第18章 一些FORMAT秘诀

Common Lisp的FORMAT函数和扩展的LOOP宏，是Common Lisp在许多用户中引起强烈反响的两个特性。对于FORMAT函数，有些人喜欢它，而另一些人讨厌它。[1]

FORMAT的爱好者们因为它的强大威力和简洁而喜欢它，它的反对者们则由于其潜在的误用和不透明性而讨厌它。复杂的FORMAT控制字符串有时就像是一行乱码，但FORMAT仍然受到一些Common Lisp程序员们的欢迎，他们希望能够生成少许人类可读的输出而无需手工编写大量的输出生成代码。尽管FORMAT的控制字符串可能是晦涩难懂的，但至少单一的FORMAT表达式还不致使事情变得太糟。例如，假设你想要将一个列表中的值以逗号分隔打印出来，可以写成下面这样：

```
(loop for cons on list
    do (format t "~a" (car cons))
    when (cdr cons) do (format t ", "))
```

这还不算太糟，但任何读到这些代码的人不得不在大脑里解析它，然后发现它所做的无非是向标准输出打印list的内容。另一方面，你可以立即说出下面的表达式正在以某种形式向标准输出打印list：

```
(format t "~{~a~^, ~}" list)
```

如果你关心该输出的具体形式，那么需要仔细分析控制字符串；但如果你只是想要第一时间估计出这段代码的用途，那么这是立即可以做到的。

不管怎么说，你应当至少可以读懂FORMAT，并且在你加入支持或反对FORMAT的阵营之前，有必要先知道它究竟能干什么。理解FORMAT的基础也是重要的，因为其他标准函数，诸如下一章将讨论的用来抛出各种状况的函数，都使用FORMAT风格的控制字符串来生成输出。

进一步说，FORMAT支持三种相当不同类型的格式化：打印表中的数据，美化输出S-表达式，以及使用插入的值生成人类可读的消息。现在将表格中的数据作为文本打印已经有些过时了，它是Lisp几乎和FORTRAN一样老的象征之一。事实上，一些可以用来在定长字段中打印浮点值的

① 当然，多数人认识到不值得在一门编程语言里将它实现出来，并且可以没有障碍地使用或不使用它。另一方面，有趣的是Common Lisp所实现的这两种特性，本质上是使用了不基于S-表达式语法的领域相关语法。FORMAT控制字符串的语法是基于字符的，而扩展的LOOP宏采用了由LOOP关键字所描述的语法。对于FORMAT和LOOP“不够Lisp化”这一常见批评恰恰反映了Lisp程序员们真的很喜欢S-表达式语法。

www.TopSage.com

指令相当直接地来源于FORTRAN的编辑描述符，它们在FORTRAN中用来读取和打印组织成定长字段的数据列。不过，将Common Lisp作为FORTRAN的替代品来使用超出了本书的范围，因此我不会讨论FORMAT的这些方面。

美化输出同样超出了本书的范围——并不是因为它们过时，而只是因为这是一个太大的主题。简单地说，Common Lisp精美打印器是一个可定制的系统，用来打印包括但不限于S-表达式的块结构数据，其中需要变长的缩进和动态添加的断行。它在需要的时候是非常有用的东西，但在日常编程中并不常用。[①]

因此，我将聚焦在FORMAT中可以使用插入的值来生成人类可读字符串的那部分内容。即便以这种方式限定范围，仍然谈及大量内容。你不必要求自己记住本章中所描述的每一个细节。只使用少量FORMAT用法通常就够了。我将首先描述FORMAT最重要的特性，究竟对它理解到何种程度完全取决于你自己。

18.1 FORMAT 函数

如同你在前面章节里看到的，FORMAT函数接受两个必要的参数：一个是用于输出的目的地，另一个是含有字面文本和嵌入指令的控制字符串。任何附加的参数都提供了用于控制字符串中指令并插入到输出中的值。我把这些参数称为格式化参数（format argument）。

