第23章　面向对象的编程

面向对象的编程(即OOP)，是让非编程人员感到很玄妙的一个字眼。虽然它听起来既神秘又复杂，但实际上就是我们在Director里一直在做的事情。OOP就是把程序与数据结合在一起的编程。让行为附属于角色，就是OOP。程序是行为里的Lingo，数据是角色的属性，即演员、位置和行为的自定义属性。

OOP使我们能够只写一段程序，却能把这段程序用于多个不同的对象。在Director里，这些对象通常是角色，但也不一定必须是角色。对象可以只是一组数据，如数据库里的一个条目或游戏里的一只敌舰。

23.1　什么是对象

行为是面向对象的程序，专门被定义为具有面向Director角色的性质。我们可以用父代剧本(parent script)创建非面向角色的对象(有时被称作代码对象)。

提示　代码对象是内存里独立的代码段，它管理自己内部的数据。代码对象与其他Director对象有一些共性。代码对象有称为属性(property)的内部变量，并像行为那样可以对消息有所反应。但与Lingo环境里的其他对象不同的是，我们看不到代码对象，并且它们也不是现成的。

在Lingo里，我们通过编写父代剧本的方式定义对象模板。可以通过向父代剧本发送消息而由那个父代剧本创建一个对象。由同一模板可以创建任意多个对象。Director手册把从父代对象创建的对象称作子对象(child object)。

通过向某个Lingo对象发送消息，可以让它做出回应。该对象的处理程序确定它应当对什么消息做出回应。对象的处理程序的内容可以是我们能够用Lingo所编写出的任何东西。Director的对象还有一个内置的数据管理系统，它与很多传统的编程语言的结构相似。除了发送给对象的消息以及从对象里发出的消息外，对象与其他成份是完全相互独立的。用OOP的术语来说，能够管理其自身的数据，并把其内部的运行与外界隔绝的代码叫做封装(encapsulation)。按照这个标准，Lingo对象就是封装的。

如果你有C++等OOP语言的编程经验，可能要花一些时间才能适应Lingo。Lingo没有任何现成的“类(class)”库，也不对对象结构制定许多规则，而是使用它自己的独特的术语。表23-1列出OOP术语和与它们等价的Lingo术语。

表23-1　面向对象的编程与Lingo术语

OOP术语	等价的Lingo术语
基类(base class)	祖先剧本(ancestor script)
类(class)	父代剧本(parent script)
实例变量(instance variable)	属性变量(property variable)
类实例/对象(class instance/object)	子对象(child object)
方法/成员函数(method/member function)	处理程序(handler)

参见第14章“创建行为”里的14.1节“控制单个角色”，可以获得更多有关行为和对象的信息。

23.2 使用对象的原因

用Lingo的对象可以做的任何事情几乎都可以用 Director的基于剪辑室的环境轻易地完成，那么为什么还要使用对象呢？因为对象有效率高、灵活等优点，并能优化地使用计算和存储资源。以下是使用对象的五大理由。

23.2.1 对象能更好地组织程序

对象能够更好地组织程序，因为父代程序既包含程序代码，又包含代码所依赖的变量。如果与打印报告或与播放数字视频文件相关的代码和变量是在同一位置，用对象可以更容易地维护它们。

23.2.2 对象能够管理全局变量的空间

同全局变量一样，对象的属性变量能一直保存着它们的值；但与全局变量不同的是，这些属性只供它们所属的对象使用。因此这些属性不属于全局变量。

假设有一个产品目录的程序。在每类产品(如电子产品或服装)里，都记录着用户已经购买的产品的数量、用户曾经关注过的产品的数量以及每位用户在每类产品上所花的时间等数据。这些数据能够帮助我们分析用户的消费意向。要想长久地保存这些数据，所有这些变量都必须是全局变量，或是某个全局变量列表里的一员。在这种情况下，我们将得到大量的全局变量，它们都用产品的类别、序号以及时间等一些相关信息命名，并可以在这个程序里被随意使用。如果真的是这样，用不了多长时间，我们就会在无意中把一个全局变量的名称用于两个不同的数据。

