第五部分 使 用 行 为

第14章 创 建 行 为

虽然行为在Director 6里才被引入，但它们已经成为用Lingo实现各种功能的标准方法。对行为的理解 —— 包括如何编写行为，它们是怎样起作用的，怎样使用它们 —— 是发挥Lingo编程的强大功能的关键。

14.1 控制单个角色

行为所做的主要的事情是控制单个的角色。我们编写行为，然后将它们赋予剪辑室中的一个或多个角色，让这些行为“接管”角色。

正像“行为”这个名字一样，行为的任务是告诉角色应有怎样的行为。例如，行为中的on mouseUp处理程序告诉角色当用户点击角色时，角色应有怎样的行为。

14.1.1 角色消息

许多消息是从各种Director事件传给行为的。“mouseUp”是最简单的例子。行为剧本首先获得这个消息，所以可以在帧和电影剧本响应之前使用它们。

下面列出了可以发送给行为的全部消息。它同第 12章“学习Lingo”所给出的清单类似，但是这里仅列出了行为可以接受的消息：

■ mouseDown —— 当光标置于角色之上时，用户按下鼠标按钮。

■ mouseUp —— 当光标置于角色之上时用户按下鼠标按钮，并在光标仍在角色之上时释放鼠标按钮。

■ mouseUpOutside —— 当光标置于角色之上时用户按下鼠标按钮，并在光标离开角色后释放鼠标按钮。

■ mouseEnter —— 光标进入角色区域。在油墨设置成 Copy时，角色区域是围绕角色的矩形区域。若油墨设置为Matte，角色区域是角色被蒙版的区域。

■ mouseLeave —— 鼠标离开角色区域。

■ mouseWithin —— 当光标在角色区域时，每一个帧循环都发送该消息。

■ prepareFrame —— 在帧即将显示在舞台上之前，每一个帧循环都发送该消息。

■ enterFrame —— 在帧刚刚显示在舞台上之后，每一个帧循环都发送该消息。

■ exitFrame —— 在下一帧即将开始之前，每一个帧循环都发送该消息。

■ beginSprite —— 在角色首次出现或再次出现在舞台上时，发送该消息。

■ endSprite —— 在角色即将离开舞台之前，发送该消息。

■ keyDown —— 在用户按下键盘上的某个键时，发送该消息。此消息仅发送给可编辑的文本演员或域。

■ keyUp —— 用户按下键盘上的某个键并释放。此消息仅发送给可编辑的文本演员或域。

上述的每一个消息都可以在行为中被同名的处理程序处理。除了这些事件处理程序，还可以向行为里添加进几个特殊的函数。每一个都使我们能够进一步自定义行为。

■ getBehaviorDescription —— 此函数可以定义行为的文本描述。定义的文本显示在行为监察窗的底部。此函数使用简单，我们只需要创建一个字符串，然后用 return发送即可。

■ getPropertyDescriptionList —— 此函数创建包含行为参数的属性列表。当行为被拖到角色上时，使用这个函数可以使库行为显示一个带有参数设置的对话框。

■ runPropertyDialog —— 当行为被拖到角色上时，此函数允许我们使用 Lingo语言自动设置行为的参数。

■ getBehaviorToolTip —— 工具提示(tool tips)是指我们把光标掠过一些屏幕元素时，显示帮助文本的黄色小方框。把这种文本添加到行为里后，如果把这个行为被添加到库面板里，那么库面板将把此文本显示为工具提示。

14.1.2 属性

行为是一种面向对象的编程方法。面向对象的意思是程序和数据存储在同一地方。对行为来说，程序指的是行为的处理程序，数据指的是行为所使用的变量。这些变量被称作属性(property)。

属性是介于局部变量和全局变量间的一种变量。局部变量仅存在于单个处理程序中，全局变量存在于整个影片中。然而，当属性存在于整个行为剧本中时，此行为中的所有处理程序都可以使用它，但是外部的处理程序通常却不能使用它。

我们可以像创建全局变量那样创建属性，所使用的不是 global命令，而是property命令。定义属性的最好位置是行为剧本顶部的前几行。

属性中包含对行为来说十分重要的数据。例如，如果行为要求角色横向移动穿过舞台，就可能需要两个属性 —— horizLoc和horizSpeed。这两个属性将分别对应于角色当前的横向位置和角色每一帧移动的像素数。

虽然第一个属性horizLoc可能只从角色开始出现时才起作用，但是第二个属性 horizSpeed可能是当我们将行为拖动到角色上时所能够设置的内容。这时， horizSpeed可能既是参数又是属性。

14.1.3 使用me

因为行为是面向对象的，所以行为里的每一个处理程序必须在它的第一个参数里含有对它所属于的对象的引用。听起来很糊涂？但就是这样。

行为剧本自身只是演员中的一些文本。当我们将它赋予角色时，它仍是一串文本，只是角色知道它有一种关系。但是，在我们播放电影，并且角色出现在舞台上时，一个对象就被创建了。

这个对象被称为行为的一个实例(instance)。它是行为的一种拷贝。 Lingo代码被调入内存中的新位置，并且属性变量被创建。现在，此实例就可以赋予并控制角色了。

如果同样的行为要赋予两个不同的角色，而这两个角色同时出现在舞台上，实际上这种情况下的这两个角色拥有同一行为的两个不同实例。这两个实例拥有相同处理程序的拷贝，

并且都具有同名属性，但是属性值存储在不同的位置，并可以有不同的数值。

实例的概念要体现在行为代码中。每一个处理程序需要知道它是行为的一部分。为了在代码中表明这一点，应该把参数 me放在行为里的所有处理程序的第一参数的位置。

创建一个新的剧本演员，并将它设置为 Behavior(行为)类型。现在，给它添加如下的简单剧本：

```
on mouseUp me
  put me
end
```

将这个行为命名为Test Behavior(测试行为)，并将它赋予舞台上的任一角色(只需要一个简单的图形就可以了)。播放影片并点击这一角色。我们将看见像这样的结果显示在消息窗口里：

```
-- <offspring "Test Behavior" 4 33b1e64>
```

这行末尾的实际数据将会有变化。此数据是行为实例在内存中的位置，它对我们的编程过程并不重要。但是，专用变量 me却很重要，因为它指向对象的实际实例。将它当作行为中所有处理程序的第一参数，以表明这些处理器是行为的一部分。

如果仍觉糊涂，不妨这样想想：通常行为中各处理程序的第一参数 me将所有的行为联系在起来。me指代的就是这个行为，这与 sprite指代的是角色、member指代的是演员是一样的。me也有属性，最有用的属性是 spriteNum。我们可以使用这个属性得到此行为目前所属的角色的角色通道编号。

```
on mouseUp me
  put me.spriteNum
end
```

如果我们没有将 me放在处理程序名的后面，剧本根本不会起作用。实际上，在我们试着关闭剧本窗口时，它会给我们一个错误信息，因为如果 me不在处理程序名后，Director甚至不知道me指的是谁。

若想了解更多关于面向对象编程的信息，请参看第 23章“面向对象的编程”里的 23.3节“用Lingo创建对象”。

14.2 创建简单的行为

简单的行为可以简单到只有一个只含一条 Lingo命令的处理程序。另一方面，复杂的行为又可以复杂到一个事件处理程序就有长达几百条语句，并且包括许多个自定义的处理程序。