FORMAT的第一个参数，用于输出的目的地，它可以是T、NIL、一个流或一个带有填充指针的字符串。T是流*STANDARD-OUTPUT*的简称，而NIL会导致FORMAT将输出生成到一个字符串中并随后返回。[②]如果目的地是一个流，那么输出将写到该流中。而如果目的地是一个带有填充指针的字符串，那么格式化的输出将被追加到字符串的结尾，并且填充指针也会作适当调整。除了FORMAT在目的地是NIL时返回一个字符串以外，其他情况下FORMAT均返回NIL。

第二个参数，控制字符串，在本质上是一段用FORMAT语言写成的程序。FORMAT语言完全不是Lisp风格的——其基本语法是基于字符而不是S-表达式的，并且它是为简洁性而非易于理解而优化的。这就是为什么一个复杂的FORMAT控制字符串可以最终看起来像是一行乱码。

多数FORMAT指令简单地以一种或另一种形式将参数插入到输出中。某些指令，诸如~%，可以导致FORMAT产生一个换行而不会使用任何参数。而其他的指令，如同你将要看到的，可以使用超过一个参数。有个指令甚至允许你在参数列表中跳动从而多次处理同一个参数，或是在特定情况下跳过特定参数。在讨论特定指令之前，我们首先了解一下指令的一般语法。

---

① 对于精美打印器感兴趣的读者可以阅读Richard Waters的论文“XP: A Common Lisp Pretty Printing System”。它是一个对后来合并到Common Lisp中的精美打印器的描述。你可以从ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1102a.pdf下载它。

② 稍有混淆的一点是，许多其他的I/O函数也接受T和NIL作为流标识符，但带有不同的语义：作为流标识符，T代表双向流*TERMINAL-IO*，而NIL在作为输出流时代表*STANDARD-OUTPUT*，作为输入流时代表*STANDARD-INPUT*。

www.TopSage.com

TopSage.com

18.2 FORMAT 指令

所有的指令都以一个波浪线（~）开始并终止于单个字符。字符可以使用大写或小写来书写。某些指令带有前置参数（prefix parameter），它们紧跟在波浪线后面，由逗号分隔，用来控制诸如在打印浮点数时小数点后打印多少位之类的事情。例如~$指令，它是用来打印浮点值的指令，默认情况下在小数点之后打印两位数字。

```
CL-USER> (format t "~$" pi)
3.14
NIL
```

不过，通过使用前置参数，可以指定它打印出的小数位数，比如五位小数，像下面这样：

```
CL-USER> (format t "~5$" pi)
3.14159
NIL
```

前置参数的值既可以是写成十进制的数字，也可以是字符，字符的书写形式是一个单引号后接想要的字符。前置参数的值还可以通过两种方式从格式化参数中获得：前置参数v导致FORMAT使用一个格式化参数并将其值用作前置参数，而前置参数#将被求值为剩余的格式化参数的个数。例如：

```
CL-USER> (format t "~v$" 3 pi)
3.142
NIL
CL-USER> (format t "~#$" pi)
3.1
NIL
```

我将在18.7节中给出一些如何使用#参数的实际例子。

你也可以完全省略前置参数。不过，如果你想要指定一个参数但不指定它前面的那个，你必须为每个未指定的参数加上一个逗号。例如~F指令，它是另一个用来打印浮点值的指令，也接受一个参数来控制需要打印的数的十进制位，但这是第二个参数而不是第一个。如果你想要使用~F将数字打印为五个十进制位，那么可以写成这样：

```
CL-USER> (format t "~,5f" pi)
3.14159
NIL
```

