并未包括我们想要的方法。若基础类来自一个库，或者由别的什么东西控制着，RTTI 便是一种很好的解决方案：可继承一个新类

型，然后添加自己的额外方法。在代码的其他地方，可以侦测自己的特定类型，并调用那个特殊的方法。这样做不会破坏多形性以及程序的扩展能力，因为新类型的添加不要求查找程序中的 switch 语句。但在需要新特性的主体中添加新代码时，就必须用 RTTI 侦测自己特定的类型。

从某个特定类的利益的角度出发，在基础类里加入一个特性后，可能意味着从那个基础类衍生的其他所有类都必须获得一些无意义的“鸡肋”。这使得接口变得含义模糊。若有人从那个基础类继承，且必须覆盖抽象方法，这一现象便会使他们陷入困扰。比如现在用一个类结构来表示乐器（Instrument）。假定我们想清洁管弦乐队中所有适当乐器的通气音栓（Spit Valve），此时的一个办法是在基础类 Instrument 中置入一个 ClearSpitValve()方法。但这样做会造成一个误区，因为它暗示着打击乐器和电子乐器中也有音栓。针对这种情况，RTTI 提供了一个更合理的解决方案，可将方法置入特定的类中（此时是 Wind，即“通气口”）——这样做是可行的。但事实上一种更合理的方案是将 prepareInstrument()置入基础类中。初学者刚开始时往往看不到这一点，一般会认定自己必须使用 RTTI。

最后，RTTI 有时能解决效率问题。若代码大量运用了多形性，但其中的一个对象在执行效率上很有问题，便可用 RTTI 找出那个类型，然后写一段适当的代码，改进其效率。

11.5 练习

(1) 写一个方法，向它传递一个对象，循环打印出对象层次结构中的所有类。

(2) 在 ToyTest.java 中，将 Toy 的默认构建器标记成注释信息，解释随之发生的事情。

(3) 新建一种类型的集合，令其使用一个 Vector。捕获置入其中的第一个对象的类型，然后从那时起只允许用户插入那种类型的对象。

(4) 写一个程序，判断一个 Char 数组属于基本数据类型，还是一个真正的对象。

(5) 根据本章的说明，实现 clearSpitValve()。

(6) 实现本章介绍的 rotate(Shape)方法，令其检查是否已经旋转了一个圆（若已旋转，就不再执行旋转操作）。