14.2.1 浏览行为

简单的浏览处理程序常被赋予按钮或位图角色。它使用 on mouseUp处理程序触发一个浏览命令。下面是一个例子：

```
on mouseUp me
  go to frame 7
end
```

此处理程序并不比用行为监察窗创建的行为更有用。但是，使用单个属性，我们可以创建用于许多不同按钮上的行为。

```
property pTargetFrame

on getPropertyDescriptionList me
  return [#pTargetFrame: [#comment: "Target Frame:",¬
      #format: #integer, #default: 1]]
end

on mouseUp me
  go to frame pTargetFrame
end
```

提示 注意在第一个处理程序中较长的那一行第一部分末尾的连续字符(¬)。Lingo行可以任意长，但是印在书上不可能任意长。所以在编写程序时不必考虑这个字符，只需输入较长的一整行就可以了。若想在 Lingo中使用连续字符，则在 Mac系统下按Option+Return组合键，在Windows系统下，按Alt+Enter组合键。如果只按下Return或Enter，一行语句将变成两行并产生错误。

这个行为使用 pTargetFrame属性以确定按钮将使影片跳到哪一帧。此属性在 on getPropertyDescriptionList函数中使用，以至我们可以在该行为的 Parameters(参数)对话框中选择帧的编号。

注释 pTargetFrame属性也可以只命名为argetFrame。但是，有时让我们的变量名提供一个类型提示是很有帮助的。有些编程人员喜欢在属性变量前面加一个“ p ”。“ g ”一般是全局变量的前缀。有些程序开发人员更愿意将“ i ”放在属性变量的前面，以代表“ instance(实例) ”。它们确实不是Lingo语言的一部分，但却是一种提供方便的惯例。

on getPropertyDescriptionList函数中所包含的列表是一个属性列表，其中包含一个或多个其他属性列表。列表的第一项(在本例中也是唯一的一项)的属性名是属性的名字，由于它前面添加了一个“ # ”，而成为了符号。作为属性值列表的内容有三项：第一是 #comment(注释)属性，是Parameters对话框将为此参数显示的字符串；第二项是 #format(格式)，告诉Parameters对话框接受哪种类型的值；最后一项是参数的缺省值。

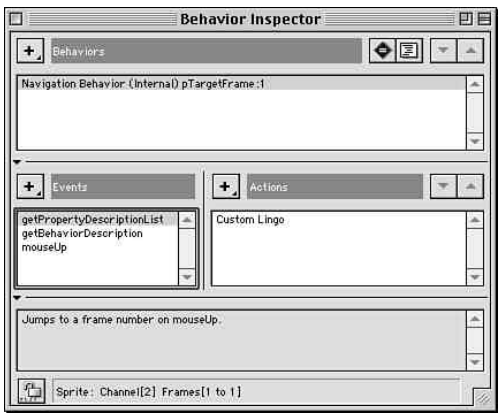

图14-1 行为监察窗显示着带有描述信息的自定义行为

当我们拖动行为并把它放到舞台上或剪辑室中后， Parameters对话框将出现，其中带有标

着“ Target Frame: ”栏的。此栏可以接受任何数值，起初它会显示缺省值“ 1 ”。

此行为现在可以赋予给许多角色，但是每次的目标帧要不同。这种可重复使用性是行为的强大功能之一。我们可以给一部影片装上许多浏览按钮，但只需要这一个行为。

要完成这个行为，我们要给它加上 on getBehaviorDescription函数。当给角色添加该剧本后，这个函数将在行为监察窗中显示一些信息性质的文本：

```
on getBehaviorDescription me
  return "Jumps to a frame number on mouseUp."
end
```

图14-1显示了这个行为赋予角色时的行为监察窗。

14.2.2 掠过行为

Director中一种常见行为是针对于按钮或其他的图形的。当光标置于其上时，它们的外观会发生改变。这种行为通常称作掠过(rollover)。

使用on mouseEnter和on mouseLeave处理程序，很容易实现掠过行为。当光标进入角色区域时，位图将发生改变；在光标离开它时，位图恢复原样。此行为的创建可以如下例一样简单：

```
on mouseEnter me
  sprite(1).member = member("button rollover")
end

on mouseLeave me
 sprite(1).member = member("button normal")
end
```

我们只要仅在通道1的角色上使用此行为，并仅使用 button rollover和button normal两个位图演员，此行为就能很好地发挥作用。因为它只能在某一指定的实例中使用，所以它还不能算是真正的行为。

要想改进这一行为，能做的第一件事让它自动分析出它被哪一个角色使用。角色的编号是我们熟知的行为实例 me的一个属性。不同于将角色的编号硬编码为“ 1 ”，我们可以使用句法me.spriteNum。它作为一个变量被使用，此变量包含了角色使用的角色通道的编号。如果它恰是角色1，将包含“ 1 ”这个数字；否则将包含任一角色通道的编号。

```
on mouseEnter me
   sprite(me.spriteNum).member = member("button rollover")
end

on mouseLeave me
  sprite(me.spriteNum).member = member("button normal")
end
```

现在我们已经有了可以赋予任何通道的任何角色的行为。但是，它仍然使用了两个硬编码演员作为正常状态和掠过状态。我们能够很容易识别按钮的正常状态，因为它可能是用来设置按钮初始状态的演员。我们可以获得这个演员，并将它存储在 on beginSprite处理程序的一个属性里。

```
property pMemberNormal

on beginSprite me
```

```
  pMemberNormal = sprite(me.spriteNum).member
end

on mouseEnter me
  sprite(me.spriteNum).member = member("button rollover")
end

on mouseLeave me
  sprite(me.spriteNum).member = pMemberNormal
end
```

onbeginSprite处理程序通过me.spriteNum使用角色的编号，在剪辑室中得到指定给角色的演员。它存储在属性变量pMemberNormal中，然后作为正常状态的演员在on mouseLeave处理程序中使用。

现在仅剩下了一个硬编码元素：掠过状态的演员。无论使用哪个角色、哪个正常状态的演员，它总是名为button rollover的演员。

通常，完成这种行为有三种技术。第一是假设按钮的掠过状态的演员总是紧接着正常状态的演员，插在演员表中的下一位置。所以如果正常状态演员的编号是 67，掠过状态演员的编号就是68。

```
property pMemberNormal, pMemberRollover

on beginSprite me
  pMemberNormal = sprite(me.spriteNum).member
  pMemberRollover = member(pMemberNormal.number + 1)
end

on mouseEnter me
  sprite(me.spriteNum).member = pMemberRollover
end

on mouseLeave me
  sprite(me.spriteNum).member = pMemberNormal
end
```

on beginSprite处理程序获得当前的演员，并像往常一样将它放在 pMemberNormal中。它也获得了该演员的编号，并对其增加 1，创建一个新的演员对象。该演员总是在演员表中紧接着的下一个。它存储在pMemberRollover中。

此行为仅在角色的正常和掠过状态置于连续的演员位置时才能正常运行。这比前面的剧本灵活得多。但是，使用演员名甚至可以使它更加灵活。我们可以得到正常状态的演员名并对它加以扩展，如加上“ rollover ”。

```
property pMemberNormal, pMemberRollover

on beginSprite me
  pMemberNormal = sprite(me.spriteNum).member
  pMemberRollover = member(pMemberNormal.name&&"rollover")
end

on mouseEnter me
  sprite(me.spriteNum).member = pMemberRollover
end
```

```
on mouseLeave me
  sprite(me.spriteNum).member = pMemberNormal
end
```

