第九部分　完成一件作品

第33章　调　　试

如果在我们制作的Director影片里有多行Lingo，很可能需要调试它。调试即纠正程序错误。程序错误可能是由不正确的Lingo程序行导致的明显的错误信息，也可能是一些不明确的提示，如“ something just not working right ”。

注释　程序错误叫做“ 虫(bug) ”，这是由于早期的程序错误真的是一只虫：一只蛾子飞进了计算机，导致电路短路而引起了程序错误，因此这个名字就沿用下来。

33.1　编写正确的程序

在编程序时注重细节是大有裨益的。这就是说应当在程序中多加注释，为处理程序和变量选用有实际意义名称，并且把程序合理地分割成多个剧本演员。

合理编写的程序出现错误的机会要少些。如果其中的确有程序错误，经详细注释的程序能增加找到并纠正错误的机会。

33.1.1　为程序加注释

在恰当的地方加上恰当的注释是一种技巧。不能为每一行都加注释，因为那样会使程序窗口太拥挤，而且降低了程序的可读性；而不加注释又会导致将来调试或修改程序时的困难。

以双连字符“ -- ”开头的内容就是注释。注释可以自成一行，也可以跟在某一程序行的末尾。

在程序中可以添加三种注释。可以在处理程序或一段程序前写一段注释，可以在一行或几行程序前写注释行，可以在某个程序行的末尾写一些注释。

1. 段注释

在处理程序前面写几行注释是使用Lingo和其他编程语言时的一种技巧。就像下面的注释一样，可以解释处理程序的作用，写明需要什么参数，将返回什么类型的值，并指定何时及如何使用该处理程序。

```
-- This handler will add a number to the gScore global
-- and place the global in the text member. It will
-- also check to make sure that the score is not less
-- than 0, and make it 0 if it is. The number passed in
-- should be an integer.
-- INPUT: integer
-- OUTPUT: none
-- EFFECTS: gScore, member "score"

on changeScore n
```

```
  gScore = gScore + n
  if gScore < 0 then gScore = 0
  member("score").text = string(gScore)
end
```

段注释的主要优点在于它们与实际的程序是分开的。由于 Lingo程序就像英语一样，因此没有必要添加太多单行注释。

当其他编程员需要把我们的程序段用在他们的程序里时，段注释也很有帮助。他们可以读注释，而不是读程序段本身。在这种情况下，段注释里应当包含参数和返回值，以及受该处理程序影响的全局变量、演员和角色等。

2. 行注释

在重要的程序行之前放置单行注释是另一种常用的方法。注释的目的是阐明下面一行程序的作用和意义。

请看下面的例子：

```
on changeScore n

  -- add n to the score global
  gScore = gScore + n

  -- make sure score isn't less than 0
  if gScore < 0 then gScore = 0

  -- place score in text member on Stage
  member("score").text = string(gScore)
end
```

尽管在上面这个例子里对每行都加了注释，但典型的情况是只对个别行加注释。这样，注释将不会占据程序的太多空间。

注释可以有效地提醒我们处理程序都做了哪些工作。这对于调试和将来修改处理程序都有帮助。

3. 简短注释

简短注释是指与程序代码行同在一行的注释，它解释该行代码的作用。有时它提供与该行内的某个变量或函数相关的特定信息。

请看下面的例子：

```
on changeScore n
  gScore = gScore + n -- gScore is a global
  if gScore < 0 then gScore = 0 -- make sure it is less than 0
  member("score").text = string(gScore) -- show on screen
end
```

该处理程序的第一行的简短注释提供了关于这一行的某一部分特定信息。其余的注释与行注释的作用是一样的。

当我们认为不需要加上完整的注释时，就可以写一些简短注释，它们对于读程序的人来说是一些提示。而且同行注释一样，在我们返回头修改程序时，它们会很有帮助。

使用注释的最佳方案是结合使用上述三种注释。可以在某个行为前或影片剧本里的每个处理器前写一个段注释，然后为重要的和巧妙的程序行加一些行注释和简短注释。

参见第12章“学习Lingo”里的12.6节“编写Lingo程序”，可以获得更多有关编写程序的

信息。

33.1.2 使用描述性的名称

第12章曾经讲到应当为处理程序和变量取一些描述性的名称。我不知道如何强调才能让读者理解这一点的重要性。以下介绍一些技巧，可以帮助我们选择一些更有描述性的名称。

1. 由多个单词组成的名称

在Lingo编程里有一种习惯，就是把多个单词连在一起使用，作为处理程序的名称。除第一个单词外，每个单词都是首字大写。

请看下面的例子：

```
on addNumberToScore numberToAdd
  gTotalScore = gTotalScore + 1
end
```