换一种方法，可以把每类产品制作成一个对象，并把已购买的产品的数量、用户曾关注过的产品的数量以及用户在那类产品上所花的时间设置成属性变量。电子产品对象里可以有属性变量productsBought，服装对象里也可以有这个变量。这些变量都不会变成全局变量，因为它们只供它们所属的对象使用。这样，我们就可以放心地把同一个变量名用于不同数据了，并且在程序里可以引用每个对象里的变量。

23.2.3 对象易于测试

对象是独立的。由于它不依赖于外部的任何程序，因此在整个软件的其他部分制作完成之前就可以被测试。在完成一个对象的编程后，就能够通过向它的所有处理程序发送消息而测试它了。如果这些处理程序能够设置正确的属性变量或返回正确的值，就说明对象的功能正常，并且可以集成到大规模的程序里了。易于单独测试是封装的另一个优点。

23.2.4 对象使编程更容易、更高效

由于有继承的概念，对象使得编程更容易、更高效。“继承”是OOP的术语，它指一段程序能够使用另一段程序里的处理程序和变量的能力。用继承的方法可以为相似的编程问题重复使用现有的程序，而不必每次一切都从零开始编写程序。

在Lingo里，继承的方式是这样的：父代剧本里定义了处理程序和属性。由父代剧本所创建的对象可以继承这些处理程序和属性。由同一个父代剧本可以创建多个对象。如果有必要，这些对象还可以分别拥有它们各自独有的处理程序和属性。

祖先剧本是Lingo里使用继承概念的另一个级别的剧本。父代剧本可以与祖先剧本链接。这样，祖先剧本里的所有处理程序和属性都变为了父代剧本的一部分，随后又能够继续传递给子对象。子对象既继承了祖先剧本里的处理程序和属性，又根据需要继承了父代剧本里的处理程序和属性。在本章后面的23.6节“使用ancestor”里，将详细讲述祖先剧本。

23.2.5 对象可以被重复使用

如果在编程时始终贯穿着“重复使用”这个思想，最终就会得到很多个对象，把这些对象加以组合，就能够处理大部分常规的编程任务。Director有让影片与多个演员表链接的能力。我们可以制作一个包含库对象的程序库，并轻易地把这些对象与影片链接。我们所创建的任何对象都有被重复使用的潜力。如果这个对象是外部演员表库，重复使用它是非常简单的。再比如，我们创建了一个负责测验的评分工作的对象，就可以把它用于任何测验，只要根据具体情况把某些变量修改一下就可以了。

23.3 用Lingo创建对象

要创建对象，首先要创建父代剧本。正如前一节所述，父代剧本定义子对象所使用的处理程序和属性。我们试图解决的编程问题是确定父代剧本里应当包括的处理程序和属性。所有父代剧本都需要的基本处理程序是new处理程序。

new处理程序用父代剧本创建一个新对象。父代剧本就像是文字处理软件里的模板。在文字处理软件里，我们可以由一个模板得到任意多个相同的文件；同样，调用某个父代剧本里的new处理程序就可以创建任意多个相同的对象。我们每向一个父代剧本发送一个 new消息，就由这个父代剧本创建一个新对象。

new处理程序的句法是这样的：

```
on new me
  return me
end
```

在由on开头的一行里，使用me是很重要的。me带有一个指针，它指向由父代剧本所创建的对象在内存里的位置。在new里使用return命令行也是很重要的，它返回一个指向对象的指针。如果不写return命令行，在创建了对象之后，却无法和它交换信息。

Script窗口是创建父代剧本的地方。在这个窗口里，可以找到Director的“Parent script(父代剧本)”类型。如果我们意外地用一个“影片剧本(Movie script)”创建了一个对象，并向它发送消息，Director将在全部影片剧本里寻找处理这个消息的处理程序，而那个对象却不能得到这个消息。

