第六部分　用Lingo控制素材

第16章　控 制 文 本

虽然与图像、声音和数字视频文件比起来，文本好像更为基本，它仍然是大多数计算机程序与用户交流信息的主要方法。 Director 7有两个主要的演员用以处理文本，外加十几种用于处理这些演员和字符串的 Lingo命令和函数。如何使用这些演员、命令和函数是本章的主题。

16.1　使用字符串和子字符串

无论是在变量中的字符串，还是在域和文本演员内的字符串，都可以用字符串命令、子字符串表达式和字符串函数来控制。一些字符串常量中也可以包括 return和tab这样的通用字符。

16.1.1　建立字符串

第13章“ 重要的 Lingo句法 ”中，对字符串命令和函数做了一个简要的概括。下面是用于建立字符串的命令和操作符的更为详细的清单：

■ & —— 用于连接两个字符串。

■ && —— 用于连接两个字符串，并在它们之间插入一个空格。

■ put...after —— 将一个字符串添加到另一字符串中。它可以与子字符串表达式一块儿使用，以将字符插在某一字符串中间。

■ put...before —— 将一个字符串放到另一字符串中，位置在已有的字符之前。它可以与子字符串表达式一块儿使用，以将字符插入到字符串中间。

■ put...into —— 将字符串放在另一个变量中。它能与子字符串表达式一起使用，用另一字符串中的字符替换某一字符串中的字符。

■ delete —— 从一个字符串中除去某一子字符串。

所有这些命令都可以使用子字符串表达式扩展它们的功能。子字符串表达式是像 char、word、line和item这样的元素。delete命令需要子字符串表达式，而 put命令的三个变种可以使用子字符串表达式插入或替换现存的文本。

例如，如果有一个简单的字符“ abcdefg ”，我们想在c和d之间插入 xyz，可采用下面的方法：

```
s = "abcdefg"
put "xyz" after s.char[3]
put s
-- "abcxyzdefg"
```

然而，我们也能像下面这样来做：

```
s = "abcdefg"
put "xyz" before s.char[4]
put s
-- "abcxyzdefg"
```

你是否想用x替换字母c呢？那很容易：

```
s = "abcdefg"
put "x" into s.char[3]
put s
-- "abxdefg"
```

但是，是否你想用更多的字符代替字母 c呢？put命令不要求替换的字符串和原始子字符串有相同的字母个数。

```
s = "abcdefg"
put "xyz" into s.char[3]
put s
-- "abxyzdefg"
```

我们也能用短一些的字符串替换长一些的子字符串。

```
s = "abcdefg"
put "xyz" into s.char[2..6]
put s
-- "axyzg"
```

16.1.2 子字符串表达式

可以注意到，在前一节里， char子字符串能够接受单一数字的参数表示的一个字符，也能用a..b符号表示一系列字符。所有子字符串表达式都有这一特征。

第13章已经介绍了子字符串表达式。下列是子字符串表达式更详细的清单和描述：

■ char —— 能够用它在字符串中指定单个字符或一组连续的字符。

■ word —— 能够用它在字符串中指定一个单词或一个词组。单词之间由空格或 Tab、回车符等不可见字符分隔。表达单个单词时不要求包括这些分隔符，但是表达词组时，则要包括这些分隔符。

■ item —— 可以用它在一个字符串中指定某一个项目或一组项目。项目之间由对应于itemDelimiter属性的字符分隔。这个属性最初设置成逗号，但也可以改变。表达单个项目不要求包括分隔字符，但表达一组项目时要求包括分隔字符。

■ line —— 可以用它在一个字符串中的指定一行或几行。行之间由回车符分隔。域中的任何自动回行被忽略。表达单行时不要求包括回车符，但表达多行时则要包括回车符。

■ paragraph —— Director 7的新子字符串，与line相同。

子字符串表达式能够结合使用，从而在字符串中描述特定的子字符串的位置，如myString.line[1].word[1..2].char[3..4]。

我们也能使用旧的Director 6句法来引用字符串。前面的例子能够用下列语句描述： char 3 to 4 of word 1 to 2 of line 1 of myString。

16.1.3 比较字符串

在复杂的程序中,常需要比较字符串。我们能够使用以前常用的操作符“ =”来查看两个

字符串是否完全相同，但也能用各种操作符和函数执行其他类型的比较，如下面这些操作符：

■ = —— 比较两个字符串，如果二者相同，返回 TRUE。此操作符忽略大小写。

■ < —— 比较两个字符串，如果第一个字符串按字母顺序排列在第二个字符串之前，返回 TRUE。此操作符忽略大小写。

■ <= —— 比较两个字符串，如果第一个字符串按字母顺序排列在第二个字符串之前，或二者相同，则返回TRUE。此操作符忽略大小写。

■ > —— 比较两个字符串，如果第一个字符串按字母顺序排列在第二个字符串的后面，则返回TRUE。此操作符忽略大小写。

■ >= —— 比较两个字符串，如果第一个字符串按字母顺序排列在第二个字符串的后面，或二者相同，则返回TRUE。此操作符忽略大小写。

■ <> —— 比较两个字符串，当它们不同时，返回 TRUE。此操作符忽略大小写。

■ contains —— 比较两个字符串,如果第一个字符串包括第二个字符串则返回 TRUE。此操作符忽略大小写。

■ starts —— 比较两个字符串，如果第一个字符串的开始几个字符与第二个相同，则返回 TRUE。此操作符忽略大小写。

■ offset —— 它并不是操作符，而是可以像 contains一样使用的函数。如果某一字符串包含在另一个中间，它返回此字符串在另一字符串中的字符位置；或者如果某字符串不在另一字符串内，则返回0。

contains和starts操作符与“ = ”的语法使用相同，下列是几个例子：

```
s = "Hello World."
put (s contains "World")
-- 1
put (s contains "Earth")
-- 0
put (s starts "World")
-- 0
put (s starts "Hello")
-- 1
```

contains函数即使在较长的字符串或域内也能相当快地完成任务。它可用于在几百甚至几千个的域演员里寻找一个词组或关键词。这使它对数据库程序十分有用。

除了在某些情况下，一般 contains函数还比要求精确匹配的 =操作符宽容得多。如果我们想知道某个演员的名字中是否有“ button ”这个词， contains能找到名为“ button normal ”、“ button down ”、“ button rollover ”等等的演员，而=只能找到完全相同的演员名。

16.1.4 字符串函数

除了命令和操作符外，许多函数都可以用于从字符串中获得信息，将字符串转化为其他的变量类型，或者将其他变量类型转化成字符串。下面是这些字符串函数的详细清单：

■ chars —— Director的旧版本里的函数，它使我们能够从某一字符串中获得一系列字符。例如：chars(myString,3,5)与myString.char[3..5]相同。

■ charToNum —— 此函数将单个字符转成它的 ASCII代码数字，这个数字是计算机用于存储字符的数字。

■ count——这个字用作子字符串的一个属性，它返回在一个字符串中的子字符串的总数，如myString.count。

■ float——给定一个字符串时，此函数试着将此字符串转换成一个浮点数。“ 4.5 ”转化成4.5000。

■ integer——在给定一个字符串时，此函数试着将此字符串转换成一个整数。“ 4 ”被转成4。

■ length——此函数返回字符串中字符的个数。它也能用作字符串的一个属性。

■ numToChar——与charToNum相反，此函数获得一个用字符的ASCII数值来表示的整数，并将它转换成单个字符组成的字符串。

■ offset——此函数带有两个字符串参数。它返回的数值是指第一个字符串的第一个字符在第二个字符串中出现的位置。如果在第二个字符串中根本没有出现第一个字符串，则它返回0。

■ stringP——此函数测试一个变量是否是一个字符串。如果是，则返回 TRUE。

■ string——此函数把其他类型的变量(整数、浮点数、列表等等)作为参数，返回表示这些类型的字符串。

■ value——此函数把一个字符串作为参数，并试着将它转换成 Lingo语言描述的型式。例如，“ 4 ”返回4，“ 4.0 ”返回4.0000。

charToNum和numToChar函数是两个十分有用的函数。每个字符对应一个叫作 ASCII码的数字。这些数字的列在附录D“ 图表 ”中。

与大写“ A ”字符对应的数字是 65，“ B ”是66等等。与小写“ a ”相对应的是97，小写“ b ”是98等等。了解这些，我们就能创建一个简单的处理程序，以将混有小写字符的字符串转换成所有都是大写字符的字符串。

```
on allCaps text
  repeat with i = 1 to text.length
    thischar = charToNum(text.char[i])

    -- check to see if it is a lower case letter
    if thischar >= charToNum("a") and¬
        thischar <= charToNum("z")  then

      -- subtract 32, to make it upper case
      thischar = thischar - 32

      -- replace the character
      put numToChar(thischar) into text.char[i]
    end if
  end repeat
  return text
end
```