最后这两个行为的不同仅在于它们如何设置 pMemberRollover属性。在刚才这个例子中，在正常状态的名字后添加了“ rollover ”。两个&的作用是在名字与“ rollover ”间插入一个空格。所以，如果正常状态的演员命名为 button1，掠过演员必须命名为 button1 rollover。在剪辑室中，只要命名正确，不必将它放在任何特定的地方。

另一种确定哪个演员用于掠过状态的方法是在每次把行为赋予角色时，在 Parameters对话框中设置这项属性。on getPropertyDescriptionList处理程序用来完成这项工作。

```
property pMemberNormal, pMemberRollover

on getPropertyDescriptionList me
return [#pMemberRollover: [#comment: "Rollover Member:", ¬
        #format: #member, #default: VOID]]
end

on beginSprite me
  pMemberNormal = sprite(me.spriteNum).member
end

on mouseEnter me
  sprite(me.spriteNum).member = pMemberRollover
end

on mouseLeave me
  sprite(me.spriteNum).member = pMemberNormal
end
```

在on getPropertyDescriptionList函数中，使用#member格式。这将在行为的 Parameters对话框中显示一个弹出菜单，其中含有所有当前演员。我们也可用 #string使用户可以手动输入演员名。或者，可能已经用 #bitmap将对话框的弹出菜单限制为只显示位图演员了。因为在这里确实没有合理的缺省状态，于是使用 VOID。

14.2.3 动画行为

既然我们知道如何改变角色在舞台上的位置，我们就可以把这个方法用于某个行为中，并使它让角色在剪辑室中活动起来，而不使用推算处理。

这时，有三个事件有规律地发生并将消息发送给角色的行为： prepareFrame、enterFrame、exitFrame。exitFrame是最好的通用动画处理程序，因为它能首先显示未经改动的角色，然后在下一次帧循环时，被on exitFrame处理程序中的命令所改动。下面是一个简单的例子：

```
on exitFrame me
  sprite(me.spriteNum).locH = sprite(me.spriteNum).locH + 1
end
```

此行为仅获得了角色的横向位置，并在每一次帧循环时添加 1。如果在这个帧剧本的 on exitFrame处理程序中有go to the frame命令存在，则播放头被置于当前的帧上，并且这个行为在每一帧循环时，让角色移动一个像素。

要不想硬编码每一帧角色移动的像素数，可以通过属性设置来完成。当拖动行为到角色

上时，我们可以使用on getPropertyDescriptionList函数设置该属性。

```
property pSpeed

on getPropertyDescriptionList me
  return [#pSpeed: [#comment: "Speed:", ¬
        #format: #integer, #default: 1]]
end

on exitFrame me
  sprite(me.spriteNum).locH = sprite(me.spriteNum).locH + pSpeed
end
```

我们现在将此行为拖动到角色上，它提示我们输入一个有关运动速度的数字，缺省设置是“ 1 ”。这段代码里对输入值为负没有什么限制，只是角色向左边移动而已。

在on getPropertyDescriptionList函数中，我们还能使用另一个技巧，使我们在 Parameters对话框中放置一个滑动条，而不只是一个普通的文字输入域。我们要做的工作是将 #range属性加到#pSpeed的属性列表中，并增加一个其中包括 #min和#max属性的小列表。这个小列表规定了滑动条的边界。

```
property pSpeed

on getPropertyDescriptionList me
  return [#pSpeed: [#comment: "Speed:", #format: #integer, ¬
        #default: 1, #range: [#min:-50, #max:50]]]
end

on exitFrame me
  sprite(me.spriteNum).locH = sprite(me.spriteNum).locH + 1
end
```

图14-2显示了带有滑动条的 Parameters对话框。像这样的滑动条是控制参数值设置范围的极好工具。我们也可以不用滑动条，而用一个线性列表制作一个弹出菜单，例如，将[1,5,10,15,-5,-10]作为#range属性。这些为pSpeed属性设置的值在弹出菜单中是有效的。

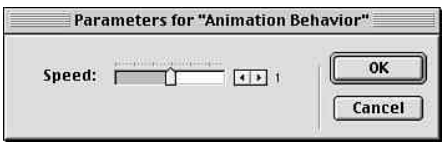

图14-2 Parameters对话框可以包含像滑动条这样的界面元素，而非一般文本框

14.3 简单的按钮行为

简单的按钮行为完成以下几项工作。第一，当按钮被按下时，它允许我们使用另一个演员，表示按钮的按下状态。第二，它能区分在角色上释放鼠标和在角色外释放鼠标这两种不同情况。这样，用户按下按钮而不想激活此按钮时，可以接着将鼠标移开并释放。第三，当一个按钮被成功地按下时，此按钮将执行某些任务，如简单的浏览任务。

像这样的行为应该有两个参数。第一个是确定按钮按下状态的属性；第二个定义行为的浏览功能。这个行为的on getPropertyDescriptionList处理程序如下：

```
property pMemberNormal, pMemberDown, pTargetFrame

on getPropertyDescriptionList me
  list = [:]
```

```
  addProp list, #pMemberDown, [#comment: "Down State Member:", ¬
    #format: #member, #default: VOID]
  addProp list, #pTargetFrame, [#comment: "Target Frame Label:", ¬
    #format: #marker, #default: VOID]
  return list
end
```

首先，注意变量 list是通过列出一个个属性而被建立的，然后被返回。这比写成一行的return命令更容易阅读。

在此列表中，属性的设置要使用两个特殊的格式。第一是 #member，它提供一个弹出菜单，其中包含可以使用的全部演员；第二是 #format，它也提供一个弹出菜单，其中包含可以使用的剪辑室标志。它还增加了 #next、#previous和#loop标志，可以与go to frame命令配合使用，以找到下一个、前一个和当前的标志。

on beginSprite处理程序只获得角色的第一个演员，即用作按钮的正常状态的演员。

```
on beginSprite me
  pMemberNormal = sprite(me.spriteNum).member
end
```

接下来的三个处理程序针对三个可能的鼠标按钮活动： mouseDown、mouseUp和mouseUpOutside、on mouseUp和mouseUpOutside处理程序的区别在于前者还要执行按钮要做的动作，而后者只将角色的演员设置回正常状态。

```
on mouseDown me
  sprite(me.spriteNum).member = pMemberDown
end

on mouseUp me
  sprite(me.spriteNum).member = pMemberNormal
  action(me)
end

on mouseUpOutside me
  sprite(me.spriteNum).member = pMemberNormal
end
```

在on mouseUp处理程序中并不只执行这个动作，此处理程序还要调用另一个称作 action的处理程序。它是我们要为行为专门创建的自定义处理程序。它不必一定要命名为 action，但这样做看上去是一个很合适的选择。

```
on action me
  go to frame pTargetFrame
end
```

这个自定义处理程序仅仅需要执行“ go to frame pTargeFrame ”命令。该属性必须包含帧标志名或一个能被go to frame使用的特殊符号：#next、#previous或#loop。