处理程序以及两个变量都使用了由多个单词连在一起构成的描述性的名称。可以看出，这样就不需要外加任何注释了，因为这个处理程序的作用以及原理都很明显。

像这样使用较长的名称似乎有一些缺点，比如占用了程序窗口里更多的空间，需要更多的录入时间等。但是这却免去了添加行注释或简短注释的必要性。而如果添加注释，将需要录入更多的文字。

2. 第一个字母的约定

在这本书里，我们始终都能看到一种命名约定，即所有全局变量(global)的第一个字母为g，所有属性变量(property)的第一个字母为p。这种约定使我们能够很容易地记住这个变量产生的位置以及使用的范围。

对其他变量的第一个字母可以其他约定。有时， l可以做为局部变量(local)的第一个字母。但是，大多数创作者都习惯于把局部变量写作唯一没有这种前缀的形式。

在父代剧本里，我们还会看到字母 i用在属性变量的前面。 i代表实例(instance)。

也可以在处理程序前面使用这种前缀。过去，在父代剧本里，这种方法很常用，如 m用来表示处理程序。在行为剧本里，也可以把字母 m用在处理程序的前面。

如果只有你自己读你的程序，或者在你们小组内的编程人员能达成一致，可以规定你们自己的前缀方案。 b可以用于行为(behavior)处理程序， m可以用于影片(movie)处理程序， p可以用于父代剧本(parent script)处理程序。对这种自定义前缀应当在说明书里解释一下，以便于其他程序员了解这些约定。

3. 使用描述性的演员名称

如果能够为变量选择描述性的名称，那么在为演员命名时也应当使用这种技巧。二者的不同之处在于演员的名称里可以加空格。

我们很容易为演员取没有具体意义的名称，毕竟放置在剪辑室里的演员是用它的角色的编号指代的，而无需在 Lingo用它的名称指代。但是为演员取恰当的名称仍旧能给我们带来方便。现在就开始为演员表内的几十个演员取名，你就会理解到其中的原因。

演员名称应当形象、唯一。十几个文本演员都叫做 Text只会把事情弄得更乱。演员的名称应当表现出何时和怎样使用该演员。例如，一个文本演员用来为游戏记分，另一个文本演员用在Game Over帧里，它们的名称可以分别为 Score Text in Game和Score Text for Game Over。

此外，在演员表里按用途整理演员的顺序也很有好处。可以按演员的类型或按如何以及

何时使用它们进行整理。

参见第9章“ Director环境 ”里的9.2节“ 参数设置 ”，可以获得更多有关演员表窗口参数设置的信息。

4. 使用变量常数

另一个增加程序的可用性和可读性的方法是用变量代表常用的数字和对象。例如，如果由于某种共同的原因， 2这个数字在程序里被使用了多次，我们也许需要把一个变量的值设为2，然后就使用这个变量。请看下面的例子：

```
on moveFromKeyPress charPressed, position
  case charPressed of
    "i": position.locV = position.locV - 2
    "m": position.locV = position.locV + 2
    "j": position.locH = position.locH - 2
    "l": position.locH = position.locH + 2
  end case
  return position
end
```

这是某程序里的一个处理程序，用户按某个键，就可以移动角色在舞台上的位置。位置每次变化的量是2。但是，如果想要把这个值变成 3，将需要在程序里修改四处。而使用下面的程序后，只需在程序里修改一处：

```
on moveFromKeyPress charPressed, position
  diff = 2
  case charPressed of
    "i": position.locV = position.locV - diff
    "m": position.locV = position.locV + diff
    "j": position.locH = position.locH - diff
    "l": position.locH = position.locH + diff
  end case
  return position
end
```

再举一个例子。假设需要多次引用某个角色，可以把这个角色对象放在一个变量里，然后使用那个变量。例如，在程序里可以这样写：

```
thisSprite = sprite(34)
```

这样，在该处理程序里的任何地方，都可以用 thisSprite来代表这个角色，而必不再用sprite(34)。一旦需要为它改用角色33，只需要在程序里修改一处。

对于全局变量也可以使用这种方法。可以把一个全局变量设为一个常数、字符串或对象，然后在整个程序里引用这个全局变量。如果将来需要修改那个常数，只需要在程序里修改一处。当然，这种方法的缺点是在使用该全局变量的每个子程序里都需要用 global命令声明一次。

参见第12章里的12.6节“ 编写Lingo程序 ”，可以获得更多有关编写程序的信息。

33.1.3 编写无错误的程序

你说这是不可能的？也许不可能完全无错误，但却可能十分接近，至少在这些错误影响用户之前就可以改正这些错误。

通常会发生这样的情况：我们在编写程序时，忽然意识到从前犯了某个错误，于是通知

用户能采取措施就采取措施，否则就停用那个程序，以免造成危害。

1. 无错误FileIO程序

如果想要用编程的方法创建一个文件，并向其中加入一些文本，只需要用几个有关 FileIO Xtra的简短命令。请看下面的例子：

```
on writeFile name, text
  fileObj = new(Xtra "FileIO")
  fileName = displaySave(fileObj, "Save"&&name, name& "txt")
  createFile(fileObj,fileName)
  openFile(fileObj,fileName,2)
  writeString(fileObj,text)
  closeFile(fileObj)
  fileObj = 0
end
```