当用父代剧本创建对象时，最好把它们存储为全局变量。这样，在影片的任何位置想要使用这个对象时，都能得到它。

同其他全局变量一样，全局对象在影片里一直存在着。当一个对象从父代剧本里生成后，它就独立存在于内存里。这个对象不需要再去读取父代剧本里的信息。当我们运行另一部不含有该对象的父代剧本的影片时，该对象仍旧有效。这个对象使用的是属于它自己的一套在

父代剧本里定义的程序代码，并已经存储在内存里了。在创建了对象后，如果再修改父代剧本，将不会影响已经存在于内存里的对象。如果需要，必须由父代剧本重新创建一个对象。如果想要更新由同一个父代剧本创建的多个对象，必须创建新的对象，再逐个替换旧的对象。

有了这些背景知识，就可以开始创建对象了。创建一个新的、空的父代剧本，方法是先创建一个剧本演员。点击Script窗口里的Info按钮，并用Type下拉菜单把剧本的类型属性改为Parent。在剧本的Name栏里输入minimal这个名称。父代剧本必须有名称，因为我们将要在其他Lingo代码里引用它。

在这个最短的父代剧本里输入如下代码，并把该剧本演员命名为 minimal：

```
on new me
  return me
end
```

在消息窗口里输入如下信息，就可以创建一个新对象：

```
minimalObj = new(script "minimal")
```

我们所输入的代码行用 minimal剧本创建了一个对象，并把它放在了全局变量 minimalObj里。把对象存储在全局变量里是一种好方法，这样，在任何需要使用它的时候都能得到它。

在消息窗口里输入如下信息，再按回车键，可以检查是否真的创建了一个对象。 Director返回这个对象变量的内容，从而证实已成功地创建了对象：

```
put minimalObj
-- <offspring "minimal" 2 8daa90>
```

虽然成功地创建了对象minimalObj，但现在还不能用它来完成任何任务。除了 new外，它不能接收任何消息，也没有用来存储数据的属性变量。

下面，添加一段程序，使一个按钮成为创建新对象的工具，这样就不必在每次播放该影片时都在消息窗口里输入命令了。下面的程序编写了一个 on mouseUp处理程序，它负责创建对象：

```
on mouseUp
  global gObj
  set gObj = new(script "minimal")
end
```

把影片回转到起点再播放它，并通过点击舞台上的按钮创建一个新对象。在消息窗口里输入put gObj并按回车键，可以检查是否成功地创建了这个对象。

为该对象创建一个on hello处理程序，使得这个对象能够发出一声系统警告声。编辑父代剧本，使其变为以下内容：

```
on new me
  return me
end

on hello me
  beep
end
```

把影片回转到起点再播放，并通过点击舞台上的按钮创建对象的一个新实例。现在可以在消息窗口里输入如下信息。如果一切正常，计算机应当发出一声系统警告声。

```
hello (gObj)
```

参见第14章里的14.2节“创建简单的行为”，可以获得更多有关行为的信息。

23.4 创建对象的属性

到目前为止，我们已经创建了一个简单的对象，如果向它发送 hello消息，它就发出系统警告声。不过这个对象的功能还是太少了。要向对象里添加更多功能，所需要的不仅仅是处理程序。这个最短的对象缺少能够执行某种功能的第二种成份：属性。对象的属性变量可以用来存储任何类型的数据，并且对于这个对象的每个实例来说，它都是唯一的。

应当像在行为里声明全局变量和属性那样在父代剧本的顶部声明对象的属性变量：

```
property pProp1, pProp2, pProp3...
```

在父代剧本内部，只要使用这些属性的名称就可以引用它们了。在父代剧本外部，需要使用“点句法”引用它们。例如，可以用 gObj引用某个对象，如果想要得到它的属性 pProp1的值，使用gObj.pProp1就可以了。

下面是一个简单的父代剧本，在new处理程序里设置了一个属性。我们随后可以用 on test处理程序查看那个属性是什么：

```
property pTest

on new me
  pTest = "Hello World. "
  return me
end

on test me
  put pTest
end
```