每个由此处理程序处理过的字符串都返回同样的字符串，但是用大写字母代替了小写字母。

```
put allCaps("This is a test.")
-- "THIS IS A TEST."
```

我们能够很容易地编写另一个处理程序，与这个过程相反，将字符串中所有字母转换成

小写字母。

前面的处理程序还使用了length属性。这个关键词能用作属性或函数。它返回字符串中字符的总数。

```
s = "Hello World."
put s.length
-- 12
put length(s)
-- 12
```

我们也能使用count属性，以得到某个字符串中的子字符串数目。

```
s = "Hello World."
put s.word.count
-- 2
put s.line.count
-- 1
```

length函数可以与子字符串表达式联合使用，以返回某一子字符串中的字符数目。

```
s = "Hello World."
put length(s.word[1])
-- 5
```

offset函数所做的事情都能用repeat循环和一些“ = ”运算来完成。然而，这个函数的运算速度很快。如果我们想在一个十分长的字符串中找到某一个字符串的第一次出现的，就应当使用offset。

只在一个字符串中找到另一个字符串的第一次出现， offset的功能似乎不应当这么有限。实际上，可以用offset函数在字符串中找到某个字符串出现的所有位置。下面的处理程序可以为我们完成这个任务：

```
on findStringInString substring, text

  -- initialize list and character position
  list = []
  currentChar = 1

  repeat while TRUE
    -- find in remaining string
    loc = offset(substring,text.char[currentChar..text.length])

    -- see if none left
    if loc = 0 then exit repeat

    -- add char pos to list
    add list, currentChar+loc-1

    -- move char pointer forward
    currentChar = currentChar+loc
  end repeat
  return list
end
```

在消息窗口中，可以这样使用该函数：

```
s = "Hello World."
put findStringInString("o",s)
-- [5, 8]
```

前面清单里的最后一个字符串函数是 value函数。如果我们想将一个字符串转换成一个数字，value常常比integer和float更好用。因为它将字符串转换成这两者中更合适的一个，而不是将它强行转换成一种类型或另一种类型。同样， value在字符串上执行数学函数计算的功能是十分强大的，如将字符串“ 1+1 ”返回成2。

我们甚至能用value调用Lingo函数。假如在影片剧本中有下面的这个处理程序：

```
on test
  return "testing!"
end
```

现在在消息窗口中试试这个函数：

```
put value("test()")
-- "testing!"
```

我们能够制作一个完整的“ 计算器 ”影片，只用value确定计算结果。图16-1显示了这样的一个计算器。第一个域是可以编辑的，使用户可以更改这个等式。

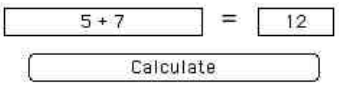

图16-1 由四个角色组成一个简单的计算器：两个文本域，一个“ = ”图形和一个“ Calculate (计算) ”按钮

所需要的Lingo程序只是按钮行为里一个简单的、只包含两行语句的处理程序。如果愿意，我们甚至可以将它缩成一行：

```
on mouseUp
  answer = value(member("question").text)
  member("answer").text = string(answer)
end
```

16.1.5 字符串常量

虽然像a这样的字符表达起来十分容易，但表达 Tab和回车符这样的字符则不容易。我们能够断定它们分别对应于哪个 ASCII码数字，并使用numToChar来表示，但是Lingo语言中有一些用于表示一些字符的内置常量。

■ SPACE —— 与“ ”相同。它的主要用途是使程序代码看起来更加整齐。

■ TAB —— 对应于ASCII码字符9，是由键盘上的Tab键产生的字符。当我们从使用了 Tab作分隔符的电子表格中输入一些文本时，它便派得上用场。

■ RETURN —— 对应于ASCII码字符13，由Mac上的Return键和Windows主键盘上的Enter键产生。它也是用于字符串中分隔行的符号。

■ QUOTE —— 因为引号在Lingo语言中用来定义字符串，所以我们需要用此字符在字符串中添加真正的引号。

■ EMPTY —— 对应于“ ”，或者长度为0的字符串。

■ BACKSPACE —— 对应于ASCII字符8，由Mac上的Delete键或Windows中的Backspace键产生。它用于翻译用户的键盘输入。

■ ENTER —— 对应于ASCII字符3，如果数字键盘设置成正确的使用状态，它由数字键盘的Enter键产生。

QUOTE常量是在Lingo编程中常常用得到的。如果我们想创建一个使用引号的字符串，可以像下面一样将QUOTE与&连接起来使用：

```
s = "The computer replied, "&QUOTE&"Hello World."&QUOTE
put s
-- "The computer replied, "Hello World." "
```

16.1.6 文本的引用

Director 7 有一个被称为ref的新的变量结构。此变量使我们能够重引用文本演员里的子字符串。例如，我们可以这样做：

```
put member("myText").line[2].word[2]
-- "is"
put member("myText").line[2].word[2].font
-- "Times"
```

或者，可以用ref属性设置一个变量，作为该子字符串的引用变量。然后，可以使用此变量以得到其他属性。

```
r = member("myText").line[2].word[2].ref
put r.text
-- "is"
put r.font
-- "Times"
```

目前，仅有text、font、fontStyle和fontSize属性能够用于ref引用变量。除text属性外，其他的属性都可以被设置为新的值，而所做的改变将关系到文本演员。

参见第13章里的13.2节“使用字符串变量”，其中包括更多有关使用字符串的信息。

16.2 使用文本演员和域

虽然文本演员和域的内容主要是文本，但是它们也有许多其他属性，会影响文本在舞台上的显示效果。文本的格式(如式样、字体和字号等)虽不是字符串的一部分，但对这些演员的文本却有很大的影响。

16.2.1 域

域演员的主要属性是它所包含的文本。我们可以在 Lingo语言中用演员的text属性得到它的文本。我们也能用Lingo语言的field句法，像对待字符串那样，得到任一个域的文本。

使用field可以直接将put命令和子字符串表达式用于域里的内容，而不必一定要首先将这些内容存在一个字符串变量中。例如，如果某一域包含“ Hello World ”文本，我们能够执行下列命令：

```
put "-" into char 6 of field 1
```

值得注意的是，在这种功能里不能使用 Director 7的新的“点”句法。例如，不能这样写：put “i” into field(1).char[6]，但却能这样写：put “i” into member(1).char[6]。这是因为field被认为是过时的语句，所以在Director6至7的升级过程中，并没有补充它的“点”句法。

对域演员来说，只使用旧式句法也能做一些事情，如在域中设置字符的字体。如果我们想设置整个演员的字体，可以这样使用“点”句法：

```
member(1).font = "Times"
```

然而，如果只想设置域里的几个字符、几个单词或几行文本，就需要使用 field句法。

```
set the font of word 6 to 9 of field 1 = "Geneva"
```

可以像这样设置域的许多属性。下面详细列出了这些属性：

■ font——字符的字体。应当用字符串“ Times ”或“ Arial ”等指定字体。

■ fontSize——字符的字体的大小。应当是一个整数，如9、12或72等。

■ fontStyle——字体的式样。在所有需要的式样间，应当用逗号分隔，如“ bold ”或“ bold, underline ”。也可以“ plain ”、“ bold ”、“ italic ”、“ underline ”、“ shadow ”或“ outline ”等式样。最后两种仅用于Mac。如果任何式样都不用，就用“ plain ”。

