第7章 基本函数：car、cdr、cons

在 Lisp 中，car、cdr 和 cons 都是基本函数。cons 函数用于构造列表，car 和 cdr 函数则用于拆分列表。

在详细分析 copy-region-as-kill 函数的过程中，将看到 cons函数以及 cdr函数的两个变种：setcdr 和 nthcdr。（参见8.5节“copy-region-as-kill函数”。）

cons 函数的函数名不是没有理由的：它是“construct”(构造)一词的缩写。相比之下，函数名 car 和 cdr 的来历则很深奥：car是“Contents of the Address part of the Register”（寄存器地址部分的内容）短语的首字母缩写；而 cdr (读作“could-er”)是“Contents of the Decrement part of the Register”（寄存器后部内容）短语的首字母缩写。这些短语是指在开发 Lisp 的早期使用的计算机上的特定硬件部分。除了有点陈腐外，这些短语与 25 年后接触 Lisp 的任何人都完全无关。尽量如此，仍然有一些勇敢的学者已经开始对这些函数采用其他似乎更有道理的名字，但是这些术语仍旧被使用。特别是，因为这些术语常用在Emacs Lisp 源代码中，所以在这本书中继续使用它们。

7.1 car和cdr 函数

一个列表的 car，简单地说，就是返回这个列表的第一个元素。因而列表 (rose violet daisy buttercup)的 car 就是 rose。

如果你在 GNU Emacs 的 Info 中阅读这份文档，对下面的表达式求值就可以看到上面说的这个例子：

```
(car '(rose violet daisy buttercup))
```

对这个表达式求值之后，rose 一词将出现在回显区中。

很明显，car函数的一个更为合适的名字可能是first。人们也的确经常这样建议。

car不将第一个元素从列表中移走，它仅仅报告列表的第一个元素是什么。car应用到一个列表之后，列表依旧是它自己。用术语来说，car 是“非破坏性”的。这种特性是非常重要的。

一个列表的 cdr 就是这个列表的其余部分（除第一个元素外的其余部分），也就是说，cdr 函数返回列表中第一个元素后的所有内容。因而，列表 '(rose violet daisy buttercup) 的 car 是 rose，而这个列表的其余部分就是 (violet daisy buttercup)，这正是 cdr 返回的值。

用通常的方法对下面的表达式求值就可以看到这一点：

```
(cdr '(rose violet daisy buttercup))
```

当对它求值时，(violet daisy buttercup) 将出现在回显区中。

像 car 一样，cdr 也不从列表中移走任何元素——它仅仅返回包含列表的第二个和随后的所有元素列表。

顺便提一下，在这个例子中，花的列表带有单引号。如果没有这个单引号，Lisp 解释器将试图通过调用 rose 作为一个函数来对这个列表求值。在这个例子中，我们并不希望这样做。

很明显，cdr 函数的一个更为合适的名字可能是 rest。

（这里有一个教训：今后当你给一个新函数取名字时，要仔细考虑你所做的一切，因为在你无法预见的将来，你可能被这个名字弄得痛苦不堪。这份文档中继续使用这些名字的原因是 Emacs Lisp 的源代码中使用了它们，而且如果我不使用它们，读者在阅读这些代码时将很困难；但是你自己确实要避免使用这些术语。那样的话，今后的人们会感激你的。）

当对一个由符号组成的列表使用 car 和 cdr 时，例如对列表 (pine fir oak maple)使用car和cdr时，由 car 返回的列表的元素是符号 pine，没有任何括号在它两边。pine是这个列表的第一个元素。然而，列表的 cdr 是一个列表本身，即 (fir oak maple)。用通常的办法对下面的例子求值就可以看到这种不同：

```
(car '(pine fir oak maple))

(cdr '(pine fir oak maple))
```

另一方面，在一个列表的列表中，其第一个元素本身就是一个列表。car 返回作为一个列表的这第一个元素。例如，下面的列表包含三个子列表，一个是猛兽的列表，一个是食草动物的列表，一个是海洋动物的列表：

```
(car '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

在这个例子中，列表的第一个元素（或者列表的 car）就是猛兽的列表，(lion tiger cheetah)。其余部分就是 ((gazelle antelope zebra) (whale dolphin seal))。

```
(cdr '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

值得在此说明的是，car 和 cdr 函数都是“非破坏性”的——也就是说，它们不改变它们所作用的列表。这一点对于如何使用这两个函数非常重要。

而且，在第1章中讨论原子时，曾经说过，在 Lisp 中，某些类型的原子(例如数组)能够被分割成几个部分；但是这种分割的机制与分割一个列表的机制是不同的。只要是论及 Lisp，列表中的原子就是不可分的。（参见1.1.1节，“ Lisp 原子”。）car 和 cdr 函数用于分割一个列表，并且是 Lisp 的基本操作。因为它们不能分割一个数组或者对数组的一部分操作，所以一个数组被认为是一个原子。相反，另外一些基本的函数(例如 cons)，能够组成或构建一个列表，但是不能构建一个数组。（数组是由与数组相关的函数来处理的。参见《*GNU Emacs Lisp 技术手册*》中的“数组”一节。）