在消息窗口里可以做如下测试：

```
gObj = new(script "Test Object")
test(gObj)
-- "Hello World. "
```

本章曾经讲过，可以从一个父代剧本创建出多个相同的对象。随后为这些对象添加不同的属性，就会使它们各自带有不同的特征。这样，我们就不必重新编写不同对象间的相同的处理程序和属性，从而提高了程序的效率，同时仍旧可以根据编程的需要，灵活地创建有不同特征的对象。

下面的父代剧本与刚才那个相似。不过在这里，在 new处理程序里为一个参数设置了pTest属性：

```
property pTest

on new me, val
  pTest = val
  return me
end

on test me
  put pTest
end
```

现在，在消息窗口里做以下测试。它演示了由一个父代剧本创建两个独立的对象的情况，

以及如何使两个对象的内部属性互不相同：

```
gObj1 = new(script "Test Object 2", "Hello World. ")
gObj2 = new(script "Test Object 2", "Testing... ")
put gObj1
-- <offspring "Test Object 2" 2 4abfdac>
put gObj2
-- <offspring "Test Object 2" 2 4abfe24>
test(gObj1)
-- "Hello World. "
test(gObj2)
-- "Testing... "
```

参见第14章里的14.1节“控制单个角色”，可以获得更多有关属性的信息。

23.5 使用OOP

在Director 7里，经常不知不觉地就使用了 OOP，想避免都不可能。每一段行为都是用来控制某个角色或帧的OOP程序。

过去，创建父代剧本是为了控制角色，并让它们按照特定的方式行动。现在行为已接管了这项任务，因此父代剧本的实用性降低了。

我们仍旧可以把父代剧本用于非视觉效果的任务。如果想要创建一个词汇软件，可以把单词做为对象来存储。毕竟单词是有许多属性的，如拼写、定义、同义词等等。下面是一个简单的父代剧本，它可以用来创建单词对象：

```
property pWord
property pDefinition

on new me, theword
  pWord = theword
  return me
end

on setDefinition me, def
  pDefinition = def
end

on define me
  return pDefinition
end
```

利用消息窗口可以看到如何使用它：

```
gWord = new(script "Word Object", "Clever")
put gWord.pWord
-- "Clever"
setDefinition(gWord, "Skillful in thinking. ")
put define(gWord)
-- "Skillful in thinking. "
```

现在我们可以向其中添加更多属性了，如同义词、反义词、同音 /形异义词、变移单词(由颠倒字母顺序而得到的词或词组)、常见的拼写错误等等。还可以添加更多的处理程序来接受和处理这些属性，或创建一些通用的可以接受和返回任何属性的处理程序。

可以把这种OOP逻辑用于任何类型的数据。例如，有一个关于雇员的数据库。每一个对

象可以有雇员的姓名、地址、电话号码、出生日期、社会保险号码等属性。

反过来，可以自定义设置父代剧本里的处理程序，来处理这类数据。

参见第14章里的“创建简单的行为”，可以获得更多有关行为的信息。

23.6 使用ancestor

父代剧本可以使用一种特殊的属性，即 ancestor。它指另一个父代剧本。

当我们定义和使用这个 ancestor属性时，也就给了对象使用那个祖先剧本里的全部处理程序和属性的权利。这样，我们可以得到使用着不同的父代剧本，却使用着相同的祖先剧本的对象。换句话说，它们共享某些处理程序，却不共享另外一些处理程序。

假设我们想要用对象记录商店里的货物。商店里的货物存在的一个问题是它们有不同的属性。例如，水果有保鲜的问题；罐头食品虽然没有保鲜问题，但却有体积问题，它们在有些货架上能放置，在另外一些货架上却不能放置。

在下面这个祖先剧本里，有两类产品所共有的属性。其中还有一个处理程序可以对它们进行一些处理：

```
property pProductName
property pAisleNumber

on new me
  return me
end

on whereIs me
  return pProductName&&"is in aisle"&&pAisleNumber
end
```