■ foreColor——字符的颜色。域里的任意一个子字符串都能使用影片里的调色板里的颜色。

除了这些应用于域的子字符串的属性外，还有许多属性可以应用于整个域演员。下面详细地列出了这些属性：

■ alignment——这一项可被设置成“ left ”、“ right ”或“ center ”，以改变域的对齐方式。

■ autotab——使用TRUE或FALSE以改变同名的演员属性。若它为TRUE，且域可编辑时，用户可使用Tab键在域间快速移动。

■ bgColor——能够用以设置域演员的背景色。可以使用新的rgb和paletteIndex结构。

■ border——可以用以设置或改变域周围的边框的宽度。0表示删除边框。

■ boxDropShadow——能够用以设置或改变域周围的边框的阴影。0表示删除阴影。

■ boxType——用以改变域演员的类型。同域演员的属性对话框中的内容一样，可选项有#adjust、#scroll、#fixed和#limit。

■ color——与foreColor相同，但能够使用新的rgb和paletteIndex结构。整个域必须一次设置。

■ dropShadow——用以设置或改变域的文本周围的阴影。0表示删除阴影。

■ editable——用以改变演员的可编辑属性。它可以为TRUE或FALSE。当它为TRUE，且影片正在播放时，用户能够在域中点击并编辑文本。

■ lineHeight——用以设置以像素为单位的整个文本域的行高。典型的方式是，行高被设置成比字号大几磅。

■ margin——用以改变文本域的内部边空属性。

■ wordWrap——通过设置它为TRUE或FALSE，用以表示自动回行为开或关状态。

如果某域被设置为滚动(Scrolling)类型，下面的几个命令和属性使我们能够控制域：

■ scrollByLine——此命令强迫域上下滚动一些行。例如，scrollByLine member(1), 2，表示向下滚动两行。若用负数，则向上滚动。

■ scrollByPage——与scrollByLine相同，只不过它是以页为单位滚动。一页是指域在舞台上的可见的那几行。若使用负数，则向上滚动。

■ scrollTop——此属性对应于滚动的域与顶部的距离(以像素为单位)。如果一个域所用的行高为12，它滚动一行，the scrollTop是12。

这三个滚动域命令的强大功能在于它们也能对设置成“ fixed ”类型的域起作用。这就是说即使滚动条元素并不在屏幕上，我们也能使用Lingo语言对域进行滚动。

可编辑的文本域有一个特殊的特征，即用户可以选中其中的文本。一些Lingo代码涉及这方面内容。我们能够得到选中的文本的位置、选中的文本的内空，甚至设置选中的文本。

■ hilite——此命令能够用以设置一个可编辑文本区域中的选中的文本，如hilite word 2 of

member “myField”。

■ the selection —— 此属性返回在当前激活的可编辑文本域中选中的文本。在此属性后不要放置演员的引用变量；它是独立的。

■ the selStart —— 它返回当前选中的第一个字符的编号。我们能够用它设置选中的文本。在此属性后不要放置演员的引用变量；它是独立的。

■ the selEnd —— 它返回当前选中的最后一个字符的编号。我们能够用它去设置选中的文本。在此属性后不要放置演员的引用变量；它是独立的。

最后，对域来说，下面几个函数能够用以发现屏幕位置和域中字符之间相互关系：

■ charPosToLoc —— 此函数有一个域演员和一个数字作为参数，并返回一个点，此点对应字符在演员中的位置。此点与演员的左上角相关联，不考虑正在发生的任何滚动。

■ linePosToLocV —— 此函数有一个域演员和一个数字作为参数，并以像素为单位，返回从演员的顶部到行位置的距离。

■ locToCharPos —— 此函数有一个演员和一个数字作为参数，并返回此域中位于这个点上的字符的编号。此点与演员的左上角相关，不考虑正在发生的任何滚动。

■ locVtoLinePos —— 此函数有一个演员和一个数字作为参数，并返回距域顶部的距离为那个数的行的编号。

■ the mouseChar —— 返回光标下的字符的编号，不考虑它在哪个域中。

■ the mouseWord —— 返回光标下的单词的编号，不考虑它在哪个域中。

■ the mouseLine —— 返回光标下的行的编号，不考虑它在哪个域中。

■ the mouseItem —— 返回光标下的项目的编号，不考虑它在哪个域中。

虽然the mouseChar及其相似的属性会告诉我们哪一个编号的子字符串在光标之下，但它不能告诉我们该子字符串属于哪个域。我们能够使用其他的函数，以得到更精确的信息。下面是一个在任何时候都能告诉我们光标下的是哪个域和哪个子字符串，而不考虑滚动情况的例子：

```
on underCursor
  s = the rollover
  loc = the mouseLoc

  -- is there a sprite under the cursor?
  if (s > 0) then

    -- is there a field attached to that sprite?
    if sprite(s).member.type = #field then

      -- subtract the loc of the sprite to get relative loc
      loc = loc - sprite(s).loc

      -- add any field scrolling
      loc.locV = loc.locV + sprite(s).member.scrollTop

      -- get the character number
      c = locToCharPos(sprite(s).member,loc)

      -- figure out the character
      ch = (sprite(s).member.text.char[c])
      put "The cursor is over character" &&c&&" ("&ch&")"
```

```
    end if
  end if
end
```

16.2.2 文本演员

文本演员有一系列与域相似的属性和函数。然而，有时句法会有所不同。

例如，我们仍然能够设置演员中的字体、字号、式样和任何子字符串的颜色，但是必须使用新的“点”句法。font和fontSize的功能与我们想象的一样。例如：

```
member("myText").char[2..5].font = "Times"
member("myText").char[2..5].fontSize = 18
```

然而，字体式样的使用方式稍有不同。我们不是提供一个字符串，如“ bold, underline ”，而需要提供一个列表，如[#bold，#underline]。用[#plain]表示不使用任何式样。

对文本上色也有所不同。我们能使用 color属性设置整个文本演员或其中一个子字符串的颜色。下面是几个例子：

```
member("myText").color = rgb(40,120,0)
member("myText").char[2..5] = rgb("#6699CC")
member("myText").word[7].color = paletteIndex(35)
```

文本演员没有任何边框、边空或阴影。然而，它们确实与域有一些共同的属性，不过它们大都使用不同的值。它们还有如下一些新属性：

■ alignment —— 它可被设置成#left、#right 或#center，以改变演员的对齐方式。文本演员也有#full设置，用来让文本强制齐行。

■ autotab —— 使用TRUE或FALSE，以改变同名的文本属性。当它为 TRUE且演员可编辑时，用户可以使用Tab 键在文本演员间快速移动。

■ boxType —— 使用它能够改变文本演员的类型。同文本演员的 Properties对话框中的内容一样，可选项有#adjust、#scroll和#fixed。

■ editable —— 使用它能够改变演员的可编辑属性。它可以是 TRUE或FALSE。当它为TRUE，且影片正在播放时，用户能够在演员中点击并编辑文本。

■ fixedLineSpace —— 它与域的lineHeight属性相同，但允许我们为不同的行设置不同的数值。

■ charSpacing —— 增加字符的间距，单位为像素，缺省值为 0。

■ kerning —— 如果我们不希望在文本变化时 Director自动调节文本演员的字符间距，可以将它设置成 FALSE。

■ kerningThreshold —— 在特定的演员里，把它设置为 kerning属性在缺省状态下应当起作用的最小字号。

■ leftIndent —— 文本距演员的左边的距离，以像素为单位。

■ rightIndent —— 文本距演员的右边的距离，以像素为单位。

■ firstIndent —— 本段文本的首行的左边的距离，以像素为单位。

■ antiAlias —— TRUE或FALSE，取决于是否想让演员中的文本以光滑、无锯齿的效果显示。

当文本演员被设置为可编辑状态时，也可以有被选中的区域。处理这个问题的 Lingo函数

与那些处理域的函数是不同的。

为得到选中的文本，应当使用 selectedText属性。它返回一个 ref结构。从那里，我们能够得到选中区域的文本字符串和一些字体信息：

```
r = member("myText").selectedText
put r.text
-- "the"
put r.font
-- "Times"
put r.fontSize
-- 12
put r.fontStyle
[#plain]
```

selection 属性返回一个列表，此列表包含选中区域的第一个和最后一个字符的编号。我们也能设置文本演员的选中区域，只要该演员可以编辑、影片正在播放且文本演员内有输入光标。这也就是说，不能使用消息窗口成功地设置 selection。

```
on preselectText
  member("myText").selection = [6,9]
end
```

文本演员也能够告诉我们位于某点的是哪个字符。这种功能的函数与域的函数十分不同。最基本的函数是 pointToChar，它告诉我们在某点之下是哪个字符。它可以这样使用：pointToChar(sprite 1, point(x,y))。

提示 注意，此函数使用角色作为引用变量，而以非演员作为引用变量。要确定光标下是哪个字符，可以用mouseLoc表示这个点。

此函数还有几个相似的函数： pointToWord、pointToItem、pointToLine和pointToParagraph。它们基本上以同样的方式执行，但是使用不同的子字符串表达式。