你还可以使用冒号和@修饰符（modifier）来调整某些指令的行为，它放在前置参数之后、指令的标识字符之前。这些修饰符可以细微地改变指令的行为。例如，使用冒号修饰符，以十进制输出整数的~D 指令将在输出数字时每三位用逗号分隔，而“@”修饰符可以使~D在数字为正时带上加号。

```
CL-USER> (format t "~d" 1000000)
1000000
NIL
CL-USER> (format t "~:d" 1000000)
1,000,000
NIL
```

www.TopSage.com

```
CL-USER> (format t "~@d" 1000000)
+1000000
NIL
```

在合理的情况下，可以组合使用冒号和@修饰符来同时得到两种调整。

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

在两个修改行为不能有意义地组合在一起的那些指令中，同时使用两个修饰符要么是未定义的，要么给出第三个含义。

18.3　基本格式化

现在来看特定指令。我将从几个最常用的指令开始，包括一些前面章节里已经提到的。

最通用的指令是~A，它使用一个任何类型的格式化参数，并将其输出成美化（人类可读）形式。例如，字符串输出成没有引用标记或转义字符的形式，而数字输出成正常的方式。如果你只是想产生一个人能看懂的值，那么这个指令是最合适的。

```
(format nil "The value is: ~a" 10)          → "The value is: 10"
(format nil "The value is: ~a" "foo")       → "The value is: foo"
(format nil "The value is: ~a" (list 1 2 3)) → "The value is: (1 2 3)"
```

一个紧密相关的指令是~S，它同样使用一个任何类型的格式化参数并输出它。不过，~S试图将输出生成为可被**READ**读回来的形式。这样，字符串将被包围在引用标记中，而符号在必要的时候是包限定的，等等。那些不带有可读表示的对象将被打印成不可读对象语法#<>。使用一个冒号修饰符，~A和~S指令都可以将**NIL**输出成()而不是**NIL**。~A和~S指令也都接受最多四个前置参数，它们用于控制是否在值的后面（或者当使用@修饰符时在值的前面）添加占位符，但这些参数只在生成表格数据时才是真正有用的。

其他两个最常用的指令是用来产生换行的~%，以及用来产生新行的~&。两者的区别在于，~%总是产生换行，而~&只在当前没有位于一行开始处时才产生换行。这对于编写松散耦合的函数特别有用，其中每个函数都生成一块输出，然后这些输出需要以不同方式组合在一起。例如，如果一个函数生成了以换行（~%）结尾的输出，而另一个函数生成了一些以新行（~&）开始的输出，那么当依次调用它们的时候就不必担心会产生一个额外的空行。这两个指令都可以接受单个前置参数来指定想要产生的换行的个数。~%指令会简单地输出那些换行符，而~&指令会输出$n-1$或n个换行，具体取决于它是否在一行的开始处输出。

不太常用的相关指令是~~，它导致**FORMAT**产生一个字面波浪线。与~%和~&相同，它可以通过一个数字来参数化地控制产生多少个波浪线。

18.4　字符和整数指令

除通用指令~A和~S外，**FORMAT**还支持一些指令用来以特定方式输出指定类型的值。这些指

www.TopSage.com

令中最简单的是~C指令，它用来输出字符。它不接受前置参数，但可用冒号和@修饰符进行修改。在不进行修改的情况下，除了它只能工作在字符上以外，其行为和~A没有区别。修改后的版本更加有用。使用冒号修饰符，~:C可以将诸如空格、制表符和换行符这些不可打印的字符按它们的名字输出。当你想要向用户输出关于某些字符的信息时，这个指令是非常有用的。例如，下面的形式

```
(format t "Syntax error. Unexpected character: ~:c" char)
```

可以产生下面的信息

```
Syntax error. Unexpected character: a
```

但也可以像下面这样：

```
Syntax error. Unexpected character: Space
```

使用@修饰符，~@C将按Lisp的字面字符语法输出字符。

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

同时使用冒号和@修饰符，~C指令可以打印出额外的信息：如果该字符要求特殊的按键组合，那么在键盘上输入该字符的方式也将打印出来。例如，在Macintosh上，在特定应用中可以通过按下Control键，然后输入@来键入一个空字符（在ASCII或ISO-8859-1和Unicode等ASCII超集中字符编码为0的字符）。在OpenMCL中，如果使用~:@C指令来打印空字符，那么它将告诉你下面的信息：

```
(format nil "~:@c" (code-char 0)) → "^@ (Control @)"
```