在大多数情况下这段程序是正确的。但如果出现意外情况怎么办？例如，当 Save对话框出现时，如果用户按了 Cancel键会出现什么情况？在前面的处理程序里，程序直接前进到createFile命令，并产生了一个无效的文件名。其结果是这个文件从未被建立，什么事情都没有发生。但用户并不知道这一切。

最好检测一下Cancel键是否被按下了。如果是， filename变量内将含有一个空字符串。我们可以进行检测，并报告给用户，然后在执行其他命令之前退出该处理程序。

但如果由于某种原因影片不能创建这个文件怎么办？当用户选择了一个现已存在的文件名时就会发生这种情况。程序也应当对此进行检测。

FileIO有两个函数使我们能够检测错误。第一个是 status函数，它可以返回一个数字。如果这个数字是0，表示一切正常；否则就表示有错误。然后，可以把这个错误代码送给第二个函数，即error函数，以便显示一句错误信息。

下面的处理程序是在刚才的处理程序的基础上添加了复杂的错误检测部分。创建文件过程的每一步都被检测，看看有没有错误，并及时处理错误。

```
on writeFile name, text
  fileObj = new(Xtra "FileIO")

  -- check to make sure object was created
  if not objectP(fileObj) then
    alert("FileIO failed to initialize")
    return FALSE
  end if

  fileName = displaySave(fileObj, "Save"&&name, name&".txt")

  -- check to see if a filename was returned
  if filename = " " then
    alert "File not created. "
    return FALSE
  end if

  createFile(fileObj,fileName)

  -- check to see if file was created ok
  errorNum = status(fileObj)
  if errorNum <> 0 then
```

```
  alert "Error: "&&error(fileObj,errorNum)
  return FALSE
end if

openFile(fileObj,fileName,2)

-- check to see if file was opened ok
errorNum = status(fileObj)
if errorNum <> 0 then
  alert "Error: "&&error(fileObj,errorNum)
  return FALSE
end if

writeString(fileObj,text)

-- check to see if file was written to ok
if errorNum <> 0 then
  alert "Error: "&&error(fileObj,errorNum)
  return FALSE
end if

closeFile(fileObj)

-- check to see if file was closed ok
if errorNum <> 0 then
  alert "Error: "&&error(fileObj,errorNum)
  return FALSE
end if

fileObj = 0
return TRUE
end
```

可以注意到，每次发现错误都从处理程序返回一个 FALSE，不过，如果这个处理程序完成了它的任务，则会返回一个TRUE。这些结果不必显示出来，但这一点是非常有用的。调用这个on writeFile处理程序的任何处理程序就可以把它作为一个函数调用，并得到一个答案，以了解是否真正保存了文件。

大约可以返回18种错误信息。表33-1列出了能够由FileIO Xtra返回的全部错误信息。如果使用了error函数后，返回的错误代码未列在该表里，则返回“ Unknown Error ”。

参见第16章“ 控制文本 ”里的16.10节“ 使用文本文件和FileIO Xtra ”，可以获得更多有关读写文件的信息。

2. 无程序错误的Shockwave程序代码

尽管FileIO有status和error函数，其他类别的 Lingo也有处理错误的函数。 Shockwave Lingo命令有netError函数。这个函数可以判断在 netDone函数返回TRUE后发生了什么情况。

下面的on exitFrame程序替换了第 22章“ 使用Shockwave和使用因特网 ”里的 22.2节“ 从因特网上获取文本 ”里的程序。同从前一样，它检查 netDone，然后进一步检查netError。如果操作成功，该函数返回 OK。如果不成功，则返回一个数字。该数字与表 33-2里的数字相对应。

表33-1 FileIO Xtra使用的错误代码

代 码	信 息
124	文件作为只写文件被打开
-123	文件作为只读文件被打开
-122	文件已经存在
-121	有一个已打开的文件
-120	没有找到目录
-65	驱动器里没有磁盘
-56	没有这个驱动器
-43	没有找到文件
-42	已打开太多文件
-38	文件没有打开
-37	文件名无效
-36	I/O错误
-35	未找到卷
-34	卷已满
-33	文件目录已满
0	OK
1	内存寻址错误