与域函数不同的是，这些文本演员函数根据实际舞台位置来判断事物，自动计算由角色位置和滚动产生的差异。

文本演员与域的不同之处还在于它们在内存中的表示方法不同。文本演员可以是 rich text或HTML text。域有一个text属性，它包含演员的普通的、无格式的文本；文本演员也是如此。然而，它们同时还拥有 rich text和HTML属性，分别称为rtf和html。如果我们创建一个简单的文本演员，将单词“ Testing ”放入其中，然后试着在消息窗口中访问这些属性，就会得到如下结果：

```
put member(1).text
-- "Testing"

put member(1).rtf
-- "{\rtf1\mac\deff3 {\fonttbl{\f3\fswiss Geneva;}{\f20\froman
Times;}}{\colortbl\red0\green0\blue0;\red0\green0\blue224;\red224\green0
\blue0;\red224\green0\blue224;}{\stylesheet{\s0\fs24 Normal Text;}}\pard¬
\f3\fs24{\pard \f20\fs36\sl360 Testing\par}}"

put member(1).html
-- "<html>
<head>
<title>Untitled</title>
```

```
</head>
<body bgcolor="#FFFFFF">
<font face="Times, Times New Roman" size=5>Testing</font></body>
</html>
"
```

rtf和html属性频繁地被更新，以反映文本演员的变化。更好的是我们能够直接设置其中的每一个属性，并且文本演员将发生变化，以与它们相一致。

很少有人了解rich text格式，但是许多人知道HTML。在Lingo语言中创建自定义HTML文本，然后将它应用到文本演员上，这可能是最强大的功能，也是目前还未被充分利用的Director 7功能。

16.3 创建文本清单

有一些界面元素使用域和文本演员。其中的一个有时被称为文本清单(text list)。它与一组单选按钮类似，但只需要一个角色。这个角色包含一个文本域，在这个文本域上，用户能够点击以从清单中选中一行文本。

图16-2显示了这样一个文本清单的例子。

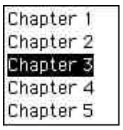

图16-2 一个小的文本清单，用户选中的文本行被突出显示

为了创建一个文本清单，首先要创建域演员。图16-2中的域有几行文本，使用了两个像素的边空和一个像素的边框。尽管选中的文本被设置为突出显示，但其中的文本行是不可编辑的。

这个行为很简单。它只需要一个属性，而且是为了使用方便。我们需要多次引用角色的演员，所以将它设置成属性使用起来十分方便：

```
property pMember

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

选中动作可以在on mouseUp或on mouseDown处理程序中进行。它调用一个自定义处理程序，以确定被点击的行，然后再调用另一个处理程序选中此行：

```
on mouseDown me
  -- get the number of the line clicked
  clickedLine = computeLine(me,the clickLoc)

  -- select that line
  selectLine(me,clickedLine)
end
```

接下来的处理程序根据点击的位置、角色的位置和域的滚动位置计算出被点击的行。在这个例子中的域是不滚动的，但下面额外的一行语句可以供将来我们希望有一个滚动文本清单时使用。

```
on computeLine me, loc
  -- get the vertical location minus the top of the sprite
  verticalLoc = loc.locV - sprite(me.spriteNum).locV

  -- add any amount that the field has been scrolled
```

```
  verticalLoc = verticalLoc + pMember.scrollTop

  -- return the results of locVtoLinePos
  return locVtoLinePos(pMember,verticalLoc)
end
```

我们可以用the mouseLine代替整个的on computeLine函数，但这样做会被认为十分草率。如果用户点击某一行，然后在Director执行包括the mouseLine的语句之前，又将鼠标迅速地移开，这会造成点击的行不同于选中的行。

on selectLine处理程序可以简单到只有一行，如果使用 hilite命令，并将演员设置为域，则就不必再做其他事情了。

```
on selectLine me, clickedLine
  -- use a simple hilite command to hilite the line
  hilite line clickedLine of field pMember
end
```

如果再看一下图 16-2中的这个图像，会发现显示结果并不十分美观。最重要的是，突出显示状态没有撑满整个选中行，它只是停在这行的最后一个字符上。如果从左边的边空一直到右边的边空都被置为突出显示状态，那么会好看得多。使用 char引用变量，而不是使用 line引用变量，外加hilite命令，可以实现这个目的。所要做的就是将每行末尾的不可见的回车符包括hilite状态中：

```
on selectLine me, clickedLine
  --figure out the first and last chars for hilite
  if clickedLine = 1 then
    -- first line, start with char 1
    startChar = 1
  else
    -- not first line, count chars before line
    -- and add 2 to go past return to the next line
    startChar = (pMember.text.line[1..clickedLine-1]).length + 2
  end if

  -- for last char, count chars including line,
  -- and then add 1 for the RETURN character
  endChar = (pMember.text.line[1..clickedLine]).length + 1

  hilite char startChar to endChar of field pMember
end
```

文本清单也可以只使用单选按钮行为和一些单行文本演员实现。或者，可以使用文本演员，而不是域演员，并在文本背后放置一个矩形图形，以用作突出显示状态。通过这样做，我们甚至可以添加一些程序，并允许在文本清单中进行多项选择，以使清单中项目的表现与一组复选框相似。不能用hilite实现这个目的，因为它仅允许连续的选择区域。

16.4 创建文本弹出菜单

文本清单的自然延伸就是文本弹出菜单。这些弹出菜单是开发人员多年来熟知的一种技巧。因为域可以设置为 Adjust To Fit形式。并且能够在播放时被更新，我们也就十分有信心，让它们模仿弹出菜单。

请看一下图16-3，它显示了一个文本域，其中只有一行文本。该域已经被设置成 1个像素的边框、2个像素的边空和2个像素的方框阴影的形式，以使它看上去更像能够点击的样子。

赋予该域的行为通过在其中放置更多的文本行，可以改变它的外观。因为此域被设置成Adjust To Fit的状态，所以当文本行增多后，域就得以扩大。图 16-4显示了用户点击时的域。其中不仅添加了更多文本，而且要用 hilite命令显示在鼠标被释放时被选中的是哪一项。

图16-3 一个简单的域可以制作成看上去像没有被激活的弹出菜单

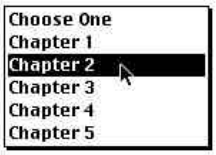

图16-4 由于更多的文本被放入其中，这个域弹出菜单被扩展

为完成这个十分巧妙的技巧，行为首先要获得当域被激活时需要放入域中的项目清单。这个过程可以用许多方法完成。对本例中的行为来说，域在一开始就拥有完整的清单，只是当角色刚出现时，除了第一行外，其他所有行都被隐藏起来。该域将被设置成“ fixed ”帧类型，并在最后一行末包括一个额外的回车符。下面是行为的起始部分：

```
property pMember -- the field used in the popup
property pText -- the complete text of the popup
property pSelection -- the selected text
property pPressed -- whether the user is making a selection
property pLastHilite -- the last line highlighted

on beginSprite me
  -- get some properties
  pMember = sprite(me.spriteNum).member
  pText = pMember.text
  pSelected = pText.line[1] -- assume first line is default
  pPressed = FALSE

  -- set the field to the selected item
  pMember.text = pSelected

  -- set the field rectangle
  setMemberRect(me)
end
```

当用户点击此域时，动作开始，弹出菜单出现。下面是这个动作的处理程序：

```
on mouseDown me
  pPressed = TRUE
  openPopup(me)
end


on openPopup me
  pMember.text = pText
  setMemberRect(me)
  pLastHilite = 0
end
```

```
-- This handler will adjust the field to be the size
-- of the text contained in it
on setMemberRect me
  memRect = pMember.rect
  numLines = pMember.text.lines.count
  if pMember.text.line[numLines] = " " then numLines = numLines - 1
  memRect.bottom = memRect.top + (numLines * pMember.lineHeight)
  pMember.rect = memRect
end
```

on exitFrame处理程序现在要不断检测鼠标的位置，以确保突出显示的是正确的项目：

```
on exitFrame me
  if pPressed then
    -- What line is the cursor over
    thisLine = getLine(me)

    -- is it over a different line than before?
    if thisLine <> pLastHilite then
      selectLine(me,thisLine)
      pLastHilite = thisLine
      pSelection = pText.line[thisline]
    end if
  end if
end
```

当鼠标按钮被释放时，弹出文本要消失，域将恢复成它以前的状态。另外，如果鼠标在某个角色上释放，表示很可能选中了某一项：

```
on mouseUp me
  pPressed = FALSE
  closePopup(me)
  makeSelection(me)
end

on mouseUpOutside me
  pPressed = FALSE
  closePopup(me)
end

-- set the popup to the current selection
on closePopup me
  pMember.text = pSelection
  setMemberRect(me)
end
```

on getLine是个十分有用的处理程序，它用于分辨光标当前落在哪一行上。此函数是对 the mousLine 函数的替换。该函数在早期的 Director版本中的运行很好，但在 Director 7中，如果光标在某一行中超过回车符，它就不正常了。