尽管如此，并非所有的Lisp都实现了~C指令的这个方面。而且就算它们实现了，结果也可能不是精确的。例如，如果在SLIME中运行OpenMCL，那么C-@键组合将被Emacs劫持，并调用set-mark-command。[①]

那些致力于输出数字的格式化指令构成了另一个重要的分类。尽管你可以使用~A和~S来输出数字，但如果想要更好地控制它们被打印的形式，那么就需要使用特定于字符的指令了。这些数值指令可以分成两个子类别：用来格式化整数值的指令以及用来格式化浮点值的指令。

有五个紧密相关的指令可以格式化整数值：~D、~X、~O、~B和~R。最常用的是~D指令，它以十进制输出整数。

```
(format nil "~d" 1000000) → "1000000"
```

如同我前面提到的，使用冒号修饰符会在输出中添加逗号。

```
(format nil "~:d" 1000000) → "1,000,000"
```

而使用@修饰符，它总是打印一个正负符号。

```
(format nil "~@d" 1000000) → "+1000000"
```

① ~C指令的这个变体在像Lisp Machine这样的平台上更有意义，其中键击事件是由Lisp字符所表示的。

www.TopSage.com

并且这两个修饰符可以组合使用。

```
(format nil "~:@d" 1000000) → "+1,000,000"
```

第一个前置参数可以指定输出的最小宽度，而第二个参数可以指定一个用作占位符的字符。默认的占位符是空格，而占位符总是插入在数字之前。

```
(format nil "~12d" 1000000)    → "     1000000"
(format nil "~12,'0d" 1000000) → "000001000000"
```

这些参数在格式化日期这样的固定宽度格式时是很有用的。

```
(format nil "~4,'0d-~2,'0d-~2,'0d" 2005 6 10) → "2005-06-10"
```

第三和第四个参数是与冒号修饰符配合使用的：第三个参数指定了用作数位组之间分隔符的字符，而第四个参数指定了每组中数位的数量。这些参数默认为逗号和数字3。这样，你可以使用不带参数的~:D指令来将大整数输出成用于美国的标准格式，但也可以使用~,,'.,4D将逗号改成句点并将分组从3调整到4。

```
(format nil "~:d" 100000000)       → "100,000,000"
(format nil "~,,'.,4:d" 100000000) → "1.0000.0000"
```

注意，你必须使用逗号来保留未指定的宽度和占位符参数的位置，从而允许它们保持各自的默认值。

除了将数字分别输出成十六进制、八进制和二进制之外，~X、~O和~B指令与~D指令的工作方式相同。

```
(format nil "~x" 1000000) → "f4240"
(format nil "~o" 1000000) → "3641100"
(format nil "~b" 1000000) → "11110100001001000000"
```

最后，~R指令是通用的进制输出指令。它的第一个参数是一个介入2和36（包括2和36）之间的数字，用来指示所使用的进制。其余的参数与~D、~X、~O和~B指令所接受的四个参数一样，并且冒号和@修饰符也以相同的方式修改其行为。当不使用任何前置参数时，~R指令还有一些特殊行为。我将在18.6节里讨论它。

18.5　浮点指令

有四个格式化浮点值的指令：~F、~E、~G和~$，其中前三个是基于FORTRAN的编辑描述符的指令。它们多数用于以表格形式格式化浮点值，所以我将跳过这些指令的多数细节。尽管如此，你可以使用~F、~E和~$指令将浮点值插入到文本中。通用浮点指令~G组合了~F和~E指令的特性，从而使得其只在生成表格输出时才真正有意义。

~F指令以十进制格式输出其参数（该参数应当是一个数字[①]），并可以控制十进制小数点之后的数位数量。此外，~F指令在数字特别大或特别小时允许使用科学计数法。而~E指令在输出数