表33-2 Shockwave Lingo netError代码

代 码	含 义
0	一切正常
4	错误的MOA类。所需要的网络或非网络Xtra没有正确地安装或根本没有安装
5	错误的MOA界面，见4
6	错误的网址或错误的MOA类。所需要的网络或非网络Xtra没有正确地安装或根本没有安装
20	内部错误。如果Netscape浏览器检测到一个网络错误或内部错误，由该浏览器的 netError()返回
4146	不能与远程主机建立连接
4149	由服务器提供的数据属性的格式未知
4150	连接意外地提前关闭
4154	由于时间已到未能完成操作
4155	没有足够的内存来完成该事务
4156	应答请求的协议表明应答里有一个错误
4157	事务未被鉴别
4159	无效的网址
4164	不能创建套接字
4165	找不到所要求的对象(网址有可能错误)
4166	类代理失败
4167	传送被客户机故意中断
4242	下载被netAbort(url)停止
4836	下载由于未知的原因而停止，很可能是由于网络错误或下载被放弃

```
on exitFrame
  global gNetID
  if netDone(gNetID) then
   err = netError(gNetID)
    if err = "OK" then
      text = netTextResult(gNetID)
      put text
      go to the frame + 1
```

```
    exit
   else
    if err = 4165 then
     alert "Could not find that URL. "
    else
    -- handle error somehow
    end if
   end if
  end if
  go to the frame
end
```

这个例子查找错误代码4165。当找不到文件时，通常返回这个错误。我们也许想要用处理其他错误的代码替换那个注释行。

参见第22章里的22.2节“从因特网上获取文本”，可以获得更多有关从因特网上获取文本的信息。

3. Shockwave声音错误的Lingo代码

Shockwave声音演员也能返回错误代码。它们通过使用 getError()函数返回这些错误。它所使用的参数是SWA演员的名称。

getError可以返回一个数字，而getErrorString可以返回一个字符串以便显示给用户。表33-3列出了这些错误。

表33-3 Shockwave声音错误的代码

getError代码	getErrorString信息	含　义
0	OK	没有错误
1	内存	没有足够内存来调用声音
2	网络	发生了一个网络错误
3	播放	不能播放声音
99	其他	

参见第17章“控制声音”里的 17.3节“使用 Shockwave音频”，可以获得更多有关Shockwave声音的信息。

4. Flash错误的Lingo

也可以对从因特网上流式传输的Shockwave Flash演员使用getError。这里不返回数字，而是返回符号。表33-4列出了全部内容。

表33-4 用getError得到的Flash演员错误

返回的符号	含　义
FALSE或0	没有错误
#memory	没有足够的内存来调用该演员
#fileNotFound	找不到文件
#network	发生了一个网络错误
#fileFormat	文件不是Flash影片
#other	返回了其他错误

参见第20章“控制矢量图形”里的20.1节“使用Flash演员的Lingo”，可以获得更多有关使用Flash演员的信息。

5. 其他错误的Lingo

除了那些与特定类型的素材或Lingo类别相关的函数外，还有一些函数可以用来检测变量，以确定它们属于哪种类型。例如，如果我们把一个数字当成列表对待，这就将产生一个错误信息。实际上，我们可以在试图使用某个变量之前先查看它是不是一个列表。

这些函数大都以字母 P结尾。它代表预先(predicate)。下面列出这些函数以及它们所检测的内容：

- listP() —— 该变量是列表吗？
- integerP() —— 该变量是整数吗？
- floatP() —— 该变量是浮点数吗？
- stringP() —— 该变量是字符串吗？
- symbolP() —— 该变量是符号吗？
- objectP() —— 该变量是一个对象(如演员、角色、列表或 Xtra事例)吗？

以下是消息窗口里的一些例子：

```
x = 1
put integerP(x)
-- 1
put floatP(x)
-- 0
put listP(x)
-- 0
put stringP(x)
-- 0
put objectP(x)
-- 0
x = [1,2,3]
put integerP(x)
-- 0
put listP(x)
-- 1
put objectP(x)
-- 1
x = "abc"
put integerP(x)
-- 0
put stringP(x)
-- 1
put objectP(x)
-- 0
x = member(1)
put listP(x)
-- 0
put objectP(x)
-- 1
```

listP函数是很有用的，但只使用它还不够完整。列表是属性列表还是线性列表？另一个叫做ilk的函数将告诉我们列表的类型。可能的值是#color、#date、#list、#proplist、#point和#rect。

以下是消息窗口里的一些例子：

```
x = [1,2,3]
put ilk(x)
-- #list
```

```
put x.ilk
-- #list
x = [#a: 1, #b: 2, #c: 3]
put x.ilk
-- #propList
x = point(100,150)
put x.ilk
-- #point
x = rect(0,0,10,10)
put x.ilk
-- #rect
x = the systemDate
put x
-- date( 1999, 1, 5 )
put x.ilk
-- #date
x = rgb(0,0,0)
put x.ilk
-- #color
```