这就完成了简单的按钮行为。在此章的后面，我们将介绍怎样为它添加更多的功能，包括掠过状态、除浏览以外的动作、声音甚至确定按下和掠过状态演员的不同方法。

14.4 使用完整的行为

行为剧本包括很多部分。但是，它们都是可选项。若我们在写一个只负责单个角色的单独动作的简单行为，可能只使用一个带有一条命令的处理程序。但是，对于完整的行为，我

们将希望包括所有的可选函数。

14.4.1 行为的描述

行为的描述仅仅用于我们想在行为监察窗的底部显示一些什么信息。然而，由于行为的描述是作为on getBehaviorDescription函数在剧本中存在的，所以它也可以在剧本中担当注释的角色。

on getBehaviorDescription 函数要完成这个任务所作的全部工作就是返回一个字符串。用下面的一行语句就能够实现这个任务：

```
on getBehaviorDescription me
  return "This behavior plays a sound when clicked."
end
```

然而，大多数可能的情况是我们想返回更多的信息，而不止七个单词。这时，可以将这行语句扩展为我们所需要的众多内容，但是有一个或许更好的方法，就是首先将字符串赋值到某一变量，然后返回变量的值。

```
on getBehaviorDescription me
  desc = ""
  put "This behavior plays a sound when clicked."&RETURN after desc
  put "Choose a sound to play as the Sound paramater."&RETURN after desc
  put "Choose an action as the Action parameter." after desc
  return desc
end
```

这个例子书写和编辑起来都比较容易，对编辑这个剧本的人员来说，也更便于阅读。

如果要将行为按照任一形式分类的话，那么，行为描述也是一个可以容纳我们的名字或公司名称的好地方。

14.4.2 行为属性描述列表

虽然行为描述只是起到点缀的作用，但是对于需要自定义的行为，行为属性描述列表是一个必要的元素。我们可以使用 on getPropertyDescriptionList函数创建一个Parameters对话框，对任何行为来说，它都可以包括大量的界面元素。

由on getPropertyDescriptionList返回的属性列表包含许多更小的列表。每一个更小的列表对单个参数进行定义。它使用四个属性完成这项工作： #comment、#format、#default和#range。下面是本章前面就出现过的一个例子：

```
on getPropertyDescriptionList me
  return [#pSpeed: [#comment: "Speed:", #format: #integer, ¬
      #default: 1, #range: [#min:-50, #max:50]]]
end
```

在这个例子中，主列表包括一个属性定义： pSpeed。在这个属性中，我们又能看到需要定义的四个属性。

comment属性只是一个短字符串，它用作在 Behavior Parameters(行为参数)对话框中标识这个参数。如果你喜欢这种式样，在字符的末尾加一个冒号作为结束符是比较好的。

#format有许多的设置形式。许多设置在 Parameters对话框中放置一个特定类型的弹出菜单。例如，#member设置放置的弹出菜单，可以让我们选择影片中使用的任一演员。表 14-1中

列出了不同#format属性设置。

表14-1 #format属性使用的设置

设 置	Parameters对话框结果	可能的数值
#integer	文本输入区域	该值被转换成整数
#float	文本输入区域	该值被转换为浮点数
#string	文本输入区域	字符串
#boolean	复选框	TRUE或FALSE(1或0)
#symbol	文本输入区	符号
#member	列出所有演员的弹出菜单	演员名
#bitmap	列出位图演员的弹出菜单	演员名
#filmloop	列出影片片段演员的弹出菜单	演员名
#field	列出域演员的弹出菜单	演员名
#palette	列出调色板演员和Director内置调色板的弹出菜单	带有调色板名的字符串
#sound	列出声音演员的弹出菜单	演员名
#button	列出按钮、单选按钮和复选框演员的弹出菜单	演员名
#shape	列出图形演员的弹出菜单	演员名
#vectorShape	列出矢量图形的弹出菜单	演员名
#font	列出字体演员的弹出菜单	演员名
#digitalVideo	列出数字视频文件的弹出菜单	演员名
#script	列出剧本演员的弹出菜单	演员名
#text	列出文本演员的弹出菜单	演员名
#transition	列出内置过渡的弹出菜单	过渡演员名
#frame	文本区域	帧编号
#marker	剪辑室中所有的标志，外加“next”、“previous”和“loop”。	字符串或#next、#previous、#loop符号
#ink	油墨列表	油墨编号

大部分#format类型得出的结果是放置在 Parameters对话框中的一个弹出菜单。在实在没有可以使用的值的情况下，取而代之的是显示一个文本区域。例如，如果有一个 #sound设置的参数，而在影片中并没有声音演员，弹出菜单就由一个文本框代替。

#range属性总是属于可选项，但对限制选择的范围十分有用。对它的使用有两种方法 :作为项目的弹出菜单使用或是作为带有最大和最小值的滑动条使用。

Parameters对话框的滑动条以 [#min: a, #max: b]格式创建。滑动条的值从 a到b。在与像["a", "b", "c"]这样的线性列表一起使用时，列表中的值在弹出菜单中作为选择项使用。

任何参数都需要#default属性。但是，若我们不打算使用缺省值，可以由 " "或0代替。

14.4.3 自动属性设置

在少数情况下，我们想创建一个行为，此行为可以半自动地设置它的参数，而不是每次使用它时都要进行设置。在把行为赋予某个角色时， on runPropertyDialog处理程序使我们能够拦截引发Parameters对话框的消息。我们能够用它检测属性，甚至改变它们，也可以用来决定Parameters对话框是否显示出来。

下面是一个行为的例子。其中有 on runPropertyDialog和on getPropertyDescriptionList两个处理程序。当把该行为赋予角色时， on runPropertyDialog处理程序运行并用 alert命令发送一条消息。该处理程序包含一个行为的属性列表作为第二参数，并重新对它的值做了设置。然后，它使用 pass命令表明 runPropertyDialog消息应当继续传递，以便像平常一样打开

Parameters对话框。

```
property pFrame, pBoolean

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pFrame, [#comment: "Frame:", ¬
     #format: #integer, #default: 0]
  addProp list, #pBoolean, [#comment:"Boolean:", ¬
     #format: #boolean, #default: TRUE]
  return list
end

on runPropertyDialog me, list
  setProp list, #pFrame, the frame + 1
  alert "I will now set the pFrame property to the next frame."
  pass
  return list
end
```

该行为中的属性通过第二参数 list传给on runPropertyDialog处理程序。此变量包括像这样的一些东西：[#pFrame: 0, #pBoolean:1]。

提示 alert命令产生一个带有字符串消息和一个OK按钮的普通对话框。这是显示信息的简单快捷的方法。

因为这只是一种普通的属性列表， setProp命令可用于改变属性列表中的某一属性值。pBoolean属性仅用于使这个例子更丰富一些。setProp命令所能影响的唯一属性是pFrame。

在变量list拥有一个新值后，我们要记住用 return命令返回这个值，以使这个变化作用到行为中。pass命令可以在处理程序的任何位置放置，因为它只告诉 Director，当此处理程序运行完后，调用它的消息(在这个例子中是 runPropertyDialog消息)还要继续使用。结果是此消息引发正常的Parameters对话框的出现。

没有pass命令，Parameters对话框就永远不会出现。有时，我们正希望on runPropertyDialog处理程序设置所有的参数并忽略对 Parameters对话框中的设置。