① 技术上来讲，如果该参数不是一个实数，那么~F应当像使用~D指令那样来格式化它，而如果该参数根本不是一个数字，其行为应当像~A指令那样，但并非所有实现都很好地遵守了这一约定。

www.TopSage.com

字时总是使用科学计数法。这两个指令都接受一些前置参数，但你需要关注的只有第二个参数，它控制在十进制小数点之后打印的位数。

```
(format nil "~f" pi)   → "3.141592653589793d0"
(format nil "~,4f" pi) → "3.1416"
(format nil "~e" pi)   → "3.141592653589793d+0"
(format nil "~,4e" pi) → "3.1416d+0"
```

~$指令和~F指令相似，但更简单一些。如同其名字所示，它用于输出货币单位。不带有参数时，它基本上等价于~,2F。为了修改十进制小数点之后打印的位数，你可以使用第一个参数，而第二个参数用来控制十进制小数点之前所打印的最小位数。

```
(format nil "~$" pi)    → "3.14"
(format nil "~2,4$" pi) → "0003.14"
```

~F、~E和~$三个指令都可以通过使用@修饰符来使其总是打印一个正负号。[①]

18.6 英语指令

用来生成人类可读消息的最有用的一些**FORMAT**指令，是那些用来产生英文文本的指令。这些指令允许将数字输出成英语单词，基于格式化参数的值来输出复数标识，并且为**FORMAT**的输出分段应用大小写转换。

我在18.4节中所讨论的~R指令，当不指定输出进制来使用时，它将数字打印成英语单词或罗马数字。当不带前置参数和修饰符使用时，它将数字输出成基数词。

```
(format nil "~r" 1234) → "one thousand two hundred thirty-four"
```

使用冒号修饰符，它将数字输出成序数。

```
(format nil "~:r" 1234) → "one thousand two hundred thirty-fourth"
```

而当使用@修饰符时，它将数字输出成罗马数字。同时使用@和冒号时，它产生“旧式风格”的罗马数字，其中4和9被写成IIII和VIIII而不是IV和IX。

```
(format nil "~@r" 1234)  → "MCCXXXIV"
(format nil "~:@r" 1234) → "MCCXXXIIII"
```

对于那些以给定形式无法表示的过大数字，~R将回退到与~D相同的行为。

为了生成带有正确复数化单词的消息，**FORMAT**提供了~P指令。如果某对应的参数不是1，它就简单地输出一个S。

```
(format nil "file~p" 1)  → "file"
(format nil "file~p" 10) → "files"
(format nil "file~p" 0)  → "files"
```

不过，一般情况下你将使用带有冒号修饰符的~P，这会使它重新处理前一个格式化参数。

① 这只是语言标准里所说的。出于一些原因，可能是源自于同一份古老的基础代码，一些Common Lisp实现并没有正确地实现~P指令的这个方面。

www.TopSage.com

大家网 TopSage.com

```
(format nil "~r file~:p" 1)  → "one file"
(format nil "~r file~:p" 10) → "ten files"
(format nil "~r file~:p" 0)  → "zero files"
```

使用@修饰符与冒号修饰符组合使用，~P将输出y或ies。

```
(format nil "~r famil~:@p" 1)  → "one family"
(format nil "~r famil~:@p" 10) → "ten families"
(format nil "~r famil~:@p" 0)  → "zero families"
```

很明显，~P不能解决所有复数化问题，并且对于生成其他语言的消息也没有帮助，但对于那些它可以处理的情况是很有用的。而我将很快讨论的~[指令可以提供更灵活的方式来有条件地输出FORMAT中的某些部分。

最后一个用来输出英语文本的指令是~(，它允许你控制输出文本中的大小写。每一个~(都要与一个~)成对使用，由控制字符串中两个标记之间的部分所生成的输出将被全部转化成小写。

```
(format nil "~(~a~)" "FOO") → "foo"
(format nil "~(~@r~)" 124)  → "cxxiv"
```