还可以用ilk来比较变量和类型。变量是第一个参数，类型是第二个参数。在这种情况下，如果列表属于多种类型，则返回 TRUE，因为有些列表类型可以以多种方式被使用。例如，某个颜色可以是列表、线性列表和颜色列表。除了这些由 ilk单参数函数所使用的符号外，还可以使用#linearlist以测试那个变量是不是这种特定类型的列表。

```
x = [1,2,3]
put ilk(x,#list)
-- 1
put ilk(x,#linearlist)
-- 1
put ilk(x,#proplist)
-- 0
x = [#a: 1, #b: 2, #c: 3]
put ilk(x,#list)
-- 1
put ilk(x,#linearlist)
-- 0
put ilk(x,#proplist)
-- 1
x = rgb(0,0,0)
put ilk(x,#list)
-- 0
put ilk(x,#linearlist)
-- 0
put ilk(x,#proplist)
-- 0
put ilk(x,#color)
-- 1
```

参见第13章“重要的Lingo句法”里的13.10节“使用列表变量”，可以获得更多有关列表的信息。

33.1.4 处理错误的Lingo

尽管坚持使用了这些检测错误的方法，我们的最终产品里还可能存在着程序错误。不过，

还有一种方法可以把这些让人烦恼的问题最小化。 Director允许我们在发生错误的时候拦截它们。

我们可以利用这一武器发送自定义的错误信息，而不是 Director的标准信息，甚至可以利用这一武器不发送任何信息。在很多情况下，在发生了不严重的错误后，程序可以带着这些小错误继续运行。

这个方法有一些复杂。首先要设置一个叫做 the alertHook的系统属性。把它设在一个父代剧本里。设置它的理想地点是 on prepareMovie或on startMovie处理程序内。在这种情况下，the alertHook被设为一个名为Error Handling的父代剧本演员。

```
on prepareMovie
  the alertHook = script("Error Handling")
end
```

然后需要创建这个父代剧本本身。它要包含一个 on alertHook处理程序。请看下面的例子：

```
on alertHook me, error, message
  alert "Error: "&&error&RETURN&message
  return 1
end
```

on alertHook处理程序应当有第一个参数 me，随后，它还接受另外两个参数： error和message。这两个实际上都是消息。它们对应于每个 Director Lingo错误的两部分信息。第一个消息通常是Script runtime error等非常概括性的消息，第二个消息则包含关于错误的特定信息，如Handler not defined或Cannot divide by zero。

我们可以利用这两个字符串来决定接下去要做什么。例如，我们想要以某种特定方法处理Index out of range错误，就可以寻找该错误，并用某种方法处理它，然后用其他方法处理其他错误。

on alertHook所需要做的最后一件事情是返回一个值。如果返回的值是 0，Director则立即显示它想要显示的错误信息；如果返回的值是 1，只要不是致命的错误，就忽略它。

如果打算彻底忽略错误信息，至少应当检查 the runMode是否为author，并通过返回0来处理它。否则，在我们创建或运行影片时，会得到错误信息，却又全然不知。

如果想要关闭the alertHook对错误的处理，只要把the alertHook设为0就可以了。

33.2 使用Lingo的调试工具

当然，发现程序错误后的最好办法是改正它。实际上，改正程序错误是编程工作的重要组成部分。无论程序员多么有经验，任何 Lingo几乎都需要调试。专业的 Lingo编程员编程量大，程序复杂，因此程序错误出现的机会多；而初学编程的读者虽然编程量很少，但也容易出错误。

程序错误通常是以错误信息的形式出现的。图33-1是一个典型的错误信息对话框。其中的重点是Debug和Script按钮。这两个按钮将把我们带进两个功能最强大调试工具： Debugger和