```
-- get the line the cursor is over
on getLine me
  if the rollover <> me.spriteNum then
    return 0
  else
    y = the mouseV - sprite(me.spriteNum).locV
```

```
    lineNum = y/(pMember.lineHeight)+1
    return lineNum
  end if
end
```

on selectLine处理程序与文本清单行为里所使用的那个处理程序相同：

```
on selectLine me, clickedLine
  --figure out the first and last chars for hilite
  if clickedLine = 1 then
    -- first line, start with char 1
    startChar = 1
  else
    -- not first line, count chars before line
    -- and add 2 to go past return to the next line
    startChar = (pMember.text.line[1..clickedLine-1]).length + 2
  end if

  -- for last char, count chars including line,
  -- and then add 1 for the RETURN character
  endChar = (pMember.text.line[1..clickedLine]).length + 1

   hilite char startChar to endChar of field pMember
end
```

最后，on makeSelection处理程序是实际完成这些工作的处理程序。在本例中，只出现了alert 。然而，请注意必须从行编号里减去“ 1”，以得到与选择相应的编号。这一步是必须的，因为选择是从域的第二行开始的。

```
on makeSelection me
  if pLastHilite > 0 then
    alert "You picked number"&&(pLastHilite-1)
  end if
end
```

所需要的最后一个处理程序是 on endSprite处理程序。它在影片离开带有该角色的帧时被调用。此处理程序用以保证原始文本在域中被重新放置。

```
on endSprite me
  -- restore the contents of the field
  pMember.text = pText
end
```

这种弹出菜单并不如用位图制作的菜单那么美观。然而，它们创建和定义起来都很容易。如果我们需要很多不同的弹出菜单，而且外观并不是十分重要的因素时，就可以使用这种弹出菜单。

参见第15章“ 图形界面元素 ”里的 15.7节“ 创建图形化的弹出菜单 ”，了解创建弹出菜单的另一种方式。

参见第28章“ 商用软件 ”里的28.3节“ 制作调查问卷 ”，以查看有关使用弹出菜单的例子。

16.5 使用键盘输入

虽然大多数演示和游戏等多媒体作品只使用鼠标操作就可以了，但迟早有需要用户使用键盘操作的可能，如输入文本或用箭头键控制游戏等。

16.5.1 键盘Lingo语言

Lingo语言有不少处理键盘输入的函数和事件。下面是详细的清单：

■ the commandDown —— 只在Mac上的Command键或Windows上的Ctrl键按下时，返回TRUE。

■ the controlDown —— 只在Mac上的Control键或Windows上的Ctrl键按下时，返回TRUE。在Windows上，它与the commandDown相同。

■ the optionDown —— 只在Mac上的Option键或Windows上的Alt键按下时，返回TRUE。

■ the shiftDown —— 只在shift键按下时返回TRUE。

■ on keyDown —— 在可编辑的域或文本演员角色、帧或影片剧本中所使用的事件处理程序。当用户初次按下某个键时，消息被发送。如果用户一直按着键，此处理程序根据用户的计算机的键盘重复设置，不断发送 keyDown消息。当此处理程序被调用时， the key和the keyCode属性被设置。

■ on keyUp —— 在可编辑的域或文本演员角色、帧或影片剧本中所使用的事件处理程序。当用户从某个键上抬起手指时，消息被发送。如果用户一直按着键也不会重复发送消息。当此处理程序被调用时， the key和the keyCode属性被设置。

■ keyPressed —— 此函数用以测试某一字符键是否被按下。例如，如果此刻 a键被按下，keyPressed (“ a ”) 返回TRUE。

■ the key —— 此属性存在于on keyDown或on keyUp处理程序中，包含那个曾被按下并因此激活该处理程序的字符。

■ the keyCode —— 此属性存在于on keyDown或on keyUp处理程序中，包含那个曾被按下并因此激活该处理程序的键的键盘编码。它仅在箭头键和功能键被使用时是可靠的。

■ the keyboardFocusSprite —— 当舞台上有多个可编辑的域或文本演员时，此属性用于设置让文本插入符落在哪一个角色上。

事件处理程序on keyDown和on keyUp用于修改在域和文本演员中的文本输入。 Director 7新添的keyPressed函数主要用于在游戏和其他实时程序中直接使用键盘。

16.5.2 了解Return键

使用键盘的Lingo命令的第一步是建立一个简单的行为，该行为判断用户在录入文字时何时按下了Return键。它的用途是想“ 偷走 ” Return字符，致使用户不能录入第二行文本，也可以用Return字符触发某个事件。

注释 在绝大部分Windows键盘上，Return键实际上总被标记为Enter。这就容易让人糊涂，因为在多数键盘的数字键盘上也有Enter键。Mac在主键盘有一个Return键，在数字键盘上有一个 Enter键。当本文涉及 Return键时，它在 Mac上是指Return键，而在Windows里则是主键盘上的Enter键。

例如，如果我们有一个屏幕，想让用户在那儿输入自己的名字，那么记录 Return键可能是个很好的作法。首先，它阻止用户录入名字时，录入第二行文本和一些在输入名字时不应当执行的操作。其次，它标识录入的结束，而不必让用户再按“ 我已经完成录入 ”这样的按钮。

下面是捕获Return键的一个行为。我们可以将它赋予某个可编辑的域或文本演员的角色：

```
on keyDown me
  if the key = RETURN then
    alert "You typed:"&&sprite(me.spriteNum).member.text
    dontpassevent
  else
    pass
  end if
end
```

此行为只使用了 on keyDown处理程序。它检测 the key属性，以查看它是否等于常量RETURN。如果是这样，则执行一个动作。它也使用 dontpassevent命令，以阻止keydown消息从处理程序再传到其他地方。否则，它将通知可编辑的演员：“ 某一个键已被按下 ”，而那个演员将把这个键 —— 在本例中为Return —— 添加到它的文本里。

如果the key不是Return，pass命令按自己的方式传送这个消息，即添加一个字符到文本中。实际上， dontpassevent命令已暗含在该处理程序中，只需要 pass命令就可以了。然而，使用dontpassevent命令并没有什么坏处，它使处理程序代码更加清楚。

这个简单的行为只使用了 alert命令以表示Return已被按下。在实际的程序中，我们可能需要采取某种措施，如将该文本记录在一个全局变量中，并转到另一帧。如果在舞台上有多个可编辑的文本域，我们可能还需要使用 keyboardFocusSprite，将文本插入符移动到另一个角色上。

16.5.3 限制输入

因为on keyDown处理程序能够捕获任何按键操作，所以我们也能用它进行限制，只允许把特定的字符传给演员。假如我们希望用户在某一域中仅输入数字，可以编写一个只接受数字的行为。

这样一个行为需要知道哪些字符是允许输入的。最好由一个参数执行：

```
property pAllowed

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pAllowed, [#comment: "Allowed Chars",¬
    #format: #string, #default: " "]
  return list
end
```

这个行为的余下部分是on keyDown处理程序，用以测试键入的字符是否可以接受。

```
on keyDown me
  if pAllowed contains the key then
    pass
  else
    dontpassevent
  end if
end
```

这个行为的限制并不很严格。事实上， Backspace键对它没有影响。创建一个接受Backspace键的行为也是很容易的。另外，此处理程序也能搜寻 Return键并同时对它进行处理。

```
on keyDown me
  if the key = RETURN then
    alert "You typed:"&&sprite(me.spriteNum).member.text
```

```
    dontpassevent
  else if the key = BACKSPACE then
    pass
  else if pAllowed contains the key then
    pass
  else
    dontpassevent
  end if
end
```

同样的处理程序也能用于限制只允许输入字母，或只允许字母和数字，而不允许其他符号。使用相同的基本思路，我们甚至能够限制可编辑的演员所接受的字符的数量。下面是能够做这项工作的处理程序：