二者共有的属性是通道编号。这是这个商店记录货物的位置的方法。此外，所有产品都是有名称的。on whereIs处理程序返回一个包含这两项内容的字符串。

商店存储的第一类货物是水果。它是一种产品，因此它使用了祖先剧本，因此也就能够使用pProductName和pAisleNumber属性，以及on whereIs处理程序。此外，它还有保鲜期。下面是水果的父代剧本：

```
property ancestor
property pExpires

on new me
  ancestor = new(script "Product Ancestor")
  return me
end

on expiration me
  return me.pProductName&&"expires"&&pExpires
end
```

水果的父代剧本里的第一个属性是 ancestor。在on new处理程序里，它被设置为祖先剧本的一个新实例。此外，这个父代剧本有一个 pExpires属性和一个使用它的处理程序。这个处理程序还通过使用me属性和“点句法”使用了祖先剧本里的 pProductName属性。

在消息窗口里，可以看到这些程序的运行结果。当由水果父代剧本创建一个对象后，祖先剧本也就附带给了它。然后该对象就可以既使用父代剧本里的属性，又使用祖先剧本里的

属性，还可以使用这两个剧本里的处理程序。

```
gApple = new(script "Fruit Parent")
gApple.pProductName = "Apple"
gApple.pAisleNumber = 14
gApple.pExpires = "12/31/98"
put expiration(gApple)
-- "Apple expires 12/31/98"
put whereIs(gApple)
-- "Apple is in aisle 14"
```

当涉及到另一类产品时，就可以看出这种技巧的威力了。下面是罐头产品的父代剧本：

```
property ancestor
property pSize

on new me
  ancestor = new(script "Product Ancestor")
  return me
end

on size me
  return me.pProductName&&"is size"&&pSize
end
```

这个剧本与水果的父代剧本相似，但却有一种不同的属性。由于使用的是同一个祖先剧本，因此产品名称以及通道编号等属性已经处理好了。

```
gYams = new(script "Can Parent")
gYams.pProductName = "Canned Yams"
gYams.pAisleNumber = 9
gYams.pSize = "Medium"
put size(gYams)
-- "Canned Yams is size Medium"
put whereIs(gYams)
-- "Canned Yams is in aisle 9"
```

现在我们可以创建几十个、甚至上百个使用同一个祖先剧本的不同类型的父代剧本了。如果还想添加另外一个共享的属性，如订货单编号，可以把它加在祖先剧本里。这样，当再次运行影片时，所有使用那个祖先剧本的对象就都继承了这个属性。

23.7 OOP的故障排除

■ 我们经常犯的错误是在on new me处理程序里忘记写结尾处的return me语句。

■ 当使用new命令创建一个对象时，它采用的是现有的程序代码。如果我们对剧本演员做了改动，只有用new命令重新创建对象，这些改动才起作用。

■ 在父代剧本里使用的处理程序的名称不能再在其他地方用作处理程序的名称。

■ 创建父代剧本对于OOP编程人员来说也许很自然，但在很多情况下真正需要的只是行为。一个很好的规则是：只有在对象里没有可见成份(显示在舞台上的成份)时，或需要由一个对象控制多个角色时，才需要使用父代剧本。

23.8 你知道吗

■ 当我们创建了一个对象，又在消息窗口里检查它时，得到的外观奇特的结果的实际内

容是：剧本的名称、剧本的引用编号和该对象的演员位置。它可以是这样的：＜ offspring "Can Parent" 2 4abfeec＞

■ 有一个叫做actorList的系统属性。如果我们用 add向该列表里添加对象，这些对象就开始严格地在每一帧接受一次on stepFrame处理程序的调用。

■ 用父代剧本可以把某些处理程序放在内存里，以供放映机或Shockwave更换影片时使用。由于这些代码存储在全局变量里，因此它们是超出影片之外而独立存在的。我们可以用这个全局变量调用存在于父代剧本里的处理程序，尽管这个剧本并没有出现在当前影片里。