可以使用@符号来修改~(的行为，将一段文本中第一个词的首字母变成大写；使用冒号可以将所有单词首字母大写；而同时使用两个修饰符将使全部文本转化成大写形式。(这里所说的单词，指的是以字母和数字组成的字符序列，各单词间由既非字母又非数字的字符分隔。)

```
(format nil "~(~a~)" "tHe Quick BROWN foX")   → "the quick brown fox"
(format nil "~@(~a~)" "tHe Quick BROWN foX")  → "The quick brown fox"
(format nil "~:(~a~)" "tHe Quick BROWN foX")  → "The Quick Brown Fox"
(format nil "~:@(~a~)" "tHe Quick BROWN foX") → "THE QUICK BROWN FOX"
```

18.7 条件格式化

除了那些插入参数和修改其他输出的指令以外，FORMAT还提供了一些指令用来实现控制字符串之中的简单控制构造。其中之一是你在第9章里曾经用过的条件指令~[，该指令闭合于一个对应的~]，在它们之间是一组由~;所分隔的子句。~[指令的任务是选取一个子句，随后由FORMAT处理。在没有修饰符或参数的情况下，该子句用数值索引选择。~[指令使用一个格式化参数，它应当是一个数字，该指令取出第*n*个（从0开始的）子句，其中*n*是该参数的值。

```
(format nil "~[cero~;uno~;dos~]" 0) → "cero"
(format nil "~[cero~;uno~;dos~]" 1) → "uno"
(format nil "~[cero~;uno~;dos~]" 2) → "dos"
```

如果该参数的值大于子句的数量，那么不打印任何东西。

```
(format nil "~[cero~;uno~;dos~]" 3) → ""
```

如果最后一个子句分隔符是~:;而不是~;，那么最后一个子句将作为默认子句提供。

```
(format nil "~[cero~;uno~;dos~:;mucho~]" 3)   → "mucho"
(format nil "~[cero~;uno~;dos~:;mucho~]" 100) → "mucho"
```

www.TopSage.com

也可以使用一个前置参数来指定被选择的子句。尽管使用控制字符串中的字面值是没有意义的，但回顾一下作为前置参数的#代表需要处理的剩余参数的个数。这样，你可以定义一个像下面这样的格式字符串：

```
(defparameter *list-etc*
  "~#[NONE~;~a~;~a and ~a~:;~a, ~a~]~#[~; and ~a~:;, ~a, etc~].")
```

然后像这样使用它：

```
(format nil *list-etc*)                → "NONE."
(format nil *list-etc* 'a)             → "A."
(format nil *list-etc* 'a 'b)          → "A and B."
(format nil *list-etc* 'a 'b 'c)       → "A, B and C."
(format nil *list-etc* 'a 'b 'c 'd)    → "A, B, C, etc."
(format nil *list-etc* 'a 'b 'c 'd 'e) → "A, B, C, etc."
```

注意，上述控制字符串实际包含了两个“~[~]”指令，两个指令都使用了#来选择要使用的子句。第一个指令使用零到两个参数，第二个指令在需要的时候会再使用一个参数。**FORMAT**将默默忽略在处理控制字符串时没有被使用的任何参数。

如果使用冒号修饰符，那么~[将只含有两个子句。该指令使用单个参数，并在该参数为**NIL**时处理第一个子句，而在其他情况下处理第二个子句。在第9章里，你曾经使用该~[变体来生成“通过或失败”消息，像下面这样：

```
(format t "~:[FAIL~;pass~]" test-result)
```

注意，任何一个子句都可以是空的，但指令中必须含有~;作为分隔。

最后，借助@修饰符，指令~[可以只带一个子句。该指令使用一个参数，并且当它是非空时可以回过头来再次使用该参数，然后再处理其子句。

```
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 20)   → "x = 10 y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 nil)  → "x = 10 "
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil 20)  → "y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil nil) → ""
```