```
property pMaxChars

on getPropertyDescriptionList me
  list = [:]
  addProp list, #pMaxChars, [#comment: "Maximum Number of Chars",¬
    #format: #integer, #default: 10]
  return list
end

on keyDown me
  if the key = RETURN then
    alert "You typed:"&&sprite(me.spriteNum).member.text
    dontpassevent
  else if the key = BACKSPACE then
    pass
  else if sprite(me.spriteNum).member.text.length < pMaxChars then
    pass
  else
    dontpassevent
  end if
end
```

该处理程序确保只有当现有字符的数量小于可接受的最大字符数量时，才允许用户添加另一个字符。注意， BACKSPACE的检测是单独进行的，即使达到了演员的最大长度也仍旧能够使用。

16.5.4 捕获按键操作

在可编辑的域和文本演员中输入文本的方式很适合于从用户处获取信息。然而，如果我们只想捕获单个按键操作，以使用户能够在影片中控制其他事情，则不需要它们。

如果没有可编辑的演员存在，按键操作事件将传给帧，被帧剧本捕获。可以使用同样的on keyDown和on keyUp处理程序来做这件事情。

下面的处理程序是帧剧本中的处理程序，它捕获按键操作并将消息放在角色 1的域中。此消息包含the key和the keyCode属性，还将the key转换成了它的ASCII编码：

```
on exitFrame
  go to the frame
end

on keyUp me
```

```
  sprite(1).member.text =¬
    "You pressed:"&&¬
    the key&&¬
   "("&chartonum(the key)&","¬
    &the keyCode&")"
end
```

这个行为的用意不仅是为了说明我们怎样用帧剧本捕获按键操作，而且也能告诉我们怎样测试各键并查看它们对应的键代码是什么。用这个剧本试一下，我们就能看到箭头键的代码为123、124、125和126，分别对应左、右、下和上箭头键。

有了这些知识，我们能够编写一个捕获这些按键操作的行为，然后告诉某一角色根据这些操作去移动。下面是完成这项工作的帧剧本：

```
on exitFrame
  go to the frame
end

on keyDown
  case the keyCode of
    123: sendSprite(sprite 1, #move, point(-5,0))
    124: sendSprite(sprite 1, #move, point(5,0))
    125: sendSprite(sprite 1, #move, point(0,5))
    126: sendSprite(sprite 1, #move, point(0,-5))
  end case
end
```

每一个箭头键将一个 move消息发送给角色。它也发送一个 point结构，以告诉角色它的位置要改变多少。下面是可以用于为角色 1捕获这些move消息的行为。

```
on move me, dist
  loc = sprite(me.spriteNum).loc
  loc = loc + dist
  sprite(me.spriteNum).loc = loc
end
```

注释 on keyDown和on keyUp事件处理程序要求舞台已经有了焦点窗口。这就是说，它是一个激活窗口并接收键盘事件。这可能要求用户在影片放映后和使用箭头键前点击舞台。

因为帧剧本使用 on keyDown捕获键值，用户可以保持箭头键按下状态，以使角色连续移动。这个过程进行仅是因为计算机在用户按住键时，发送了大量 keyDown消息。结果与用户重复按下此键的效果相同。

这种运动形式是非常基本的屏幕活动，但是还有一种方式可以创建更加流畅的运动，我们可以使用keyPressed函数不停地查看键盘，以确定是否有键被按下。

下面是使用keyPressed检测键盘并发送move消息给角色1的帧剧本。

```
on exitFrame
  if (keyPressed(123)) then sendSprite(sprite 1, #move, point(-5,0))
  if (keyPressed(124)) then sendSprite(sprite 1, #move, point(5,0))
  if (keyPressed(125)) then sendSprite(sprite 1, #move, point(0,5))
  if (keyPressed(126)) then sendSprite(sprite 1, #move, point(0,-5))
  go to the frame
end
```

这两种做法的差异是很明显的。 on keyDown方法创建一个不平稳的、慢速的运动，而keyPressed方法可以得到流畅的运动效果。实际上，帧速度越快，运动就越流畅。

虽然the keyPressed对于箭头键和字母数字键的工作状态很好，但它没有办法用它查看像Shift和Command这样的键。对于这些键，必须对它们使用 16.5.1节中的清单描述的特殊属性。

16.6 使用rich text格式

通过使用文本演员的rtf属性，我们接触到它的 rich text格式。这是一种完全独立的语言，现在属Microsoft所有。创建一个包含单词“ Testing ”的文本演员，其中用到了18磅的Times字体。下面是使用消息窗看到的最终的 rtf:

```
put member(1).rtf
-- "{\rtf1\mac\deff3 {\fonttbl{\f3\fswiss Geneva;}{\f20\froman
Times;}}{\colortbl\red0\green0\blue0;}
{\stylesheet{\s0\fs24 Normal Text;}}\pard¬
\f3\fs24{\pard \f20\fs36\sl360 Testing\par}}"
```

正像我们看见的一样， rich text格式需要许多控制结构，以定义式样和色彩。如果要详细列举每个结构的含义，将需要整整一本书。因为 RTF现在已经很少使用， HTML已替换它作为文本标准，所以我们不必再详细了解它。

然而，我们应该知道我们也能创建自己的 rich text格式化代码，并通过设置rtf属性，替换演员中的文本。例如，用“ Courier ”代替前面那些复杂的代码里的“ Times ”，那么文本演员看起来还是一样的，只不过改用了 Courier字体。

16.7 使用HTML和表格

文本演员的html属性更容易使用。实际上，我们能够使用 Lingo语言创建HTML代码，并将它应用到演员上。后面的例子表明了这个属性使用起来是非常容易的。

16.7.1 一个简单的HTML应用程序

下面是同样的Testing演员的html属性：

```
put member(1).html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#FFFFFF">
<font face="Times, Times New Roman" size=5>Testing<br>
</font></body>
</html>
"
```

我们可以看到， Director总要证实正确的“ <html> ”、“ <head> ”、“ <body> ”和“ <title> ”存在。title总是“ Untitled ”，而“ bgcolor ”实际上反映了演员的背景颜色。

html属性是很容易编辑的，尤其在我们已经了解了 HTML语言后更是如此。我们甚至能省略一些标签， Director会帮助我们填写它们。在消息窗口中，用演员表中的一个单独的文本演员试试下面这段程序：

```
member(1).html = "Testing"
put member(1).html
```

```
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#FFFFFF">
Testing</body>
</html>
"
```

我们可以看到，大部分适当的标签都是由 Director添加的。然而，如果我们想用自己的HTML标签修改这个文本，则需要将这些标签加入进去。试试下面这个程序：

```
member(1).html = "<B>Testing</B>"
put member(1).html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#FFFFFF">
&lt;B&gt;Testing&lt;&#47;B&gt;</body>
</html>
"
```

我们可以看到， Director没有正确地翻译“ 粗体 ”标签，以使文本加粗。相反，而把它看作文字。舞台上的文本看上去将是“ <B>Testing<B> ”。为了让Director识别该粗体标签，它需要找到“ <body> ”标签。

```
member(1).html = "<HTML><B>Testing</B></HTML>"
put member(1).html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#CCCCCC">
<b>Testing</body>
</html>
"
```

现在它正确地将“ 粗体 ”标签识别为标签。而演员的背景颜色已被设置成灰色，以模仿浏览器的缺省的灰背景。为使在某些情况下，产生不同于灰色的效果，我们也要设置自己的body标签。

```
member(1).html = ¬
  "<HTML><BODY BGCOLOR=#FFFFFF><B>Testing</B></BODY></HTML>"
put member(1).html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#FFFFFF">
<b>Testing</body>
</html>
"
```

现在文本演员按我们要求的显示出来。我们还能添加有关尺寸和外观的字体标签，以设置文本的字体。我们甚至能够添加表格标签以创建表格。

16.7.2 应用表格

用 HTML创建表格是Director一个强大的新功能。它使我们能够用 Director 6中几乎不可能的方式创建高度格式化的文本。

为创建表格，要做的所有工作是用 HTML语言构造它，并将它应用到文本演员的 html属性上。下面是一个简单的例子。创建一个文本域，将之命名为“ html text ”，然后在里面放置如下的文本：

```
<html>
<body bgcolor="#FFFFFF">
<table border=1>
<TR><TD>
Test1
</TD><TD>
Test2
</TD></TR>
<TR><TD>
Test3
</TD><TD>
Test4
</TD></TR>
</table>
</body>
</html>
```

现在，创建一个空文本演员并放在舞台上。将它命名为“ html member ”。使用消息窗口，我们能将HTML文本应用到文本演员上。

```
member("html member").html = field "html text"
```

舞台上的文本演员将如图 16-5中那样显示。

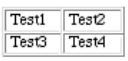

图16-5 包含一些简单HTML属性的文本演员，它组建了一个带有边框的表格

几乎所有的HTML表格的特殊功能都能用于文本演员。使用一个影片处理程序，我们可以创建适合任何需要的表格。下面的处理程序把几个列表做成了一个表格：

```
on makeTable memberName, headings, width, data
-- start with <HTML> and <BODY> tags
  htmlText = "<HTML><BODY BGCOLOR=FFFFFF>"