图33-1 一个典型的Director错误信息对话框，其中显示了错误类型、错误信息以及Script、Debug和Cancel按钮

Script窗口。

除了Debugger和Script窗口外，还有一个Watcher窗口，它可以显示影片里的变量在变化过程中的值。消息窗口也是一个有价值的调试工具。

33.2.1 使用Debugger

有两种情况将要用到Debugger。第一种是在得到错误信息后，按下其对话框里的 Debug按钮时，会调出Debugger对话框。它与图33-2相似。

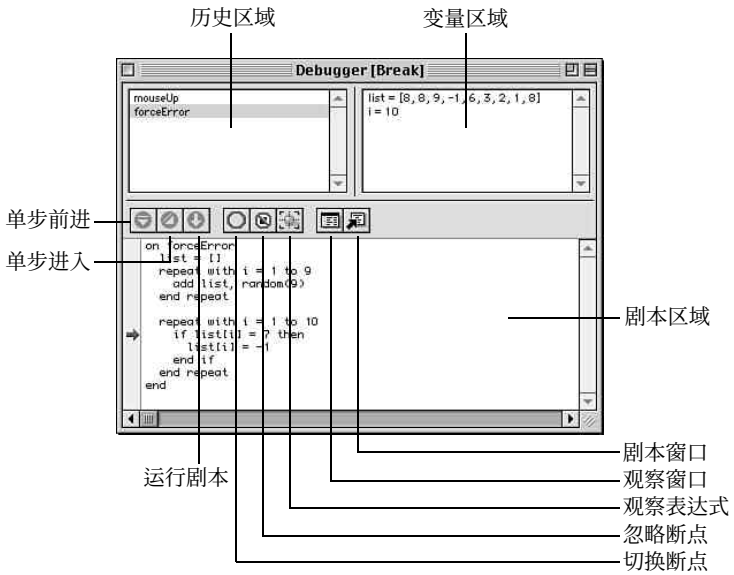

图33-2 Debugger窗口

Debugger窗口内有三个矩形区域和一组按钮可供我们检查影片当前的状态或单步执行程序。在图33-2的Debugger窗口里，我们可以看到被调用的 on forceError处理程序的程序代码。执行这些代码时发生了错误，错误信息就是图 33-1所显示的信息。

Debugger窗口的左上方的区域显示着处理程序的清单。该清单里的最后一个处理程序是当前正在执行的，在这里是on forceError。它上面的内容是导致当前局面的处理程序。在本例中，on mouseUp处理程序引发了该动作，并调用了 on forceError。

该窗口的右上方的区域显示着当前处理程序所使用的局部变量和全局变量的清单。紧接着是变量的值。

Debugger窗口内最大的区域是程序区域。它显示当前处理程序的程序，还有一个绿箭头指示着刚刚执行完的语句。

在本例中，我们可以看到 Debugger停在了if list[i]=7 then这一行。其错误是 Index out of range，因此最大的嫌疑是 list里的i，它的值大于 list里面的元素的个数。的确是这样， i是10，但list里只有9个元素。

使用Debugger的另一种方法是设置断点。在图 33-2里可以看到Toggle Breakpoints(切换断点)按钮。这个按钮也出现在 Script窗口里。在Script窗口里，还可以通过点击Lingo代码的任何一行左边的灰色区域来设置断点。

设置了断点后，当执行到带有断点的那一行时，影片停止，而且屏幕上出现 Debugger窗口。这与发生错误时出现的Debugger窗口是不同的，因为可以继续逐行地运行代码。

图33-2里的单步前进、单步进入和运行剧本三个按钮使我们能够继续运行程序。第一个按钮表示运行到下一行。 Lingo程序可以正常地执行，改变舞台上的变量和对象。用这个按钮可以观察到程序缓慢地执行。

如果在单步运行程序时遇到一行，它的内容是调用另一个处理程序，只需用一步就可以完成这个处理程序的运行。但是，如果使用的是单步进入按钮，我们就可以逐行地从那个处理程序里走过。

最后一个按钮运行剧本能使程序以全速再次运行。 Debugger不再跟踪它。不过，如果遇到另一个断点，它就会停下来。

这就是Debugger窗口内存在着Toggle Breakpoint按钮的原因，即我们可以在前方某处设置断点，再按运行剧本按钮，使程序前进到那一点时再停止。

33.2.2 使用Watcher窗口

除Debugger窗口外，我们还可以使用Watcher窗口查看表达式和变量的值。图 33-3是Watcher窗口。

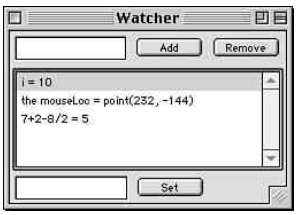

图33-3 Watcher窗口使我们能够在影片运行时跟踪表达式的变化

Watcher窗口的主要区域内列着我们想要查看的表达式。在它上面的区域里，我们可以输入新的表达式，并把它们添加到清单里。在下面，我们可以输入一个数字、字符串或其他值，然后在清单里选择一个变量，把它设为那个值。

实际上我们可以输入任何能得到一个值的 Lingo程序代码，并观察它在程序运行过程中的变化。这意味着我们可以使用加和减等运算，以及 mouseLoc等影片属性。

Watcher窗口的主要作用是帮助我们跟踪有可能出现问题的变量。在影片运行时我们可以监视它们，看看是在哪里出问题的。

Watcher窗口使用起来很方便，与使用 put语句并在消息窗口里测试变量的方法相比，它更简单些。

33.2.3 使用Script窗口

在调试时不能忽略Script这个重要的窗口。单单是查看程序，就能发现许多程序错误。

实际上，每当我发现一个新程序错误后，最先查看的就是这个窗口。当屏幕上出现如图33-1所示的错误信息对话框时，我就按 Script按钮，并查看程序。在大多数情况下，立刻就能发现错误并立即改正，而不必使用 Debug窗口或使用其他什么技巧。