14.4.4 工具提示

只有当我们打算在库面板中使用行为的时候，才使用 on getBehaviorTooltip函数。如果这样，将这个函数放置在那里，这样我们能够定义在叫作工具提示(Tool tip)的小黄色框中显示什么文本。

下面是一个例子。它与 on getBehaviorDescription处理程序的工作方式基本相同。我们只需要它返回一个字符串。对于工具提示来说，字符串应尽量短，以保证它们出现时，很好地显示在屏幕上而不至覆盖太多的地方。

```
on getBehaviorTooltip me
  return "Button Behavior"
end
```

14.5 完整的按钮行为

现在我们对如何创建一个复杂按钮行为有了足够的了解。这种行为需要执行如下的许多

任务：

■ 在用户按下鼠标时改变状态。我们应当确定“ 按下状态 ”的演员的选择方法，可选的演员有三种：演员表中的下一个演员，或者与原始演员同名但带有扩展词“ down ”的演员，或是一个特殊的演员名。

■ 在鼠标掠过它时改变状态。我们应当确定“ 掠过状态 ”的演员的选择方法，可选的演员有三种：演员表中最初的两个演员，与最初演员同名的却带有“ rollover ”扩展词的演员，或是一个专用演员名。

■ 在用户按下时播放声音。声音名从演员表当前的声音演员中选择。

■ 在鼠标掠过它时播放声音。声音名从演员表当前的声音演员中选择。

■ 在掠过它时改变光标。光标从内置光标中选择。

■ 在按钮被成功点击时，播放声音。

■ 在按钮被成功点击时，使用 go to或play命令让影片播放另一帧。同时，也能执行 play done命令。

■ 在按钮被成功点击时，调用指定的 Lingo命令。

像这样的行为是我们在大型影片中经常需要的。它考虑到可能使用的所有不同按钮，甚至可以帮助我们创建不是按钮的掠过角色。

14.5.1 创建参数

首先，创建on getPropertyDescriptionList处理程序。通过一个接一个的属性，构造这个处理程序，我们能够看到有多少属性要被使用，并给它们合适地命名。然后，可以返回到剧本的开头并添加属性的声明。

从创建属性列表开始：

```
on getPropertyDescriptionList me
  list = [:]
```

现在，想一下14.5节开始部分的列表的第一项：让某一角色显示“ 按下状态 ”。

除了这一项中列出的三个选择外，我们应该还有第四个选择：非按下状态。下面是添加这项属性的一行代码。属性的名字为 pDownState，我们将用一个弹出列表选择按下状态的类型。这些选择项要用#range属性列出。

```
addProp list, #pDownState, ¬
   [#comment: "Down State", #format: #string,¬
    #range: ["No Down State", "Member + 1",¬
          "Append 'down' ", "Name Down State"],¬
    #default: "No Down State"]
```

注意，缺省状态被设为“ No Down State ”。我们要为所有的属性设置缺省状态。

按下状态选项之一是为一个按下状态的演员命名。我们需要有一个以这个演员的名字作为一个参数的弹出菜单。

```
addProp list, #pDownMemberName,¬
   [#comment: "Down Member", #format: #bitmap, #default: ""]
```

正如我们对按下状态所做的一样，我们要为掠过状态提供相似的功能。但是，因为我们已经有一种方法让按下状态演员作为演员表中的下一个演员，那么掠过状态演员应该也可以是当前演员后的第二个。这样，如果我们愿意，就可以在演员表中将 normal(正常状态)、

down(按下状态)和rollover(掠过)位图排在一起。

```
addProp list, #pRolloverState, ¬
   [#comment: "Rollover State", #format: #string, ¬
    #range: ["No Rollover", "Member + 2", "Append ‘ rollover ’ ",¬
          "Name Rollover", "Cursor Change"],¬
    #default: "No Rollover"]
```

正像包含按下状态一样，一旦选中“ Name Rollover ”,我们就有一个列出位图的弹出菜单以供使用。

```
addProp list, #pRolloverMemberName, ¬
   [#comment: "Rollover Member", #format: #bitmap, #default: ""]
```

掠过还有另一个选项： Cursor Change(光标变换)。如果这个选项被选中，我们要知道哪个光标已被选为掠过光标。

```
addProp list, #pRolloverCursor, ¬
[#comment: "Rollover Cursor", #format: #cursor, #default: ""]
```

在按钮被点击或按钮被掠过时，还要提供声音的播放方法。要有一个复选框，以确定是否播放声音，然后还要有列出声音的弹出菜单。

```
addProp list, #pPlayDownSound, ¬
   [#comment: "Play Down Sound", #format: #boolean, #default: FALSE]

 addProp list, #pDownSound, ¬
   [#comment: "Down Sound", #format: #sound, #default: " "]
```

同样的属性也要在掠过动作中存在：

```
addProp list, #pPlayRolloverSound, ¬
   [#comment: "Play Rollover Sound", #format: #boolean, #default: FALSE]

 addProp list, #pRolloverSound, ¬
   [#comment: "Rollover Sound", #format: #sound, #default: " "]
```

现在,剩下的工作是要定义当按钮成功点击时，该做些什么事情。首选项是要有一些浏览功能。这个浏览功能可以是go to frame、play frame或是play done。同样，应该也可以选择没有浏览，并应将它设成缺省状态。

```
addProp list, #pActionNavigation, ¬
    [#comment: "Action Navigation", #format: #string, ¬
    #range: ["None", "go to frame", "play frame", "play done"],¬
    #default: "None"]
```

如果go to frame或play frame被选中，则要有一个包含这一帧的名字的属性。 play done不需要某一帧的名字。

```
addProp list, #pActionFrame,¬
   [#comment: "Action Frame", #format: #frame, #default: " "]
```

当按钮成功点击时，也应该有一个声音播放选择。这需要有两个以上的属性，与按下状态和掠过状态的声音属性类似。

```
addProp list, #pPlayActionSound, ¬
   [#comment: "Play Action Sound", #format: #boolean, #default: FALSE]

addProp list, #pActionSound, ¬
   [#comment: "Action Sound", #format: #sound, #default: " "]
```

最后的属性是在成功点击时发生的另一个动作。它是一个 Lingo命令或自定义处理程序的名字。这个动作要使用名为 do的专用命令，我们可以在用户按下按钮时，执行这个命令。它可以像beep(系统警告声)一样简单，也能够像各种自定义的影片处理程序的名字一样复杂。用在on getPropertyDesciptionList中时，它只要是一个字符串就可以了。

```
addProp list, #pActionLingo, ¬
  [#comment: "Action Lingo", #format: #string, #default: " "]
```

现在，on getPropertyDescriptionList完成了。只要以return list结束就可以了。那么，我们应该返回去并分析一下，行为的属性声明应该是什么呢？

在on getPropertyDescriptionList处理程序中使用的所有属性应该包含在其中，同时还要包括我们认为需要的其他属性。还应该有包含 normal(正常状态)、down(按下状态)和rollover(掠过状态)演员引用的属性。我们也要有一个称为 pPressed的属性，它在按钮保持按下时为真；而在所有其他时候为假。

```
property pNormalMember, pDownMember, pRolloverMember, pPressed
property pDownState, pDownMemberName
property pPlayDownSound, pDownSound
property pRolloverState, pRolloverMemberName, pRolloverCursor
property pPlayRolloverSound, pRolloverSound
property pActionNavigation, pActionFrame
property pPlayActionSound, pActionSound, pActionLingo
```