7.2 cons函数

cons 函数可以构造列表，它的作用与 car 和 cdr 函数正好相反。例如，cons 函数能够被用于将三个元素的列表(fir oak maple)变成四个元素的列表：

```
(cons 'pine '(fir oak maple))
```

对这个列表求值后，你将看到

```
(pine fir oak maple)
```

出现在回显区中。cons 函数将一个新元素放到一个列表的开始处，它往列表中插入元素。

cons 必须有一个待插入元素的列表[⊖]。绝对不能从一无所有开始。如果正在创建一个列表，首先至少需要提供一个空列表。下面是一系列 cons函数，它们构建了一个花的列表。如果你在GNU Emacs的Info 中阅读这份文档，可以用通常的方法对下面的每一个表达式求值，表达式的值是打印在“⇒”之后的文本，你可以将其读作“求值得”：

```
(cons 'buttercup ())
     ⇒ (buttercup)

(cons 'daisy '(buttercup))
     ⇒ (daisy buttercup)

(cons 'violet '(daisy buttercup))
     ⇒ (violet daisy buttercup)

(cons 'rose '(violet daisy buttercup))
     ⇒ (rose violet daisy buttercup)
```

在第一个例子中，空列表是()，由 buttercup 组成的列表因此构造出来。就像你能够看到的，空列表不在构造后的列表中出现。你所看到的是(buttercup) 列表。空列表不作为列表的一个元素，因为空列表中什么也没有。一般来说，一个空列表是不可见的。

第二个例子中，(cons ' daisy ' (buttercup)) 通过将 daisy 放到buttercup 之前构建了一个新的、两元素的列表；而第三个例子则将 violet插入到 daisy 和 buttercup 之前构建了一个三元素列表。

查询列表的长度：length函数

通过使用 Lisp的 length函数，你能够找出一个列表中有多少元素。例如，

```
(length '(buttercup))
     ⇒ 1

(length '(daisy buttercup))
     ⇒ 2

(length (cons 'violet '(daisy buttercup)))
     ⇒ 3
```

⊖ 实际上，可以将一个元素 cons 进一个原子来生成一个带点的偶对。带点偶对的内容超过了本书的范围，详情请参考《*GNU Emacs Lisp* 技术手册》。

在上面的第三个例子中，cons 函数被用于构建一个三元素列表，这个列表随后被传递给 length 函数作为其参量。

也可以用 length 函数来计算空列表中元素的个数：

```
(length ())
     ⇒ 0
```

就像你会想到的，空列表中没有任何元素。

一个有趣的试验是，如果你试图找出一个非列表对象的长度，将会发生什么？也就是说不给 length 函数传递参量，哪怕是一个空列表，那会发生什么？

```
(length )
```

如果你对这个表达式求值，你看到的将是一个错误消息：

```
Wrong number of arguments: #<subr length>, 0
```

这意味着这个函数接收了错误数量的参量（0），而它希望得到其他数目的参量。在这个例子中，length 函数希望得到一个参量，这个参量是一个函数正欲求其长度的列表。（注意，一个列表是一个参量，即使这个列表有许多元素也是如此。）

错误消息中的“#<subr length>”是函数名。它是以一种特殊形式写出来的，“#subr”是指 length 函数是一个用 C 语言编写的、而不是用 Emacs Lisp 编写的基本函数。（“subr”是 “subroutine”（子例程）一词的缩写。）更多关于子例程的资料，参见《*GNU Emacs Lisp* 技术手册》中的“什么是函数？”一节。

7.3 nthcdr函数

nthcdr 函数与 cdr 函数联系在一起。它所做的就是重复地取列表的 cdr。

如果你取列表 (pine fir oak maple) 的cdr，你将得到列表 (fir oak maple)。如果你对返回值重复取 cdr，你将得到 (oak maple) 列表。（当然，由于函数不改变列表，因此对一个原始列表重复取 cdr 将得到原始的cdr。还可以对列表的 cdr 取 cdr，等等。）如果你继续这样做，最后你将返回一个空列表，在这个例子中，最后显示的不是一个空列表 ( )，而是显示 nil。

总的来说，下面是一系列重复的 cdr，“⇒”后面的文本显示的是返回值：

```
(cdr '(pine fir oak maple))
     ⇒(fir oak maple)

(cdr '(fir oak maple))
     ⇒ (oak maple)

(cdr '(oak maple))
     ⇒(maple)

(cdr '(maple))
     ⇒ nil

(cdr 'nil)
     ⇒ nil
```

```
(cdr ())
     ⇒ nil
```

你也可以无需输出值而连续使用 cdr，就像这样：

```
(cdr (cdr '(pine fir oak maple)))
     ⇒ (oak maple)
```