  -- beginning of table
  put "<TABLE BORDER=0><TR>" after htmlText

  -- place headings as TH tags
  repeat with i = 1 to count(headings)
    put "<TH WIDTH="&widths[i]&">" after htmlText
    put "<B>"&headings[i]&"</B></TD>" after htmlText
  end repeat
  put "</TR>" after htmlText

  -- add each row
  repeat with i = 1 to count(data)
    put "<TR>" after htmlText
```

```
    -- add a row
    repeat with j = 1 to count(data[i])
      put "<TD>" after htmlText
      put data[i][j]&"</TD>" after htmlText
    end repeat

    put "</TR>" after htmlText
  end repeat

  -- close table and html
  put "</TABLE></BODY></HTML>" after htmlText

  member(memberName).html = htmlText
end
```

此处理程序用一个演员名和三个线性列表作为参数。第一个列表包含表头的文本；第二个包含用基于HTML的像素描述的栏宽；第三个列表包含一组较小的列表，其中每一个代表表格中的一行。在这些较小的列表中的项的数目与其他两个列表中的数目相同。下面是使用on makeTable处理程序的一个处理程序：

```
on testTable
  headings = ["Name","Address","Phone","Birthday","City"]
  widths = [100,150,60,60,80]
  data = []
  add data, ["John Doe", "123 Street Road", "555-3456", ¬
    "7/28/65", "Seattle"]
  add data, ["Betty Deer", "654 Avenue Blvd", "555-1234", ¬
    "9/11/68", "Los Angeles"]
  add data, ["Robert Roberts", "9346 Dead End Pl.", ¬
    "555-9999", "1/8/67", "New York"]
  makeTable("html table",headings, widths, data)
end
```

使用这两个处理程序的结果在图16-6中显示。

这种表格的一个优点是文本可以在栏内自动换行。这比使用带有 Tab键的多个行形成的表格要好得多。当表格太长，在屏幕里容纳不下时，我们甚至可以将像这样的表格放进滚动的文本演员中。

Name	Address	Phone	Birthday	City
John Doe	123 Street Road	555-3456	7/28/65	Seattle
Betty Deer	654 Avenue Blvd.	555-1234	9/11/68	Los Angeles
Robert Roberts	9346 Dead End Pl.	555-9999	1/8/67	New York

图16-6 此表格用Lingo语言产生，并被应用到文本演员上

16.8 使用HTML和超文本

Director 7 的另一个新功能是给文本演员方便地添加超文本的功能。实际上，这项工作十分简单，我们几乎根本不需要任何 Lingo语言。

16.8.1 设置和使用超链接

指定某个文本代表一个链接，只要使用演员和文本监察窗就可以了。我们可以在舞台上或Text Cast Member编辑窗口中编辑文本。

图16-7显示了此动作的过程。舞台上有一个文本演员，其中“ dog ”被选中。文本监察窗已将“ man ’ s best friend ”作为选中的文本的超链接数据。结果，文本演员中的单词“ dog ”

被置于下划线和蓝色着色状态，作为超文本存在，它是 Web浏览器中典型的超文本格式。“quick”单词也已被设置成超文本。

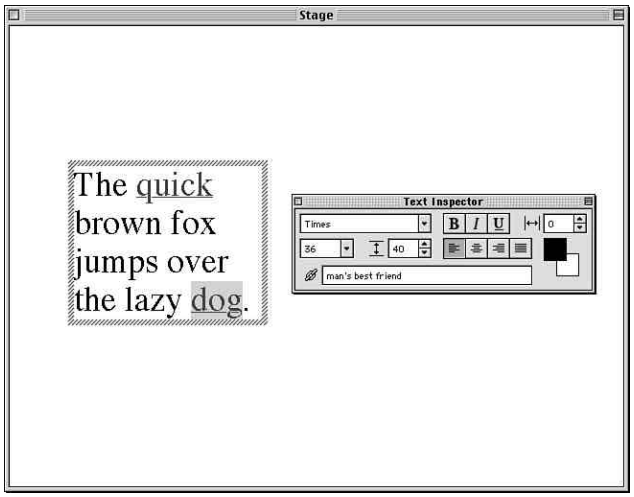

图16-7 文本监察窗用于在文本演员中设置超文本

在超文本设置完后，无论何时文本被点击，超链接数据都可以通过 Lingo语言得到。文本的式样是自动的。此外，光标将自动变成手指光标。

提示 我们可以通过对演员属性useHyperlinkStyles的设置，关闭自动式样和光标的变化。然后，我们还能够单独改变文本的式样和颜色。

为了在超文本上实际做一些事情，我们要创建一个简单的行为，以在用户点击超链接时捕获所发送的信息。on hyperlinkClicked处理程序得到这些消息，同时，特定链接的超链接数据被存储。下面是一个例子：

```
on hyperlinkClicked me, data
  put data
end
```

这个简单的处理程序在消息窗口中放置超链接数据。这个代码行自身并不十分有用，但却能用于在另一个域中放置数据。像这样的处理程序能用于创建一个简单的词汇表函数。用户可以点击某个带有超链接的单词，它的定义将出现在舞台的底部。

```
on hyperlinkClicked me, data
  member("Glossary").text = data
end
```

我们也能用这个剧本做一些事情，如让影片跳到另一帧。此帧的名称应当是 data参数。

```
on hyperlinkClicked me, data
  go to frame data
end
```

如果我们正在创建一个 Shockwave短程序，此超文本可以用于使浏览器跳到另一个网址。然后，它将像 Web页上普通的超链接一样工作。超链接数据必须是一个有效的网址，如 http://clevermedia.com。

```
on hyperlinkClicked me, data
```

```
  gotoNetPage(data)
end
```

还有一个参数也可以使用在 on hyperlinkClicked处理程序里。它返回一个小型列表，此列表包含被点击的超链接里的第一个字符和最后一个字符的位置。使用一个普通的子字符串表达式，我们就能够通过此列表得到超链接的实际文本。下面是修改过的超链接词汇表剧本：

```
on hyperlinkClicked me, data, pos
  definition = data
  hyperword = sprite(me.spriteNum).member.char[pos[1]..pos[2]]
  member("Glossary").text = hyperword&":"&&definition
end
```

16.8.2 用Lingo修正超链接

除了用文本监察窗设置超链接外，我们也能用 Lingo语言设置超链接。文本演员的hyperlinks和hyperlink属性能够用来添加、修改和删除超链接及它们的数据。

hyperlinks属性返回一系列包含每个链接的第一和最后一个字符位置的小型列表。例如，图16-7中的例子显示了两个链接。如果我们使用消息窗口得到 hyperlinks，其结果如下：

```
put member(1).hyperlinks
-- [[5, 9], [41, 43]]
```

虽然这些属性对于确定文本中存在什么超链接十分有用，但实际上，我们不能用这种方式设置超链接。相反，而应使用 hyperlink属性。下面是一个例子：

```
put member(1).char[5..9].hyperlink
-- "fast, speedy"
```

我们能以相似的方式为某些新的演员设置超链接数据：

```
member(1).char[5..9].hyperlink = "no definition"
put member(1).char[5..9].hyperlink
-- "no definition"
```

用这种方式设置超链接，却不指定超文本的精确的字符位置，是不可取的。然而，在发生这种情况时，Director确实是按逻辑进行处理的。如果我们试着设置超链接的子集，如前面例子中的字符6~8，整个链接就被改变了。

要消除超链接，将超链接数据设置成“ ”或空字符串就可以了。

```
member(1).char[5..9].hyperlink = EMPTY
```

在一组还没有超链接的字符上设置超链接，将会创建一个超链接。下面的剧本在一个文本演员中查找某个单词，并将所有出现的这个单词都设置成超链接。

```
on makeHyper memberName, hyperword, hyperdata
  text = member(memberName).text
  repeat with i = 1 to text.word.count
    if text.word[i] = hyperword then
      member(memberName).word[i].hyperlink = hyperdata
    end if
  end repeat
end
```

16.9 使用文本文件和FileIO Xtra

在Director中使用文本文件总比想象的要困难得多。在 Director 7中也没有什么不同。它要

求我们使用FileIO Xtra。这个Xtra的用途是添加一些读、写文件的Lingo命令。

使用这样的一个Xtra就像使用行为一样，只不过Xtra是通过变量进行引用，而不是通过角色。下面是一个例子。为了阅读一个文本文件，可以使用下面这些命令：