提示 我们可以将属性声明分成几行(像前面例子中显示的一样)，也可写成一长行。这两种写法都是全局声明的正确写法。

14.5.2 编写事件处理程序

既然所有创建行为的基本步骤都已经介绍过了，那么我们就可以开始实际编写事件处理程序了。从逻辑上说，应该从on beginSprite处理程序开始。

这个处理程序应该对属性进行设置，这些属性包含引用下面三个被使用的演员的变量：pNormalMember、pDownMember和pRolloverMember。它们设置成什么，取决于参数属性。例如，如果pDownState设置为“ Member+1 ”，我们要得到最初演员的演员编号，并将其加 1，以得到所要的演员。

下面是 on beginSprite处理程序。它首先获得角色当前用到的演员，并将它存为pNormalMember。然后，查看pDownState属性，以确定在pDownMember属性中放些什么。对于pRolloverState和pRolloverMember属性所做的也一样。

```
on beginSprite me
  pNormalMember = sprite(me.spriteNum).member

  case pDownState of
    "No Down State":
      pDownMember = member pNormalMember
    "Append 'Down' ":
      pDownMember = member(pNormalMember.name&&"Down")
    "Member + 1":
      pDownMember = member(pNormalMember.number + 1)
    "Name Down State":
      pDownMember = member pDownMemberName
```

```
  end case

  case pRolloverState of
    "No Rollover":
      pRolloverMember = pNormalMember
    "Cursor Change":
      pRolloverMember = pNormalMember
    "Append 'Rollover' ":
      pRolloverMember = member (pNormalMember.name&&"Rollover")
    "Member + 2":
      pRolloverMember = member (pNormalMember.number + 2)
    "Name Rollover":
      pRolloverMember = member pRolloverMemberName
  end case

  pPressed = FALSE
end
```

on beginSprite处理程序做的最后一项工作是将pPressed属性设成FALSE。注意，在这个行为中，并不需要像pDownState、pDownMemberName和pRolloverMemberName这样的属性，因为它们的用意是确定pDownMember和pRolloverMember是什么。

图14-3展示了该行为的Parameters对话框。我们可以看到on getPropertyDescriptionList中列出的每一个属性。

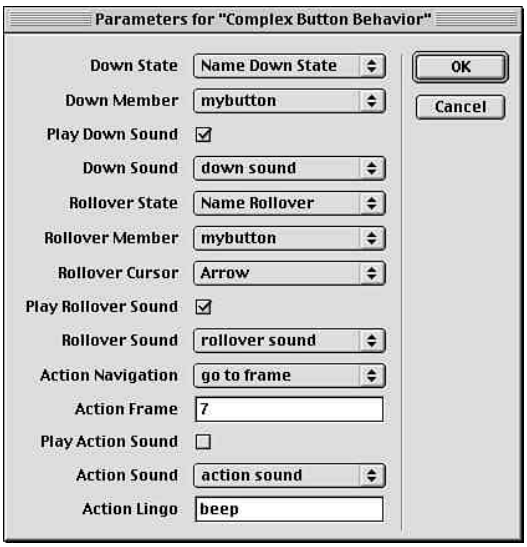

图14-3 复杂的按钮行为的Parameter对话框。它显示了按功能排列的属性，而不是按文章中讨论的顺序排列的

像这样的一个综合的行为需要我们使用大部分的基本事件处理程序： on mouseDown、on mouseUp、on mouseUpOutside、on mouseEnter和on mouseLeave。最后两个处理程序用于确定角色在什么时候被掠过。

在on mouseDown 处理程序中，要放置当角色被点击时所要执行的命令。尤其要设置按下状态的演员，根据需要播放一种声音，以及设置 pPressed属性。

```
on mouseDown me
```

```
  pPressed = TRUE
  sprite(me.spriteNum).member = pDownMember

  if pPlayDownSound then
    puppetSound pDownSound
  end if
end
```

与点击鼠标直接有关的还有 on mouseUp处理程序。正是在这里需要调用执行按钮的动作的自定义处理程序。还需要将 pPressed属性设置为FALSE。我们不能假设按钮的动作将导致离开当前帧，所以将角色的演员设回到非按下状况比较合适。因为当按钮被释放，接收到 on mouseUp消息时，光标一定仍位于角色之上，我们应该不将角色设置成正常状态，而应设置成掠过状态。

```
on mouseUp me
  pPressed = FALSE
  sprite(me.spriteNum).member = pRolloverMember
  doAction(me)
end
```

与on mouseUp相伴的是on mouseUpOutside。如果执行这个处理程序，表示起初当光标置于角色之上时，用户按下了按钮，但在光标离开角色之后，才释放鼠标。这是不执行某种动作的标准用户界面方式。用户显然不希望该动作发生，所以不应该像 on mouseUp处理程序中那样调用on doAction处理程序。另外，我们知道，光标并不在角色之上，所以应该将演员设置回正常状态。

```
on mouseUpOutside me
  pPressed = FALSE
  sprite(me.spriteNum).member = pNormalMember
end
```

两个掠过处理程序 —— on mouseEnter和on mouseLeave，实际上要做更多的事情。第一个处理器要完成将演员设置成掠过演员的工作。但是，如果用户已经点击并保持鼠标按钮呈按下状态，则按下状态演员将代替它而被使用。

另外，如果需要， on mouseEnter 处理程序还可以播放声音。如果 pRolloverState属性被设置成 “ Cursor Change(光标变化) ”，cursor命令将用来改变光标。

```
on mouseEnter me
  if pPressed then
    sprite(me.spriteNum).member = pDownMember
  else
    sprite(me.spriteNum).member = pRolloverMember
  end if

  if pPlayRolloverSound then
    puppetSound pRolloverSound
  end if

  if pRolloverState = "Cursor Change" then
    cursor(pRolloverCursor)
  end if
end
```

注释 cursor 命令可将光标变换成任一种专用的内置光标。在本例里，它从Parameters对话框的弹出菜单中获得设置，我们也可以使用数字设置光标，如280表示手形光标；4表示一个钟/表形光标；0表示光标复位。

on mouseLeave 处理程序必须取消on mouseEnter处理程序所做的工作。这是相当简单的，只需要将演员恢复成正常状态，并在必要时将光标复位就可以了。

```
on mouseLeave me
  sprite(me.spriteNum).member = pNormalMember

  if pRolloverState = "Cursor Change" then
    cursor(0)
  end if
end
```

上面讨论了所有需要的事件处理程序，仅留下自定义的 do Action处理程序。这个处理程序要查看pActionNavigation的三个需要被处理的可能值。在每一种情况下，该处理程序都执行一些必要的命令。因为某个浏览命令会使影片立刻离开角色所在的当前帧，因此在这件事情发生前，最好使用cursor(0)语句复位光标。

除了浏览以外， on doAction处理程序的任务是确定何时需要播放声音。它还用于确定是否有任何Lingo命令已作为pActionLingo的属性被输入，它用 do 命令运行那条命令或处理程序。