有时，该窗口内会出现一个问号，试图指出错误的确切位置。不过，由于 Director并不一定了解程序的真正意图，它的猜测不一定总是正确的。

在Script窗口里也可以为Debugger设置断点，当然，也可以用来输入有用的注释，以帮助我们更快、更容易地发现隐蔽的错误。

33.2.4 使用消息窗口

当我们想要编写无错误的程序以及调试有错误的程序时，可以把消息窗口用于多种用途。

首先，可以用消息窗口测试简短的、我们不太有把握的程序。它就相当于其他编程环境里的命令行解释器。例如，我们不必去猜测某个文本演员的 type是不是#text，而是可以去检测它。

```
put member("score").type
-- #text
```

如果忘记了systemDate里是什么内容，只要测试就可以了：

```
put the systemDate
-- date( 1999, 1, 5 )
```

这样就可以十分有信心地编程序了。

此外，可以用消息窗口来测试单个的影片处理程序。只要输入它们的名称和某些参数，再按回车键就可以了。如果该处理程序是一个函数，可以在它前面写一个 put，就可以把结果放在消息窗口的下一行里了。

```
put addToNumbers(4,5)
-- 9
```

我们还可以在消息窗口里测试全局变量。因此如果想要知道全局变量 gScore现在的值是什么，只要测试一下就可以了：

```
put gScore
-- 405
```

我们甚至可以在消息窗口里设置全局变量：

```
gScore = 10000000
```

使用消息窗口的另一种方法是在程序里放置 put命令。这样可以把变量的内容送到消息窗口内。这与使用Watcher窗口很相似，但还可以记录这些值。

注释 在影片完成之前，一定要把用于调试的put语句变为注释或删除。向Message窗口输送大量信息将显著地降低影片的速度。

33.2.5 使用跟踪

消息窗口也是一种叫做跟踪 (Trace)的调试方法的主要工具。使用跟踪方法时， Director在每执行完一行Lingo程序和每个变量发生变化时，都向消息窗口发送信息。图 33-4是一次跟踪过程的实例。

注释 清注意，消息窗口里的第一行实际是我们输入进去的。在按回车键后，“ 跟踪 ”就开始了。Director错误地认为forceError消息来自演员23128。不过，这并不影响从下一行开始其余的跟踪。

要打开跟踪功能，只要点击消息窗口里的 Trace按钮就可以了。它是图 33-4所示的五个按钮的中间的那一个。

此外，还可以通过把 the trace设置为TRUE或FALSE来打开或关闭跟踪功能。可以在消息窗口里做这件事，也可以把它插入处理程序里。用这两种方法的任何一种都可以开始在某个

敏感程序行前开始跟踪，并在它的后面暂停。

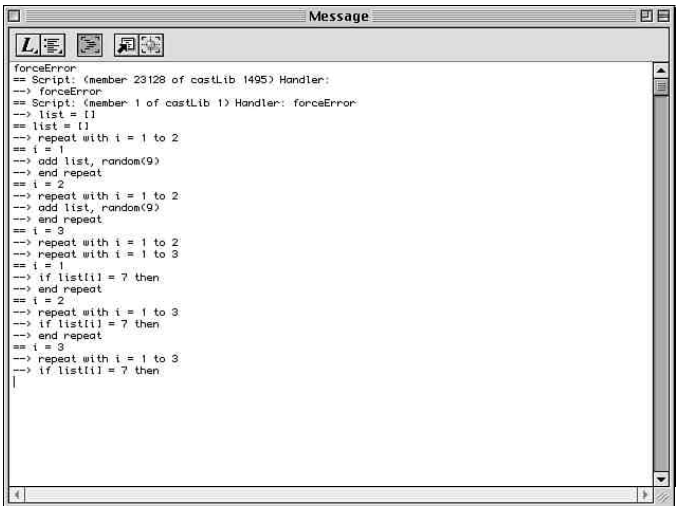

图33-4 消息窗口里显示了跟踪的信息

33.2.6 使用其他调试方法

调试程序在Director里的效果很好，但如果当影片以放映机或 Shockwave的方式运行时出现程序错误怎么办呢？

在这种情况下，没有Debugger可以使用。不过我们仍旧可以让特定位置的程序输出信息，让我们了解变量的设置，或了解影片现在运行的是哪部分程序。

对于放映机来说，alert命令可以很方便地用于这个目的。我们可以把 alert用在程序的敏感部位，以分析程序运行的情况。例如，程序正确地执行了 getNetText等系列命令，我们想查看Shockwave影片所获取的文本是否正是我们想要的。除了得到文本和继续执行外，我们也许还想试一试下面的方法：

```
text = netTextResult(gNetID)
alert text
```