```
fileObj = new(xtra "FileIO")
openFile(fileObj, "myfile.txt")
text = readfile(fileObj)
closeFile(fileObj)
```

变量fileObj用于存储这个Xtra的一个实例。第一行的new命令用于创建该实例，一个指向它的指针放在变量fileObj中。我们可以把fileObj当作行为中的me。现在讨论FileIO的这个实例。

一个FileIO的实例能够打开、创建、写、读和删除文件。在前面的例子中，该对象使用openFile命令打开一个文件，用readFile命令读取这个文件内容，然后用closeFile命令关闭这个文件。

下面列出了FileIO Xtra所使用的全部命令：

■ new —— 创建Xtra的一个新实例。

■ fileName —— 返回当前被Xtra的这个实例控制的文件名。

■ status —— 返回在Xtra中使用的最后那个命令的错误代码。0表示无错误出现。

■ error —— 取一个对象和整数作参数。该整数是错误代号。它返回一个描述错误内容的字符串。

■ setFilterMask —— 取一个对象和一个字符串作参数。该字符串定义在 Open(打开)和Save(存储)对话框中显示的文件类型。在Mac上，传送一个包含1个或多个4个字母的文件类型，如“ TEXT ”或“ JEPGGIFF ”。在Windows中，此字符串应该是一个由逗号分隔的列表，以代替文件描述和类型，如“ Text Files, *.txt，GIF Files, #.gif ”。

■ openFile —— 用于打开文件来读、写或两者都要执行。我们在执行大部分其他的文件功能前要调用这个函数。它带有另外的参数，如1表示读，2表示写，0表示读写都要执行。

■ closeFile —— 关闭与文件对象相连的文件。当我们用完这个文件时，必须调用此函数。

■ displayOpen —— 用于显示Mac或Windows上的Open对话框，并使用户能够浏览和选择一个文件。它返回文件路径。

■ displaySave —— 用于显示Mac或Windows上的Save对话框，并使用户能够浏览和选择文件的存储目的地。它返回文件路径。

■ createFile —— 需要将一个对象和代表文件名或文件完全路径的字符串赋予该命令。在文件还不存在的情况下，必须在openFile之前调用它。

■ setPosition —— 有一个整数作额外的参数。它在文件中设置下一次读操作将要发生的位置。

■ getPosition —— 返回文件中当前的读操作的位置。

■ getLength —— 返回当前打开的文件长度。

■ writeChar —— 向文件中写一个字符。

■ writeString —— 向文件中写一个完整的字符串。

■ readChar —— 从文件中读取单个字符。

■ readLine —— 从文件中读取从当前位置到下一个Return字符之间的所有字符。它将Return

字符包括在返回值中。

■ readFile —— 从文件中读取从当前位置到文件结束的所有字符。

■ readWord —— 从文件中读取从当前位置到下一个空格或不可见字符间的所有字符。

■ readToken —— 此函数有三个参数：对象、一个跳过的字符和一个中断字符。它从文件中读取从当前位置到中断字符间的字符，无论在什么时候碰见要求跳过的字符时，它都跳过它们。

■ getFinderInfo —— 仅用于Mac，它用于获得文件类型和当前打开文件的创建者。并将获得的信息以9个字符的字符串返回，在文本类型和创建者间包含一个空格，如“ TEXT ttxt ”。

■ setFinderInfo —— 仅用于Mac，它用于设置文件类型和当前打开文件的创建者。我们必须使用一个9个字符的字符串，如“ TEXT ttxt ”。

■ delete —— 此命令删除当前打开的文件。

■ version —— 返回Xtra的版本，以下面的格式使用： put version (xtra "FileIO")。

■ getOSDirectory —— 是这个Xtra的一部分，但不要求任何引用变量，以下面的格式使用：put getOSDirectory()。它返回Mac系统文件夹或Windows目录的路径。

文件的读和写操作需要有多行程序。然而，一个这种功能的处理程序可以反复使用许多次，下面是一个处理程序，它提示用户输入一个用于读操作的文本文件，然后返回该文本文件的内容：

```
on openAndReadText
  -- create the FileIO instance
  fileObj = new(xtra "FileIO")

  -- set the filter mask to text files
  if the platform contains "mac" then
    setFilterMask(fileObj, "TEXT")
  else
    setFilterMask(fileObj, "Text Files,*.txt,All Files,*.*"┐ )
  end if

  -- open dialog box
  filename = displayOpen(fileObj)

  -- check to see if cancel was hit
  if filename = " " then return " "

  -- open the file
  openFile(fileObj,filename,1)

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

  -- read the file
  text = readFile(fileObj)

  -- close the file
  closeFile(fileObj)
```

```
  --return the text
  return text
end
```

如果我们想用这个处理程序读取我们已知其名称的文件，只要将 filename变量作为一个参数传递给处理程序，并取消对 setFilterMask和displayOpen命令的引用就可以了。

与这个处理程序的功能相对的处理程序是用于向文件中存储文本的。此处理程序负责所有任务，直到把新文件的文件类型设置为 Mac的SimpleText文件。被放置在文件中的文本作为参数被传送。

```
on saveText text
  -- create the FileIO instance
  fileObj = new(xtra "FileIO")

  -- set the filter mask to text files
  if the platform contains "mac" then
    setFilterMask(fileObj, "TEXT")
  else
    setFilterMask(fileObj, "Text Files,*.txt,All Files,*.* ")
  end if

  -- save dialog box
  filename = displaySave(fileObj, " " , " ")

  -- check to see if cancel was hit
  if filename = " " then return FALSE

  -- create and open the file
  createFile(fileObj,filename)
  openFile(fileObj,filename,2)

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

  -- write the file
  writeString(fileObj,text)

  -- set the file type
  if the platform contains "Mac" then
    setFinderInfo(fileObj, "TEXT ttxt")
  end if

  -- close the file
  closeFile(fileObj)
  return TRUE
end
```

on saveText和on openAndReadText处理程序可以用各种方法自定义，以适应各种不同的用途。例如，我们可以使用 getOSDirectory()函数，找到操作系统的路径，并向 Preferences(预置)文件夹中存储文件。还可以用 the pathname 属性获得Director应用程序或放映机的路径，并

把文件存储在那里。

16.10 文本和字符串的故障排除

■ 记住，当比较字符串或使用 contains或offset这类比较函数时， Lingo语言对大小写不敏感。所以，“ Abc ”与“ aBc ”相同。

■ 在Mac和Windows上，普通字母和数字的ASCII码是完全相同的，所有非符号字体也是完全相同的。事实上，几乎所有 32~127之间的ASCII字符对所有字体来说都是相同的。然而，127以上的字符在不同字体上是十分不同的，并且甚至不同平台的相同字体也是不同的。还有几个特殊的符号，如货币符号和重音符号等。参见第35章“跨平台问题”里的35.1.1节“字体”，会得到有关字体映射的信息。

■ 如果在Director 6.5或更早期的版本中使用 the mouseLine，注意它与在Director 7.0中的表现十分不同。如果光标移到某行上的 Return字符右边，这将表示下一行。

■ 如果在Director 6.5或更早的版本中对域使用 hilite命令，注意它与Director 7.0有所不同。如果我们试着使某一域的最后一行突出显示，只有该行的字符被突出。这种突出显示状态并不延伸到域的整个宽度。

■ 如果对可编辑的域或文本演员的输入进行限制时，记住应当允许使用 BACKSPACE字符。否则，如果用户录入出现错误时，可能不能清除。

16.11 你知道吗

■ 我们能够强制Director作为我们的RTF-HTML的转换器，方法是输入 RTF文件，然后获得那个演员的html属性。反向转换也可以。

■ 在消息窗口中，使用put interface (xtra “ FileIO ”)格式，可以获得所有FileIO的命令和函数的清单。该函数对多数 Xtra都起作用。

■ 如果不想用put命令将文本插入文本演员的前面、后面或中间，我们能够使用一些未记入正式文件的Lingo语言来实现： setContentsBefore、setContentsAfter和setContents。例如，我们可以写成： member ("myText").setContentsBefore ("abc")，以将“ abc ”放在演员中的文本之前，或只用普通的member ("myText").setContents ("abc")，以完全代替演员中的内容。

■ 如果我们输入一个在 Windows中创建的文本文件，可能看见在每一行开始处有一些额外的块字符。它们是新行字符(newline characters)。Director和现在的大部分文字处理程序已不使用它们。为了去掉它们，只要编写一个循环查看程序，以查看每个字符是否与 numToChar(10)不同，如果匹配，则删除它。或者，我们可以在循环查看中使用 offset函数，以快速地发现并删除它们。