同时也应该注意到，这段程序不仅只是将 pActionFrame用作标志的名称，而是检测它是否能被当成一个数字处理。处理程序中使用 integer函数做这件工作。该函数试着将字符串转成一个数字。如果转换成功，它将这个数字当作要跳到的那一帧的编号，而不是一个标志字符串。如果需要，对play frame选项也可以采用同样的作法。

```
on doAction me
  if pActionNavigation= "go to frame" then
    cursor(0)
    if integer(pActionFrame) > 0 then
      go to frame integer(pActionFrame)
    else
      go to frame pActionFrame
    end if
  else if pActionNavigation = "play frame" then
    cursor(0)
    play frame pActionFrame
  else if pActionNavigation = "play done" then
    cursor(0)
    play done
  end if

  if pPlayActionSound then
    puppetSound pActionSound
  end if

  if pActionLingo <> " " then
    do pActionLingo
  end if
end
```

该行为现在是一个功能强大的多用途剧本，它可以在当前的影片和其他的影片中一次又一次地被使用。收集各种像这样的通用行为，并将它们存储在我们自己的行为库中以备后用是一个很好的想法。

14.6 创建动画行为

使用Lingo语言，动画不再必须要通过几个帧来发生。相反，它能够作为影片片断存在于某一帧内。这种可能性是无限，而后面的两个例子表明了其中一些可能性。

14.6.1 墙面回弹

改变角色的位置很容易实现。沿一条直线移动物体并不是件复杂的事情，因为剪辑室和推算已经能够让我们很容易地进行这项工作。但是，使用 Lingo，我们可以创建一种行为，拥有该行为的角色对环境会有所反应，如从墙面回弹。

有这样功能的行为实现起来比较简单。照例，首先创建 on getPropertyDescriptionList处理程序，其中只要三个属性就足够了：横、纵向运动速度和反弹物体的矩形。

```
property pMoveX, pMoveY, pLimit

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pMoveX, [#Comment: "Horizontal Movement",¬
    #format: #integer, #range: [#min:-10,#max:10], #default: 0]
  addProp list, #pMoveY, [#Comment: "Vertical Movement",¬
    #format: #integer, #range: [#min:-10,#max:10], #default: 0]
  addProp list, #pLimit, [#Comment: "Limit Rectangle",¬
    #format: #rect, #default: rect(0,0,640,480)]
  return list
end
```

对横向和纵向的运动速度的设置使用滑块十分合适。因为，我们大概不会对它们使用无限值。在这种情况下，属性被限制为每次增加或减少 10个像素。

注释 在Lingo和其他语言中，在变量名中使用X和Y是十分普遍的。X值一般用于表示横向位置和运动，而Y值一般用于表示纵向位置和运动。

pLimit属性应当为rect结构。使用格式类型#rect会把为这个属性输入的文本数值强制放在rect结构内。

要使该行为起作用，只需要一个事件处理程序。 on exitFrame处理程序一般用于类似的动画。我们可以获得角色当前的位置并为它添加一个点。 point和rect结构的变量都可以像其他变量一样被加上和减去。在Message窗口中试一下下面的例子：

```
p = point(100,150)
p = p + point(40,20)
put p
-- point(140, 170)
```

这个技术使得程序很简练。我们不必将角色的位置分成横向和纵向成份。下面就是这个处理程序：

```
on exitFrame me
  -- get the old location
```

```
  currentLoc = sprite(me.spriteNum).loc

  -- set the new location
  newLoc = currentLoc + point(pMoveX, pMoveY)

  -- set the sprite location
  sprite(me.spriteNum).loc = sprite(me.spriteNum).loc + ¬
    point(pMoveX, pMoveY)
end
```

我们也要查看pLimit属性，以确定角色是否已经撞击了矩形的一边，并将发生转向。一个rect结构，如pLimit属性，可以被分成四个属性：left、right、top和bottom。我们必须检测left和right，确定是否某边被撞击，并检测top和bottom，看它们是否被撞击。

如果的确发生了撞击，该方向的运动以及 pMoveX或pMoveY属性将反向，也就是正值变为负值，反之亦然。

限制角色的矩形可能很小，角色也可能从极限位置开始运动。为了保证在这两种情况下该处理程序都能正常运行，每一种情况都要单独处理。当角色撞击右墙，将获得 pMoveX属性的绝对值，然后将其转成负数形式。在角色撞击左墙时，获得它的绝对值并保持正数形式。大多数情况下，其结果是当墙被撞击时，该属性的符号发生改变。

```
on exitFrame me
  -- get the old location
  currentLoc = sprite(me.spriteNum).loc

  -- set the new location
  newLoc = currentLoc + point(pMoveX, pMoveY)

  -- check to see if it has hit a side of the limit
  if newLoc.locH > pLimit.right then
    pMoveX = -abs(pMoveX)
  else if newLoc.locH < pLimit.left then
    pMoveX = abs(pMoveX)
  end if

  -- check to see if it has hit top or bottom of the limit
  if newLoc.locV > pLimit.bottom then
    pMoveY = -abs(pMoveY)
  else if newLoc.locV < pLimit.top then
    pMoveY = abs(pMoveY)
  end if

  -- set the sprite location
  sprite(me.spriteNum).loc = sprite(me.spriteNum).loc + ¬
    point(pMoveX, pMoveY)
end
```

现在，我们已经有了一个行为，它使某一角色匀速运动，并在必要时在墙面处反弹。只要角色在舞台上存在，该动画就不会停止。

14.6.2 添加重力

现在的“回弹”行为看起来，好像作用于角色上的动作发生在理想宇宙的外围空间中。

在角色撞击屏幕边缘时，角色发生回弹，它不会丧失速度或由于重力而下坠。

为了创建更真实的影片，我们有时要将诸如能量损失和重力因素考虑进来。毕竟，在现实生活中，扔向墙壁的球将撞击墙面，落到地下并发生回弹。

添加重力因素十分容易，但一般要有两个属性。第一是角色所受的重力大小；第二是角色向下运动的速度。

当我们在现实环境中释放一个物体，它首先缓慢下落，然后在继续下落过程中加速，因为重力加快了物体向地面的下落速度。加速度是速度的变化，而非速度本身。

加速度是第一个属性：pGravity。另外，还需要pSpeedDown属性，以记录角色向下运动的速度。对于行为的能量损失特征来说，需要的只是pLoseEnergy属性，它可以是真或假。

为创建该行为，可以从刚才构造的“回弹”行为开始。我们要为property声明添加几个属性：

```
property pGravity, pSpeedDown, pMoveX, pMoveY, pLimit, pLoseEnergy
```

现在，我们要在on getPropertyDescriptionList处理程序中添加两个新参数：

```
addProp list, #pGravity, [#Comment: "Gravity",¬
  #format: #integer, #range: [#min:0,#max:3], #default: 0]
 addProp list, #pLoseEnergy, [#Comment: "Lose Energy",¬
  #format: #boolean, #default: FALSE]
```

pGravity属性被设置成1时，是最佳的状态，但是滑块调节最大可以达到3。0被用作缺省状态，它表示无重力影响的效果。

在角色开始时，要设置pSpeedDown属性，所以我们要创建on beginSprite处理程序。

```
on beginSprite me
  pSpeedDown = 0
end
```

然后，每一次on exitFrame处理程序运行，由于重力的作用，下落速度要增加。

```
pSpeedDown = pSpeedDown + pGravity
```

而且，当角色撞击地面后，根据牛顿定理，该属性必须反向。我们可以在on exitFrame处理程序的上面这一行后面添加一行语句，以使pMoveY属性反向。

```
pSpeedDown = -abs(pSpeedDown)-1
```