18.8 迭代

另一个你已经见过的**FORMAT**指令是迭代指令~{。该指令可以让**FORMAT**在列表的元素或者隐式的**FORMAT**参数列表上进行迭代。

不带修饰符时，~{使用一个格式化参数，它必须是一个列表。和~[指令必须与一个~]指令配对使用一样，~{指令也必须和一个闭合的~}成对使用。两个标记间的文本将作为一个控制字符串来处理，它们从~{指令所使用的列表中取得其参数。只要被迭代的列表尚有元素剩余，**FORMAT**将重复处理该控制字符串。在下面的示例中，~{使用单个格式化参数，列表(1 2 3)，然后处理控制字符串"~a, "，重复操作直到该列表的所有元素都已使用。

```
(format nil "~{~a, ~}" (list 1 2 3)) → "1, 2, 3, "
```

可是在输出中，列表最后一个元素后面跟了一个逗号和一个空格，这显得很让人讨厌。你可

www.TopSage.com

以使用~^指令来修复这点。在一个~{指令体内，当列表中没有元素剩余时，~^将令迭代立即停止且无须处理其余的控制字符串。这样，为了避免在列表的最后元素之后打印出逗号和空格，你可以在它们前面添加一个~^。

```
(format nil "~{~a~^, ~}" (list 1 2 3)) → "1, 2, 3"
```

在迭代过程的前两次里，处理~^时，列表中尚有未处理的元素。到了第三次的时候，在~a指令处理了3之后，~^将令**FORMAT**跳出迭代而不会打印出逗号和空格。

使用@修饰符，~{将把其余的格式化参数作为列表来处理。

```
(format nil "~@{~a~^, ~}" 1 2 3) → "1, 2, 3"
```

在一个~{...~}的主体中，特殊前置参数#代表列表中需要被处理的剩余项的个数，而不是剩余格式化参数的个数。你可以将#和~[指令一起使用，来打印用逗号分隔、并在最后一个列表项之前带有and的列表，就像下面这样：

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2 3)) → "1, 2, and 3"
```

不过，当列表是两个元素长度时，由于它添加了一个额外的逗号，上述表达式的输出并不正常。

```
(format nil "~{~a~#[~;, and ~:;, ~]~}" (list 1 2)) → "1, and 2"
```

你可以通过多种方式来修复这个问题。下面的方法利用了~@{嵌入到另一个~{或~@{指令中时所具有的行为——它迭代由外层~{迭代的列表中的剩余元素。你可以将它与~#[指令组合使用，从而使得下面的控制字符串按照英语语法来格式化列表：

```
(defparameter *english-list*
  "~{~#[~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~}")

(format nil *english-list* '())        → ""
(format nil *english-list* '(1))       → "1"
(format nil *english-list* '(1 2))     → "1 and 2"
(format nil *english-list* '(1 2 3))   → "1, 2, and 3"
(format nil *english-list* '(1 2 3 4)) → "1, 2, 3, and 4"
```

尽管这个控制字符串已经接近于只能写不能读的程度了，但如果你能花一点时间还是不难理解它的。外层的~{...~}将使用并迭代在一个列表上。整个迭代体随后由一个~#[...~]构成。这样每次通过迭代所产生的输出将取决于列表中待处理项的个数。通过使用~;子句分隔符来分拆~#[...~]指令，你可以看到它由四个子句所组成，而且因为它前置了一个~:;而不是普通的~;，所以最后一个是默认子句。第一个子句用于当还有零个元素需要处理时，其为空是合理的——如果没有元素需要处理了，那么迭代就该停止了。第二个子句处理只有一个元素的情况，它带有一个简单的~a指令。两元素的情况由"~a and ~a"所处理。而默认子句用来处理三个或更多元素的情形，它由另一个迭代子句所构成，这一次使用~@{来迭代由外层~{处理的列表的剩余元素。该迭代的主体可以正确处理三个或更多元素列表的控制字符串，在这种情况下是正确的。因为该~@{循环将消耗掉所有剩余的列表元素，而外层循环只迭代一次。

www.TopSage.com

如果你想要在列表为空时，打印出诸如<empty>这样的特殊标识，那么有几种方式可以做到这点。最简单的一种可能是把你想要的文本放在外层~#[的第一个（确切地说是第零个）子句里，并在外层迭代的闭合~}上添加一个冒号修饰符——该冒号会强制迭代至少运行一次，就算列表是空的，在这种情况下FORMAT将处理条件子句中的第零个子句。