现在，我们可以看到文本，并判断它是否正是我们想要的。当然，在进行完这个测试后，可以删除alert命令。

Shockwave还允许我们使用netStatus命令。该命令可以把一条信息放在 Netscape Navigator浏览器底部的信息区域。

我们还可以用netStatus向用户显示有用的信息，如光标所在的按钮的潜在目的或动作。这与HTML和JavaScript对它的使用是相似的。

33.3 测试程序

今天，很多重要的软件在发布时仍带有程序错误，这不能说不是一件憾事。这样的事情也许本可以避免。是否能够彻底地测试软件呢？由于某些项目的设置不同，就会导致有上千种不同的计算机设置：中央处理器类型、中央处理器速度、操作系统的类别、操作系统的版本、显示卡生产厂、显示卡驱动程序的版本、显示器设置、硬盘速度、 CD-ROM速度、内存

量、系统设置、系统功能扩展等等。

在每一台计算机上都进行测试是不可能的。不过，可以在各种不同的计算机上测试，发现大部分问题。

33.3.1 及早测试并经常测试

任何一位计算机学教授或软件项目经理都会告诉我们：及早测试并经常测试。应当记住这句话。

一旦有程序值得测试，就要开始测试。如果是在编写处理程序，在可能的时候要单独测试每个处理程序。例如，我们编写了一个处理程序，它的功能是计算两点间的距离，那么为什么不用消息窗口试着计算一下呢？发现错误了吗？结果与预想的一样吗？

尽管这种方法看上去很繁琐，但想想如果不测试会发生什么情况。一个大型的 Lingo程序可能包含几十个影片处理程序、很多行为剧本以及很多已命名的(或错误地命名的)演员。如果我们一口气把所有程序编完，从未进行过测试，那么当程序编写完成后，很少有机会一次成功。相反，如果时常花一分钟的时间在各个环节测试一下，那么当写完最后一句程序，软件基本上就可以运行了。

33.3.2 内部测试

内部测试有时也叫做 alpha测试。当程序编写完成时，我们自己可以先对它进行测试，这算是内部测试的第一步。大多数程序错误都能被查出来。

下一步是让我们的公司或机构内的其他人来测试。如果在学校里，可以让实验室的同伴或同学帮助测试，也可以请一些朋友来测试。

与下一阶段的测试相比，我们对本阶段的测试有着更多的控制权。我们可以同使用这个软件的人面对面地交谈，甚至请他们把他们所发现的错误演示给我们看。

当发生错误后，我们也许想要使用 alertHook Lingo命令给出详细的报告。这使我们能够更容易地找到错误，又由于我们可以在产品发布之前删除这些代码，因此它们对最终产品不会有影响。

33.3.3 beta测试

当我们自己以及我们的同事都做完测试后，就可以开始 beta测试了。这意味着要把软件交给我们的公司或机构以外的人员使用。大多数时候我们甚至不认识这些人。

有些创作者在私下进行beta测试，仅有选定的几个人参加测试。这些人可以从公司的忠实顾客里寻找，也可以发广告寻找。

有些公司则选择公开测试的方法，任何人都可以下载他们的软件进行测试。但这会导致有很多人下载，却只有很少人能够返回关于错误的报告。不过这种测试的覆盖面的确很广。

尽管我认为应当在软件制作全部完成并进行某些内部测试后再进行 beta测试，但有很多公司在开发软件期间就进行 Beta测试。这使他们在开发阶段就能测试一些新功能并得到实际用户的反馈信息。

不过，对于小型作品来说，就没有必要进行 beta测试了。此外，如果你制作的是Shockwave短程序，应当想到，在程序发布后，想要更新它是多么的容易，因为它只存在于一

个地方：你的网站。如果发现了错误，即使是严重的错误，只要改正以后再把新的版本放到网站上就可以了。

33.4 你知道吗

■ 试着在要放置断点的地方放置 Lingo命令nothing。nothing很适合做这件事。我们可以把nothing放在一个条件语句内部，并在那里设置断点，这样可以仅在特定的条件下进行调试。例如：

```
if i = 8 then
  nothing
end if
```

■ 也可以在寻找键盘操作(如the shiftDown)的条件语句里放置nothing命令。这样可以在程序里的某个特定点进行测试，但只有我们需要时才测试。

```
if the shiftDown then
  nothing
end if
```

■ 可以在Debugger窗口里的处理程序历史区域点击，以便在当前处理程序刚运行完时就返回那一点，以查看以前的处理程序。这也会改变第二个区域内反映选定的处理程序状态的变量。