在得到的速度值里加“-1”，是为了修正一种矛盾。在角色撞击地面并且pSpeedDown变成负值后，该角色的位置直到下一次经过on exitFrame处理程序时才会再次发生变化。于是那时，该角色的位置在再次改变之前被添加了1。所以，如果pSpeedDown是22，角色撞击底边时，它变为-22。然后，它又一次经过此处理程序并被增加了1，则值变成-21。

结果是该角色先是每次移动22像素，然后每次则移动-21像素。由于上述原因，多了一个像素。而为pSpeedDown加-1，可以使二者相等。

要制作能量损失效果，首先要检测pEnergyLoss是否为真。然后，当角色撞击地面时，要对pSpeedDown添加1。但是，在角色的速度接近0时，要注意不要做这个操作，因为这种情况下，角色将渐渐被吸进地下。

```
if pSpeedDown < -1 then pSpeedDown = pSpeedDown + 1
```

下面介绍最后的行为。请注意处理顶边撞击的代码已经去掉。因为既然有重力影响，撞击顶边几乎是不可能的。另外，它会使处理重力的程序更加复杂。所以，该行为似乎只在两边是墙，下面是地面，而上面为天空的空间里行动。

```
property pGravity, pSpeedDown, pMoveX, pMoveY, pLimit, pLoseEnergy
```

```
on getPropertyDescriptionList me
  list = [:]
  addProp list, #pGravity, [#Comment: "Gravity",¬
    #format: #integer, #range: [#min:0,#max:3], #default: 0]
  addProp list, #pLoseEnergy, [#Comment: "Lose Energy",¬
    #format: #boolean, #default: FALSE]
  addProp list, #pMoveX, [#Comment: "Horizontal Movement",¬
    #format: #integer, #range: [#min:-10,#max:10], #default: 0]
  addProp list, #pMoveY, [#Comment: "Vertical Movement",¬
    #format: #integer, #range: [#min:-10,#max:10], #default: 0]
  addProp list, #pLimit, [#Comment: "Limit Rectange",¬
    #format: #rect, #default: rect(0,0,640,480)]
  return list
end

on beginSprite me
  pSpeedDown = 0
end

on exitFrame me
  -- Accelerate due to gravity
  pSpeedDown = pSpeedDown + pGravity

  -- get the old location
  currentLoc = sprite(me.spriteNum).loc

  -- set the new location
  newLoc = currentLoc + point(pMoveX, pMoveY+pSpeedDown)

  -- check to see if it has hit a side of the limit
  if newLoc.locH > pLimit.right then
    pMoveX = -abs(pMoveX)
  else if newLoc.locH < pLimit.left then
    pMoveX = abs(pMoveX)
  end if

  -- check to see if it has hit top or bottom of the limit
  if newLoc.locV > pLimit.bottom then
    pMoveY = -abs(pMoveY)
    pSpeedDown = -abs(pSpeedDown)-1
    if pLoseEnergy then
      if pSpeedDown < -1 then pSpeedDown = pSpeedDown + 1
    end if
  end if

  -- set the sprite location
  sprite (me.spriteNum).loc = newLoc
end
```

14.7 在行为间交换信息

行为并不是只能控制它们自己的角色；实际上它们也能向其他的角色和行为发送指令和信息。要做这项工作，要用到两个特殊的命令。

第一个是sendSprite，它随同另外的信息一起，向特定的角色发送消息。例如：

```
sendSprite(sprite 1, #myHandler, 5)
```

这行语句向角色1发送myHandler消息。如果该角色有on myHandler处理程序，那么它就会运行。另外，数字5作为me后面的第一参数传给此处理程序。这可能看起来像下面这样：

```
on myHandler me, num
  put "I got your message:"&&num
end
```

还有一个更有用的例子，它可以将两个角色联系在一起，以致当其中一个被鼠标拖动时，另一个跟着同步运动。第一个角色的行为看上去可能像下面这样：

```
property pPressed

on beginSprite me
  pPressed = FALSE
end

on mouseDown me
  pPressed = TRUE
end

on mouseUp me
  pPressed = FALSE
end

on mouseUpOutside me
  pPressed = FALSE
end

on exitFrame me
  if pPressed then
    -- calculate move amount
    moveAmount = the mouseLoc - sprite(me.spriteNum).loc

    -- move this sprite
    sprite(me.spriteNum).loc =  sprite(me.spriteNum).loc + moveAmount

    -- move another sprite
    sendSprite(sprite 2,#move,moveAmount)
  end if
end
```

这是个相当基础的拖动行为。在用户在其上点击时， pPressed变量被设置成TRUE。这使on exitFrame处理程序中的三行语句能够得以执行。这三行语句计算当前光标的位置和当前角色位置的数量差，并使该角色移动这段数量以便与光标匹配，然后将这个运动数值送给编号为2的角色。

第二个行为被赋予角色2：

```
-- get message from another sprite to move
on move me, moveAmount
  sprite(me.spriteNum).loc = sprite(me.spriteNum).loc + moveAmount
end
```

这个处理程序从第一个角色接收一个消息，并使用与消息包括在一起的数字，改变它的

位置。结果显示，在第一个角色被拖动时，两个角色一起移动。

行为间交换信息的另一个方式是使用 sendAllSprites命令。它实际上与 sendSprite相同，但不需要第一个参数。

```
sendAllSprites(#myHandler, 5)
```

我们可能猜得到，这个命令将消息和信息发送给当前帧中的所有角色。其中任何一个有 on myHandler处理程序的角色都接收这个消息并使用它。如果某一行为没有这个处理程序，则此消息被忽略。

14.8 创建行为的故障排除

■ 如果要创建行为，应当确认把剧本类型设置成“ Behavior ”。否则，我们甚至不能将该演员赋予某个角色。

■ 记住在getPropertyDescriptionList处理程序的末尾使用return语句。如果没有返回其中定义的属性列表， Director将无法生成Parameters对话框。

■ 当使用sprite(x)句法格式引用角色时，这个角色编号要完整地包括在这个圆括号中。代码sprite(x)+1是错误的，而应写成sprite(x+1)的形式。

■ 当为getPropertyDescription处理程序建立一个列表时，每一个表项都必须有 #comment, #format和#default属性。即使在逻辑上没有 #default属性的地方，也要为这个属性安排一个 0、VOID或空字符串这样的值。

14.9 你知道吗

■ 在某一个行为的参数描述中可以为值的范围设置 #min和#max属性，也可以包括一个#increment属性，以确定每点击一次箭头键，滑动条的值要改变多少。

■ 在Director6和更早的版本中，连续字符(¬)更有用一些，其为程序行的长度是有限的，必要时必须强制换行。当我们使用这个字符时，应确认不要在其后放一个额外的 Return(回车)，否则Director会认为我们使用的是两个独立的行。

■ 如果有重叠的角色，并且要确认掠过行为仅在掠过角色的可见部分时才起作用，在 on mouseEnter处理程序中设置演员之前，要查看以确信这个掠过状态的编号与该角色的编号相同。

■ 在为回弹行为确定墙、地板和天花板位置时，我们可能要相对于舞台尺寸对它们进行计算。使用(the stage).drawRect得到舞台尺寸的矩形。用(the stage).draRect.width和(the stage).drawRect.height得到右墙和地板的位置。