在这个例子中，Lisp 解释器首先对最内层的列表求值。最内层的列表带有引号，因此它仅仅将这个列表本身传递给 cdr。这个 cdr 将一个由原来列表的第二个元素以及其后的其他元素组成的列表传递给外部的 cdr。这个 cdr 则将产生由原始列表的第三个元素以及其后的其他元素组成的列表。在这个例子中，cdr 函数被重复使用，并返回一个由原始列表的除头两个元素之外的元素组成的列表。

nthcdr 函数的功能就像重复调用 cdr 函数一样。在后续的例子中，参量 2 以及一个列表被传递给nthcdr函数，返回值是由原始列表中除头两个元素之外的元素组成的列表，这与重复两次使用 cdr 函数得到的结果完全一样：

```
(nthcdr 2 '(pine fir oak maple))
     ⇒ (oak maple)
```

使用 4 个元素的原始列表，可以看到当给 nthcdr 函数传递不同的数字参量时会发生什么情况，例子中使用的参量包括0、1 和 5：

```
;;留下列表全部。
(nthcdr 0 '(pine fir oak maple))
     ⇒ (pine fir oak maple)

; ; 返回移去第一个元素的列表。
(nthcdr 1 '(pine fir oak maple))
     ⇒ (fir oak maple)

; ; 返回移去 3 个元素的列表。
(nthcdr 3 '(pine fir oak maple))
     ⇒ (maple)

; ; 返回移去 4 个元素的列表。
(nthcdr 4 '(pine fir oak maple))
     ⇒ nil

; ; 返回一个移去所有元素的列表。
(nthcdr 5 '(pine fir oak maple))
     ⇒ nil
```

值得一提的是，就像 cdr 函数一样，nthcdr 函数也不改变原始列表——这个函数是非破坏性的。这一点与 setcar 和 setcdr 函数截然不同。

7.4 setcar函数

从函数的名字，你可能已经猜测到，setcar 和 setcdr 函数将一个列表的 car和 cdr 设置为一个新的值。不像 car 和 cdr 函数不改变原始列表，setcar 和 setcdr 这两个函数实

际上改变了原始列表。了解它们如何工作的一个途径就是不断尝试。我们将从 setcar 函数开始。

首先，构造一个列表并用 setq 函数将这个列表赋值给一个变量。下面是一个动物的列表：

```
(setq animals '(giraffe antelope tiger lion))
```

如果你是在 GNU Emacs的Info 中阅读这份文档，就可以用通常的方法对上面的表达式求值，将光标置于表达式末尾，键入 C-x C-e。(我写到这里的时候就是这么做的。在计算环境中安装解释器的好处之一就在于此。)

当对变量 animals 求值时，我们看到它被绑定到列表 (giraffe antelope tiger lion)上：

```
animals
     ⇒ (giraffe antelope tiger lion)
```

这也就是说，变量animals 指向了列表(giraffer antelope tiger lion)。

接下来，要给 setcar 函数设置两个参量，一个是变量 animals，一个是带引号的符号 hippopotamus；这是通过编写一个三元素列表 (setcar animals 'hippopotamus) 并随后用通常的方法对它求值来完成的：

```
(setcar animals 'hippopotamus)
```

对这个表达式求值之后，再对变量 animals 求值。你将会看到 animals 指向的列表已经改变了：

```
animals
     ⇒ (hippopotamus antelope tiger lion)
```

这个列表的第一个元素giraffe已经被 hippopotamus 取代了。

因此我们能够看到：setcar 函数不是像 cons 函数那样在列表中增加一个元素；它用新元素 hippopotamus 取代原来的元素 giraffe，它改变了列表。

7.5 setcdr函数

setcdr 函数与函数 setcar 相似，不同之处仅在于这个函数替换列表的第二个以及其后的所有元素，而不是列表的第一个元素。

为了了解这个函数是如何工作的，通过对下面的表达式求值来将驯养动物的列表赋值给一个变量：

```
(setq domesticated-animals '(horse cow sheep goat))
```

如果现在对这个变量求值，将返回一个列表：(horse cow sheep goat)。

```
domesticated-animals
     ⇒ (horse cow sheep goat)
```

接下来，要给 setcdr 函数设置两个参量，一个是变量名(这个变量有一个列表作为其值)，另一个参量是一个列表(它是第一个列表的 cdr 将被设置的值)：

```
(setcdr domesticated-animals '(cat dog))
```

如果对这个表达式求值，列表 (cat dog) 将出现在回显区中。这就是这个函数的返回值。但是我们感兴趣的是这个函数的附带效果，对变量 domesticated-animals 求值就可以看到这个附带效果：

```
domesticated-animals
     ⇒ (horse cat dog)
```

的确，驯养动物的列表已经从 (horse cow sheep goat) 变成了 (horse cat dog)。也就是说，列表的 cdr 已经从 (cow sheep goat) 变成了 (cat dog)。

7.6 练习

通过对几个 cons表达式求值，来构建一个四元素的、有关鸟的列表。试一试，当你对列表本身使用 cons 函数时会发生什么？用一种鱼取代这个列表的第一个元素。用其他鱼的列表取代这个列表的其余部分。