```
(defparameter *english-list*
  "~{~#[<empty>~;~a~;~a and ~a~:;~@{~a~#[~;, and ~:;, ~]~}~]~:}")

(format nil *english-list* '()) → "<empty>"
```

令人惊奇的是，~{指令还通过不同的前置参数和修饰符组合提供了更多的变体。我不会详细讨论它们，简单而言，你可以使用一个整数前置参数来限制迭代的最大数量，以及通过一个冒号修饰符，列表（无论是一个实际列表还是由~@{指令所构造出的列表）中的每个元素其本身都必须是一个列表，并且后者的元素将用作~:{...~}指令中控制字符串的参数。

18.9 跳，跳，跳

一个更简单的指令是~*指令，它允许你在格式化参数列表中跳跃。在它的基本形式中，没有修饰符的情况下，它简单地跳过下一个参数，使用它而不输出任何东西。不过更常见的情况是，它和一个冒号修饰符一起使用，这使它可以向前移动，从而允许同一个参数被再次使用。例如，你可以使用~:*将一个数值参数按单词打印一次，再按数值打印一次：

```
(format nil "~r ~:*(~d)" 1) → "one (1)"
```

或者，你也可以组合~:*与~[来实现一个类似于~:P的不规则复数形式的指令。

```
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 0) → "I saw zero elves."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 1) → "I saw one elf."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 2) → "I saw two elves."
```

在这个控制字符串中，~R将格式化参数打印成一个基数。然后~:*指令回过头来使该数字再用作~[指令的参数，并在该数字是0、1或其他任何值时分别选择不同的子句。[①]

在一个~{指令中，~*可以跳过或恢复列表中的项。例如，可以像下面这样只打印一个plist中的键：

```
(format nil "~{~s~*~^ ~}" '(:a 10 :b 20)) → ":A :B"
```

还可以给~*指令一个前置参数。当没有修饰符或使用冒号修饰符时，该参数指定了向前或向后移动的参数个数，默认为1。当使用@修饰符时，该前置参数指定了一个用来跳跃的、绝对的、

① 如果你觉得“I saw zero elves”这种说法有点奇怪，那么可以使用一个更加精巧的格式字符串，其使用了~:*的另一种用途：

```
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 0) → "I saw no elves."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 1) → "I saw one elf."
(format nil "I saw ~[no~:;~:*~r~] el~:*~[ves~;f~:;ves~]." 2) → "I saw two elves."
```

www.TopSage.com

以零开始的参数索引，默认为0。在你想要使用不同的控制字符串来为同一组参数生成不同的消息，并且不同的消息需要使用不同顺序的这些参数时，~*的@变体可能是有用的。[1]

18.10　还有更多……

还有更多的内容。我还没有提到~?指令，它可以从格式化参数中获取控制字符串；还有~/指令，它允许你调用任意函数来处理下一个格式化参数。还有所有用于生成表格和优美打印输出的全部指令。但在本章中所讨论的这些指令对于目前来说应当足够使用了。

在下一章里，你将接触Common Lisp的状况系统，类似于其他语言的异常和错误处理系统。

① 这类问题可能在试图本地化一个应用程序并将人类可读消息翻译成不同语言时出现。FORMAT可以对这些问题中的一些提供帮助，但绝不意味着它是一个全功能的本地化系统。

www.TopSage.com